Computer Architectures for Mobile Computer Vision Systems

by

Jason Lavar Clemons

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2013

Doctoral Committee:
Professor Todd M. Austin, Chair
Professor Scott Mahlke
Assistant Professor Silvio Savarese
Assistant Professor Thomas F. Wenisch

© Jason Lavar Clemons

All Rights Reserved
2013

To my parents and wife.
Thank you.

Acknowledgments

I would like begin by thanking my advisor, Professor Todd Austin, for his guidance during my graduate career. It was his persistence that motivated my return to graduate school and since my return he has been there to help me grow as a researcher. His advice on research, paper writing, and presentations has been invaluable and he has given me a strong foundation as I move forward in the world of research, and his advice on life has seen me through challenging times. I hope to pass on what he has taught me.

The members of my committee, Professors Silvio Savarese, Thomas Wenisch, and Scott Mahlke, also deserve a large amount of gratitude. Their support and suggestions has greatly improved this dissertation and insured my research was of the highest quality. Each of them has been a great resource during my graduate career. I would especially like to thank Professor Savarese for his guidance on understanding computer vision. It is my belief that everything I understand about computer vision is because of the foundation he gave me.

I have been fortunate enough to work with very talented people over the past few years. I had the opportunity to work closely with Andrew Jones, Robert Perricone, Max Seiden, Haishan Zhu, Andrea Pelligrini, Ying-Ze Bao and Yixin Luo here at Michigan. They are all very talented and I hope to work with them again soon. I was given the opportunity to intern at NVIDIA where I worked closely with James Fung and Jason Sanders, both of whom are very talented and I wish them well in their careers. I would like to acknowledge NVIDIA's support and thank them for the tools and software they donated to my research. I would also like to thank Vinay Sharma, Erik Rainey, Mike Polley, Branislav Kisacanin, Bruce Flinchbaugh, Goksel Dedeoglu and Vivienne Sze from Texas Instruments for two incredible summer internships and projects. I would like to acknowledge the support that Texas Instruments has given me in the pursuit of my graduate degree, they have been very generous with tools and software.

I would like to thank all of the fellow Computer Science and Engineering students, who did not formally work with me, yet have become my good friends: Andrew DeOrio, Anthony Gutierrez, Hector Garcia, Jin Hu, Bill Arthur, Debapriya Chatterjee, Shamik Ganguly, Rawan Khalek, Shashank Mahajan, Biruk Mammo, Andreas Moustakas, Ritesh Parikh,

Shobana Sudhakar, Korey Sewell and Dan Zhang. I don't think I could have done this without their friendship.

I would also like to thank Professor Valeria Bertacco for helping me through some tough times and giving advice when I needed it. She helped me with a major event in my life and I can not thank her enough.

Finally, I would like to thank my family. They have all been there to support me when I needed it. I would like to especially thank my mother who has been there since I was young encouraging me to pursue my dreams and my wife has stood by me these past few years and helped pick me up when I fell down.

Table of Contents

Dedication ii

Acknowledgments iii

List of Tables viii

List of Figures ix

Abstract xi

Chapter 1 Introduction 1
1.1 Rise of Mobile Computer Vision 3
1.2 Mobile Computer Vision: A Brief Overview 4
1.2.1 Mobile Computing 4
1.3 Computer Vision 6
1.3.1 Mobile Computing Applied to Vision 8
1.4 Overview of This Thesis 9
1.4.1 Visual Sonification Application 10
1.4.2 Mobile Vision Benchmarking 12
1.4.3 Targeted Singular Vector Computation for Vision Applications . . . 12
1.4.4 EFFEX Mobile Vision Architecture 13
1.4.5 EVA: Efficient Vision Architecture 13
1.4.6 Organization of the Remainder of the Thesis 14

Chapter 2 Michigan Visual Sonification System (MVSS) 15
2.1 Chapter Introduction 15
2.2 Motivation Behind Visual Sonification 15
2.2.1 Contribution of This Chapter 16
2.3 Related Work 17
2.4 A System for Visual Sonification 18
2.4.1 Scene and Depth Estimation 18
2.4.2 Scene Segmentation 19
2.4.3 Bag-of-Words Analysis 20
2.4.4 Audio Signature Generation 21

2.5 Experiments . . . 22
2.5.1 Experiments . . . 22
2.5.2 Results . . . 24
2.6 Chapter Conclusion . . . 27

Chapter 3 Michigan Embedded Vision Benchmarking Suite (MEVBench) . . . 30
3.1 Introduction . . . 30
3.1.1 Contribution of This Chapter . . . 31
3.2 Previous Work . . . 32
3.3 Benchmark Details . . . 33
3.3.1 Feature Extraction . . . 34
3.3.2 Feature Classification . . . 37
3.3.3 Multi-image Processing . . . 38
3.3.4 Recognition Applications . . . 39
3.4 Benchmark Characterization . . . 41
3.4.1 Experimental Setup . . . 41
3.4.2 Dynamic Code Hot Spot Analysis . . . 42
3.4.3 Computational Performance . . . 45
3.4.4 Memory System Performance . . . 47
3.4.5 Multithreaded Performance . . . 50
3.5 Chapter Conclusion . . . 52

Chapter 4 Targeted Singular Vector Computation for Vision Applications . . . 54
4.1 Introduction . . . 54
4.1.1 Contribution of This Chapter . . . 55
4.2 Related Work . . . 55
4.3 Method . . . 57
4.3.1 Preliminaries . . . 57
4.3.2 Computing Right Singular Pairs . . . 57
4.3.3 Computing Left Singular Vectors . . . 59
4.3.4 Application to Homogenous Equations . . . 60
4.3.5 Application to Matrix Factorization . . . 60
4.4 Experiments . . . 62
4.4.1 Homography Estimation . . . 62
4.4.2 Structure From Motion . . . 65
4.5 Discussion . . . 67
4.6 Conclusion . . . 68

Chapter 5 The EFFEX Mobile Vision Feature Extraction Platform . . . 70
5.1 Chapter Introduction . . . 70
5.1.1 Contribution of This Chapter . . . 71
5.2 Related Work . . . 72
5.3 Feature Extraction Algorithms . . . 73
5.3.1 Algorithm Performance Analysis . . . 74
5.4 Proposed Architecture . . . 76

5.4.1 Support for Heterogenous Parallelism . . . 76
5.4.2 Functional Units for Feature Extraction . . . 78
5.4.3 Patch Memory Architecture for Fast Pixel Access . . . 80
5.5 Performance Analysis . . . 82
5.5.1 EFFEX Performance Model . . . 82
5.5.2 Benchmarks . . . 82
5.5.3 Simulation Results . . . 84
5.6 Chapter Conclusion . . . 86

Chapter 6 The EVA Mobile Vision Platform . . . 88
6.1 Introduction . . . 88
6.1.1 Contribution of This Chapter . . . 89
6.2 Background Review On Mobile Vision Applications . . . 89
6.3 Application Traits . . . 91
6.3.1 Vector Reduction Operations . . . 91
6.3.2 Diverse Parallelism . . . 93
6.3.3 One and Two Dimensional Locality . . . 93
6.4 EVA Hardware . . . 94
6.4.1 EVA System Architecture . . . 94
6.4.2 Custom Accelerators . . . 96
6.4.3 Tile Memory Architecture . . . 101
6.4.4 Heterogenous Chip Architecture . . . 102
6.5 Experimental Setup . . . 104
6.5.1 EVA Model . . . 104
6.6 Experiments . . . 106
6.6.1 Benchmarks . . . 106
6.6.2 Single Core Results . . . 106
6.6.3 Multicore Results . . . 109
6.6.4 Comparisons To Other Approaches . . . 111
6.7 Related Work . . . 112
6.8 Chapter Conclusion . . . 114

Chapter 7 Conclusions And Future Work . . . 116
7.1 Thesis Summary . . . 116
7.2 Future Research Directions . . . 118
7.3 Conclusion . . . 120

Bibliography . . . 121

List of Tables

Table

3.1 Benchmarks in MEVBench. 34
3.2 Configurations for profiling MEVBench. 42

4.1 Symmetric transfer error in homography Estimation 64

5.1 **EFFEX Configuration** 83
5.2 Area estimates for the EFFEX 9-core processor 83

6.1 EVA Accelerator Instructions 96
6.2 **EVA Configuration** 105
6.3 Area estimates for the EVA cores 106

List of Figures

Figure

1.1 Performance Scores For Last Three Generations of iPhone 2
1.2 Resolutions of the Cameras for Four Generations of iPhone 2
1.3 Number Of Smart Phone And Tablet Computer Sales Increasing Over Time 3
1.4 Device Width Versus Generation 4
1.5 Battery Capacity and Life Versus Device 5
1.6 Power Drawn By Mobile Systems Versus Desktop 6
1.7 Performance Versus System Application 6
1.8 Inverse Relationship of Computer Vision and Computer Graphics 7
1.9 Typical Computer Vision System 8
1.10 Mobile And Desktop Performance On Vision Applications 10
1.11 Mobile Vision Performance Improvement Cycle 11

2.1 Visual Sonification Concept 17
2.2 Michigan Visual Sonification System Block Diagram 19
2.3 Thresholded Depth-Based Connected Components Segmentation Results . 20
2.4 Subject Training GUIs 23
2.5 Object Classification Experiment Phase 1 User Interface 23
2.6 Object Classification Experiment Phase 2 User Interface 24
2.7 Localization Experiment Phase 1 User Interface 24
2.8 Localization Experiment Phase 2 User Interface 25
2.9 Sample Sonification Results 25
2.10 Sonification of a different pose of the person model 26
2.11 Accuracy Versus Sonification Technique 26
2.12 Accuracy Versus Dictionary Size 27
2.13 Object Spatialization Results 28
2.14 MVSS Execution Time 28
2.15 MVSS Work Distribution 29

3.1 Augmented Reality Example 31
3.2 Typical Computer Vision Software Pipeline 31
3.3 MEVBench Execution Time 46
3.4 MEVBench IPC for Varied Core Capability 46

3.5 Vector Instruction Impact 47
3.6 MEVBench Memory Activity Per Instruction 48
3.7 Patch Memory 49
3.8 Performance (cycles) vs Number Of Threads 50
3.9 Branch Divergence 51
3.10 Average Mutlithreaded IPC 52

4.1 Effect of Number of Inlier Points on Symmetric Transfer Errors 63
4.2 Speedup Provided By SEVS Over NRC and LAPACK for Homography Estimation 65
4.3 Accuracy and Efficiency of SEVS for SFM Factorization 67

5.1 Overview of Feature Extraction 74
5.2 Feature Extraction Execution Time Distribution 75
5.3 EFFEX Architecture 77
5.4 One-to-Many Compare and Binning Unit 79
5.5 Convolution MAC Unit 79
5.6 Gradient Unit 80
5.7 Patch Memory Access Pattern 81
5.8 Patch Memory Address Computation 81
5.9 9-Core EFFEX Speedup 84
5.10 Scalability for SIFT Running on EFFEX 85
5.11 EFFEX Normalized Performance Running SIFT 85
5.12 Performance versus Overall Cost Running SIFT 86

6.1 Vision Software Pipeline 90
6.2 Vision Software Pipeline with Image Capture 90
6.3 Number of Dot Product Operations 92
6.4 Thread Level Parallelism Within Vision Algorithms 94
6.5 An EVA based System Overview 95
6.6 Dot Product Accelerator Overview 97
6.7 Tree Compare Accelerator Overview 98
6.8 Max Compare Accelerator Code 100
6.9 Monopoly Compare Accelerator Overview 101
6.10 Tile Cache Overview 103
6.11 EVA Coordinating and Supporting Core Configurations With Area Constraint 104
6.12 EVA Single Coordinating Core Speedup 107
6.13 EVA Single Coordinating Core Accelerator Usage 108
6.14 EVA Single Coordinating Core Normalized Energy 109
6.15 EVA Single Coordinating Core Normalized Committed Instructions 110
6.16 Multicore Configuration performance of EVA 111
6.17 EVA Coordinator Cores Scaling 112
6.18 Comparison Of EVA with Other Solutions 112

Abstract

Mobile computing devices are increasing in number and capability. The features of these devices are driving the development of applications in new areas. In particular, computer vision benefits from both the rise in compute performance and the increase in imaging capability that mobile devices such as smart phones and tablet computers are experiencing.

Mobile vision is enabling many new applications such as face recognition and augmented reality. However, the performance of mobile processors is limiting the capability of mobile vision computing. With current systems, developers are forced to modify their algorithms to account for the decreased computational performance available on smart devices. Thus current systems must make a tradeoff between algorithm accuracy (and capability) and execution time. They also must take into account the energy constraints imposed on mobile devices due to their use of batteries; thus even high performing algorithms can only run for a limited time.

This dissertation presents an in-depth analysis of mobile computer vision applications and proposes novel hardware and software optimizations with the goal to increase mobile computer vision processing capability. First we present the Michigan Visual Sonification System, a new mobile vision application that provides navigational aide to the visually impaired. The development of this application gives insights into the nature of mobile vision applications including the tradeoffs between performance and energy on mobile processors. We then present MEVBench, a mobile vision benchmark suite that we built to determine the computational characteristics of various mobile vision kernels. This analysis exposes the vector reduction operations, the imbalanced task or thread parallelism and the 2D spatial locality in memory accesses, all which we exploit in the pursuit of highly efficient mobile vision architectures.

Armed with a deeper understanding of computer vision processing, the core of this thesis focuses on software and hardware based optimization to improve the efficiency of mobile vision processing. We begin the optimization with a software optimization known as Single Eigenvector Solver (SEVS). SEVS is an algorithm that reduces the computation of singular pairs, as used in homography and structure from motion, by up to 30% by computing only

what is needed to produce the required singular pairs. We remove many time consuming components of computation utilized in the classical approach. We then move to optimizing the hardware. We begin the hardware optimizations with EFFEX, a heterogeneous multicore architecture that utilizes vector reduction functional units and a 2D memory controller to improve the efficiency of feature extraction. This technique improves feature extraction execution time by up to 12x while reducing energy consumption. Finally, we close with the Efficient Vision Architecture (EVA). EVA expands the EFFEX architecture by adding more custom accelerators for vision operations beyond feature extraction. It also utilizes the tile cache to allow for both 1D and 2D spatial locality in cache accesses. EVA improves the performance of EFFEX by 4x while cutting energy consumption in half.

Overall, this dissertation demonstrates that an application specific approach to processor design can create a flexible programmable design with significant efficiency improvements in mobile vision performance when compared to currently available mobile processors. These works enable the development of richer more capable mobile vision systems.

Chapter 1
Introduction

The table was a large one, but the three were all crowded together at one corner of it

---

Alice's Adventure's In Wonderland
Lewis Carroll

Microprocessors have become a ubiquitous part of the world today. The number of microprocessors in the world continues to increase as many systems integrate microscopic digital circuits to create advanced features such as music players with voice recognition and video game systems capable of rendering near photo-realistic scenes. These devices have moved from multi-room sized systems capable of doing basic mathematical operations to devices that fit within a few cubic millimeters and perform complex analysis [83]. The world has reached a point where there are more microprocessors in the world than there are humans [8].

Mobile phones have been a driving force for the increase in the number of microprocessors. Mobile phones have continued to add applications enabled by improvements in the computational capability of mobile processors [7]. Starting originally as devices merely capable of placing telephone calls, they have evolved to smart devices capable running productivity applications and browsing the internet. Figure 1.1 demonstrates how this performance trend continues in the most recent generations of smart phones.

The improved mobile processor performance has helped usher in a revolution in tablet computing. This revolution was started with the introduction of the Apple iPad and continues with newer tablets such as the Apple iPad 2 and Amazon Kindle Fire [3]. These devices utilize multicore microprocessors to perform tasks such as streaming content and video editing with touch screen interfaces. The newest devices contain more than four mobile processors to deliver never before seen performance in mobile applications [119].

There are many new features helping drive the growth of mobile devices. Many devices are now capable of turn by turn navigation using Global Positioning System (GPS) hardware. The ability to adjust the screen in response to devices orientation is enabled with

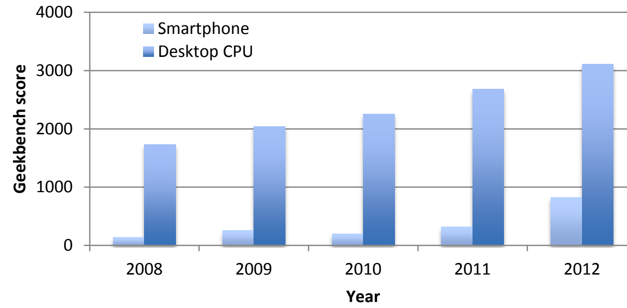


Figure 1.1 Performance Scores For Last Three Generations of iPhone [52]. The performance of each generation of smart phone seems to increase. The figure shows how each new generation of iPhone continues to increase in computational capability as an example of this trend.

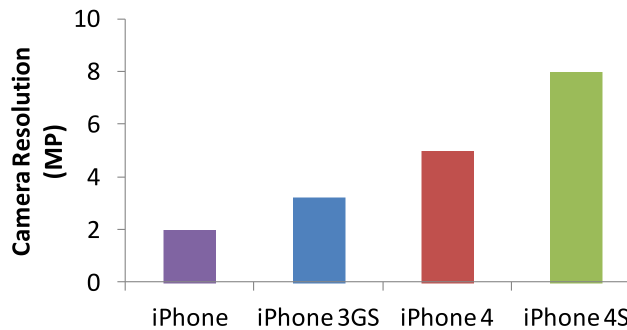


Figure 1.2 Resolutions of the Cameras for Four Generations of iPhone [7]. The capabilities of the cameras on mobile computing platforms continues to increase with each generation. The figure shows how the resolution of the camera has increased with each generation of iPhone.

accelerometers. It is common for smart phones and tablet computer to have cameras for taking photographs or running video chat applications [7] [56]. Figure 1.2 demonstrates how the imaging capability of mobile devices have been increasing with each generation. Newer smart phones are capable of capturing high resolution still images and recording high definition (HD) video.

The increase in features and capability of mobile devices has given rise to an explosion in sales. Figure 1.3 shows that the increase in mobile devices is expected to continue with sales of smart phones exceeding one billion units by the year 2015 [35]. The number of tablet computer sales are expected to exceed three hundred million in the same time frame, adding even more feature rich mobile computing devices with cameras to the growing pool.

The increase in the compute performance along with the increase in device capability, such as image recording, is giving rise to many new applications in mobile computing. Computer vision has seen a large increase in use in the mobile space because of the improvements made in mobile device features. Computer vision benefits from both the increase in compute performance and the increase in imaging capability. It is uniquely poised to be a major factor

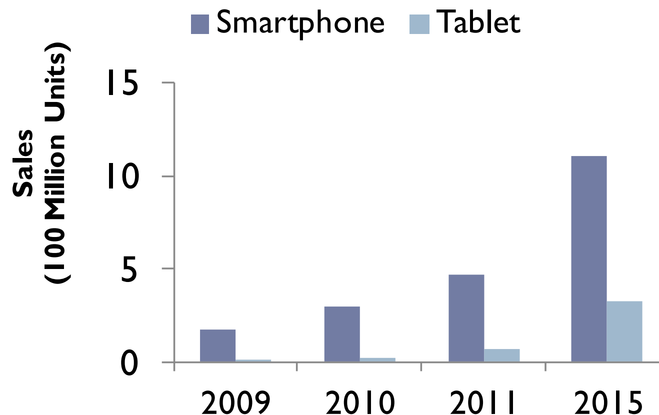


Figure 1.3 Number Of Smart Phone And Tablet Computer Sales Increasing Over Time. The number of smart phone sales continue to increase each year, expected to exceed one billion by 2015. Tablet computer sales are expected to exceed three hundred million by the year 2015.

in the coming mobile revolution. Thus, there is an immediate need to better understand the mobile computer vision workload and develop efficient computer architectures for these critical applications in the mobile space.

1.1 Rise of Mobile Computer Vision

In general, the increase in computing performance has enabled many fields, such as computer vision, to advance over the years. Computer vision brings together multiple fields such as machine learning, image processing, probability, and artificial intelligence to allow computer systems to analyze and act upon scenes. Computer vision has been utilized to enable many different applications including self driving cars [98], robotic surgery [60], facial recognition [141] and visual search [27]. The computer vision community has utilized the increase in computing power to develop advanced algorithms such as Dense Tracking and Mapping by Newcombe et al. that is capable using a single camera to perform real-time analysis and the reconstruction of scenes [106].

The rise in mobile device computing power and imaging capability has driven a renewed interest in mobile computer vision. Mobile computer vision includes applications such as face detection, augmented reality and object tracking. These applications are seeing large growth with mobile computing companies making them a key component of their strategies [30]. This growth in mobile vision has enabled a large number of new applications, but has also exposed a weakness in the performance of current mobile processors when executing computer vision applications.

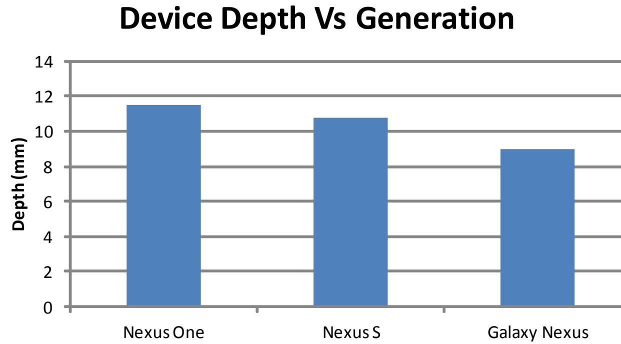


Figure 1.4 Device Width Versus Generation. The plot shows how the width of the Google Nexus smart phone has decreased each generation [56][57][58]. This size decrease limits both the space for components and the ability to add cooling components such as heat sinks into the devices. This adds extra constraints on to mobile processors in terms of physical area and heat dissipation or power.

1.2 Mobile Computer Vision: A Brief Overview

1.2.1 Mobile Computing

Mobile processors have a more constrained design space than desktop machines. Typical cost analysis of microarchitectures look at throughput, latency, area, or power to measure the quality of a design. While these constraints hold true for mobile processors, the area and power metrics take on an increased importance. The importance of area in a mobile processor is driven by consistent reduction in mobile device size. Figure 1.4 illustrates how the size of a device such as smart phone continuously decreases with each generation. The decrease in mobile device also has an impact on the second key characteristic of mobile systems, battery size.

Mobile systems primarily utilize batteries to provide power. However, devices such as smart phones and tablet computers have limited space for battery placement leading to limited limited device battery life. Figure 1.5 demonstrates how the battery capacity of devices decreases as they become smaller. The laptop is the least mobile system on in the figure and has a large storage capacity, capable delivering over one hundred hours at the low current of 200 mA. At nearly 30% of the capacity of the laptop, the batteries in tablet computers are capable of delivering energy for close to twenty four hours. However, smart phone batteries are only capable of delivering 200 mA for less than ten hours. These numbers are assuming only 200 mA where devices such as the pandaboard can have a peak current draw of more than 710 mA, severely reducing the battery life.

In order to accommodate the limited capacity of batteries, mobile systems typically minimize their power usage to ensure adequate system up time. We used a power meter to

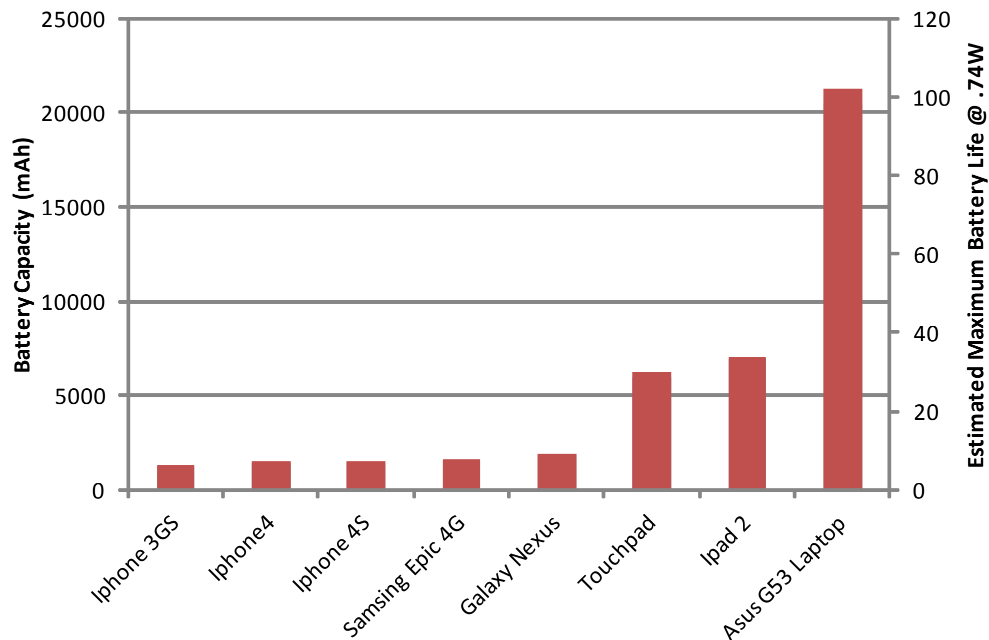


Figure 1.5 Battery Capacity and Life Versus Device. The figure shows the battery capacity of various mobile devices [6][18][61][23][73][56]. The left axis is the capacity of the battery in mAh for the given device. The right axis shows the estimated battery life for that device assuming it draws 200mA continuously. The smart phones have not seen much improvement in battery capacity as can be seen as the transition from the iPhone 3GS to the iPhone 4S. There is a large jump going from smart phones to tablets due to the large size of the devices allowing a larger battery.

measure the average power drawn by various systems 30 seconds after their boot sequence was completed. Figure 1.6 shows the dramatic difference between the power drawn in a mobile system versus that in desktop systems.

Figure 1.7 shows the performance of processors from various application spaces. The mobile processor computation capability, measures in Drhystone MIPS, is over an order of magnitude less than than either the desktop or laptop processors. The decrease in performance is a consequence of minimizing the power in mobile processors. To decrease the power consumption various architectural features, such as highly speculative execution are left out of the designs. Highly speculative execution is valuable for performance but wastes power when the predicted path of execution is not utilized.

Applications in the mobile space have a different emphasis in their design. Size and energy are more critical than in their traditional processor counterparts. As such, energy must be optimized along with area and performance based on each system application. This creates a unique design challenge for any mobile system.

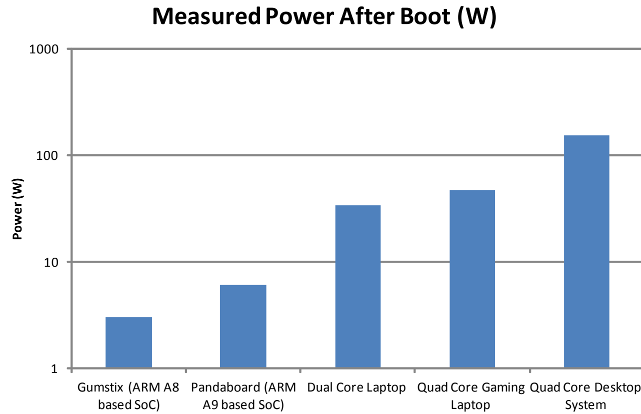


Figure 1.6 Power Drawn By Mobile Systems Versus Desktop. The figure shows the power drawn as systems become more mobile. The system to the far right is a Quad Core desktop with an Intel Core 2 Quad and a NVIDIA discrete GPU. The next two systems are laptops, the first containing a quad core Intel i7 Mobile chip and discrete NVIDIA Mobile GPU and the second laptop contains an AMD Turian X2 with integrated GPU. The last two systems are mobile like devices the first being a system based on an Arm A9 SoC and the being a system based on an Arm A8. The difference in power demonstrates the dramatic difference in power usage for a mobile system when compare to stationary systems or desktop computers.

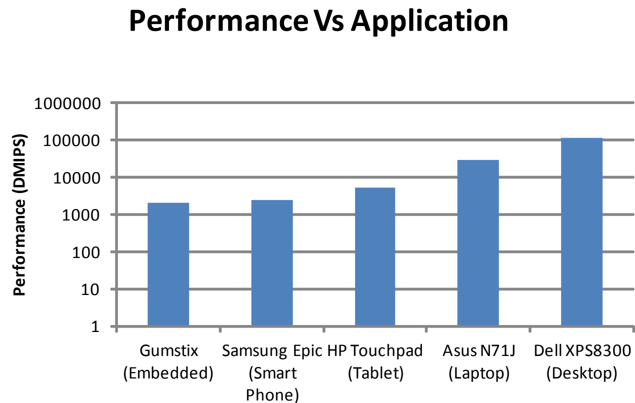


Figure 1.7 Performance Versus System Application. The figure shows computation capability of example systems from various application spaces [43][63][12][123][1][107][135][40]. The figure shows that desktop and laptop processors have far more computational capability than embedded processors. The tablet processors have more power than smart phone processors due partially to the ability to increase the battery size.

1.3 Computer Vision

Computer vision allows computer systems to analyze and act upon scenes. Computer vision algorithms continue to enter the mainstream from the ability of cameras to locate human faces when taking a picture to reconstructing a landmark in 3D from a set of vacation images [131]. News programs and sporting events use computer vision techniques to create

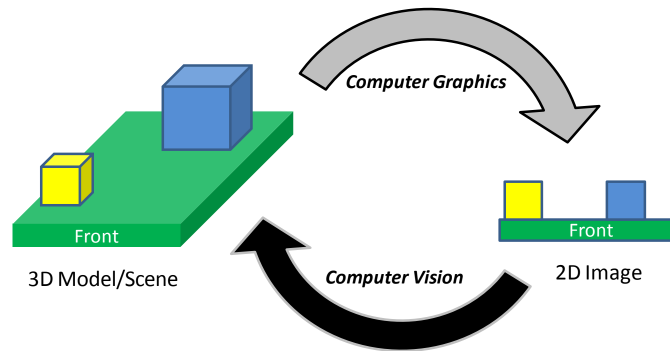


Figure 1.8 Inverse Relationship of Computer Vision and Computer Graphics. Computer vision maintains a relationship to the field of computer graphics. In computer graphics the system has a 3D model as seen on the left and computes how to project that 3D scene onto the 2D plane of a display as seen on the right. In computer vision the problem begins with the 2D information on the right and attempts to reconstruct the 3D model and/or context on the left.

items such as the first down line in an American football game. Computer vision is apart of our lives whether we are aware of it or not.

Many computer vision applications have a graphics component, such as rendering the results of 3D reconstruction. This has led to computer vision being typically linked with computer graphics and the two together are referred to as visual computing [67]. However, a better summation of the relationship between them is computer vision as the inverse of computer graphics. In computer graphics, an application has a 3D model of the world and displays this information by projecting onto the 2D plane of the screen. In computer vision the system is given information about the projection in the form of image data and attempts to extract information about the 3D world that created the projections. Figure 1.8 graphically depicts the inverse relationship these two field share.

The typical computer vision system involves three components, as shown in Figure 1.9. The first is the scene under study. The second is an sensing device that can be used to analyze the scene. The third is a computational device that can perform the analysis of the scene based on the data from the sensor. The computation devices generates two possible forms of data, information such as visual cues, and interpretations of information such as actions being performed or the presence of objects. The two forms of data can each be used to refine the other until the output of the vision system is computed with a predefined amount of certainty. The result can be the 3D location for every pixel within and image or the certainty that a person is performing jumping jacks. With proper selection of the information and the interpretation algorithm, computer vision systems can be applied in a large number of applications.

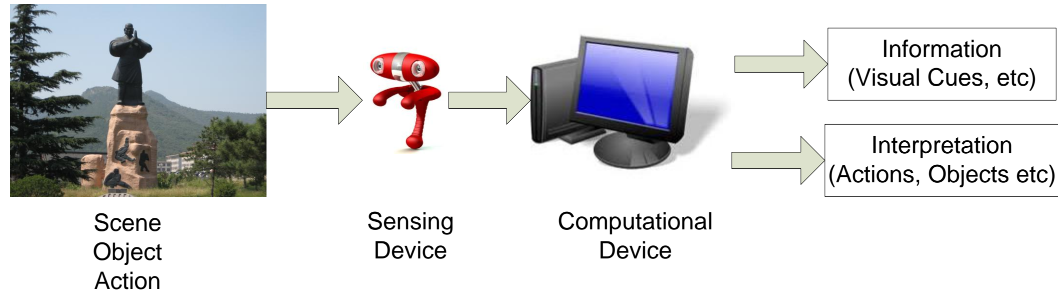


Figure 1.9 Typical Computer Vision System. The figure shows the basic components of a computer vision system. The left most component is the scene or object of study. The next required component shown in the figure is the sensing device used to collect data from the scene. The third component is the computation device. The device computes information such as visual cues and reasons on this information to generate interpretations of the scene such as objects present or actions being performed.

1.3.1 Mobile Computing Applied to Vision

Mobile devices create a unique opportunity for computer vision. The mobile devices allow computer vision to be applied in new environments where traditional computing systems can not be utilized. For example, navigation applications that allow the user to point their smart phone's camera at a landmark and have interest point information displayed [55]. Some smart phones use computer vision to provide security features such as unlocking the phone only when an clear image of a registered user is visible. Computer vision has also been used to help create navigational aides for the visually impaired [145] [31]. Smart phones can snap a picture of QR codes, which encode anything from a URL to interesting facts about a location into 2D bar codes. Mobile computing is poised to become a dominating platform for vision.

To improve the quality of their results, many vision systems utilize a iterative computation process. Thus in computer vision there is a trend to increase computation to increase application quality. For example, FAST and BRIEF can be used to quickly locate and describe salient points in images known as feature points [25]. However, the feature points are not very robust; if the image is rotated or the illumination in the scene changes the feature descriptions change completely. The algorithm known as the Scale Invariant Feature Transform (SIFT) can be utilized to compute more robust feature points however, the computation time is over an order of magnitude longer [33]. Many mobile systems today utilize FAST and BRIEF due to constraints on the computation capabilities of modern mobile processors despite the availability of the more robust SIFT.

Many new mobile devices are beginning to support features that utilize computer vision such as high dynamic range (HDR) imaging capability [129]. To produce HDR images

multiple images are captured and processed using computer vision algorithms to produce a more vibrant single image [97]. This can lead to high quality images; however, due to the processing time on mobile systems, the time between pictures can be in seconds [144]. Mobile vision continues to be utilized in the automotive industry as automobiles support more features, such as lane departure detection, driver state and pedestrian detection. These algorithms are being implemented on the same mobile processors that are utilized in smart phones.

Even as there are many applications for mobile systems today, there is a growing demand for improved processing performance to account for future mobile vision systems. As unmanned aerial vehicles are becoming more autonomous they will require higher performance mobile vision processing to complete achieve this autonomy [41]. This requires increase vision processing with minimal energy to allow for maximal mission time. Autonomous vehicle are another area of growth for mobile vision processing. Computer vision is key for providing self driving cars the ability to deal with everyday traffic [90]. New products such as Google Goggles will increase the use of mobile vision [55]. This system which places a mobile processor, display and camera on the users glasses could enable high quality gesture recognition control of electronic devices or the real time analysis of scenes. However, the current processing performance limitations of mobile platforms may hinder the development of these new vision algorithms.

Computer vision algorithms can involve large amounts of image processing and machine learning which can be compute intensive even on desktop systems. When these same algorithms are applied to mobile systems, there is considerable performance loss in terms of execution time. Figure 1.10 shows how much performance is lost when running applications from MEVBench [33] and going from the Intel Core 2 Quad desktop processor to the ARM A9 mobile processor. In order to reach real-time performance on a mobile system, the mobile processor requires a large increase in performance while still minimizing the energy required to do so.

1.4 Overview of This Thesis

This thesis performs an in-depth analysis of mobile computer vision applications by leveraging the process presented in Figure 1.11 and proposes novel architectural improvements with the goal to increase mobile computer vision processing capability by orders of magnitude. We develop a new application of mobile vision that can be used as a navigational aide to the visually impaired. We characterize and analyze this new application along with other mobile

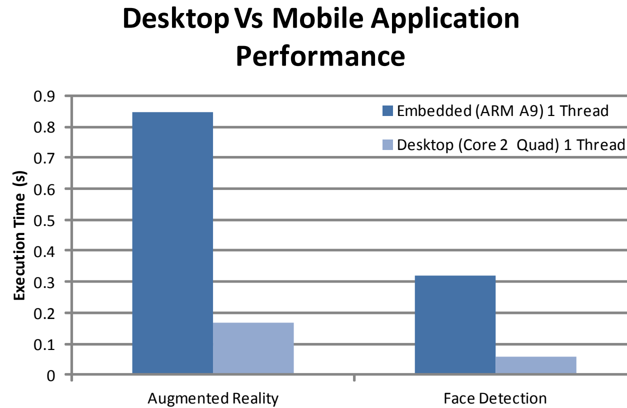


Figure 1.10 Mobile And Desktop Performance On Vision Applications. The figure shows the stark difference in current mobile and desktop processor performance on vision applications. Two vision applications from MEVBench [33], face detection and augmented reality execution times on an Arm A9 and an Intel Core 2 Quad. The mobile ARM A9 is much slower at executing this vision applications when compared to the Core 2 Quad.

computer vision codes. The results of this analysis drives our work to improve mobile vision systems by decreasing the energy usage while also decreasing the computation time. To achieve this we utilize mobile vision application specific designs to optimize compute performance while minimizing energy and area costs. The solutions are further constrained by the rate at which new computer vision algorithms are being developed; the solutions must maintain a level of programmability that facilitates deployment of new mobile algorithms while also achieving high performance on established mobile vision algorithms. There are many cases where the mobile vision algorithms are not running on dedicated vision systems, but instead run on systems such as smart phones which run tasks such as web browsers or music players as well. Thus, any proposed solutions must be amenable to multitasking systems. With this in mind, our work proposes architectural enhancements for the mobile computer vision space.

1.4.1 Visual Sonification Application

Visual Sonification is the process of converting scenes into 3D audio data to provide navigational aid to the visually impaired. We developed the Michigan Visual Sonification System (MVSS) for this purpose [31]. The system utilizes many aspects of computer vision to generate audio signatures based on the visual features of an object. The audio signatures are representative of an object's appearance, and are augmented with 3D synthetic spatialization to provide the users with an indication of the object's orientation and distance. We

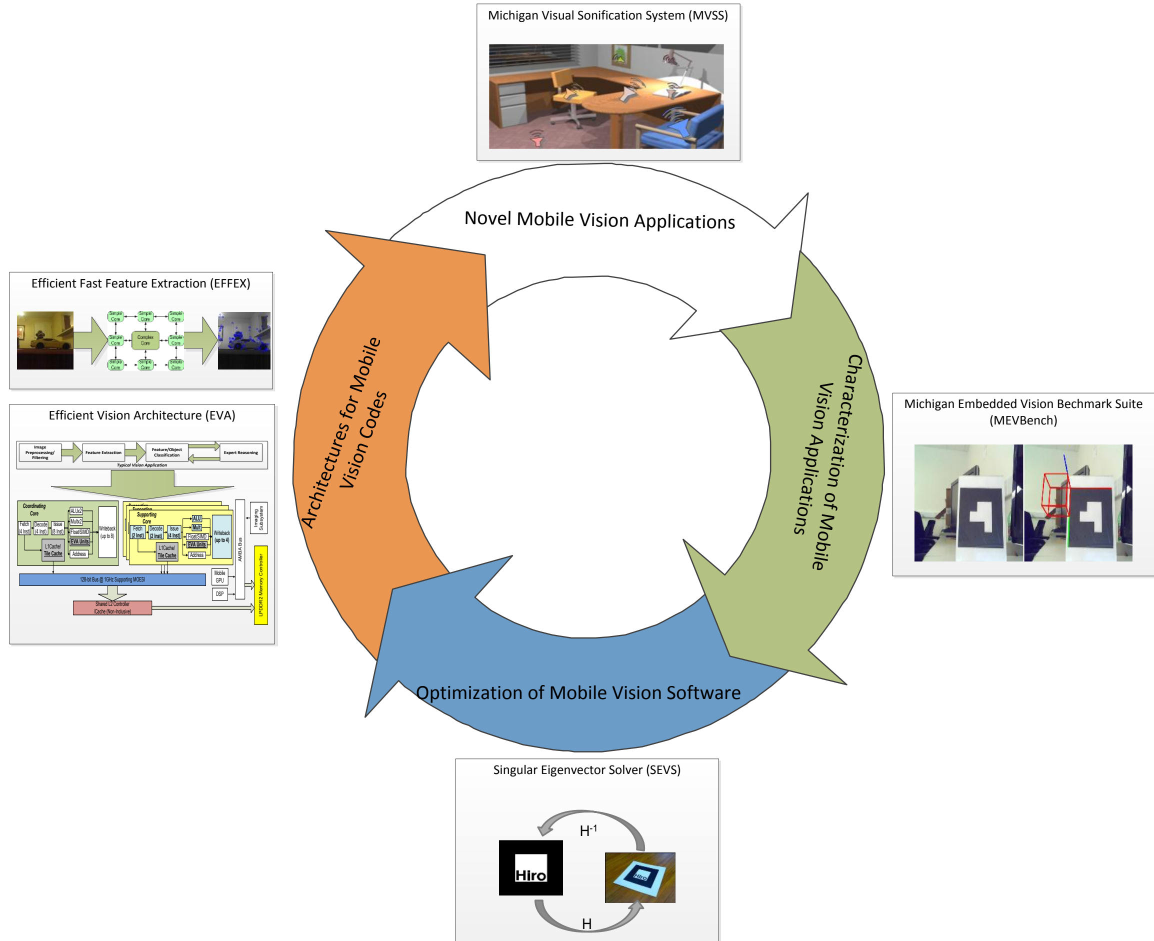


Figure 1.11 Mobile Vision Performance Improvement Cycle. The figure shows the cycle used for improving the performance of mobile vision systems and the components of this dissertation that correspond to each phase. The first phase is the development of novel mobile vision applications, such as *MVSS*. This is followed by characterization of the codes used by mobile vision applications (using *MEVBench*) . The analysis is proceeded by optimizations of the software used in by the mobile vision applications as demonstrated in *SEVS*. Once software has been fully optimized, hardware optimizations are applied to further increase performance as demonstrated by *EFFEX* and *EVA*. The increased performance allows for new applications.

demonstrate that the human brain is capable of recognizing object categories and locations based on these signatures. We build MVSS to gain insight into the nature of mobile vision processing. This application helped develop an understanding of computer vision and its application in the mobile vision space. The development of this application exposed how mobile vision systems must balance application capabilities with the constraints of mobile processing performance. We show that current mobile processors do not provide enough capability for the system to achieve real-time execution for this system. Our experiments indicate that a modern dual core mobile processor requires at least 20x improvement in processing performance to accommodate real time performance.

1.4.2 Mobile Vision Benchmarking

While mobile vision application use is on the rise, there are few benchmark suites available that take into account the developments and constraints within this space. The mobile vision space has a wide range of useful applications such as MVSS, augmented reality, face detection, and object detection that must be considered when analyzing system performance. For example, MVSS is composed of several computer vision modules and each module is subject to unique processing characteristics that must be taken into account when optimizing for mobile systems. Also, mobile systems are experiencing an increase in number of cores available [119], thus an effective benchmark suite must take this into account. To this end, we developed MEVBench, a mobile vision benchmark suite that contains multithreaded versions of many mobile computer vision applications [33]. It also contains multiple benchmarks for the core components of mobile vision which allow for characterizations that are applicable to most applications in mobile vision. We evaluate the performance of MEVBench on various platforms, both mobile and desktop processors. Our analysis shows that mobile vision applications contain exploitable parallelism at both the thread and data parallelism levels. We show that parallel mobile vision architectures need to support heterogenous computation to efficiently accommodate parallel and serial execution components present in mobile vision. We expose that exploitation of 2D spatial locality can improve performance. We introduce a new measure of control complexity in branch divergence, and demonstrate that it can help guide architectural decisions for mobile vision specific hardware.

1.4.3 Targeted Singular Vector Computation for Vision Applications

Software is a common target for optimizing applications. In the case of mobile computer vision, it is essential to optimize algorithms and code due to the constraints placed on mobile systems. Eigen decomposition, the computation of the eigenvalues and eigenvectors of a matrix, is commonly used in mobile vision applications, such as augmented reality, structure from motion, and camera calibration. In this thesis we present the Singular Eigenvector Solver (SEVS) technique for computing these eigen pairs efficiently. This technique looks utilizes the fact that in many cases we only want a small subset of eigen pairs. Utilizing this knowledge along with eigen decomposition methods we can restrict our computation to only what is required to compute the subset. We apply our technique to various components of augmented reality such as homography estimation and demonstrate how it allows for a faster computation of eigen pairs with user controlled accuracy. We demonstrate that for a given

precision, our technique is more accurate while reducing the computation when compared to the classical algorithm.

1.4.4 EFFEX Mobile Vision Architecture

We utilize hardware to improve the mobile vision performance further. Our work in characterizing mobile vision applications shows that the performance of feature extraction is a key bottleneck for mobile processors. Thus, we develop the Efficient Fast Feature Extraction (EFFEX) Architecture [32]. We observe that there is imbalanced task level parallelism available in feature extraction. There is also data parallelism in the form of vector reduction operations. This class of operations takes in one or more vector inputs and produces a scalar result. This operation takes place in dot product, 2D gradient and one-to-many compare operations. Furthermore, the image data is utilized as input to feature extraction which creates a memory access pattern that has 2D spatial locality. As a result of these insights the EFFEX architecture is a heterogenous multicore with custom vector reduction functional units and a 2D patch memory controller. The patch memory controller allows for efficient access to image data while the custom functional units provide efficient computation of vector reduction operations such as the inner product. The multicore component allows the architecture to exploit thread-level parallelism while the heterogenous configuration allows for the complex core to handle serial components while coordinating the bulk of the work on the simple cores. This early work demonstrates a 90x improvement in feature extraction performance.

1.4.5 EVA: Efficient Vision Architecture

In order to move beyond feature extraction we develop the Efficient Vision Architecture (EVA). EVA expands on the EFFEX architecture to include features for computer vision processing phases beyond feature extraction. Computer vision applications maintain their large degree of parallelism beyond the early feature extraction phase. In particular, the later phases maintain the task level parallelism present in feature extraction but that coordination continues to cause a workload imbalance. We take advantage of this in EVA by using a high performance core that performs computation and coordination and energy efficient cores to take advantage of the task level parallelism. There is also a large degree of data parallelism present in mobile vision codes. This data parallelism is primarily in the form of vector reduction operations. The dot product, and one to many compare (monopoly compare) are common throughout most computer vision codes. We also found that decision trees and

computing the maximum of a vector are common in the later stages of vision application pipelines. To optimize the performance of EVA we include custom vector reduction accelerators for computing dot products, monopoly compares, decision trees, and vector maxima. We observed that many vision algorithms have memory access patterns that include both 1D and 2D spatial locality in various points during computation. Thus, EVA enhances the memory system with the tile cache to handle, allowing the cache to switch between 2D memory accesses and 1D memory accesses. We show that EVA can provide speedups of nearly 9x that of an advanced embedded processor while reducing energy demands by as much as 3x on mobile vision workloads.

1.4.6 Organization of the Remainder of the Thesis

The next six chapters describe the work in Figure 1.11 in detail. In Chapter 2 we describe the visual sonification application, MVSS. Chapter 3 analyzes the performance of mobile vision application with MEVBench. In Chapter 4 we demonstrate the use of our eigen decomposition technique, SEVS to allow the developer to tradeoff accuracy and execution time effectively. Chapter 5 presents EFFEX, our a hardware architecture designed to increase the performance of feature extraction for mobile vision systems. Within Chapter 6 we describe EVA, an architecture that expands on EFFEX to provide acceleration to all the phases of mobile vision processing through custom accelerators. Finally in Chapter 7, we present a summary of the contributions of this thesis, draw conclusions and present avenues for future research.

Chapter 2

Michigan Visual Sonification System (MVSS)

> There is no better way to thank God for your sight than by giving a helping hand to someone in the dark.
>
> ---
>
> *Light in my Darkness*
> Helen Keller

2.1 Chapter Introduction

Chapter 1 discussed that increased computational capability in the mobile space has led to the creation of new applications that utilize mobile computer vision. In this chapter we introduce a novel application of mobile vision that serves as a key application driving our work to improve the performance of mobile computer vision in terms of energy usage and execution time. Developing this visual sonification application served as a primer for mobile computer vision allowing in-depth understanding of common mobile computer vision algorithms and their applications. The process of developing a new application gave us first hand knowledge of this unique development process which includes balancing the desired features with the computational capability of the system. This application provides valuable insight into how the various components of mobile vision are combined in real applications.

2.2 Motivation Behind Visual Sonification

The World Health Organization (WHO) reports there are over thirty million blind people in the world [142]. While biomedical research is making progress towards solutions such

as prosthetic retinas, these technologies are likely to be expensive and require invasive procedures. In this chapter, we present the Michigan Visual Sonification System (MVSS) that offers a low-cost, non-surgical solution that can serve as a visual aid for the blind.

Driven by the increase in embedded computing power and the ubiquitousness of camera sensors on devices such as smart phones, there is a wide array of computer vision applications being used in low-cost, low-power settings. MVSS aims to leverage this trend in embedded vision processing to create a computer vision based solution that provides pertinent scene information to a user using *visual sonification*, the process of converting visual scene properties to audio signals.

MVSS analyzes scenes with a stereo camera pair and provides spatially registered audio feedback to a blind user to aid in the classification of different object categories. An illustration of the concept is shown in Figure 2.1. Objects are first segmented using depth information obtained from analyzing the stereo images. Scale and rotation invariant appearance features are extracted from the segmented object using SIFT [94]. In an off-line learning phase, SIFT features computed from a database of images are used to create a dictionary of visual words by employing the Bag-of-Words modeling technique [37]. For each feature set extracted from the segmented objects, the closest matching word in the dictionary is found, and the object is represented as a histogram of dictionary words. This histogram is then converted to a sound signal. During playback, the volume and spatial location of the sound source is adjusted based on the estimated location of the segmented object in the scene. The object is classified by the user using the their association of the audio signature with the sound.

A salient aspect of MVSS is that it relies on the visually impaired users acute sense of hearing to process the audio signal and recognize the source object. To achieve this, a user can be equipped with an "reference set" of sound signatures for common object categories. Since the human brain affords us an adaptive classifier that can add/subtract/merge/split classes and sub-classes on the fly, this reference set will naturally evolve and adapt to an individual user over time. We expect this approach to be more scalable and robust than attempting to build a classifier to recognize signatures of a fixed set of object classes. In MVSS, a user's ability to distinguish between sounds of different object categories is primarily limited only by the richness of the dictionary of visual words.

2.2.1 Contribution of This Chapter

MVSS is an inter-disciplinary approach to creating a visual aid for the blind. It leverages recent work in computer vision and machine learning as well as audio processing and sound

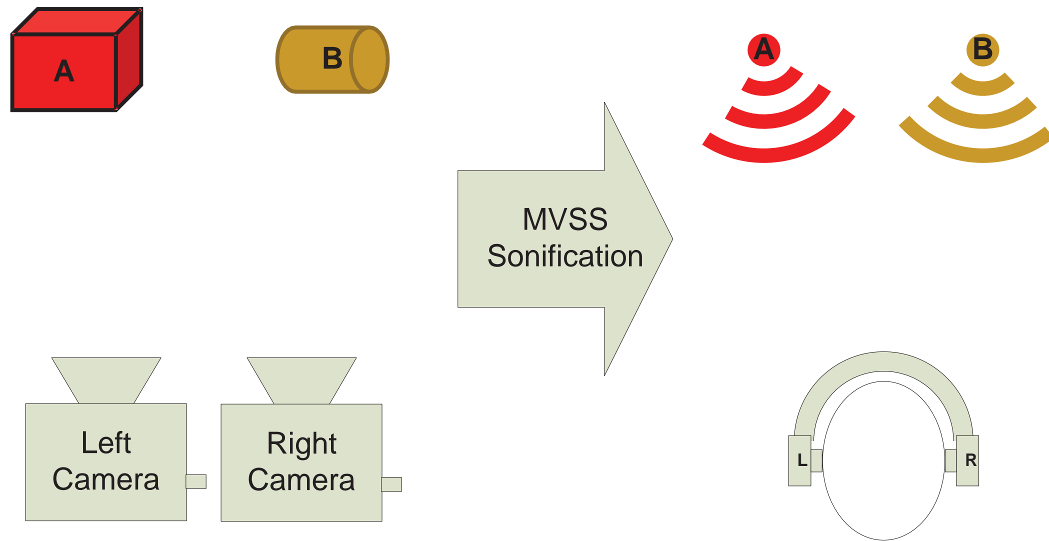


Figure 2.1 **Visual Sonification Concept** Illustration of visual sonification. The two objects, A and B, are analyzed and represented as distinct audio signals, and played back to the user preserving their relative position with respect to the camera pair.

localization. In this work, which is to be published in [31], we describe early results from a prototype of MVSS that highlight the following primary contributions of our approach:

- An efficient method for object segmentation that extends traditional connected components analysis to 3D point clouds. We show the results of this technique in this chapter.
- Exploitation of an alternate modality, namely hearing, to enhance a blind person's ability for object classification. To our knowledge, this is a novel use of a Bag-of-Words object descriptor.
- Capturing object location from scene and modifying audio signature using 3D sound processing.

2.3 Related Work

There are many groups working on implants such as retinal prostheses or sensor chips to replace degenerated tissue within the ocular system [127]. These devices are typically limited to use in patients with specific conditions such as having a large fraction of the neural pathways to the visual cortex still functioning or only the eye's photo receptors damaged. Such devices require invasive surgery to be implanted. Many are still years away from

being approved for the general public in places such as the United States. Our system uses the human ear as the interface to the brain and thus does not require surgery and is widely applicable.

The SWAN system [145] uses beacons and object specific tones to navigate blind users. The system requires prior knowledge of the environment and objects in order to produce the tones. Furthermore the system is designed to recognize the object and generates a tone for the given object based on its classifier. Such a system is limited to the object classes used to train the classifier, and is likely error-prone since object classification in unconstrained environments remains an open problem. MVSS circumvents these issues by relying on the human brain to categorize and recognize audio signals generated from segmented objects.

vOICe [101] converts grayscale image data to sound by modulating amplitude with brightness. The pitch is controlled by the pixel vertical location and time is used to represent the pixel horizontal location. Each column of the image is output at the same time. This technique does not attempt to extract relevant information from the scene, and is hence limited to low resolution images due to the conversion to sound. MVSS also relies on the user's brain to classify input signals, but lessens the demands on the user by summarizing the relevant information in the scene and provides object location information through 3D audio generation.

2.4 A System for Visual Sonification

The Michigan Visual Sonification System brings together various components of computer vision with audio processing to generate scene signatures based on the regions of interest. The system performs scene segmentation, feature extraction, 3D reconstruction, Bag Of Word's analysis, audio signature generation, and spatial audio output to allow the user to distinguish objects. An overview of the processing done by MVSS can be seen in Figure 2.2.

2.4.1 Scene and Depth Estimation

MVSS utilizes a calibrated stereo camera pair to analyze the 3D structure of the scene. For each input image pair, pre-computed remap tables are used to remove any distortion and to rectify the the images. Missing pixels are calculated using linear interpolation. In order to compute stereo disparity, we use the sum of absolute differences (SAD) correlation measure to match the pixels between the left and right images. The pixel disparity is used to compute an estimate of the 3D location for each pixel location using the re-projection matrix. This

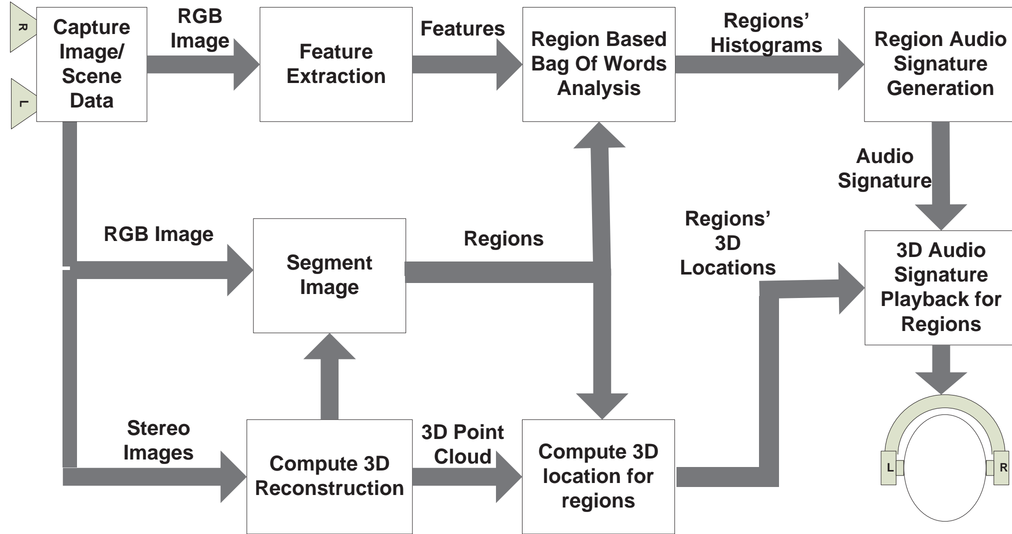


Figure 2.2 Michigan Visual Sonification System Block Diagram This figure illustrates the processing components of the MVSS.

3D point cloud is used by other computation blocks to perform the sonification of the scene.

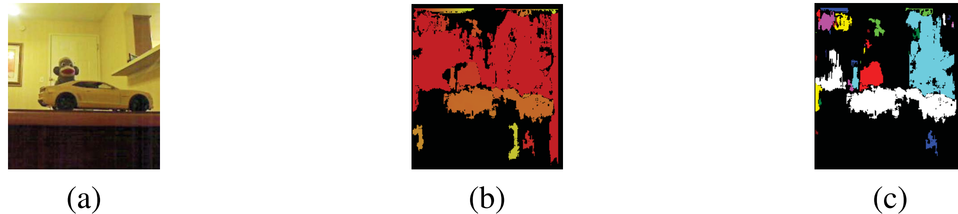
2.4.2 Scene Segmentation

The MVSS applies a region of interest (ROI) to the scene under investigation. The ROI is defined by a minimum depth and maximum depth that is of interest to the user. Using the ROI as a mask on the scene depth data, any pixel location that has an estimated depth value outside of the ROI specification are masked out. Other metrics apart from the depth can also be incorporated into the ROI.

The scene data within the ROI is now segmented into regions that correspond as best as possible to individual objects in the scene. Our object segmentation algorithm is based on the assumption that large differences in depth correspond to occlusion edges in the scene. This implies that the change in depth (distance from camera) along an object's surface is locally smooth. We apply a connected component algorithm with a depth difference threshold to perform the depth based segmentation in a single pass.

The depth difference connected component algorithm begins by setting all pixels to an "unlabeled" state. The algorithm scans across each pixel of the image in raster scan order starting at the top left most pixel and tests if the pixel is unlabeled. If the pixel is unlabeled then a new component is formed, and a corresponding queue is initialized. The depth of all the neighbors of the newly added component pixel are compared to the depth of the new component pixel. If the change in depth is within a threshold and unlabeled, the neighbors

are added to a work queue. Once the work queue is empty the algorithms searches for a new unlabeled pixel to begin a new component segmentation. Once the component raster scan reaches the lower right corner of the image, all viable components have been found. Figure 2.3 shows the results of segmenting a scene using this technique. For each generated region the centroid is computed as it is used for 3D audio generation.



(a) (b) (c)

Figure 2.3 Thresholded Depth-Based Connected Components Segmentation Results (a) Input image with a sock monkey sitting behind a model car. (b) Depth map (yellow is close, red is far). (c) Segmentation result, with each segment denoted by a color. The red region is the monkey and the large white region is the car.

2.4.3 Bag-of-Words Analysis

In Bag-of-Words (BoW) modeling, traditionally employed in text retrieval, a document is represented as a histogram of words. BoW has been successfully applied in computer vision to tasks such as object classification or topic discovery. [48]. MVSS utilizes the BoW representation to describe the segmented image regions.

When used in vision processing, the first step of BoW modeling is to define a dictionary of visual words. A visual "word" is typically defined to be a local invariant image descriptor. MVSS utilizes SIFT processing [94] to extract scale and rotation invariant image descriptors which are 128 element vectors. In an off-line learning phase, thousands of SIFT features are extracted from a database of images, which are are then vector-quantized to generate a small set of words that forms the dictionary. Thus each SIFT feature is a considered a BoW word.

During processing of an input image, we extract SIFT features for each segment computed using the procedure described in Sect. 2.4.2. The features are then compared against the dictionary to produce a feature occurrence histogram for each segmented region. The histogram is created by comparing each SIFT feature in a region against the features in the dictionary in a brute force method, and their Euclidean distance d is computed. The top N results, based on smallest distance, of the matches are added to the Bag Of Words occurrence histogram for that region using $1-d$ weight. The histogram, whose number of bins is equal to the number of features in the dictionary, is then used to describe the content of the image or region. This is analogous to reading a document and counting how often a

word occurs to determine the topic. The dictionary features must be chosen to ensure the content can be easily determined from the shape of the histogram. Feature selection can impact the results of the BoW model. The feature occurrence histograms are the primary input into the audio signature generation.

2.4.4 Audio Signature Generation

MVSS generates a 3D spatially located sound or signature for each feature occurrence histogram. The 3D location of the object relative to the camera is used to place the sound in space and the BoW occurrence histogram is used to generate the content of audio signature. A signature is only created if a region has a minimum pixel area and a minimum number of features/words. Audio signatures are generated using modulation based on the histogram bin. We employ three methods for performing this modulation.

The first modulation technique is pure sine frequency modulation. For each region, each bin of the occurrence histogram has a different single frequency generated. The frequency for a given bin is computed using linear interpolation by setting frequencies for the minimum and maximum allowed bin values. The second modulation technique, named harmonic modulation, expands the first modulation technique by adding the first three harmonics to a given bins audio. The final technique we developed utilizes the same technique to generate the base frequency as the first two techniques, however the sound is synthesized to emulate a piano key being struck. This technique is named piano modulation.

The current system outputs a single audio signature at a time with a small pause between outputs. The total length of a region audio signature output is configurable, and divided equally among the generated histogram bin frequencies. For example, for a ten bin audio signature with a one second output, each bin frequency would be output for one hundred milliseconds.

Each signature is passed to the audio system for output with a 3D location, that indicates where to simulate the signature's location in space. We utilize the default capabilities of OpenAL [36] to handle proper spatialization of the audio signature. The spatialization can be enhanced utilizing techniques from the work of McMullen [100]. In order to achieve the personal 3D effect, the system requires the user to wear headphones.

2.5 Experiments

2.5.1 Experiments

The input to the system was a stereo pair of QVGA (320 x 240 pixels) images. We use OpenCV 2.2 as the framework for our application relying on built-in functions for disparity and image rectification. For feature extraction we use the Sift++ implementation from [138]. We generated the audio using the OpenAL library 1.1 [36]. The API allows the user to specify the location, direction, and velocity of a sound source. All sound sources were placed with their direction facing the user and a velocity of zero.

The application was run on a 2.0 GHz AMD Turion x64 running 64-bit Windows 7 with 4 GB of RAM, and compiled using Microsoft Visual Studio 2010 for performance on a PC. Furthermore, the PC system used a stereo pair of calibrated Logitech C905 cameras as inputs to the system. The mobile system was a TI Blaze [77] which utilized a 1.0 GHz Texas Instruments OMAP 4430 running Android version 2.3 with 1 GB of RAM compiled with Android NDK revision 6. The mobile system used a pair of stereo cameras built into the TI Blaze device.

The dictionary for the Bag-of-Words model was created using two data sets, the INRIA Person data set [39] and a 3D object categories data set [124]. The features for the dictionary were generated using k-means clustering. For the computation performance results shown here, the cluster count for k-means was set to 10. For all experiments the audio signature output time was one second.

We performed two different experiments with two sets of subjects. The first set of subjects consisted of 15 individuals that were unfamiliar with MVSS and were trained during the experiment through feedback from answer the first 5 to 10 questions of each experiment. We called this the unfamiliar group. The second group was a pair of subjects who were given access to a training graphical user interface that showed images of objects and played the given audio signature. The training interface can be seen in Figure 2.4.

The first experiment was designed to measure the capability of users to recognize the categories of objects based on their audio signature. The experiment utilized five different object categories (care, bicycle, cellphone, monitor, and shoe). It also varied the sonification technique and the size of the dictionary used for generating region signatures. This experiment involved two phase. The first phase required the user to listen to an example sound and then choose which of two other sounds was from the same object category. An example of the user interface for this phase can be seen in Figure 2.5. In the second phase the user was asked to listen to an example sound and then a second query sound. The user was asked

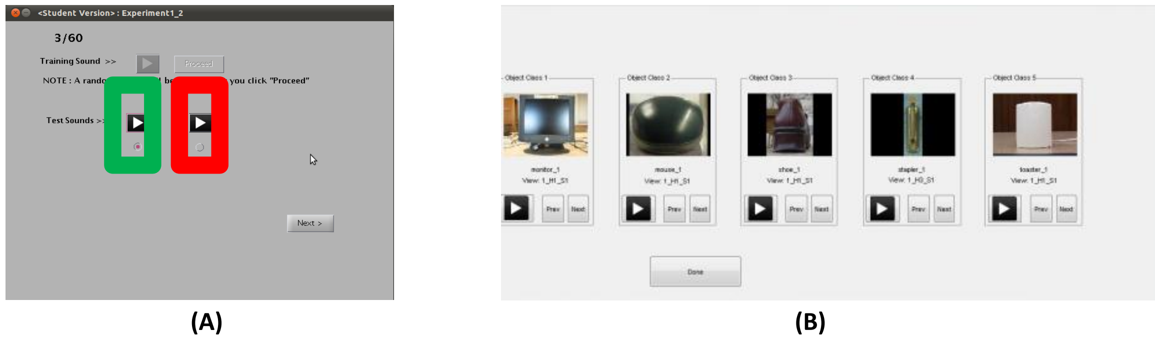


Figure 2.4 Subject Training GUI The figure shows the user interface used for training the subjects. (A) shows the screen used for the unfamiliar group during the experiment. Their training is done through feedback during the first few questions. (B) shows the training GUI the familiar group utilized. This gui allowed the familiar group to listen to example sounds for objects before beginning the experiment.

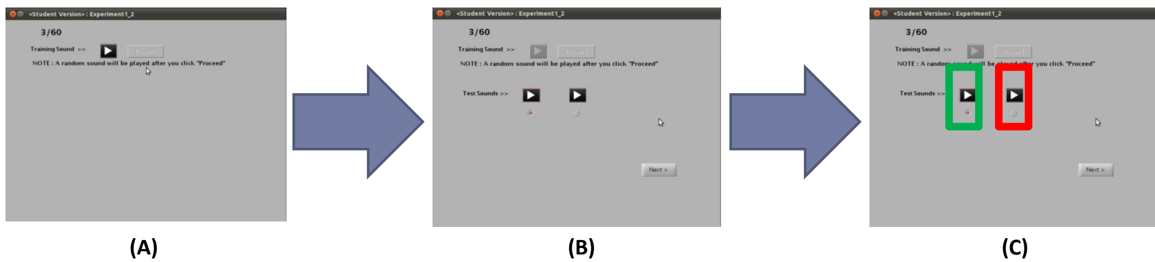


Figure 2.5 Object Classification Experiment Phase 1 User Interface The figure shows the user interface used for first phase of the object classification experiment. (A) shows the screen with the initial sound. The sound is played once the user clicks the play icon. Once the user clicks the proceed button, they are presented with a second screen with two sound icons and asked to select which sound is from the same category as the initial sound. During the training phase, the user is given feedback using screen (C) which informs them whether they have selected the sound generated by the object from the same category as the initial audio signature.

to rate the similarity of the two sounds. The user interface used for this phase can be seen in Figure 2.6.

The second experiment was designed to measure how capable users were at determining the sound source for the generated audio signatures. In this experiment we utilized a 3D coordinate system centered at the user. The positive z direction is defined as the direction the user is facing, the positive x direction is to the right of the user and the positive y direction is defined as up. In the first phase of this experiment the audio signature of a random object is played. Using 3D audio rendering techniques, the source is moved to one of three x locations, left, right, or origin of the coordinate system. The user is asked to identify the location of the source relative to the user's location. The user interface can be seen in Figure 2.7. The second phase increases the number of locations to nine by allowing the z value to take on 3 possible depths for the locations, near, midfield and far. The user is again

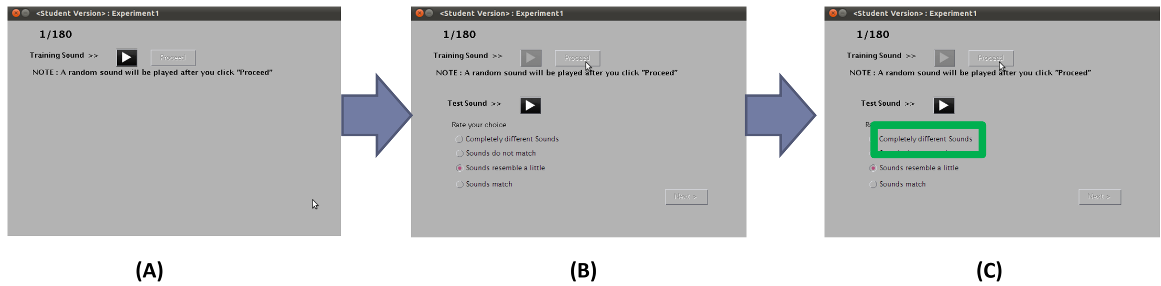


Figure 2.6 Object Classification Experiment Phase 2 User Interface The figure shows the user interface used for second phase of the object classification experiment. (A) shows the screen with the initial sound. The sound is played once the user clicks the play icon. Once the user clicks the proceed button, they are presented with a second screen with a second sound and a scale for rating the similarity of the first sound to the second. During the training phase, the user is given feedback using screen (C) which informs them whether the sounds are from the same object category.

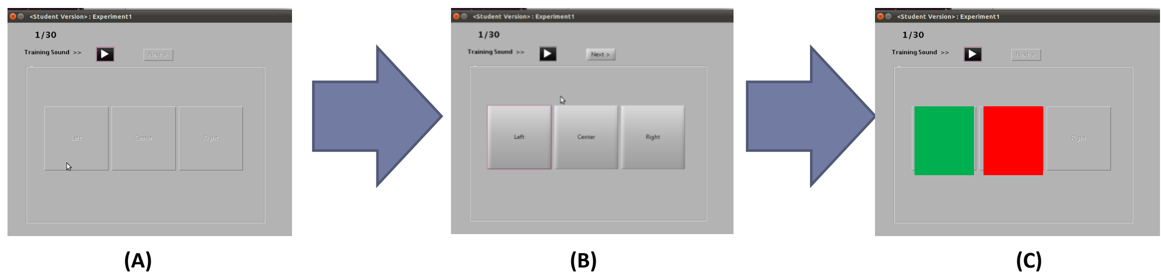


Figure 2.7 Localization Experiment Phase 1 User Interface The figure shows the user interface used for first phase of the localization experiment. (A) shows the screen with the initial sound. The sound is played once the user clicks the play icon. Once the user clicks the proceed button, they are presented with a second screen with three possible locations for the sound source and asked to identify the correct option. During the training phase, the user is given feedback using screen (C) which informs them whether they have selected the correct origin.

asked to identify the location of the source using a user interface such as that in Figure 2.8.

2.5.2 Results

Sample results from sonifying objects with MVSS can be seen in Figure 2.9, which shows three rows of images corresponding to a car (top row), a person (middle row) and a sock monkey (bottom row). The columns of Figure 2.9, correspond to the input image (left), object segments extracted from the image (middle), and the spectrograms of the sonification output for the highlighted object segments (right). Note that the spectrograms of the different objects have unique signatures, which result in perceptibly distinct sounds. These results also demonstrate how the proposed approach can help distinguish previously unseen object categories. While the dictionary is trained only on people and car images, MVSS utilizes available dictionary words to describe novel objects, such as the slightly human-shaped sock

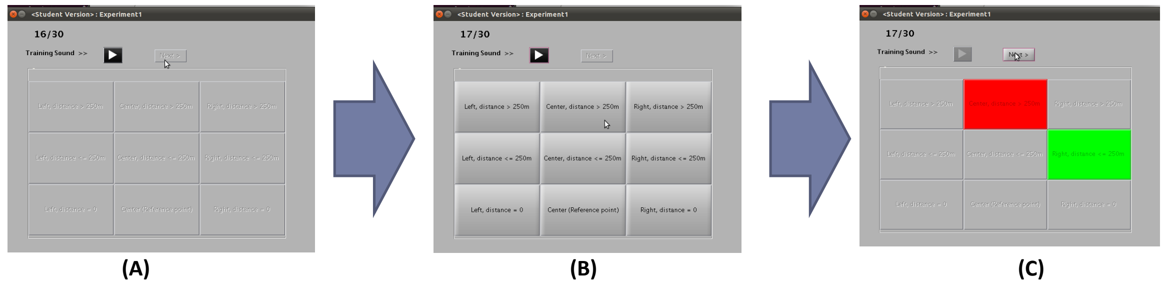


Figure 2.8 Localization Experiment Phase 2 User Interface The figure shows the user interface used for second phase of the localization experiment. (A) shows the screen with the initial sound. The sound is played once the user clicks the play icon. Once the user clicks the proceed button, they are presented with a second screen with nine possible locations for the sound source and asked to identify the correct option. During the training phase, the user is given feedback using screen (C) which informs them whether they have selected the correct origin.

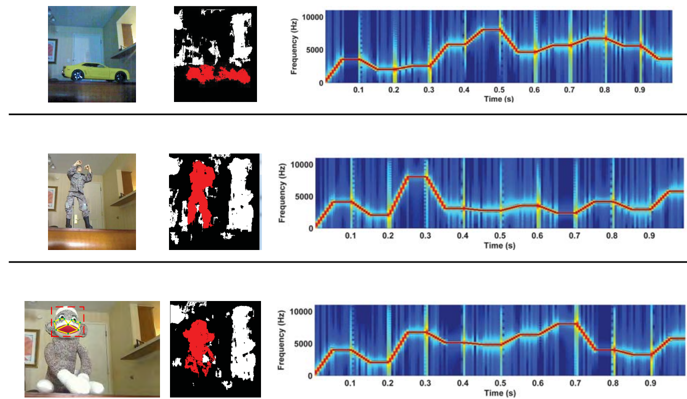


Figure 2.9 Sample Sonification Results. Sample sonification results on three different objects, namely car model (top row), person model (middle row), and sock monkey (bottom row). From left to right, the columns show input image (left), extracted image segments with sonified object highlighted in red (middle), and spectrogram of audio signal of the highlighted object (right).

monkey in this example.

The Bag-of-Words representation also enables MVSS to correctly handle different object poses and viewpoints. In Figure 2.10 we show the result for a different pose of the person model. Comparing the spectrogram (right column) with those shown in Fig. 2.9, we see that it is most similar to that of the person model (middle row). Our experiments show that visible similarity in spectrograms translates to perceptible similarities in the sound domain.

The accuracy results of varying the modulation technique and the number of bins can be seen in Figure 2.11. The accuracy was improved in the familiar group due to the use of the training graphical user interface. It can also be noted that for for the first phase of the experiment, the sine modulation appears to allow subject to answer with the highest

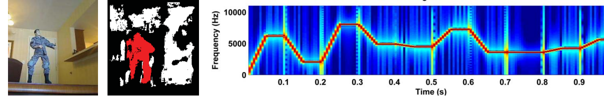


Figure 2.10 Sonification of a different pose of the person model. Compared with middle row of Fig. 2.9, note differences in object pose and similarity in corresponding audio signatures.

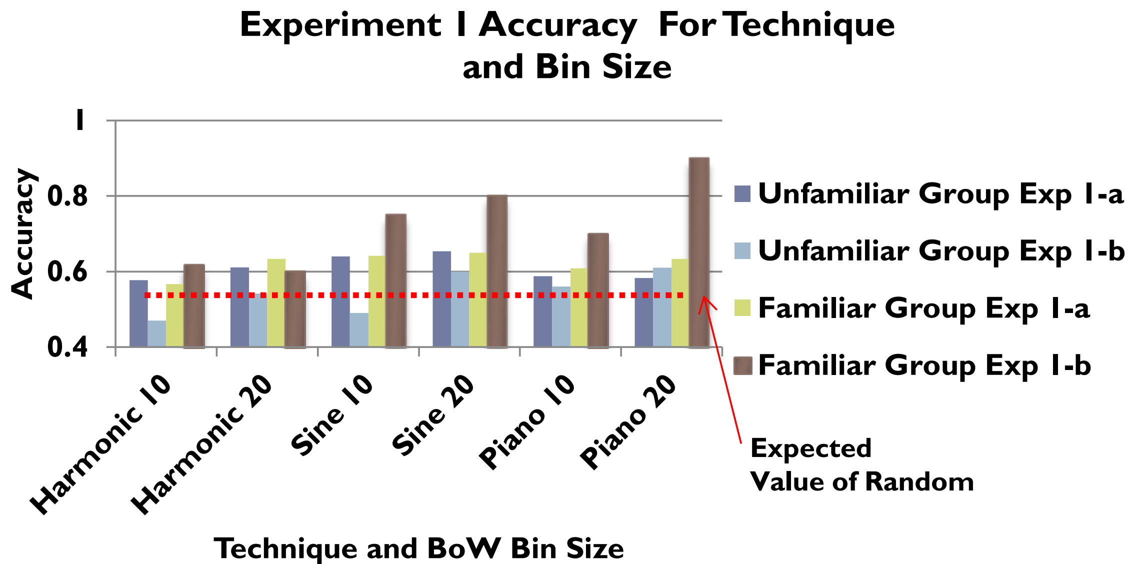


Figure 2.11 Accuracy Versus Sonification Technique The figure shows accuracy of subjects identifying object categories as the sonification technique is varied. It also demonstrates how more training can greatly improve the accuracy of subjects.

accuracy while for the second phase the piano modulation allows highest accuracy.

Figure 2.12 shows increasing the number of bins can lead to a decrease in accuracy. This is primarily due to the user being overwhelmed with the amount of information and being unable to discern differences.

Figure 2.13 shows the results from the localization experiment. Users are capable of determining the location of nearby sound source but as depth is added the accuracy drops of quickly. This is due to far field effect. The user could determine the item was far away but could not ascertain the exact location. MVSS is primarily designed to function within a 3m range of the user thus the fair field effect is not an issue.

A key characteristic of the MVSS is the performance on mobile devices. Figure 2.14 demonstrates how slow the performance of MVSS is on a traditional mobile processor. The system is over an order of magnitude away from achieving real-time performance. Figure 2.15 illustrates how much of the total execution time for MVSS is taken up by feature extraction on a typical mobile processor. Clearly, feature extraction is the most critical phase of computation for improving MVSS execution time.

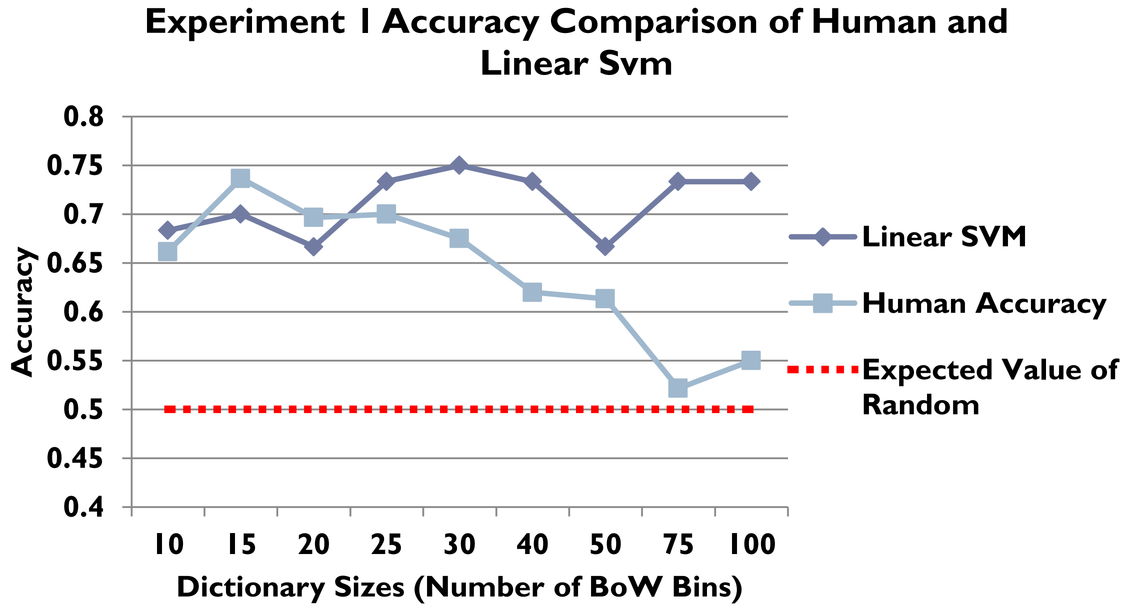


Figure 2.12 Accuracy Versus Dictionary Size The figure shows accuracy of subjects identifying object categories as the dictionary size is varied. These are the results for the familiar group averaged across all three techniques. We also show the classification performance of a linear SVM on the same data set. This shows that as the number of bins increases, the accuracy seems to increase early and then begin to decrease.

2.6 Chapter Conclusion

While methods to improve the quality of life of the visually impaired are being developed, most techniques that enhance their visual awareness involve costly and invasive surgeries and are years away. The Michigan Visual Sonification System presented in this chapter provides a low cost, noninvasive alternative that, given advances in embedded computing power, could be a practical solution in the near future. The MVSS uses “visual sonification”, whereby the the 3D scene around the user is analyzed with computer vision algorithms and visually different image regions are represented by distinct audio signals.

MVSS accomplishes the visual sonification process through a unique process. It utilizes an efficient technique for object segmentation that combines connected components with 3D depth information. The system uses a Bag-of-Words histogram representation of a segmented object to generate sound signatures that are directly interpreted by the visually impaired user. This approach leverages the power of the human brain to perform object recognition and classification. In addition to object appearance, the 3D object position is encoded into the sound signature to provide the user with spatially oriented sound sources.

Our experiments show that this technique is capable of allowing a user to determine both the location and class of an object with greater than 70% accuracy. We expose that the

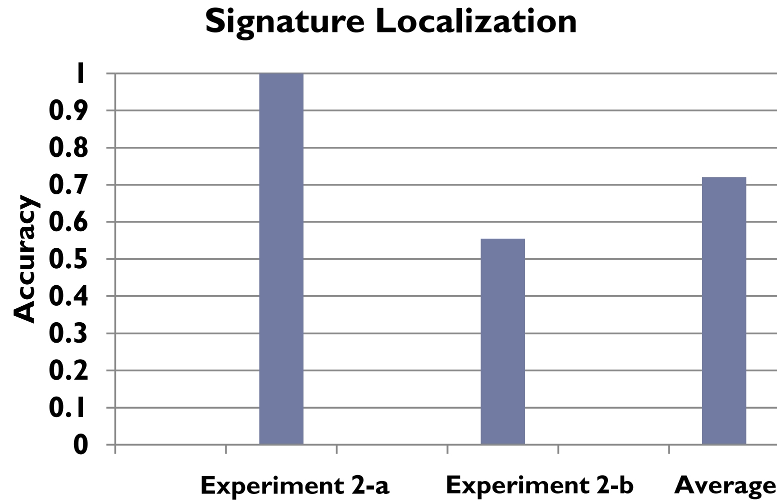


Figure 2.13 Object Spatialization Results The graph shows accuracy of subjects identifying virtual location of the audio signatures sources. The first phase, where the depth or z value is help constant, the users are perfectly capable of determining the location the source. Once distance is added in for the second phase, we see a decrease in accuracy. This is due primarily to the far field effect, which limits the users ability determine the exact location of sound sources.

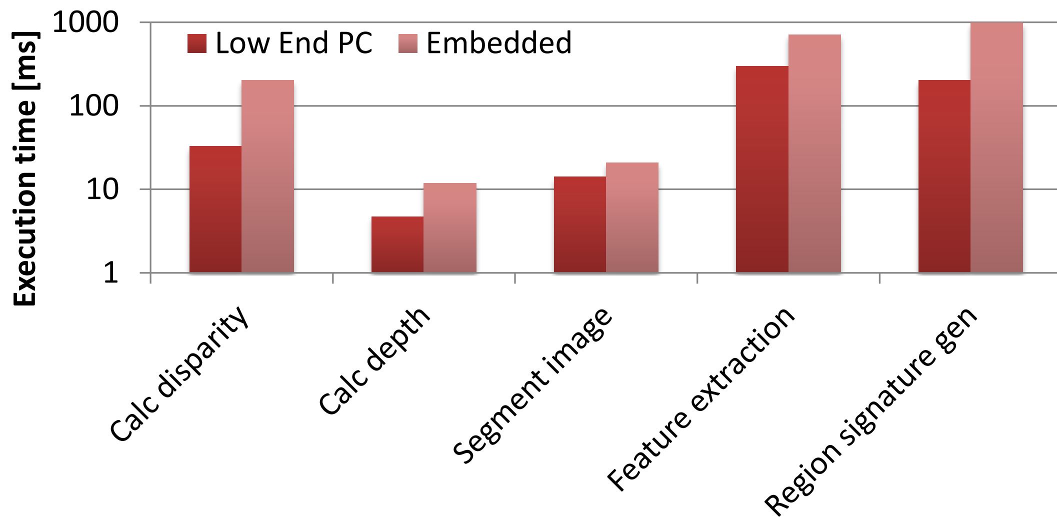


Figure 2.14 MVSS Execution Time The figure shows the breakdown of execution time of MVSS on a 2GHz Turian X2 processors with 4GB of RAM and a 1GHz Arm A9 with 1GB of RAM. The low embedded system is well under the performance required for realtime operation.

accuracy of the users is influenced by a large amount of factors, including dictionary size and audio signature generation method. We show that peak accuracy occurs at a dictionary size of 15 code words for the current sonification techniques. Our analysis shows that different methods to convert the region signatures to audio signatures impact system efficacy.

Beyond user performance, MVSS gave insights into mobile system design. The limited

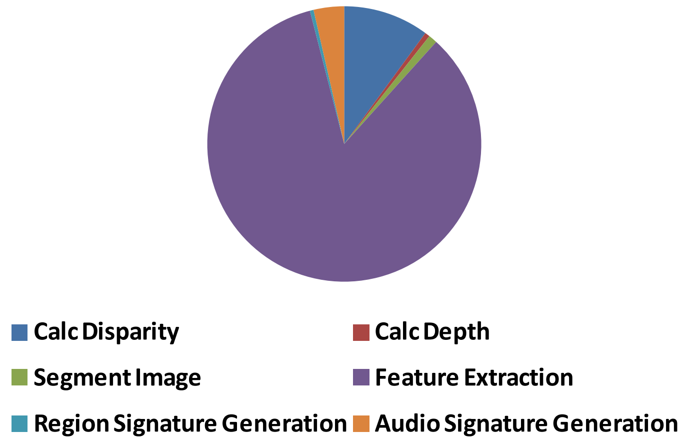


Figure 2.15 MVSS Work Distribution The figure shows percentage of execution time spent in each phase of the MVSS execution for an 1GHz Arm A9 with 1GB of RAM. The feature extraction consumes the largest amount of time during the execution.

performance of mobile systems forced tradeoffs between computation time and system capability. For example, the system utilizes low resolution QVGA images as input. Higher resolution images would allow for better segmentation and a larger number of features, however the increased image size greatly increases processing time. The segmentation algorithm in MVSS uses depth difference as the primary method for determining regions, however this has a weakness when the objects meet a supporting plane. This could be rectified with the use of support plane estimation, however the processing capability modern mobile processors hinders the inclusion of this.

In general, the performance of the MVSS algorithm on an embedded platform requires over an order of magnitude speedup to achieve real-time performance. Thus there is a need to optimize the software and the hardware to achieve this. This optimization requires characterizing the different types of computation taking place in the application. We utilize a benchmark suite in Chapter 3 to better understand the computation in the various mobile vision modules. We then turn to optimizing the algorithms and software to improve performance in Chapter 4. Our timing analysis shows that the critical path is dominated by the feature extraction component of the computation. Improving this phase of execution can greatly increase the system performance. We examine hardware techniques to improve the feature extraction performance in Chapter 5. However, feature extraction is only a single phase component of the system, so we develop hardware for general mobile vision systems in Chapter 6.

Chapter 3

Michigan Embedded Vision Benchmarking Suite (MEVBench)

> For me, it is far better to grasp the Universe as it really is than to persist in delusion, however satisfying and reassuring.
>
> *The Demon-Haunted World: Science as a Candle in the Dark*
> Carl Sagan

3.1 Introduction

Chapter 2 demonstrated an application that utilizes mobile computer vision. Developing this application helped in understanding how mobile vision systems are developed and the balancing that takes place between features and performance. In this chapter we focus on understanding the computation taking place in mobile vision codes. We utilized our experience from developing MVSS to identify the common mobile vision algorithms and develop a benchmark suite. MVSS is composed of many different mobile vision components or modules, such as feature extraction, feature classification, and 3D reconstruction thus we started with these in our analysis. We also include other applications to ensure applicability of our results to a wide array of mobile vision systems such as augmented reality systems, Figure 3.1. The analysis provided in this chapter creates a foundation for locating optimization opportunities in mobile vision codes.

Despite the rise in mobile vision applications, there are limited benchmarks and analyses of the various computation kernels that mobile computer vision applications use. An overview of a typical computer vision pipeline can be seen in Figure 3.2. The pipeline begins with processing of the scene to locate features or distinctive image characteristics such as corners, edges, or high contrast areas. These features usually have a signature computed,

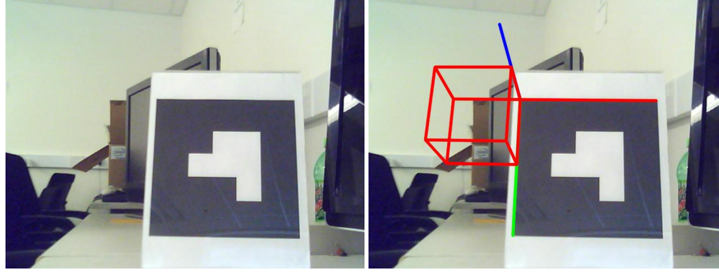


Figure 3.1 Augmented Reality Example The figure shows an example of augmented reality available on mobile platforms. The left image shows the original scene. In the right image a red cube frame has been rendered in proper perspective as though attached to the marker. Current mobile computing devices are capable of rendering detailed objects into the scene.

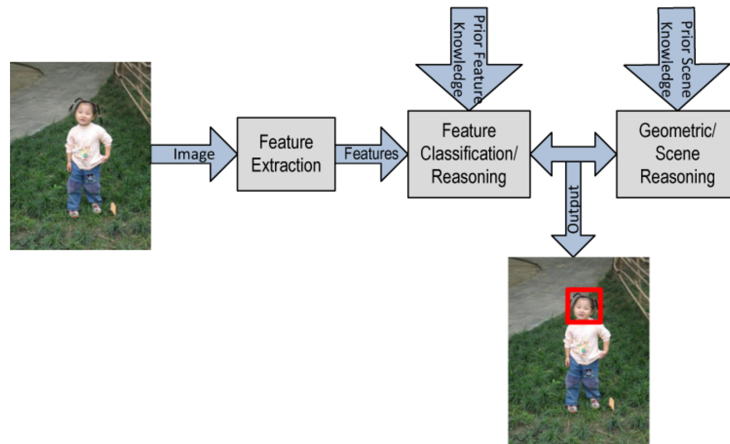


Figure 3.2 Typical Computer Vision Software Pipeline. The figure shows a typical vision pipeline. Features are extracted from an image and reasoned about based on prior knowledge. Feature reasoning is used in concert with scene reasoning to produce contextual knowledge about the scene. In this example the face is found in the image.

called the feature descriptor, that is used to reference the feature and provide comparisons. The features are then used to drive scene reasoning. In many cases they are matched to a known-object database to determine object semantic information and scene context. This information is then refined to determine contextual knowledge about the scene such as the presence of an object or a person.

3.1.1 Contribution of This Chapter

In this work, which was published in [33], we analyze many key vision algorithms that are particularly apt for the mobile computing space. We examine their architectural characteristics, including memory demands, execution time, and microarchitectural performance. Our key contributions in this work are:

- We assemble a mobile computer vision benchmark suite, composed of key applications and kernels for the mobile space. We draw on existing vision benchmarks and computer vision libraries to provide an initial collection that broadly samples the application domain.
- We perform a detailed microarchitectural analysis of the MEVBench benchmarks, indicating their hotspots, performance, memory characteristics and scalability.
- To better assess the efficiency of computer vision benchmarks on potential hardware platforms, we develop two new performance metrics that measure control complexity and 2D spatial locality.
- Finally, we present insights into how future embedded architectures can better serve the mobile computer vision application domain.

3.2 Previous Work

The computer vision community continues to work toward solving many open problems but focuses primarily on optimizing algorithm accuracy. For example, vision efforts include the Middlebury data set for disparity/depth generation [70], Pascal Visual Object Classes Challenge for classification of various objects [47], and the Daimler Occluded Pedestrian benchmark [46], which all focus on reocognition accuracy and present the core of their results using precision versus recall curves. While accuracy is important for the general problem set, the mobile space introduces additional important constraints on computation capabilities and power usage. While there have been some investigation into the mobile vision space in recent years [88], it is just now becoming a focus.

Previous vision benchmark efforts have either looked at computer vision components as a part of a larger benchmark or provided benchmarks with basic timing analysis. In this work we focus specifically on mobile computer vision applications and provide detailed information at the architecture level of their computational demands.

VisBench contains a face recognition benchmark for the development and analysis of visual computing [67]. It groups graphics and computer vision together. However computer vision and graphics are distinctly different problems. In graphics, there is a full 3D model that must be presented to users, typically in 2D. Computer vision, on the other hand, takes a set of information such as a 2D images and attempts to reconstruct the information that was lost in the transformation from 3D to 2D. While some components are similar, studying

computer vision algorithms (and in particular mobile vision algorithms) will best serve our broader purpose of designing more efficient mobile vision computing platforms.

In PARSEC, bodytrack is the only computer vision benchmark [17]. While including this benchmark into PARSEC aids in the development and optimization of general purpose processors, there are now many more important vision applications, such as augmented reality, that have become common in the realm of computer vision. Thus we provide a benchmark analysis focused only on computer vision.

There are embedded benchmarks, however they do not target the growing field of embedded computer vision. Mibench provides an embedded benchmark suite but does not contain any direct computer vision benchmarks [64]. The Embedded Microprocessor Benchmark Consortium provides benchmark suites for embedded computing, such as Coremark [44] and MultiBench [45], however none are targeted at the embedded computer vision space.

OpenCV is an open source computer vision library [20]. It provides many low level vision and machine learning algorithms for use in computer vision application development. It is widely used and has been optimized for various platforms such as ARM and x86. It is capable of providing a vision framework to develop a wide range applications, but to date it has not been thoroughly benchmarked. We used the OpenCV framework to develop our custom benchmarks.

SD-VBS is a benchmark suite that provides single threaded versions of many important computer vision applications [139]. They provide basic implementations of the algorithms for use in benchmarking. We incorporate a number of the SD-VBS benchmarks into our collection of applications, but we also broaden the effort to include a number of full-scale computer vision applications that are apt to the mobile space such as augmented reality and SURF feature extraction. In addition, we include parallelized versions of key vision kernels, as exploitation of explicit parallelism will likely be a critical factor in the design of successful mobile vision computing platforms. To our knowledge, MEVbench is the first mobile computer vision benchmark suite.

3.3 Benchmark Details

MEVBench is targeted at mobile embedded systems such as the ARM A9 and Intel Atom processors that are common in smartphones and tablets. These devices are gaining in popularity [42] and acquiring more capable cameras and mobile processors [109] [119]. Mobile embedded systems differ from typical desktop systems in that they are more concerned with size, energy and power constraints. This typically leads to lower computational power

Table 3.1 Benchmarks in MEVBench.

Benchmark	Input Type	Multithreaded
Feature Extraction		
SIFT	Image	Yes
SIFT (SD-VBS)	Image	No
SURF	Image	Yes
HoG	Image	Yes
FAST and BRIEF	Image	Yes
Feature Classification		
SVM	Feature Vectors	Yes
SVM (SD-VBS)	Feature Vectors	No
Adaboost	Feature Vectors	Yes
K Nearest Neighbor	Feature Vectors	Yes
Multi-image Processing		
Tracking (SD-VBS)	Image Sequence	No
Disparity (SD-VBS)	Image Pairs	No
Image Stitch (SD-VBS)	Image Set	No
Recognition Applications		
Object Recognition	Image	Yes
Face Detection	Image	No
Augmented Reality	Image	No

along with less memory resources. MEVBench provides full applications, such as augmented reality, along with components of common vision algorithms such as SIFT feature extraction and SVM classification. Table 3.1 summarizes the MEVBench benchmarks. The algorithms are built using the OpenCV framework unless otherwise noted. For the OpenCV benchmarks, we used the framework and some of the functions OpenCV provides, but we assemple the applications together using custom code. Furthermore, we developed a custom framework for multithreading vision benchmarks based on Pthreads. The included SD-VBS benchmarks are a subset of the SD-VBS benchmark suite that were chosen because they are suited for the mobile vision space [139].

3.3.1 Feature Extraction

Features are key characteristics of a scene or data. Typical image features include corners, edges, intensity gradients, shapes and blobs. Feature extraction is the process of locating features within a scene and generating signatures to represent each feature. This is a key component of most computer vision applications, and the quality of a feature is based on its invariance to changes in the scene. A high quality feature is invariant to viewpoint,

orientation, lighting and scale. Feature extraction quality is, in general, proportional to the algorithm's computational demand [32]. Our benchmark provides a variety of feature extraction algorithms to accommodate this characteristic vision workload. We provide a wide variety of feature extraction algorithms from the high quality and computationally intensive Scale Invariant Feature Transforms (SIFT), to the low quality but efficient FAST corner detector.

Scale Invariant Feature Transform (SIFT)

SIFT is a common feature extraction algorithm that is used to localize features and generate robust feature descriptors. SIFT descriptors are invariant to scale, lighting, viewpoint and orientation of the given feature. It is commonly used in applications that involve specific instance recognition such as object recognition, tracking and localization, and panoramic image stitching.

SIFT is a robust feature detection and extraction algorithm. SIFT first creates an image pyramid using iterative gaussian blurring [94]. A difference of gaussian (DoG) pyramid is then formed by taking the difference between the pixel intensities of two adjacent images in the initial image pyramid. The DoG pyramid is searched for extrema pixel locations that are greater than all their neighbors in both the current image and the images at adjacent scales. If this is true, the point is a potential feature point. The localization of a possible feature is further refined using 3D curve fitting. The refined potential feature points are then filtered based on their resemblance to edges and their contrast. Once a point is located, the descriptor is formed using the gradients of the image in the region around the feature point. The 128-entry feature descriptor is then normalized to aid in illumination invariance. The SIFT algorithm, while computationally expensive, provides a high level of invariance to changes in illumination, scale or rotation.

MEVBench has two different implementations of SIFT. The first is the single-threaded version from the SD-VBS benchmark suite. This version is a self-contained implementation of the feature point localization phase of SIFT which is commonly referenced as DoG localization. Furthermore, this version is optimized for code understandability. The second version is a multithreaded version of SIFT built using the OpenCV framework . This version of SIFT is based on the implementation provided by Vedaldi [138]. The MEVbanch version can be scaled by number of threads and input size.

Speeded Up Robust Features (SURF)

SURF is a commonly used feature extraction alternative to SIFT. Its native form produces a smaller feature descriptor than SIFT and takes less computation time [15]. However a comparison of the performance of the two algorithms gives mix results based on the application being used [81]; for example, SIFT performs better at rotations while SURF is slightly more viewpoint invariant. Overall, SURF is more commonly used in embedded systems because of the relatively low computational complexity. Similar to SIFT, SURF is used in applications that involve specific instance recognition such as object recognition, tracking and localization, and panoramic image stitching.

SURF uses integral images to approximate image convolutions. This allows for fast computation of regional information once the integral images have been computed. Furthermore it uses a second order derivative computation (Hessian Matrix) and box filters to localize the feature point locations. The algorithm uses multiple sizes of the filters to find feature points at different scales. The locations are then filtered using a non-maxima suppression where only the strongest signal in an area is used. The feature descriptor is computed using gradient-like computations, specifically Haar wavelets, for an oriented region. The SURF descriptor is based on the SIFT descriptor [15]. MEVBench contains a multithreaded version of SURF. This version can be scaled using thread counts and input sizes.

Histogram of Oriented Gradients (HoG)

HoG is commonly used for human or object feature detection [39]. HoG uses image gradients to describe features within an image. To locate features, the algorithm uses a sliding window technique where feature descriptors are computed at all possible locations within the image and compared against a database of possible feature descriptors. If the descriptors match, a possible object of interest has been found at the given location. HoG feature descriptors are computed using an array of cells. A histogram of the gradient directions for each cell is computed, and then the cells are grouped into blocks and normalized. The resulting histograms are concatenated together to form the descriptor. The HoG algorithm is slower than some lower quality feature extractors, but it provides good illumination invariance and a small level of rotational invariance. MEVBench contains a multithreaded version of HoG. This version can be scaled by thread count and input size.

FAST Corner Detector (FAST) and Binary Robust Independent Elementary Features Descriptor (BRIEF)

FAST was originally developed for sensor fusion applications [120]. This algorithm is designed to quickly locate image corners, a process typically used to implement position tracking. Since FAST is primarily for detecting corners we have coupled it with the BRIEF feature descriptor to complete the feature extraction. The FAST algorithm uses only pixel intensity, within the 16 nearest pixels, to locate corners. If the pixel contrast is high enough to form a proper corner, the pixel is considered a feature point. Once the detection phase is completed, the BRIEF feature descriptor is computed. The BRIEF feature detector compares the pixel intensities within a smoothed image patch to form bit vectors of the results [25]. These are then used to describe the patch where the feature point was found. It was found that as little as 16 bytes were enough for accurate matching [25]. MEVBench contains a multithreaded version of the FAST and BRIEF combination. This version can be scaled to multiple thread counts and input sizes.

3.3.2 Feature Classification

Once feature descriptors are extracted there is typically a reasoning process that takes place based on these features. A common component of this reasoning phase is feature classification. Feature classification attempts to predict some information about a feature based on the descriptor. The classification will commonly use previous data to predict information about the new data. This operation is commonly implemented using machine learning techniques. MEVBench includes three different classification algorithms that are appropriate for use in embedded vision applications.

Support Vector Machine (SVM)

SVM is a supervised learning method used to classify feature vectors or descriptors [137]. The algorithm is trained with a set of feature vectors and their known classes. SVM treats each piece of data as a point in n-space where n is the dimension of the feature vector. It then tries to find separating hyperplanes in the n-space between the various classes. The result of training is a set of vectors called support vectors that can be used to evaluate a new data point in the classification phase. The query vector is combined with the support vectors in various ways to classify the new point. MEVBench has two versions of SVM. The first is from SD-VBS which is a single-threaded version of SVM that includes both the training and classification phase [139]. The second version is a multithreaded version built using the

OpenCV framework. This version implements a linear SVM kernel that can be scaled by both number of threads and number of input feature vectors.

Adaboost

Adaboost is a supervised learning method based on decision trees [49]. The algorithm uses a group of weak learner decision trees to increase the accuracy of classification. The training process of the classifier assigns a weight to each weak learner and then the individual results are summed together to determine the final classification. The query vector is merely classified by each weak learner and the results aggregated together based on the weights. Since Adaboost is based on decision trees, the computational complexity is not typically high. MEVBench has a multithreaded version of Adaboost. The number of threads and the number of feature vectors can be varied. It implements Adaboost with decision trees with a max depth of three.

K-Nearest Neighbor (KNN)

K-Nearest Neighbor is a supervised learning method that classifies new feature vectors based on their similarity to the training set. The implementation used in this work is based on FLANN [104]. The K nearest points in the training set vote for the class of the query vector. The votes are weighted based on the similarity of the query vector to the vectors in the training set. FLANN is configured for an approximate nearest neighbor using kd-trees thus eliminating the need to do a complete search of the training set. MEVBench provides a multithreaded K-nearest neighbors implementation of FLANN, using approximate neighbor matching as this is more appropriate for resource-constrained mobile applications. The number of threads along with the number of vectors can be varied.

3.3.3 Multi-image Processing

Multi-image processing is commonly used in embedded vision applications. In these applications, multiple images or frames are used to garner information about the scene or items within the scene. For example, tracking can be used in robotics to follow an obstacle or another mobile object. Each of the benchmarks for multi-image processing enable other applications while being an application in their own right. The benchmarks here are a subset of the SD-VBS suite [139], selected based on their suitability for use in mobile vision applications.

Tracking

Feature tracking involves extracting feature motion from an image sequence. In this benchmark we use the Kanade Lucas Tomasi (KLT) tracking algorithm [96]. The features are extracted and their motions are estimated based on interframe correlations. The features used are those from Shi and Tomasi [126]. The algorithm estimates the motion based on gradient information for each feature as it moves from frame to frame. Thus there is feature extraction, matching, and motion estimation for each tracked feature. This is a single-threaded version of the application. The input sizes can be varied for this benchmark.

Disparity

Disparity is a measure of the shift in a point from one image to another. It is used in stereo imaging to estimate distance or depth. It can also be used with images from a single camera. The disparity of a point or object is inversely proportional to the depth of object. Thus disparity is used to recover the 3D information lost within a scene when it is projected to a 2D image plane. In this benchmark we use the algorithm from Marr and Poggio [99]. The algorithm uses patch-based region comparisons to match the pixels between two images. This is a single-threaded benchmark. The input sizes can be varied from small to large images for this benchmark.

Image Stitching

Image stitching takes multiple images and merges them to form a single image. This operation requires matching regions of overlap between the images and aligning them accordingly, typically by combining feature detection and matching algorithms. The two images must also be blended together since the view points may be slightly different. The image stitching benchmark is a single-threaded implementation of the algorithm from [132].

3.3.4 Recognition Applications

Recognition applications utilize vision kernels, such as feature extraction and classification, to analyze images or scenes. These will typically augment the feature processing with reasoning to extract information from a scene. For example, object detection has a geometric constraint that must be met before an object is considered found (e.g., a standing person must be upon a horizontal surface).

Object Recognition

Object recognition uses computer vision to evaluate when a trained object is present within a scene. It is common for object recogntion to use feature extraction, feature classification and geometric constraints to recognize the trained object. The MEVBench benchmark is based on the technique described by Lowe [94]. This technique uses SIFT features and matches the query image's features to the trained object features. Then the features are filtered using a geometric constraint on their location. A histogram binning technique is used to group and verify the location predicted by the features. The features are then used to compute the estimated pose of the object and the error of the location for each feature is computed. If the total error is below a threshold, the object is considered located. The object recognition benchmark is built using the OpenCV framework, and it is multithreaded to support a variable number of processors.

Face Detection

Face detection is a common application for embedded computer vision. This involves locating a face within an image. The face detection technique used in this benchmark is based on Viola-Jones method [140]. This method uses box filters to locate faces present in the image at variable scales. The face detection benchmark is a single-threaded implementation, built using OpenCV.

Augmented Reality

Augmented Reality is an increasingly popular application on mobile devices for navigation and gaming. In augmented reality, a known marker is used to determine scale and provide a reference point within a scene. The MEVBench implementation uses a black and white marker that is located using a binary threshold-based segmentation of the image similar to the technique presented by Kato et al [85]. The marker is identified using a basic binary pattern on the marker face. The projection of a virtual cube into the scene, relative to the marker, is performed by estimating the markers translation and rotation relative to the camera. This technique requires the camera calibration be known and that the image be adjusted based on this calibration. The MEVBench implementation uses the OpenCV framework to implement this technique in a single-threaded application.

3.4 Benchmark Characterization

3.4.1 Experimental Setup

In order to evaluate the benchmarks, we employed a variety of physical and simulated systems. The embedded nature of the benchmark required that we look at cores commonly used in the mobile space as well as desktop systems. For the physical system embedded target we utilized an ARM 1GHz dual-core A9 with 1 GB of RAM. This class of processor is found in many smartphones and tablet SoCs such as the NVIDIA Tegra 2 [109]. For this device we ran the benchmarks on a TI OMAP4430. We called the clock_gettime() to measure application times on this platform. For the desktop class physical system we used an Intel Core 2 Quad Q6600 processor, configured as described in Table 3.2. Application timing on the Intel-based desktop system employed the hardware timestamp counter (TSC) to capture execution cycles count. The TSC is a register on Intel x86 architectures that holds a 64-bit counter that is incremented at a set rate. We modified the Linux kernel on this system to virtualize the TSC on a per-thread basis. As such, each thread has a copy of the TSC that it swaps in and out at context switches. This allows us to have cycle-accurate data on a per-thread basis on a standard Intel processor.

We gathered detailed microarchitectural characteristics not available on most physical systems, by employing an embedded platform simulator. The simulated embedded platform is a 1 GHz Intel Atom model with 2 GB of RAM, simulated using the MARSS x86 simulator [111]. For the simulated desktop target we used MARSS to simulate an Intel Core 2 class of processor in various configurations. Table 3.2 summarizes the experimental setup for the various processors. All benchmarks were compiled using GNU g++ compiler suite version 4.4.3 with maximum optimization. We also ran Intel Vtune Amplifier XE 2011 to gather code hotspot information for the MEVBench benchmarks [75].

To assess the scalability of the algorithms, they were run with varied thread counts and input sizes. We used three different input sizes in this evaluation: small, medium and large. The small inputs for the benchmarks are based on images that are a standard Common Intermediate Format (CIF) size of 352x288 pixels. The medium inputs are the standard VGA size of 640x480 pixels. The large inputs are the full HD size of 1920x1080 pixels. All image data is in color PNG format. For classification, the small input size was 30 vectors. The medium input size was 116 vectors. and the the large input was 256 vectors. All of the vectors have 3780 entries. Vector sizes were chosen to align with the expected computation load from feature extraction.

Table 3.2 Configurations for profiling MEVBench.

Feature	Configuration
Embedded Bare Metal	
Operating System	Linux 2.6.38-1208-omap4
Processor	1 GHz Dual Core Arm A9 (OMAP4430)
Memory	1GB low power DDR2 RAM
L1 Cache	32KBi, 32KBD private 4-way associative
L2 Cache	1MB shared
Desktop Bare Metal	
Operating System	Linux 2.6.32.24 Custom Kernel
Processor	2.4 GHz Intel Core 2 Quad Q6600
Memory	4 GB PC2-5300
L1 Cache	128KBi,128KBD private 8-way associative
L2 Cache	2x4 MB shared 16-way cache associative
Simulated Base Embedded Core	
Operating System	Linux 2.6.31.4
Processor	1.0 GHz 32bit x86 in Marss
Memory	2 GB
L1 Cache	32KBi private 8-way cache associative, 24KBD private 6-way cache associative
L2 Cache	512KB shared 8-way cache associative
Simulated Base Desktop Core	
Operating System	Linux 2.6.31.4
Processor	1.0 GHz 64bit x86 in Marss
Memory	2 GB
L1 Cache	32KBi, 32KBD private 8-way cache associative
L2 Cache	2MB shared 16-way cache associative

3.4.2 Dynamic Code Hot Spot Analysis

We examined the most executed instructions in the single-threaded versions of the benchmarks within MEVBench using medium-sized inputs running on an Intel Core 2 Quad Q6600 with 4GB RAM. We looked at the operations taking place at these various hot spots to determine possible software or hardware based optimizations.

SIFT

The SIFT benchmark based on OpenCV showed that gradient computation for descriptor building and feature point orientation accounted for 70% of the computation. Furthermore, the most executed computation component of this was a vector multiply. This vector multiply was part of the 2D gradient computation required to build the descriptor. The SIFT

benchmark from SD-VBS, which contains only the feature localization, spends 67% of the time blurring the image which involves a 2D convolution.

FAST and BRIEF

The FAST and BRIEF benchmark spent 15% of the time locating corners with the FAST feature detector. The majority of the FAST algorithm is spent on a compare operation used to detect the corners. BRIEF accounts for over 30% of the execution time. The primary operations in this portion of the benchmark are integral image computations and a smoothing computations. The integral image computations apply box filters which are convolution operations. The smoothing operation also utilizes a convolution operation.

HoG

The HoG benchmark spent 20% of the time computing the integral images. The primary operation in this computation was a vector add. Also there was a significant amount of time spent on a divide operation used for normalizing feature vectors.

SURF

The SURF benchmark had a hot spot in the edge detector used to localize features and compute the feature vectors. This was a vector instruction that accounted for 39.9% of the computation in the benchmark. Also, another 40% of the time was spent fetching image data from main memory. This is due to the nature of the image region based descriptors that SURF uses.

AdaBoost

In the AdaBoost benchmark, the primary computation took place in the prediction code. The majority of the runtime calls are spent traversing the trees and performing comparison operations in the decision trees. Thus the comparison operation is the largest component of this computation.

K-Nearest Neighbor

In the K-nearest neighbor benchmark 40% of computation was spent indexing the tree to find the nearest neighbor. The other 60% of the time was primarily taken up with a vector addition to compute the classification based on the neighbor's class.

SVM

In both the OpenCV and SD-VBS SVM benchmarks, over 60% of the computation time is spent with an inner product calculation. This involves multiplying two vectors element by element and summing the results together. This is the predominant operation for training the SVM classifier as well.

Stitch

The stitch benchmark spent 53% of execution performing a non-maxima suppression. In this operation the maximum feature response value within a region is used to filter out weaker feature responses. This operation requires many 2D spatial compares. The second highest fraction of computation for this benchmark was a convolution operation for finding features. This took 33.3% of the computation time.

Disparity

The disparity benchmark was computing the integral image for 57% of the time. The primary operation in this phase was a vector add operation. No other single operation dominated the remainder of the computation.

Tracking

The tracking benchmark had a hot spot in the 2D gradient computation. This operation constitutes 56% of the computation. This was mainly a vector operation for performing a convolution.

Face Detection

The face detection benchmark spent 60% of the time evaluating the class of the object using the cascade classifier. This is a decision tree designed such that when a decision evaluates

false no other comparisons are made and the classifier returns false, but when the evaluation is true additional compares are made until the final leaf node is reached.

Object Detection

The object detection benchmark combines feature extraction, classification, and a geometric check. The hot spot for this benchmark is the same as that of feature extraction. We found that feature extraction dominates the execution timing, taking 69% of the time.

Augmented Reality

The augmented reality benchmark has two major hot spots that take a combined 28% of the computation time. The first is the location of the marker by tracing contours or edges. The second hot spot performs correction of the image based on the camera calibration data. The adjustment allows the system accurately project the scene in 3D. Over 57% of the time in augmented reality is taken up with memory reads and writes.

To summarize our hot spot analysis, the results suggest that hotspots are taking place at vector instructions thus alluding to a vector architecture being useful. There are also hotspots that take place at complex or hard to predict control flow areas such as the cascade from face detection. In those cases there is a need to deal with irregular branching patterns. This may be difficult for traditional vector machines. Among the operations being performed at hotspots, the convolution operation is used in many benchmarks thus showing that accelerating that operation may be helpful to embedded vision applications. There are also hotspots that involve comparison operation in which a single value is being compared to many other values. Many benchmarks also require many memory accesses thus an efficient embedded vision processor must have a streamlined memory system.

3.4.3 Computational Performance

We examined the runtime performance of MEVBench on various platforms such as an ARM A9 and Intel Core 2 Quad. Figure 3.3 shows the number of cycles for single-threaded runs of MEVBench on a physical Core 2 Quad as the input is scaled. The logarithmic component to the cycles is due to the nature of image data. If both the height and width are scaled, the amount of work is scaled as the product of those increases. For example, the doubling of height and width increases the number of pixels to compute by a factor of 4. Thus moving to

HD computation on embedded systems will require a significant increase in computational efficiency.

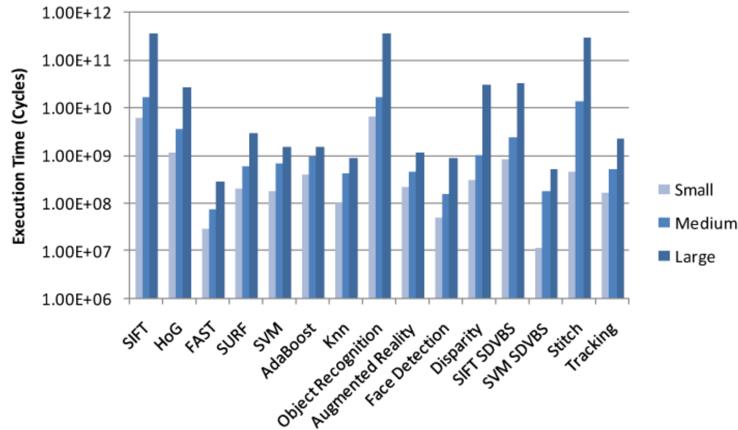


Figure 3.3 MEVBench Execution Time The figure shows the number of cycles of execution for single-threaded versions of each benchmark. This gives an indication of the amount of overall computation contained within each benchmark. The experiments were run on a modified Linux kernel on the Core 2 Quad bare metal configuration.

Figure 3.4 shows the instructions per cycle (IPC) for the simulated Core 2 and Atom cores. This figure shows the degree of instruction level parallelism the desktop cores can extract from the benchmarks when compared to the embedded Atom core. Given the amount of power and area costs to extract instruction level parallelism, the embedded processor will need to utilize more efficient computational resources to gain performance. Thus, instruction level parallelism may not be the driving performance enhancement in embedded platforms, which leaves thread-level and data-level parallelism to improve performance in this space.

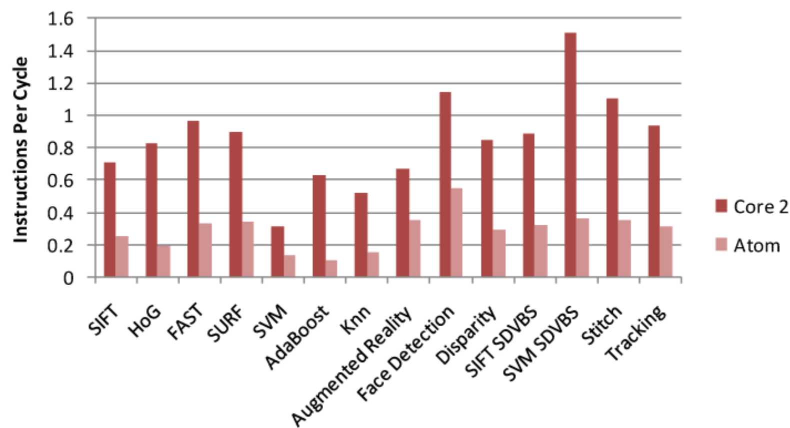


Figure 3.4 MEVBench IPC for Varied Core Capability The figure shows the IPC for each benchmark on both the simulated Atom and Core 2 cores. This marks the difference between desktop machines and embedded platforms in terms of throughput.

Figure 3.5 shows the effect of running the benchmarks on the actual Core 2 Quad when compiled with or without vector instructions. We chose this execution target because the vector width of the x86 SSE instructions is greater than the ARM SIMD engine. When the SIMD instructions are used, they are inserted automatically by the compiler. In some cases the inclusion of vector instructions hurt performance. This is due partly to the control complexity of the vision kernels such as k-nearest neighbor where the kd-tree is searched. This suggests that an efficient embedded processor for computer vision will need to support vector instructions in some cases but disable it in others where it might hinder performance.

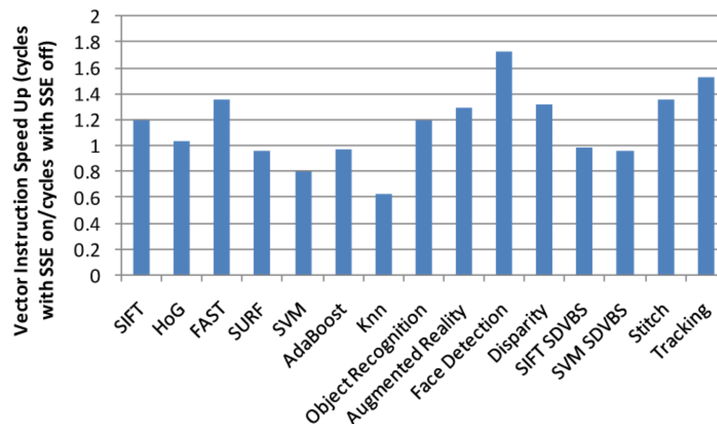


Figure 3.5 Vector Instruction Impact The figure shows the impact of using vector instructions on various versions of the benchmarks. This examines the amount of data parallelism present. The control complexity and need to rely on the memory system hinders the performance in some cases when vector instructions are activated.

3.4.4 Memory System Performance

Embedded systems have limited memory when compared to their desktop counterparts. In order to design properly for the embedded vision space these targets must efficiently serve the memory demands of vision applications.

A common limiting factor in low-cost systems is memory bandwidth. Vision algorithms rely a great deal on image data that has little temporal locality. Figure 3.6 examines the memory bandwidth used by each application, in terms of bytes activity per instruction. This includes all memory activity through the buses whether it is touched or not by the actual application. Thus a cache miss in L1 will result in two data transfers, the L2 to L1 and the L1 to processor. This is because the entire memory system is using power and all activity contributes to this. Bytes accessed per instruction are calculated on a per work element basis. As embedded vision systems move toward real time performance the memory system will

need to accommodate this amount of data movement per instruction. This metric is agnostic to the actual frame rate and designers are free to calculate the amount of data per second based on this and the number of instructions per frame. The Atom core exhibits a higher number here because of smaller caches forcing it to access full memory pages more often. Furthermore some applications access memory in a way that is not tailored to the traditional memory system.

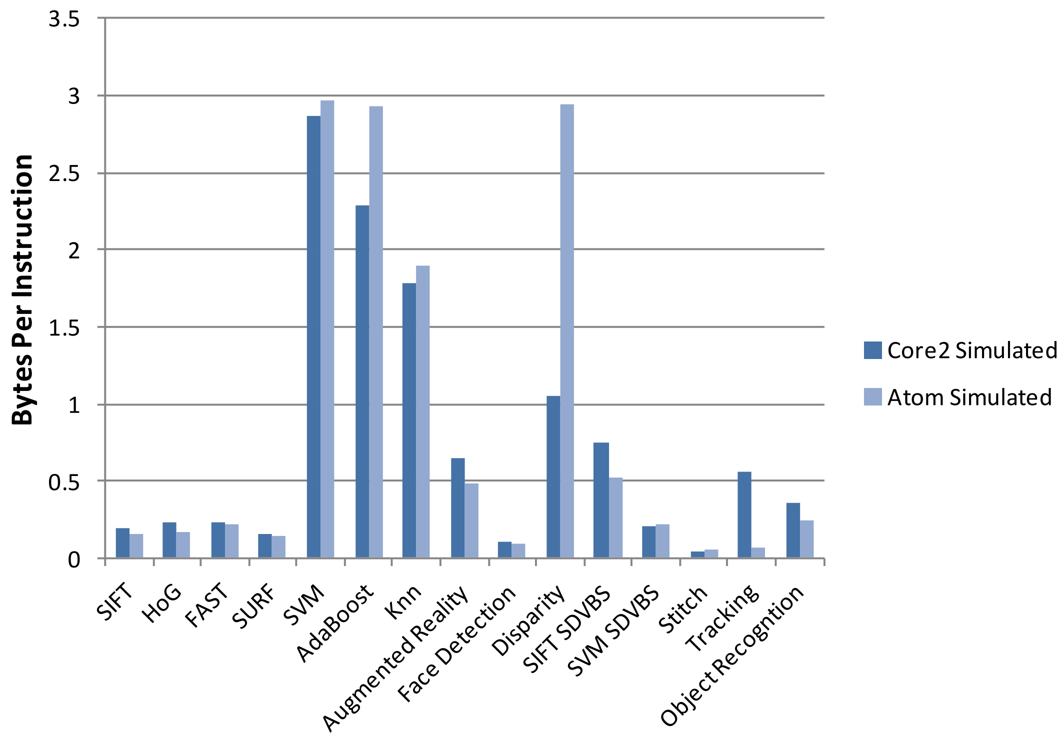


Figure 3.6 MEVBench Memory Activity Per Instruction The figure shows the memory activity in bytes per instruction for the single-threaded versions of each benchmark. This evaluates the stress on the memory system per instruction of work. The benchmarks were run on a single image or set of vectors. A high overhead indicates an inefficient memory fetch or caching policy. The algorithms often fetch memory that is not needed but it is still counted in the measured overhead. Furthermore, misses in upper level caches increase the numbers because the data needs to be transferred more than once.

Efficient memory is key to embedded performance in terms of energy and timing. In an embedded system, a cache miss can be more costly due to the slower memories. Thus we examined the L2 cache hit rate for various components of the MEVBench. We found that the misses per instruction could be improved. We also noted that many vision algorithms operate on images and thus can take advantage of 2D spacial locality. Thus, we created a cache controller that assumed all of memory contained images arranged into patches. A patches are groupings of data based on their location in the image. In classical image storage the pixel are stored in raster scan order meaning the access a pixel one row lower,

the processor must increase the memory address by at least the width of the image. In patch based memory the pixels are stored based by breaking the image into 2D regions and storing each region or patch together. Each patch was an 8 byte row by 8 byte column and data was fetched into the cache based on this arrangement with an 8-way associativity. We found that this outperformed the standard cache in the Core 2 simulation, as seen in Figure 3.7. Thus to increase performance, processors for the embedded vision space should take advantage of 2D locality present within many of the applications. Some applications exhibit a cache performance decrease with the patch cache. This is due partially to how it is accessing its data in the most time consuming portions of code. For example, Adaboost is a decision tree based algorithm that works with feature vectors. The patch cache prefetches the wrong data in this case. In SD-VBS SIFT, performance was reduced because the algorithm performs a 1D operation for a large amount of computation but the patch memory is designed for 2D data. The SD-VBS SVM has a similar issue due to the size of the vectors it uses. As such, 2D memory optimizations, while beneficial, should only be enabled for code with substantial 2D spatial locality.

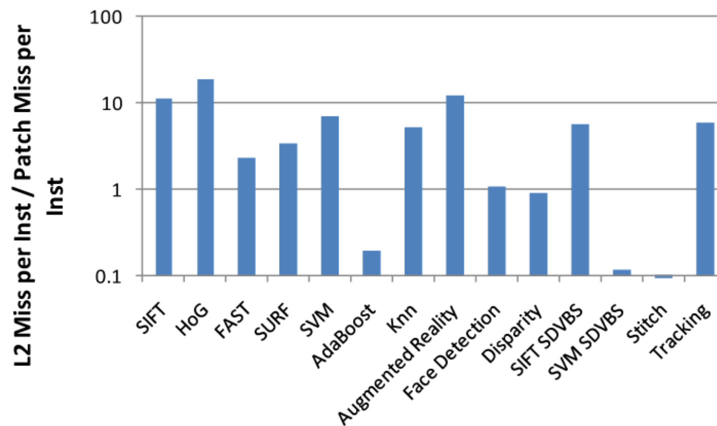
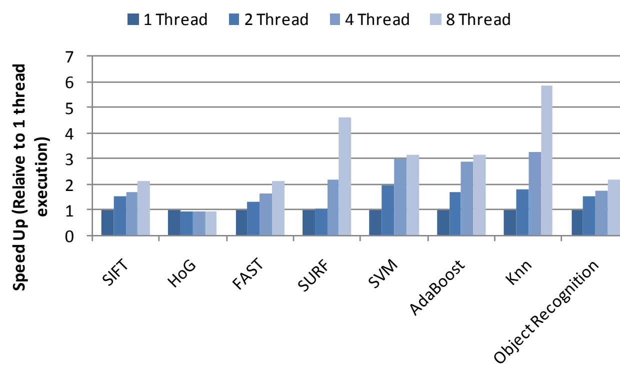


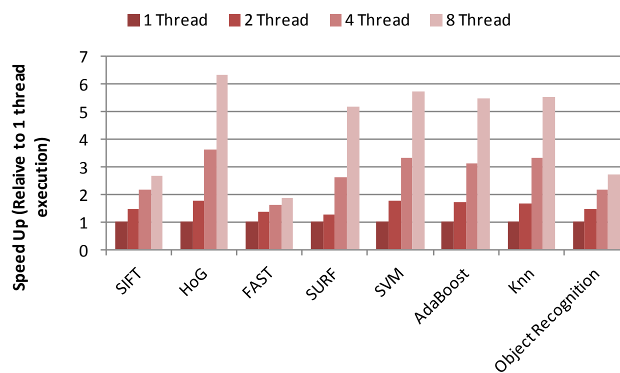
Figure 3.7 Patch Memory The figure shows the effect of using a patch based cache on various benchmarks. The Y axis is the cache misses per instruction for a traditional cache divided by the misses per instruction for a patch-based cache. A value above one means the patch based cache had higher performance while a value below one shows the traditional cache has higher performance. This demonstrates how much more efficient the patch cache can be for certain vision applications. This was done for the single-threaded versions of the benchmarks on the Core 2 simulated core with the small input sizes. Some algorithms have a worse miss rate. This is due partially to how it is accessing its data in the most time consuming portions of code. For example, Adaboost is a decision tree based algorithm that operates on feature vectors. The patch cache prefetches the wrong data in this case.

3.4.5 Multithreaded Performance

Figure 3.8 shows the performance in cycles of a dual core ARM A9 as the number of threads is increased for the medium and small input sizes. The performance of HoG is seen to get worse and plateau as the thread count reaches 4, this is due to the memory used by HoG being large and requiring swap as the number of threads increases. It should also be noted that FAST/BRIEF also suffers an issue where as the number of cores increases, coordinating the cores overtakes the execution of the feature extraction. This shows that for some algorithms, a lower number of high-performance cores may perform better than a large number of small cores.



A



B

Figure 3.8 Performance (cycles) vs Number Of Threads The figure shows the effect of using multithreaded versions of some of the benchmarks. Specifically the feature extraction, classification, and object recognition benchmarks. This is because extraction and classification are core vision components. Object recognition is used to show the potential of multithread vision applications. These plots show the performance of the 1GHz dual core ARM A9. The plot (A) on the top is for a small input size while the plot (B) on the bottom is for a medium input size. The HoG trend is caused by the large memory demand of HoG which incur page faults in the memory system.

We examined the regularity of branching to evaluate how well MEVBench algorithms

might map to architectures where multiple cores perform the same instruction in lock step, such as GPGPUs. We looked at the top 30% of dynamic branches from various benchmarks and measured how often they changed targets. Figure 3.9 shows the branch divergence measure for the various benchmarks. AdaBoost and HoG have such high transitions because they have high control complexity built into the algorithms. HoG is performing a binning operation on the entire image as a first operation to compute the feature descriptor. This binning operation is designed to decrease the amount of time and loops for future computation. Adaboost is a decision tree based classifier, thus as each feature is evaluated, branching in the tree is quite varied. Stitch has a portion where values are compared for non maxima suppression. This major operation in stitch is executed many times. Thus, this operation dominates and can lead to different control paths. The branch divergence characteristics suggest that some of the algorithms would experience significant stalls on a GPGPU.

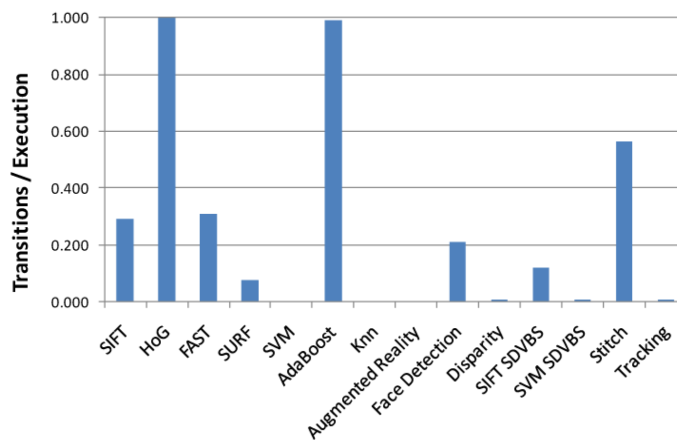


Figure 3.9 Branch Divergence The figure shows the branch divergence present in the benchmarks. This figure shows how often the top 30% of the most executed branches change their target location. This result can be used to predict the complexity of algorithmic control flow, and its amenability to dataflow architectures such as GPGPUs.

Figure 3.10 shows the average IPC for multithreaded versions the MEVBench benchmarks. FAST and BRIEF performance drops off quickly due to the coordination taking more time and the threads waiting for each other to finish. The SURF and SVM degrade because some cores do more work than others , forcing many to wait during coordination. The total IPC of the system is still always higher than the individual threads, however.

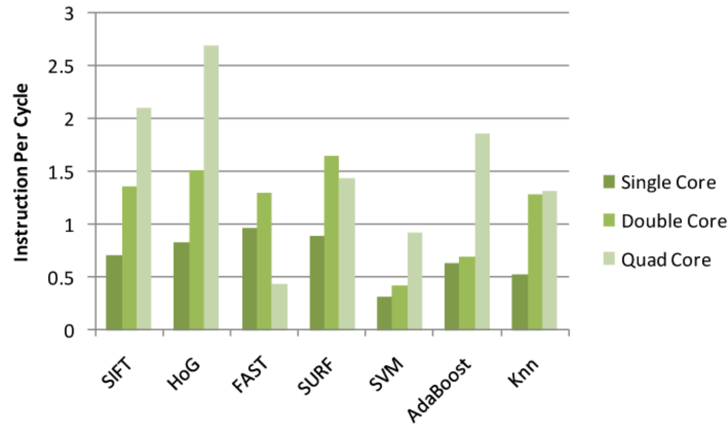


Figure 3.10 Average Mutlithreaded IPC The figure shows the average IPC of the cores based on the Intel Core 2 as the number of threads and cores are increased. This is a measure of how much the various cores affect each other's performance. The FAST performance drops off at 4 cores due to the amount of work coordinating the threads exceeding the work to actually perform the feature extraction.

3.5 Chapter Conclusion

The analysis has shown that some of the MEVBench workloads would benefit from data parallelism while others may be hindered. Therefore, a key attribute of an mobile embedded processor for this space is the ability to extract data-level parallelism when present but still be able to perform well at single-threaded applications. In some cases the algorithms' performances are limited due to the computation required to coordinate multiple threads and in others the limitation is due to code with serial dependencies. This lends itself to a multicore with at least one powerful core to deal with these single-threaded applications, plus additional (possibly simpler) cores for leveraging available explicit parallelism. Given the area, cost and power constraints, a heterogeneous multicore with lower area and power cores to support a larger core when thread level parallelism is available is a fair solution to this issue.

There is a fair bit of control complexity in the various workloads. Thus the core needs the ability to efficiently handle diverging control flow. However, the hotspot analysis showed many applications have hot spots at vector operations. For some benchmarks, control complexity rules out traditional vector machines and possibly GPGPU architectures that do not deal well with branch divergence. Thus, architectures that support specific vector instructions would be a better fit.

It was found that patch-based memory accesses can take advantage of the ample 2D spatial locality present in many vision algorithms. This should decrease energy and execution

time. The memory accesses per instruction data shows that inefficient memory management in the architecture can increase the memory bandwidth requirement. We also see that not all benchmarks see improvement from the patch memory; thus it is beneficial to allow multiple memory access modes to increase performance, decrease required memory bandwidth and decrease energy usage. We have also seen there is a performance gap between the embedded and desktop systems. This will need to be overcome to enable more accurate embedded vision applications. As mobile vision systems push toward accurate real-time computation, the need for performance will continue to increase.

This chapter addressed characterizing the computation that takes place in mobile vision codes. MEVBench is the first benchmark suite that is tailored to the mobile vision space and our analysis of it showed that there are various types of computation and hot spots in these applications. The next chapters utilize this knowledge to optimize mobile vision systems, first with software and then with hardware. Chapter 4 begins our solutions for optimization with an algorithm which can be utilized in the augmented reality to improve performance based on the hot spots presented in this chapter. Chapter 5 continues improving the performance of mobile vision systems by developing hardware optimizations to support feature extraction based on the heterogeneous nature of the workloads, data parallelism and 2D spatial locality for memory accesses. Chapter 6 expands the scope of hardware by developing accelerators for the entire mobile vision software pipeline.

Chapter 4

Targeted Singular Vector Computation for Vision Applications

Lost Time is never found again.

Poor Richard's Almanac
Benjamin Franklin

4.1 Introduction

Chapters 2 and 3 focused on better understanding mobile vision systems and characterizing the computation within mobile vision codes. The next few chapters utilize this information to provide optimizations to increase the performance of these mobile systems. Mobile systems typically have constraints on their computational capability thus we begin by optimizing the software to decrease the computational load. Our software based optimization is an algorithm that can be utilized to improve the performance of the augmented reality benchmark from Chapter 3. One of the hot spots of the augmented reality application was the computation of the homography. In this chapter, we present an efficient algorithm for solving for a singular vector which is a key component of homography estimation.

Computing singular values and singular vectors of a matrix is a crucial step in the solution to a wide variety of computer vision problems. Today, singular value decomposition (or SVD) remains the method of choice for eigen analysis in computer vision. However, in a number of vision problems, only a small subset of singular pairs are relevant and used, while the remaining computed values are discarded. This is an inefficient use of memory and computation resources.

In this chapter, we present an approach for directly computing relevant subsets of singular values and the corresponding singular vectors. Our approach utilizes the linear algebra

technique called inverse iteration, and we describe its applicability to two problems commonly encountered in vision algorithms, solving homogeneous equations using normalized Direct Linear Transformation (NDLT) [66] and matrix factorization as employed in rigid structure from motion (SFM) [84].

4.1.1 Contribution of This Chapter

The NDLT algorithm relies on SVD to compute the singular vector corresponding to the smallest singular value. In the factorization approach to SFM, SVD is used to compute left and right singular vectors corresponding to the top 3 singular values. In both these cases, replacing the traditional SVD with our approach, which we refer to as SEVS (Specific Eigen Vector Solver), straightaway produces significant computational speedups without any appreciable drop in accuracy.

We compare SEVS to LAPACK [5] optimized for an Intel architecture using Intel's Math Kernel Library (MKL) [74] and to textbook implementation from Numerical Recipes in C [114] (NRC). Our experimental analysis reveals that, for the vision applications considered, SEVS is a) at least as accurate as these benchmark SVD implementations, and b) capable of providing significant computational savings at an application level. For homography estimation, a basic C implementation of SEVS sped up the NDLT algorithm by about $4\times$ compared to NRC, and a partially optimized SEVS outperformed hardware-accelerated LAPACK by $8\times$ for the 4-point NDLT algorithm. For SFM factorization, SEVS offered a linear growth in speedup over LAPACK, with factors between $6\times$ and $9\times$ for sequences from 120 to 480 frames.

4.2 Related Work

Standard techniques to compute SVD are detailed in [71]. Common to all these algorithms is the Golub-Kahan bi-diagonalization [53]. The numerical methods for computing SVD vary in their approaches to compute the singular values and vectors of the bi-diagonal form after the Golub-Kahan step. LAPACK [5], the standard linear algebra library, provides two versions of SVD. The more robust DGESVD function permits computation of either none, left, right, or both sets of singular vectors and all the singular values. The faster DGESDD function permits computation of either all or none of both (left and right) sets of singular vectors together with all singular values. However, none of these options addresses the need of several applications which require only a few singular vectors from the row or column

space. MATLAB utilizes LAPACK, and to our knowledge, always computes all of the left and right singular vectors.

While the idea of computing a specific eigen pair (Fiedler Vector) is employed with great success in the field of spectral graph partitioning for image segmentation [125], no solutions address the problem of computing particular singular pairs in areas of computer vision such as homography estimation and structure from motion. We employ techniques from linear algebra literature to solve for specific singular pairs using the ideas of bisection and inverse iteration [54]. As we discuss in this chapter, in order to employ inverse iteration, it is important to have good initializations of a starting vector and of the singular value of interest. While the problem of finding good starting vectors has been addressed quite extensively in linear algebra, for example in [80], the latter problem has not received much attention. In our analysis, we exploit the structure of the application to define efficient means to initialize both the the starting vector and singular value such that the method converges to the singular values/vectors of interest.

The normalized DLT [66] is a widely used algorithm for homography estimation, camera calibration, triangulation, etc., that utilizes SVD to solve a system of homogeneous equations. The solution is the right singular vector corresponding to the smallest singular value. Cognizant of the wasted flops involved in computing both the left and right singular vectors, the authors of [66] recommend computing only the right singular pairs. In [65] a different homography algorithm is presented that also relies on SVD to extract a right singular vector, and here too, the observation is made regarding the computing only the right singular pairs. The method presented in this chapter goes beyond addressing this need: not only does SEVS allow computing either the left or right singular pairs, it enable the direct computation of *only* the desired singular vectors. While this chapter describes the speedups offered by SEVS to NDLT [66], it is equally applicable to homography estimation method of [65].

Low-rank matrix factorization is another common technique used in computer vision [21, 84] that relies heavily on SVD. In particular, for the structure from motion (SFM) problem, the factorization [84] of a point trajectory matrix into smaller structure and motion components is today solved by first applying SVD, and then discarding all but the 3 top singular values and corresponding left and right singular vectors. We describe the application of SEVS to directly compute only the required singular vectors and the computational benefits of doing so.

4.3 Method

4.3.1 Preliminaries

For any given matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, there exists the singular value decomposition

$$\boldsymbol{A} = \mathbf{U}_{m\times m}\Sigma_{m\times n}\mathbf{V}^T_{n\times n} \tag{4.1}$$

where Σ is a diagonal matrix of singular values, and $\mathbf{U}$ and $\mathbf{V}$ are orthonormal matrices containing the right and left singular vectors of $\mathbf{A}$, respectively [54]. The SVD of $\mathbf{A}$ relates to the eigen decomposition of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$. The singular values, $\sigma_i \in \text{diag}(\Sigma)$, of $\mathbf{A}$ are equal to the square root of the eigenvalues of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$, and left and right singular vectors of $\mathbf{A}$ are the eigenvectors of $\mathbf{A}\mathbf{A}^T$ and $\mathbf{A}^T\mathbf{A}$, respectively [54]. Thus solving for a singular pair or set of singular pairs can be made equivalent to solving a symmetric eigen problem.

We will not discuss algorithm details of SVD in this chapter; there exist well-known stable techniques to compute the SVD. In order to compute $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{D}$, SVD takes $4m^2n + 8mn^2 + 9n^3$ computations. If only the right singular pairs $\mathbf{V}$ and $\mathbf{D}$ are required, the complexity reduces to $4mn^2 + 8n^3$.

There are alternatives to compute a single eigen pair of an $m \times m$ matrix $\mathbf{C}$, provided a close approximation of the eigen value is known a priori. One method, known as inverse iteration [143], iteratively solves the system of linear equations

$$(\mathbf{C} - \hat{\lambda}\mathbf{I})\mathbf{x}_k = r_k\mathbf{x}_{k-1}, \quad k \geq 1 \tag{4.2}$$

where the vector $\mathbf{x}_k$ eventually converges to an eigen vector corresponding to the eigenvalue closest to the starting estimate $\hat{\lambda}$. In Eqn. 4.2, r_k is a scalar that normalizes $\mathbf{x}_{k-1}$, and $\mathbf{x}_0$ is initialized to any vector, usually of unit norm. We investigate the applicability of this technique as an alternative to SVD in computer vision problems.

4.3.2 Computing Right Singular Pairs

The basic steps of our approach are outlined in Algorithm. 1. Here we assume that we are interested in right singular pairs of input matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, where $m > n$. We discuss computing the left singular pair of such a matrix in Sect. 4.3.3.

We first convert our task into an eigen problem by computing $\mathbf{A}^T\mathbf{A}$, giving us a square positive semi-definite matrix C. The right eigen pairs of C correspond to the right singular

Input : $\mathbf{A} \in \mathbb{R}^{m \times n}$, Error limit ε, Indices of desired eigenpairs K
Output: Singular pairs $(\sigma_k, \mathbf{v_k})$, $k \in K$

$\mathbf{C} = \mathbf{A}^T \mathbf{A}$
for $k \in K$ **do**
  $\hat{\lambda}_k \leftarrow$ `EstimateEigenvalue`($\mathbf{C}$, k)
  $\mathbf{q}_0 \leftarrow$ `InitializeEigenvector`($\mathbf{C}$, k)
  $[\lambda_k, \mathbf{v_k}] \leftarrow$`InverseIteration`($\mathbf{C}, \hat{\lambda}_k, \mathbf{q}_0, \varepsilon$)
  $\sigma_k = \text{sqrt}(\lambda_k)$
end

Algorithm 1: Computating Subset of Right Single Pairs

pairs of A. We note that this step raises concerns about the condition number of C and is, in general, avoided in SVD. However, in our applications to computer vision problems (Sect. 4.4), we faced no such issues. We discuss alternatives and trade-offs in Sect. 4.6.

We proceed next to estimate a suitable approximation $\hat{\lambda}_k$ to the desired eigenvalue λ_k. This step is of crucial importance to the accuracy and computational efficiency of our method. In general, inverse iteration will converge to the eigenvalue closest to $\hat{\lambda}_k$. Further, the convergence rate, R, is proportional to ratio of the distance of $\hat{\lambda}_k$ to the nearest two eigen values,

$$R \propto \left| (\lambda_j - \hat{\lambda}_k)/(\lambda_k - \hat{\lambda}_k) \right| \tag{4.3}$$

where λ_k and λ_j are the closest and second closest eigenvalues to $\hat{\lambda}_k$. We describe efficient techniques for estimating $\hat{\lambda}_k$ for application in homography estimation and structure from motion in Sections 4.3.4 and 4.3.5.

The next step in our approach is to initialize a starting vector $\mathbf{q}_0$ for inverse iteration. Subject to the constraint $\|\mathbf{q}_0| = 1$, the choice of good starting vectors has received considerable attention in linear algebra. We discuss different choices of $\mathbf{q}_0$ and their trade-offs from an application driven perspective in Sections 4.3.4 and 4.3.5.

Having computed $\hat{\lambda}_k$ and $\mathbf{q}_0$, we now describe our implementation of the inverse iteration algorithm (see Algorithm 2). Inverse iteration involves computing $z_i = (\mathbf{C} - \hat{\lambda}_k \mathbf{I})^{-1} q_i$. We employ LU Decomposition to solve the linear system of equations, noting that though the LU factorization has cubic complexity, it is performed only once and $\mathbf{L}$ and $\mathbf{U}$ are re-used at each iteration. Along with the construction of $\mathbf{C}$, we discuss trade-offs of other numerically stable methods in Section 4.6. Instead of the commonly used two norm metric [54], we propose a metric based on the stricter albeit computationally friendly ∞ norm. While we do not have a proof of convergence, our extensive numerical evidence suggests that the metric

used serves its purpose in applications of interest.

Input : $\mathbf{C} \in \mathbb{R}^{m\times m}$, Eigenvalue estimate $\hat{\lambda}$, Initial eigenvector $\mathbf{q}_0$, Error limit ε
Output : Eigenvalue λ, Eigenvector $\mathbf{v}$

$err = \varepsilon + 1$
$i = 1$
while $err > \varepsilon$ **do**
  $Solve(\mathbf{C} - \hat{\lambda}\mathbf{I})\mathbf{z}_i = \mathbf{q}_i$
  $\mathbf{q}_i = \mathbf{z}_i / \|\mathbf{z_i}\|_2$
  $err = \text{abs}(\|q_i\|_\infty - \|q_{i-1}\|_\infty)$
  $i{+}{+}$
end
$\lambda = \mathbf{q}_i{}^T \mathbf{C} \mathbf{q}_i$
$\mathbf{v} = \mathbf{q}_i$

Algorithm 2: Modified Inverse Iteration

Overall, in order to compute K right singular pairs, our approach outlined in Algorithm 1 has a complexity of $O\left(mn^2 + K\left((2n^3/3) + tn^2 + \delta\right)\right)$, where t is number of iterations in Algorithm 2, and δ is the computation of $\hat{\lambda}$ and $\mathbf{q}_0$ (expected to be negligible compared to the other terms).

4.3.3 Computing Left Singular Vectors

If A is such that $m \geq n$, and a subset of the right singular vectors and singular values are computed using Algorithm 1 we can solve for the corresponding left singular vectors directly. Let us denote by $\mathbf{V}_{n\times K}$ the matrix of right singular vectors and by $\mathbf{D}_{K\times K}$ the diagonal matrix of singular values computed at the end of Sect. 4.3.2. By construction, since $\mathbf{A} \approx \mathbf{U}_{m\times K}\mathbf{D}_{K\times K}\mathbf{V}^T_{K\times n}$, we directly compute $\mathbf{U}_{m\times K}$ as follows

$$\mathbf{U}_{m\times K} = \mathbf{A}_{m\times n}\mathbf{V}_{n\times K}\mathbf{D}^{-1}_{K\times K} \tag{4.4}$$

Using Eqn. 4.4 produces the left singular vectors corresponding to $\mathbf{D}_{K\times K}$ and $\mathbf{V}_{n\times K}$ in $O(mnK + mK)$ computations over Algorithm 1.

If a subset of only the left singular pairs are of interest, and $m < n$, one can directly apply Algorithm 1 after changing the definition of $\mathbf{C}$, such that $\mathbf{C} = \mathbf{A}\mathbf{A}^T$. In general, since the order of *LU* Decomposition is cubic in the size of the matrix, our approach is most beneficial when the matrix $\mathbf{C}$ in Algorithm 1 can be made as small as possible.

4.3.4 Application to Homogenous Equations

Systems of homogenous equations of the form $\mathbf{Ax} = 0$ occur often in reconstruction problems such as homography estimation, triangulation, and camera matrix estimation. The normalized Direct Linear Transform [66] (NDLT) is a well known algorithm for solving such problems.

The typical input to NDLT is a set of m point correspondences. Assuming 2D points, the equations in $x-$ and $y-$ coordinates per point pair are reordered and stacked to form a $2m \times n$ matrix $\mathbf{A}$ such that $\mathbf{Ah} = 0$, where $\mathbf{h}$ is a $n \times 1$ vector whose elements we seek. If $\text{rank}(\mathbf{A}) < n$ the required h is a non-zero vector that lies in the 1D null space of A. Otherwise, an exact solution is not available in practice, and a vector h that minimizes the norm $||Ah||_2$ subject to the constraint $||h||_2 = 1$ is sought. In either case, the solution to this homogenous system of equations is the right singular vector of A that corresponds to the smallest singular value. This solution is typically obtained by computing the SVD of $\mathbf{A}$ and discarding all but said right singular vector. We believe that SEVS provides a more efficient way to solve this problem by directly computing only the desired singular vector.

To enable the application of SEVS, we need inexpensive techniques to compute $\hat{\lambda}$ and $\mathbf{q}_0$. Given the positive semi-definite nature of the matrix $C = A^T A$, we know that all eigenvalues of $\mathbf{C}$ are greater than or equal to 0. A safe estimate for the smallest eigenvalue is thus $\hat{\lambda} = -1$. Additionally, we examine the entries in $\mathbf{C}$ to scale $\hat{\lambda}$ appropriately. Having a suitable estimate for $\hat{\lambda}$, we found the choice of $\mathbf{q}_0$ to have little impact on the final solution as long as $||\mathbf{q}_0|| = 1$ thus any vector with a value of 1 in single entry and 0 for all other entries is sufficient.

In Sect. 4.4.1, we apply SEVS within NDLT for the task of homography estimation. For homography estimation, typically the 4-point NDLT is executed several times within a RANSAC loop. In addition, NDLT can also be applied to larger sets of points to obtain a least squares homography estimate. In Sect. 4.4.1, we describe the use of SEVS to accelerate NDLT over a wide range of input sizes.

4.3.5 Application to Matrix Factorization

The factorization method for rigid SFM [84] is another classic computer vision algorithm which is critically dependent on estimating a small set of singular vectors. The requirements for singular vector decomposition in SFM are very different from that of homography estimation. For rigid SFM, the factorization method requires recovering the left *and* right eigen vectors corresponding to the top 3 eigenvalues.

In SFM, the input matrix $\mathbf{A}$ is of size $2F \times N$ composed of x and y coordinates of N features tracked over F frames. The factorization method seeks the decomposition $\mathbf{A} = \mathbf{RS}$, where R is the $2F \times 3$ "motion" matrix consisting of one 2×3 rotation matrix for each of the F camera views, and $\mathbf{S}$ is the $3 \times N$ "structure" matrix holding the 3D coordinates of the N tracked features.

To estimate R and S, the factorization approach relies on first decomposing $\mathbf{A}$ such that $\mathbf{A} \approx \hat{\mathbf{U}}_{2F\times3}\hat{\mathbf{D}}_{3\times3}\hat{\mathbf{V}}_{3\times N}$. To the best of our knowledge, this step has so far been carried out by first computing the SVD (see Eqn. 4.1) of A, then identifying the top 3 singular values and corresponding right and left singular vectors, and finally discarding all other computed values. In this section, we describe how we use SEVS to *directly* compute only the 3 singular pairs corresponding to the largest singular values.

Apart from the measurement matrix $\mathbf{A}$ and error limit ε, the input to Algorithm 1 is the set $K = 1,2,3$ corresponding to the indices of the 3 largest eigenvalues. In order to estimate $\hat{\lambda_1}$, we rely on the trace of $\mathbf{C}$. The trace of a square matrix is the sum of its eigenvalues. Typically, the top few eigenvalues dominate the trace, with λ_1 being the largest contributor. Hence, we set $\hat{\lambda_1} = \text{tr}(C)$. The estimates for the remaining eigenvalues are set according to

$$\hat{\lambda_k} = \text{tr}(\mathbf{C}) - \sum_{i=1}^{k-1} \lambda_i \tag{4.5}$$

As long as the top 3 eigenvalues are well separated from the remaining, Eqn. 4.5 provides reasonable estimates $\hat{\lambda_k}$ for $k \in K$.

Unlike in homography estimation, for SFM, we find the initialization of $\mathbf{q}_0$ to have a bearing on both the rate of convergence and accuracy of the estimated eigenvectors. According to [78], $\mathbf{q}_0$ can be any vector as long as it contains some contribution from the desired eigenvector. Following the discussion in [78], for each $k \in K$, we set $\mathbf{q}_0 = \mathbf{e}_k$, where $\mathbf{e}_k$ is the kth column of an identity matrix of the same size as $\mathbf{C}$. We note that the vector of all ones, as used for homography estimation, and suggested in [143], gave poor results in the SFM use-case.

Using the above techniques for estimating approximate eigen values and starting vectors, one can directly apply Algorithm 1 to compute the right singular pairs for an SFM measurement matrix.

4.4 Experiments

In this section, we evaluate the use of SEVS for homography estimation (Sec. 4.4.1) and structure from motion (Sec. 4.4.2). For both applications, we first validate accuracy using relevant datasets and then evaluate computational efficiency. We compare SEVS against two benchmark SVD implementations, `svdcmp` from Numerical Recipes in C (NRC) [114] and `gesvd` from LAPACK.

In our build environment, LAPACK utilizes hardware-accelerated linear algebra routines (Basic Linear Algebra Subprograms or BLAS) provided by the Intel Math Kernel Library (MKL) [74]. On the other hand, SEVS is implemented entirely in unoptimized natural C. We utilize this pure natural C version when evaluating performance against NRC. When comparing against hardware-accelerated LAPACK, we employ BLAS matrix multiplication (`sgemm`) for two steps in the SEVS algorithm: a) computation of **C** in Algorithm 1, and (b) evaluation of Eqn. 4.4. We note that in either case, the SEVS implementation is at a disadvantage compared to both, the thoughtfully crafted NRC code, and the fully hardware accelerated LAPACK functions.

4.4.1 Homography Estimation

We used 605 homography matrices, H_i $(0 < i \leq 605)$, corresponding to perspective transformations from a real augmented reality application as our test set. We generated a random set of 500 2D points, $\mathbf{x}$, within a 640x480 coordinate space. For each known homography H_i, we created a corresponding set of points $\mathbf{x_i'}$ by applying H_i to 400 points from $\mathbf{x}$ and transforming the remaining 100 points with a random 3×3 matrix. Further, each transformed pixel coordinate was perturbed by a random noise vector of magnitude ≤ 2 pixels.

In the following experiments, we examined the problem of estimating the homography $\hat{H}_i$ between the noisy point sets $\mathbf{x}$ and $\mathbf{x_i'}$. We employ a classic two-stage approach to this problem. In Stage 1, we use the normalized DLT (NDLT) algorithm [66] within a RANSAC loop to generate an initial homography estimate and a corresponding set of inlier points. Then, in Stage 2, we employ one of two methods to refine the homography estimate: a) apply NDLT with all inliers to obtain a least squares homography solution, or b) apply Levenverg-Marquardt [92] to refine the homography estimate by minimizing the symmetric transfer error across all inliers. We analyze the use of SEVS, NRC (`svdcmp`), and LAPACK (`gesvd`) for Stages 1 and 2 of homography estimation. In addition, we also compare against using SEVS for Stage 1 followed by Levenberg-Marquardt optimization for Stage 2. We refer to this method as "SEVS+LM" in this section. For SEVS, we apply Algorithm 1 as

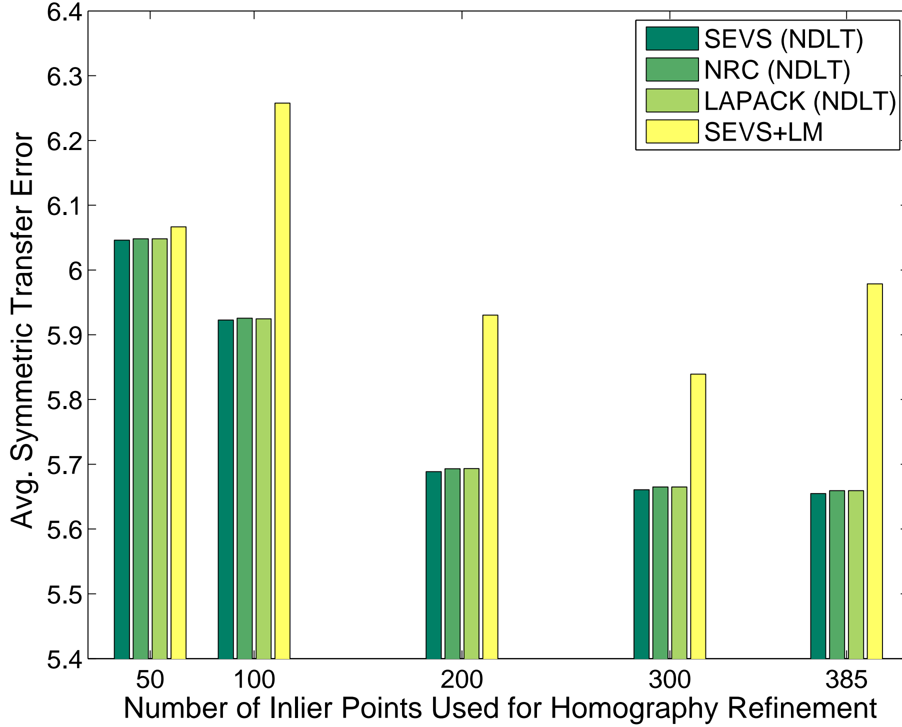


Figure 4.1 Effect of Number of Inlier Points on Symmetric Transfer Errors This figure shows the effect of the number of inlier points on symmetric transfer errors using NDLT and LM for homography refinement. Overall, NDLT produces lower error than using LM. Moreover, using SEVS in NDLT produces marginally lower errors than LAPACK and NRC.

described in Sections 4.3.2 and 4.3.4. For LAPACK, we configure `gesvd` to compute only right singular vectors.

Accuracy

In each case enumerated above, we use the symmetric transfer error [66] averaged over all 605 pairs to quantify the accuracy of the method. We summarize the accuracy numbers of the four methods in Table 4.1. These results clearly show that SEVS is at least as accurate as SVD implementations of NRC and LAPACK for the task of homography estimation. Interestingly, comparing SEVS with SEVS+LM in Table 4.1, we see that using NDLT with all available inliers produces lower symmetric transfer error than using LM for refinement.

We further analyze the refinement phase of homography estimation by varying the number of inlier points used in Stage 2 of our algorithm. In this experiment, we applied SEVS in Stage 1 to get an initial homography estimate and inlier set. Out of the 400 possible inliers in our dataset, we obtained at least 385 inliers for each of the 605 homographies.

	SEVS	NRC	LAPACK	SEVS+LM
Error	5.6549	5.6593	5.6592	5.9785

Table 4.1 Symmetric transfer error in homography estimation using SEVS, NRC, and LAPACK This figure shows the symmetric transfer error in homography estimation using SEVS, NRC, and LAPACK for singular vector estimation in NDLT.

We then varied the numbers of inliers used to refine the homography by both NDLT and LM. The results of this experiment are shown in Fig. 4.1. First, the graph clearly shows that if we increase the number of inlier points used with NDLT, the average symmetric transfer of the computed homography decreases. Second, this plot shows that SEVS is as good as, if not better than, NRC and LAPACK for use within NDLT across a wide range of input sizes. Third, we see further evidence that using NDLT with computed inliers produces lower symmetric transfer errors than LM minimization. This suggests that for homography estimation, apart from using the 4-point NDLT within RANSAC, it is also of great practical interest to be able to efficiently compute NDLT on larger point sets.

Computational Efficiency

Using, in turn, SEVS, NRC and LAPACK for singular vector estimation within NDLT, we profile the execution time of the algorithm for various input sizes. For each input size we measure the average time over 605 homography calculations, and capture the speedup offered by SEVS by computing the ratios LAPACK/SEVS and NRC/SEVS. The former indicates the computational speedup of SEVS over LAPACK, and the latter the speedup of SEVS over NRC. (We use the Query Performance Counter utility in Visual Studio 2011 to get high precision timing information).

The results of this experiment are shown in Fig. 4.2, which plots speedup factors across input sizes ranging from 4 (8×9) to 385 points (770×9). The solid lines capture the speedups for just solving $\mathbf{Ah} = 0$ within NDLT. Our pure C implementation of SEVS is consistently around $4\times$ faster than NRC for all input sizes. Compared to hardware-accelerated LAPACK, our partially optimized SEVS implementation provides dramatic speedups ($> 10\times$) at smaller input sizes. And even though LAPACK is tuned for high performance for large inputs, SEVS outperforms LAPACK by roughly $5\times$ for inputs of size 600×9 and larger. Note that these speedup numbers are relevant to the general problem of solving homogenous systems of equations which occur in many vision problems.

The dashed lines in Fig. 4.2 indicate speedups obtained for the full NDLT algorithm. We see that for the most common input size of 4 points, partially optimized SEVS speeds up

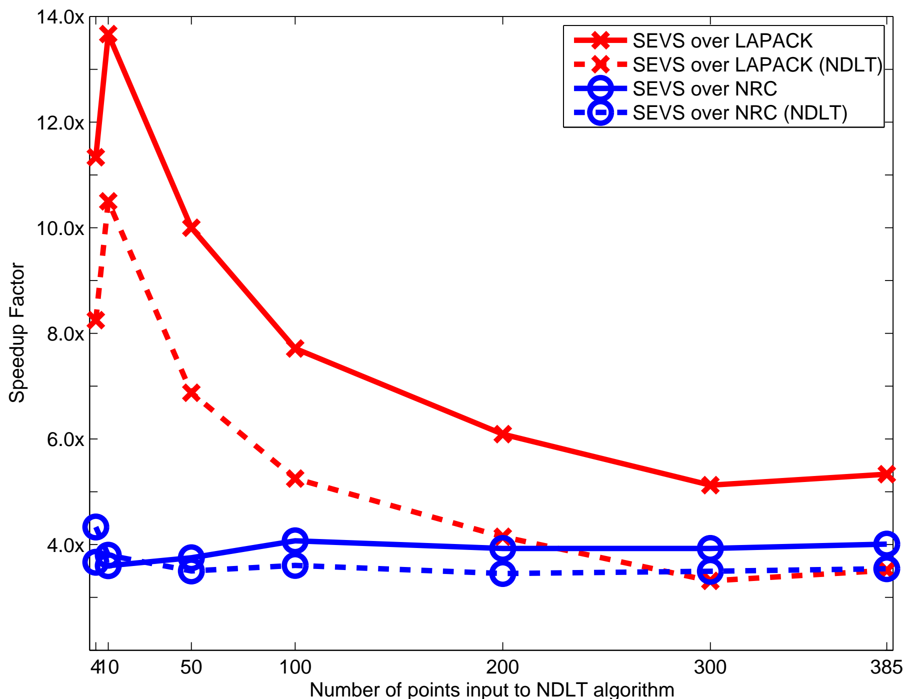


Figure 4.2 Speedup Provided By SEVS Over NRC and LAPACK for Homography Estimation This figure shows the speedup provided by SEVS over NRC and LAPACK for homography estimation using NDLT. Dashed lines indicate speedup for the full NDLT algorithm, and solid lines indicate speedup of singular vector estimation within NDLT. Note that unoptimized SEVS is roughly $4\times$ than NRC for all input sizes, and a partially optimized SEVS can be between $13\times$ and $5\times$ faster than hardware accelerated LAPACK.

NDLT roughly $8\times$ over hardware accelerated LAPACK. Our natural C code outperforms NRC by more than a factor of 4. Consider an example where, after 20 RANSAC loops, 300 inliers are generated and (based on results from Sec. 4.4.1) NDLT is applied on all 300 inliers to obtain the final result. Not considering other compute or memory overhead, our numbers show that utilizing SEVS can speedup this 2-stage homography estimation by a factor of $5\times$ over LAPACK.

4.4.2 Structure From Motion

In order to evaluate SEVS for rigid SFM, we utilized the Shark dataset [136] and a MATLAB implementation based on [112] and [134] of the factorization method for SFM. Instead of using MATLAB's SVD function, we process the data in turn with SEVS, LAPACK and NRC to compute the top 3 singular values and vectors. For SEVS we used Algorithm 1 and Eqn. 4.4 as described in Sections 4.3.2 and 4.3.4. For LAPACK we configured `gesvd` to

compute left and right singular vectors. The Shark dataset consists of 91 points tracked over 240 frames. We double the size of the data by simply wrapping the coordinates to simulate the 91 points being tracked over 480 frames. We feed different subsets of this 960×91 matrix as input to the factorization algorithm.

Accuracy

We evaluate the accuracy of factorization at the application level, by measuring the sum of squared differences between the computed feature coordinates and that of the provided groundtruth. The feature coordinates are compared in 2D image space after projecting the recovered 3D feature points using the recovered camera rotations and translations. We handle the reflection ambiguity by repeating the measurement after negating the sign of the Z coordinate and taking the smaller of the two SSD scores at each pose. For each input sequence length, we use the mean SSD values over the entire sequence to compare SEVS, LAPACK and NRC.

Holding the number of features constant $(n = 91)$ in our modified Shark dataset, we varied the sequence length between 120 and 480 in increments of 60 frames $(m = \{240, 360, \ldots 960\})$. The variation of mean SSD errors for SFM factorization across these input sizes using SEVS, LAPACK, and NRC are shown in Fig. 4.3(a). The plot shows that for all but one case SEVS produced errors less than or equal to NRC, and for two cases, gives identical results to LAPACK. Overall, these results indicate that SEVS is more accurate than NRC and comparable to LAPACK for 3D reconstruction across various input sizes.

Computational Efficiency

We created a subset of the Shark dataset consisting of every other feature point to form a dataset ("Half Shark") of size 960×46 entries. We measured the computation time for SFM factorization by SEVS, NRC, and LAPACK on different sequence lengths from the Shark and Half Shark datasets. As in the case for homography estimation (Sect. 4.4.1), we compute the ratios $\mathrm{LAPACK}/\mathrm{SEVS}$ and $\mathrm{NRC}/\mathrm{SEVS}$ to capture the speed-up factor of SEVS over LAPACK and NRC respectively.

In Fig. 4.3(b), we compare the natural C implementation of SEVS against NRC. We see that as the input matrix gets more skewed, $(m \gg n)$, SEVS offers greater returns over NRC. We see this effect in two ways in Fig. 4.3(b). First, for an SFM dataset of fixed number of points, the SEVS speed-up factor grows as the sequence length increases. Second, for

a given sequence length, the SEVS speed-up is more pronounced for datasets with fewer feature points.

Since the complexity of SEVS is linear in m compared to m^2 for NRC, we would expect SEVS to offer a more consistent speedup over NRC. However, due to inefficiencies in our C implementation, these gains become apparent only when $m \gg n$. In fact, we found that the matrix multiplications to compute step 1 of Algorithm 1 and evaluate Eqn. 4.4 account for more than 90% of time spent in SEVS. Next, we enable BLAS matrix multiplication for these steps in SEVS and compare compute efficiency against hardware-accelerated LAPACK.

The graph in Fig. 4.3(c) shows the speedup factor of SEVS over LAPACK. We see that SEVS is an impressive $6\times$ to $9\times$ faster than LAPACK for sequence of length between 120 and 480 frames. In this mode, we do not see any systematic differences between the Shark and Half Shark datasets. Indeed, in Fig. 4.3(c) we see a roughly linear speedup in m, which is expected since LAPACK grows with m^2 and SEVS with m,

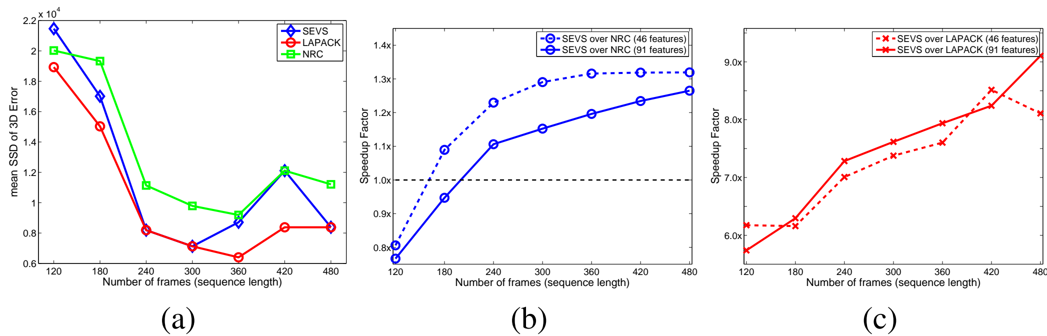


Figure 4.3 Accuracy and Efficiency of SEVS for SFM Factorization This figure shows the accuarcy and efficiency of SEVS for SFM factorization. (a) 3D reconstruction errors across different sequence lengths for SFM factorization using SEVS, LAPACK, and NRC. Note that SEVS is almost as accurate as LAPACK for 3D reconstruction using factorization. (b) Speedup provided by unoptimized SEVS over NRC for SFM factorization over different sequence lengths. Dashed lines are for 46 tracked features (Half Shark) and solid lines for 91 tracked features (Shark). Natural C implementation of SEVS offers performance gains as input matrix gets more skewed $m \gg n$. (c) Speedup provided by partially optimized SEVS over hardware accelerated LAPACK for SFM factorization over different sequence lengths. SEVS is significantly faster than LAPACK for SFM factorization, offering a linear growth in speedup factor with sequence length.

4.5 Discussion

The SEVS technique is a viable alternative to the traditional SVD methods. In this section, we discuss the specific computations in SEVS and compare against traditional SVD

methods.

In general, the construction of the Grammian matrix of the form $A^T A$ is avoided since the condition number of this product is bounded by the square of the original condition number. Thus, traditional SVD approaches compute a bi-diagonal form of A using Golub-Kahan bi-diagonlization [53]. We explored adopting this approach for defining $\mathbf{C}$ in Algorithm 1, but favored of the Grammian form for two reasons. One, in practice, the danger of ill-conditioned input matrices can be minimized via construction. For example, in homography estimation, we reject degenerate point configurations before passing them to the NDLT algorithm. Two, the cost of bi-diagonalization is at least 4 times that of computing $A^T A$. It should be noted that in the event ill conditioned matrices are possible, the bi-diagonalization can step can be introduced to deal with this at the cost of the extra computation. If we do not compute the bi-diagonalization and ill conditioned matrices are possible then a numerical stability test may be performed on the final linear system and resulting homography parameters by applying a small perturbation and determining the change in results. This can be done by setting a maximum amount of change allowed by the perturbation and if this is exceeded then the algorithm, can fall back on the bi-diagonalization method. With proper underflow and overflow checks, the release of a numerically unstable solution should be preventable.

Bi-diagonalization of the input offers another benefit in that the corresponding inverse iteration form can be solved in linear time, $O(n)$, as opposed to $O(n^2)$ for LU Decomposition. However, it would take a very large number of iterations for this difference to make up for the $4\times$ gap in computing $\mathbf{C}$. For the NDLT algorithm, SEVS typically took no more than 6 iterations to converge based on our termination criterion. For SFM, the numbers of iterations showed more variation across different inputs. We found the second and third eigenvectors took larger number of iterations, presumably due to less than ideal $\hat{\lambda}$ values.

4.6 Conclusion

Our results show that SEVS is both very accurate and extremely efficient for certain classes of vision problems, namely solving homogeneous linear equations and matrix factorization.

Overall, our experimental results validate the design choices in SEVS for the popular vision problems of homography estimation (Sec. 4.4.1) and SFM factorization (Sec. 4.4.2). For homography estimation, we showed that SEVS can be used to significantly speedup NDLT (Fig. 4.2) in two stages of the process, once within the RANSAC loop, and a second time, as a replacement to the iterative LM optimization. We showed that SEVS produced

lower symmetric transfer errors than LM (Fig. 4.1), addressing concerns about numerical stability issues during computation.

Application to SFM factorization showed that SEVS can be used to efficiently recover left and right singular vectors. Based on the 3D reconstruction error metric, SEVS produced results that were more accurate than NRC, and close to LAPACK in several cases (Fig. 4.3(a)). SEVS not only produced robust results over long sequences, it also yielded steadily increasing computational benefits across input sizes (Fig. 4.3(b),(c)).

We note that our SEVS implementation was not machine-optimized, and yet offered significant speedups across the board compared to Numerical Recipes in C and a hardware accelerated LAPACK library. We look forward to implementing SEVS with accelerated APIs such as MKL or ATLUS to fully exploit the benefit of this technique.

We have shown that for homography estimation SEVS is capable of running a speed up of over 12x more than common solutions such as LAPACK with a Intel's MKL BLAS. SEVS is as accurate as other techniques that use more complex computation. We have also shown how SEVS can be utilized as a replacement to the well known LM refinement and produce stable results. We present techniques that utilize the structure of the singular value computations to properly estimate the required eigenvalue starting points for the inverse iteration calculation. During our experiments we found no issues with numerical stability in our applications. Thus, SEVS is a technique that is poised to produce more efficient vision computation, particularly in mobile systems.

Optimizing the software is the first component of creating more efficient mobile vision systems. The following chapters focus on the use of architecture and hardware to improve vision system performance while balancing energy consumption. In particular, Chapter 5 develops hardware for efficient feature extraction while Chapter 6 expands the hardware to improve the entire computer vision software pipeline.

Chapter 5

The EFFEX Mobile Vision Feature Extraction Platform

> People who are more than casually interested in computers should have at least some idea of what the underlying hardware is like. Otherwise the programs they write will be pretty weird.
>
> *The Art of Computer Programming*
> Donald Knuth

5.1 Chapter Introduction

Chapters 2 and 3 presented first a specific application and then a benchmark suite for mobile vision. In Chapter 4 we presented how software and algorithm optimizations can improve the performance of vision codes on mobile systems. Our previous analyses exposed feature extraction as a costly component of many mobile vision systems. In this chapter, we introduce hardware solutions to improve the execution time and energy performance of this key mobile vision module.

Applications such as Google Goggles [55] and Layar [91] are starting to make use of computer vision algorithms on mobile platforms; however, the limited computational resources of mobile processors prohibits the use of many techniques found in higher performance machines. For example, the popular vision algorithm SIFT (Scale Invariant Feature Transform) [94] can take over 53s to process a 1024x768 image on an embedded processor (ARM A8) which is 34× longer than on a modern desktop machine (Intel Core 2 Quad).

A typical vision software pipeline, illustrated in Figure 3.2 of Chapter 3, takes an image or video and distills the data down to key relevant information components called *features*. The features are then processed, typically with machine learning algorithms, to gain semantic and geometric information about objects in the scene, while identifying the objects' type

and location. Objects can be observed over time to gain understanding of the context of the scene. Typically the process is iterative, once enough object and context understanding is gained, such information can be used to refine knowledge of the scene.

Features within an image are identified by the *feature extraction* algorithm, a principle component of the vision software pipeline. A capable feature extraction algorithm must distill important image information into scale, illumination, viewpoint, and rotation invariant signatures, called *feature descriptors*. Feature descriptors are vital to the algorithmic process of recognizing objects. For example to recognize a car, the feature extraction algorithm could enable the identification of the wheels, side-mirrors, and windshields. Given the relative location of these features in the image, a system could then recognize that the scene contains a car. The ”quality” of any particular algorithm lies in its ability to consistently identify important features, regardless of changes in illumination, scale, viewpoint, or rotation. In general, the more capable an algorithm is at ignoring these changes, the more computationally expensive it becomes. To demonstrate this tradeoff, the unsophisticated FAST Corner Detector [120] executes in 13 ms for a 1024x768 image on a desktop machine, but provides no robustness to changes in illumination, scale, or rotation. In contrast, the highly capable SIFT algorithm, which is illumination, scale, viewpoint and rotation invariant, processes the same image in 1920 ms, which is 147× slower.

One method to address the performance issues of feature extraction on mobile embedded platforms is to utilize cloud computing resources to perform vision computation. However, this approach requires much more wireless bandwidth compared to a system with a capable feature detector. For example, transmitting compressed SIFT features would require about 84 kB for a large number of features (over 1000) compared to 327 kB to send a compressed image. Since existing wireless mediums are already straining to carry existing data [146], there is significant value to communication mediums to perform feature extraction locally, even if cloud resources are used to analyze the feature data.

5.1.1 Contribution of This Chapter

In this chapter, we present EFFEX, an embedded heterogenous multicore processor specialized for fast and efficient feature extraction. The design is targeted for energy-constrained embedded environments, in particular mobile vision applications. It is an application-specific design, but fully programmable, making it applicable to a wide variety of feature extraction algorithms. To demonstrate the effectiveness of our design, we evaluate its performance for three popular feature extraction algorithms: FAST [120], Histogram of Gradients (HoG) [39], and SIFT [94]. Specifically, this work, which was published in [32], makes

three primary contributions:

- We perform a detailed analysis of the computational characteristics of three popular feature extraction algorithms: FAST, HoG, and SIFT. We identify key characteristics which motivate our application-specific hardware design.
- We develop a heterogenous multicore architecture specialized for fast efficient feature extraction. The architecture incorporates a number of novel functional units that accelerate common operations within feature extraction algorithms. We introduce memory enhancements that exploit two-dimensional spatial locality in feature extraction algorithms.
- Finally, we demonstrate the effectiveness of our design when running three different feature extraction algorithms. The EFFEX architecture is capable of high performance at low cost in terms of silicon area and power.

5.2 Related Work

Previous works have attempted to improve computer vision processing performance with hardware optimizations. For example, work by Wu attempted to use GPGPUs to increase the speed of computer vision algorithms [147]. While GPGPUs provide large speedups, their power usage, at 100s of watts, is too high for the embedded computing space. Furthermore, GPGPUs, along with embedded GPUs, lose valuable performance gains when there is control divergence due to shared instruction fetch units. Our technique does not suffer from control divergence performance degradation. There are also many computations, such as the summation portion of the inner product, that do not map well onto a GPGPU architecture.

Silpa describes a patch memory optimization for use in texture memory in mobile GPUs to increase performance [128]. They evaluate using bulk texture transfer mechanisms, and supported these operations with texture caching. Our technique takes the design to a lower level and looks at increased hardware support.

Kodata developed an FPGA design focused on HoG [82]. Their design showed excellent speedup, but their approach was solely targeted at HoG. Our approach utilizes an application-specific processor applicable to many computer vision problems, thus, we can provide performance benefits to a wider range of algorithms. Other hardware design efforts have also worked to speed up various aspects of computer vision algorithms, such as

Skribanowitz [130], Chang [26], and Qui [116], but similarly these efforts target a specific algorithm and do not offer the programmable benefits of EFFEX.

Prengler developed a SIMD engine targeted to vision algorithms with the capability to switch to MIMD processing [113]. Their technique reduces the large SIMD unit into many smaller SIMD units. While effective, it suffered from poor performance due to branch divergence, forcing expensive reconfigurations. In contrast, EFFEX does not require reconfiguration, and it benefits from special functional units along with memory optimizations.

The IBM Cell processor [62] is a heterogeneous architecture with some similarity to our proposed architecture. A major difference is that the Cell uses full vector processors and a conventional memory architecture. Our architecture focuses on specific vector reduction operations where vector data is reduced to smaller summary data, such as the inner product. This approach allows for smaller cores with specific vector instructions which take less area and power than a full vector processor. This also sets our approach apart from general vector processors. Furthermore, we use a memory system tuned specifically to vision algorithms.

5.3 Feature Extraction Algorithms

A typical feature extraction algorithm, as illustrated in Figure 5.1, is composed of five steps. The first step is to preprocess the image, an operation which typically serves to accentuate the intensity discontinuities (i.e., object boundaries) by, for example, eliminating the DC components (mean values) of the image. The second step scans the processed image for potential feature point locations; the specifics of this phase are highly dependent on the underlying algorithm. The third step of feature extraction works to filter out weak or poorly represented features through, for example, sorting the features found based on a key characteristic and then dropping the non-prominent results. The second and third steps implement a process typically called *feature point localization*. Once feature points are localized, the fourth step computes the feature descriptor. A *feature descriptor* is a compact representation of an image feature that encodes key algorithm-specific image characteristics, such as variations of pixel intensity values (gradients). The feature descriptor implements the illumination, scale, and rotation invariance supported by a particular algorithm. The fifth and final step of feature extraction performs another filter pass on the processed feature descriptors based on location constraints.

The quality of a feature extraction algorithm is evaluated on four major invariance characteristics: illumination, scale, viewpoint, and rotation. A very capable feature extraction

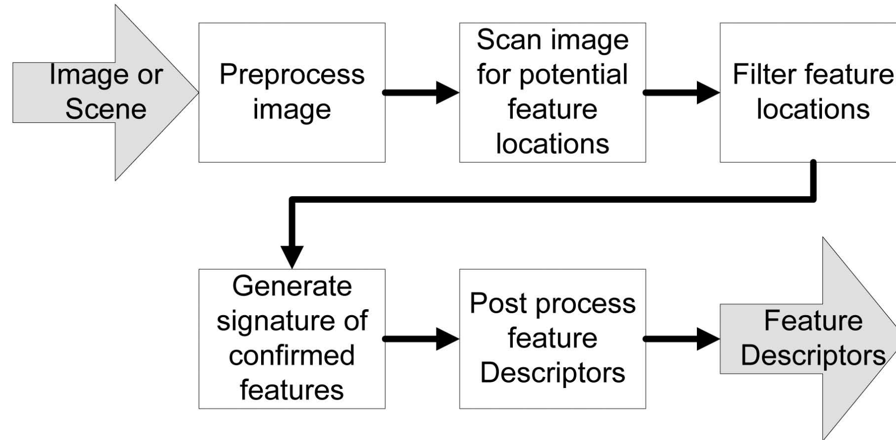


Figure 5.1 Overview of Feature Extraction This figure shows the general steps involved in feature extraction. The first three steps of the process locate feature points. The final two steps create feature vectors.

algorithm will produce feature sets for an image that are nearly identical, despite changes in lighting, object position or camera position. In this work we focus on FAST [120], HoG [39], and SIFT [94]. These algorithms represent a wide trade-off of quality and performance, ranging from the high-speed low-quality FAST algorithm to the very high-quality and expensive SIFT algorithm. In addition, these algorithms are widely representative of the type of operations that are typically found in feature extraction algorithms.

5.3.1 Algorithm Performance Analysis

We analyzed the execution of the FAST, HoG and SIFT feature extraction algorithms to determine how their execution might benefit from hardware support. We instrumented a single-threaded version of each algorithm and profiled their execution.

FAST corner detection is designed to quickly locate corners in an image for position tracking [120]. It is the least computationally intensive and the least robust of the algorithms we examine. The FAST feature matching degrades when the scene is subject to changes in illumination, object position or camera location and image noise. As seen in Figure 5.2, the majority of time in the FAST algorithm is spent performing feature point localization. The algorithm locates corners by comparing a single pixel to the 16 pixels around it. To perform this comparison, the target pixel and surrounding pixels must be fetched from memory, and then the target pixel must be compared to all the pixels in the enclosing circle. The descriptor is made by concatenating the pixel intensities of the 16 surrounding pixels. Speeding up these comparisons, through a combination of functional unit and thread-level parallelism, greatly improves the performance of FAST.

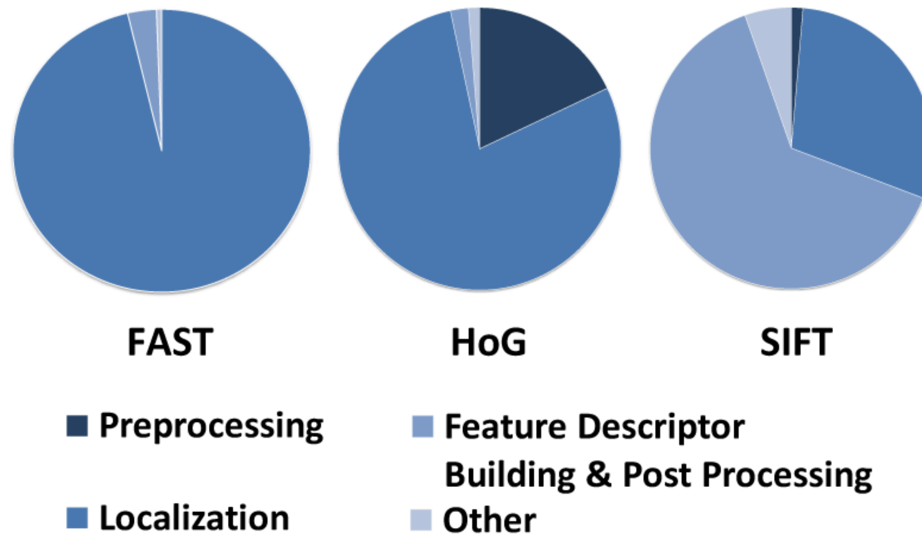


Figure 5.2 Feature Extraction Execution Time Distribution FAST spends most of the time locating the features using a comparison of a center pixel value to other pixels that form a circle around it. HoG spends most of the time localizing which involves building the descriptor for comparison. SIFT spends a large amount of time building the descriptor which involves gradient computations and binning.

HoG is commonly used for human or object detection [39]. The HoG algorithm is more computationally intensive than FAST, because it provides some illumination and rotation invariance. Figure 5.2 shows that HoG spends a significant amount of time performing feature localization and descriptor building. The descriptor is built using the histogram of the gradients of pixel intensities within a region, which are subsequently normalized. The major operations for this phase of HoG computation are the fetching of image data from memory and the calculation of histograms using integral images [140]. This phase of the HoG algorithm utilizes a sliding window of computation in which each window is independent. Consequently, much parallelism is available to exploit. The second major time component is preprocessing, which for HoG is computation of the integral image. This is comprised mainly of memory operations, the computation of the image gradient, and finally the histogram binning of gradient values based on direction. More efficient computation of these components, through functional unit support and thread-level parallelism, significantly speeds up processing.

SIFT is a feature extraction algorithm widely used for object recognition [94]. It is the most computationally expensive and algorithmically complex of the feature extraction algorithms we examine, but it provides a high level of invariance to most scene changes. Figure 5.2 shows that the largest component of time is spent in feature descriptor building. This portion of the algorithm involves computing and binning the gradient directions in

a region around the feature point, normalizing the feature descriptor, and accessing pixel memory. The operations in this phase are performed on each feature point and benefit from specialized hardware. The second largest component of SIFT is the feature point localization. This portion is dominated by compare and memory operations to locate the feature points. There are also 3D curve fitting and gradient operations to provide sub-pixel accuracy and filter weaker responses, respectively. This phase of SIFT provides ample thread-level parallelism. The preprocessing step, the third most expensive component in SIFT, involves iterative blurring of the image which is a convolution operation. The convolution requires multiplying a region of the image by coefficients and summing the result, operations which can benefit from specialized functional unit support.

5.4 Proposed Architecture

To meet the needs of feature extraction in the embedded space, a number of design criteria exist. First, the design must run feature extraction algorithms efficiently, as close to real-time analysis of high-resolution images as possible. To this end, we seek to exploit the ample explicit parallelism in these algorithms, plus employ the latency reduction capabilities of specialized functional units and memory architectures. Second, efficient execution of feature extraction algorithms must be possible at low power and area cost, as the mobile space predominantly relies on low-cost untethered platforms using batteries. To meet these challenging criteria, we employ many computational resources with a simple energy-frugal design, to lower overall power demands. Finally, the design must be able to run a variety of feature extraction algorithms to accommodate the fast-moving pace of feature extraction algorithm development. To serve this demand, we employ an application-specific processor design, aimed at providing efficient execution of feature extraction algorithms across a wide range of applications.

5.4.1 Support for Heterogenous Parallelism

All of the feature extraction algorithms that we studied demonstrated a dichotomy of internal algorithmic styles. At a high level, the algorithms are quite complex, with significant control-oriented code that displayed many irregular branching patterns, but with a large amount of the workhorse computation that exhibited repetitive patterns for feature point localization. Given these contrasting code styles, we employ a heterogeneous multicore architecture. The EFFEX high-level architecture is illustrated in Figure 5.3. Our architecture

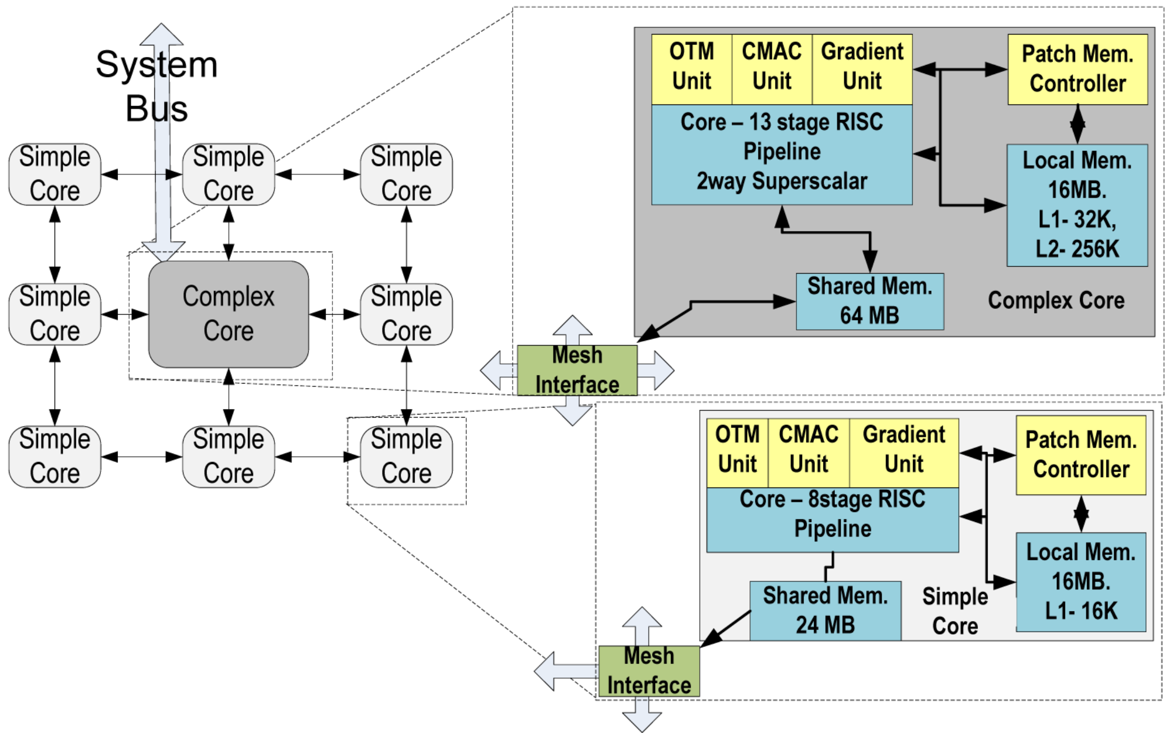


Figure 5.3 EFFEX Architecture This shows the overall EFFEX embedded architecture. There is one complex core surrounded by multiple simple cores on a mesh network. Each core has application-specific functional units and a patch memory architecture.

has a single complex superscalar core, coupled with a variable number of simple single-issue cores. The complex core is used to perform high-level tasks that require complex decision making while the simple cores plow through the vast amount of simple repetitive tasks that often require unpredictable control flow using specialized vector instruction units. Since the simple cores require only a fraction of the area of the complex core, this heterogeneous multicore design approach yields significant performance per unit area for feature extraction algorithms.

The image under analysis is shared among all of the processing units, and each simple core must have local memory to store intermediate results. For our design we use an uncached distributed shared memory in a NUMA mesh [38]. The feature extraction algorithms have a large amount of parallelism and little communication demands between the worker threads: the shared memory is primarily used to hold the image under analysis, plus work requests sent by the complex core to the simple core. Additionally, the simple cores return computed feature descriptors to the complex core through the shared memory. As such, we rely on explicit inter-processor communication through shared memory. Synchronization between the cores is implemented with barriers that reside in the shared memory.

5.4.2 Functional Units for Feature Extraction

Our analysis of the three feature extraction algorithms revealed three application-specific functional units that could be used to accelerate their execution. These functional units are useful across a broad array of feature extraction algorithms. In general, the functional units provide vector reduction instructions for the cores. Every core has one of the following functional units.

One-to-many Compare Unit Searching for the feature point locations typically involves a large number of compares between a pixel location and its neighbors. For example, SIFT compares a image pyramid pixel value to its immediate neighbors' values while FAST compares a center pixel value to surrounding pixels. The one-to-many compare (OTM) unit, illustrated in Figure 5.4, significantly speeds up this processing by performing the comparisons in parallel through the use of an instruction extension. The one-to-many compare unit takes a primary operand and compares it against 16 other values in a single operation. The unit returns the total number of values that are less than (or greater than) the instruction operand along with a basic result that is one if all compares are true, zero otherwise. The unit can also be used for histogram binning by comparing a value to the limits of the histogram bins. This makes the output equal to the number of the histogram bin. The one-to-many compare unit is loaded with image data using a single bulk load operation from patch memory (see Section 5.4.3), which takes approximately 9 cycles. The comparisons complete in 32 additional cycles. All together, a one-to-many comparison is 3.5 times faster with the specialized functional unit support, compared to executing the necessary operations on an typical ARM A8 core running at the same clock speed.

Convolution MAC The second specialized functional unit is a convolution multiply accumulate (CMAC) unit. The CMAC unit takes two floating vectors and performs an inner product operation on them, as shown in Figure 5.5. This operation is performed in SIFT and HoG for image preprocessing and for vector normalization. The vectors are loaded using a bulk load from patch memory which takes 36 cycles for two 32-entry vectors. Once the vectors are loaded the CMAC unit can complete a convolution step in 6 cycles. This is up to 96 times faster than an ARM A8 running at the same clock speed as EFFEX.

Gradient Unit Both SIFT and HoG have many image gradient computations while building the feature descriptors. We have included the gradient functional unit, illustrated in Figure 5.6, to speed up this computation. This functional unit computes the gradient of an image patch using a Prewitt version of the Sobel convolution kernel [20]. The unit operates on single-precision floating point data, and it is able to compute the gradient in both the x and y directions at the same time. The gradient unit uses patch memory to quickly load pixel operands from memory based on a pixel address operand for the gradient instruction. The

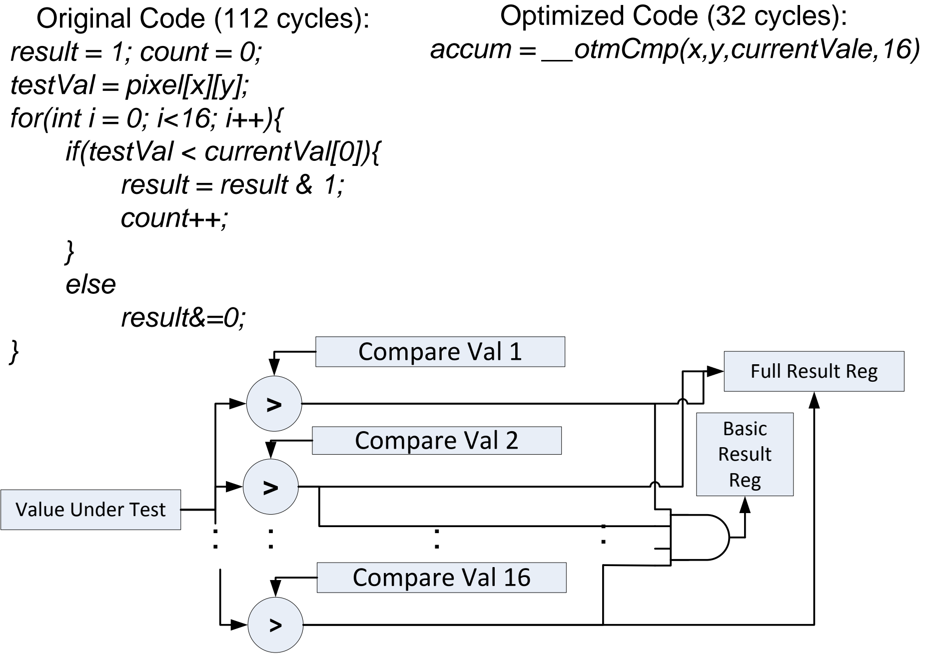


Figure 5.4 One-to-Many Compare and Binning Unit This functional unit is used frequently in SIFT and FAST. It can be used for comparing one pixel to many others all at once. It has fast access to patch memory, allowing multiple values to be retrieved and compared quickly. It can also be used for efficiently computing histogram bins.

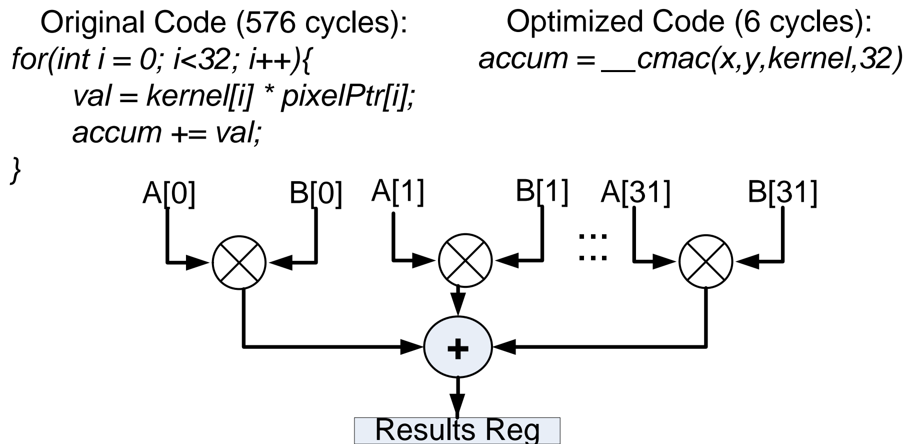


Figure 5.5 Convolution MAC Unit This unit is a MAC that takes vectors from patch memory and implements fast convolutions. This unit is used in SIFT for preprocessing and HoG for vector normalization.

gradient unit is able to perform both the x and y gradient operation for a $3x3$ pixel patch in 33 cycles which is a speedup of 3.57 over an execution on an ARM A8 processor running at the same clock speed.

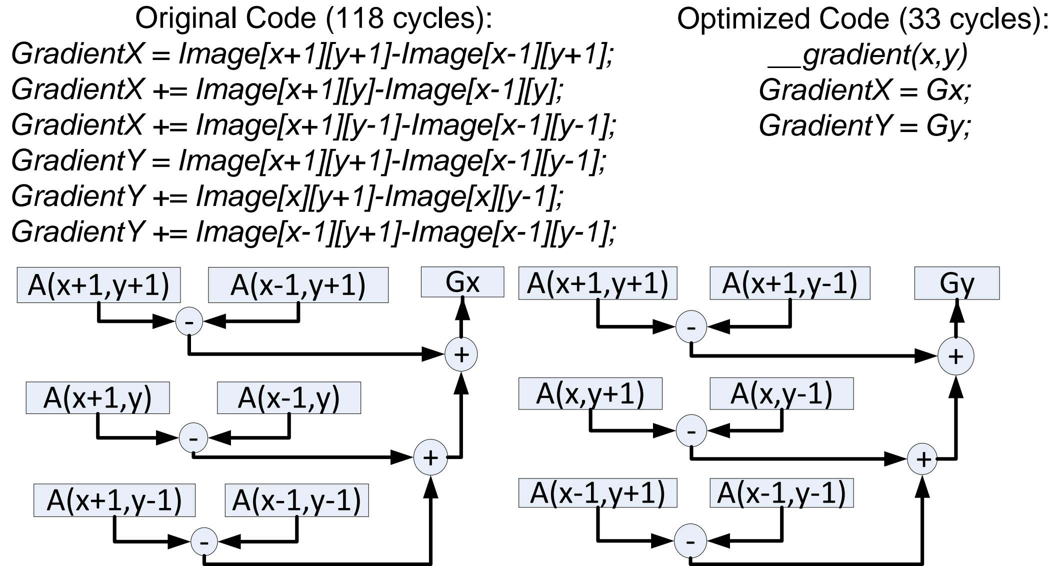


Figure 5.6 Gradient Unit This unit uses access to patch memory to quickly compute the gradient at a pixel location. This unit computes the x and y gradient at the same time. This unit is used frequently in SIFT and HoG. It is a small SIMD engine interfaced to patch memory architecture.

5.4.3 Patch Memory Architecture for Fast Pixel Access

Feature extraction algorithms generally inspect 2D regions of an image, leading to high spatial locality in accesses. However, this spatial locality is relative to the 2D space of pixel locations. The traditional approach of a scan-line ordered layout of pixels in memory, as illustrated in the left image of Figure 5.7, is an inefficient way to store pixel data. Access to pixel data in scan line order results in image data residing in multiple DRAM rows which leads to long latencies and wasted energy.

The *patch memory architecture* is a memory system design that combines hardware and software optimizations to significantly speed up access to pixel data. Software is optimized to store pixel data in region-order. In our design regions are defined as small two-dimensional patches of pixels that are n pixels wide by m pixels high which fit evenly within a single DRAM row, resulting in the memory layout shown in the right image of Figure 5.7. Any access to a single memory patch typically only requires reading one DRAM row, and subsequent reads of nearby patches can often be serviced out of the DRAM row buffer. When the region accessed spans the boundaries of the region-ordered memory, multiple DRAM rows may need to be accessed. As such, multiple DRAM row buffers (four in our design) are desired to ensure that data is quickly available.

Image data access requires translation of the pixel location to a memory address, an operation typically performed in software. Calculating addresses in patch memory requires a similar amount of computation. Given the high frequency and irregularity of pixel accesses

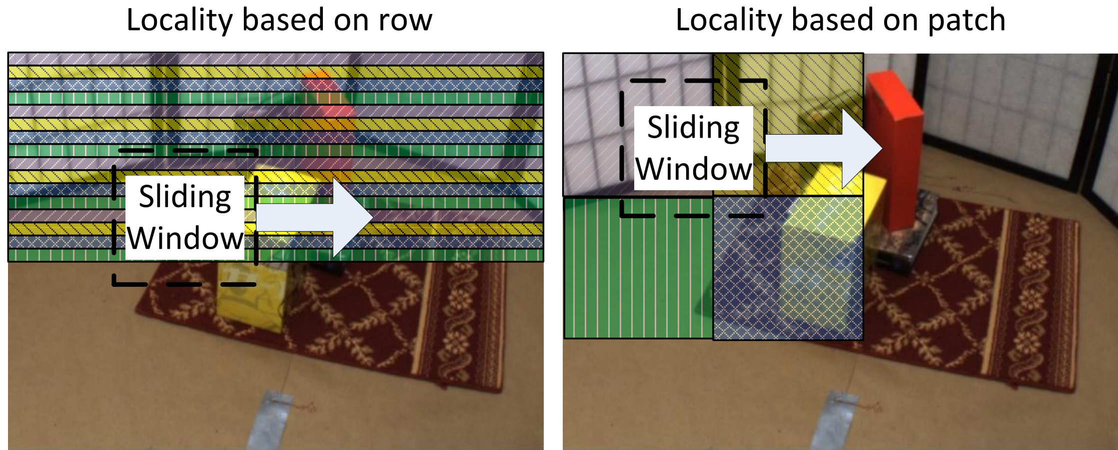


Figure 5.7 Patch Memory Access Pattern This figure illustrates the storage of image data in traditional DRAM (left) and our patch based DRAM (right). Each color/pattern is a different DRAM row containing data. In traditional DRAM storage, a single sliding window will access a large number of different DRAM rows, while in patch-based memory the window only accesses at most four DRAM rows for a reasonably sized window.

in feature extraction, we provide special patch memory instructions and hardware support to convert an integer (x, y) pixel address quickly into its corresponding patch memory address, as illustrated in Figure 5.8. The specialized address generation unit checks if subsequent accesses are in the same memory patch, and in this case omits an unnecessary recomputation of the pixel address.

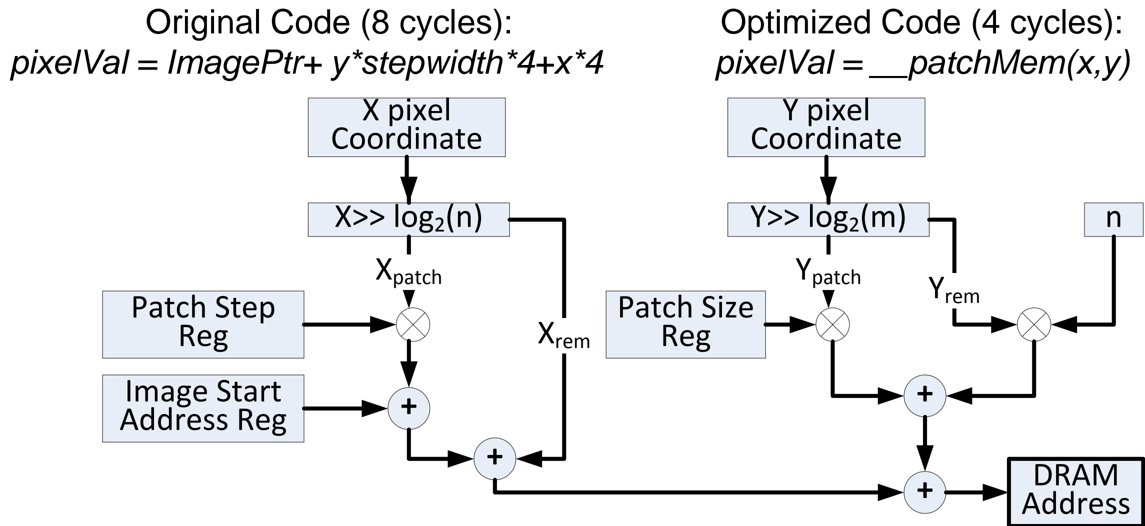


Figure 5.8 Patch Memory Address Computation This figure shows the logic required to compute the patch memory DRAM address based on the pixel x and y coordinates. Using this address computation method the memory can automatically place the data in the optimal access pattern for many of the feature extraction algorithms.

5.5 Performance Analysis

5.5.1 EFFEX Performance Model

We simulated our system using a technique similar to Graphite [103]. We modeled the EFFEX architecture using an ARM A8 like custom model for the complex cores, and an ARM A5 (with floating point) model for the simple cores, all running at 1 GHz. Each simulated processor contained a CMAC, OTM compare unit, and Gradient Unit along with a patch memory controller. The memory system was modeled such that each core had a local cached private memory and uncached access to the main shared memory. The interconnect between the cores was modeled as a mesh network. Table 5.1 lists the basic configuration parameters.

We modeled shared and local memory using a custom memory simulator with support for 16 pixel by 16 pixel patch memory in the local memories, and a traditional row-oriented DRAM architecture for the main shared memory. Power estimates for the memory system were implemented using activity counters, similar to the approach in Wattch [24]. To gauge power and area estimates for the application-specific functional units, they were synthesized from Verilog using Synopsys synthesis tools for a 65 nm process node. The CMAC unit is based on the work in [79]. The area overhead of the EFFEX cores can be seen in Table 5.2. The base complex and simple core areas and power were based on [10] and [9], respectively. For comparison, we modeled an embedded GPGPU similar to the PowerVR SGX family [95] with an 8-thread embedded GPGPU (2 sets of 4 threads) running at 110 MHz.

5.5.2 Benchmarks

The computer vision algorithms analyzed were the FAST, HoG, and SIFT feature extraction algorithms. The base implementations for SIFT, HoG and FAST are from [69], [59], and [121], respectively. We modified the algorithms to expose explicit parallelism for our heterogeneous multicore. For test inputs, we randomly selected fifteen 1024x768 images as a data set. The algorithms were compiled for Linux using the GNU GCC compiler set to maximum optimization. All GPGPUs used the CUDA SIFT implementation from [147].

The algorithms were transformed into multithreaded versions by splitting the algorithms into phases at logical serialization points. Within a phase the image processing was split into regions, and each thread performed computation on a separate region. Each thread was assigned to a different core with synchronization performed using barriers in shared memory.

Table 5.1 EFFEX Configuration

Feature	Configuration
Core Clocks:	1 GHz
Complex Core:	32 bit RISC in-order 2-way superscalar
Complex Pipeline:	13-Stage
Complex Local Cache:	32k instr. and data 256k unified L2
Simple Core:	32 bit RISC in-order
Simple Pipeline:	8-Stage, single issue
Simple Local Cache:	16k instr. and data
Local Memory Clock:	1GHz
Local Memory Size:	16MB
Shared Memory Clock:	500MHz
Complex Core Shared Mem. Size:	64MB
Simple Core Shared Mem. Size:	24MB
Total Shared Memory Size:	256MB
Memory Bus Width:	128 bits
Processor Interconnect:	Mesh

Table 5.2 Area estimates for the EFFEX 9-core processor. These estimates assume a 65 nm silicon process.

Module	Area (mm^2)
One-to-many Compare	0.0023
Gradient Unit	0.0034
Convolution MAC	0.5400
Total for functional units per core	0.5457
Complex Core	4.0000
Simple Core	0.8600
Total for 9 Core EFFEX	15.8000
SIMD Complex core w/o EFFEX	6.0000
Normal Complex core w/Embedded GPGPU	16.5000

We simulated all the algorithms running on: i) a single complex core without EFFEX functional units, ii) a complex core with EFFEX functional units, and iii) an EFFEX with a complex core and a variable number of simple cores. We also ran the SIFT algorithm on i) a 65 nm 1.2 GHz NVIDIA GTX260, ii) a 45 nm 2.67 GHz Intel Core 2 Quad Q8300 with 8 GB of RAM and iii) a simulated 110 MHz embedded GPGPU. We simulated the embedded GPGPU with the aid of an NVIDIA GT210.

5.5.3 Simulation Results

Figure 5.9 shows the speedup that EFFEX achieves over a single complex core lacking SIMD and the EFFEX enhancements. The plot shows that EFFEX is capable of achieving a significant speedup over a basic embedded processor. It also shows that for SIFT and FAST, adding 4-way 32 bit floating point SIMD instructions to EFFEX has negligible impact on the speedup. The speedup in HoG increases more with SIMD because SIMD instructions speed up a portion of the algorithm that the EFFEX enhancements do not. This shows that EFFEX can extract enough parallelism that there is typically not much need for SIMD. In the remaining experiments EFFEX is configured without SIMD support.

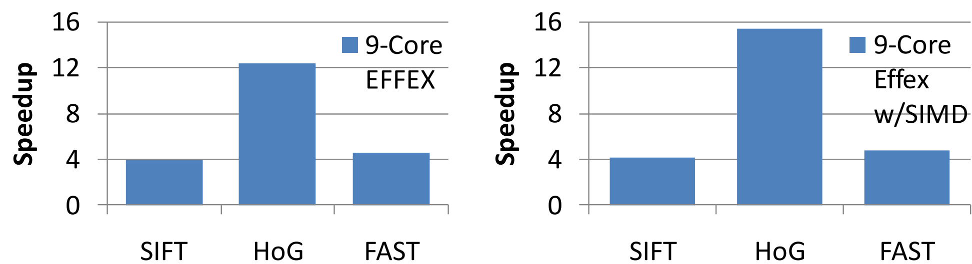


Figure 5.9 9-Core EFFEX Speedup The left graph shows the speedup of a 9-Core EFFEX configuration without general SIMD support for three algorithms versus a single complex core without EFFEX enhancements. The graph on the right shows the same comparison but for a 9-Core EFFEX with general SIMD instruction support as well.

Figure 5.10 demonstrates that as the number of simple cores increases, so does the performance of EFFEX. It can be seen that at around 4 total cores the EFFEX solution begins to outperform the embedded GPGPU. This is due primarily to the efficiency of the vector reduction operations that EFFEX performs, which run much less efficiently on the embedded GPGPU.

The 9-Core EFFEX solution has slightly less area than a typical embedded core combined with an embedded GPGPU; however, Figure 5.10 confirms that the EFFEX solution has higher performance. EFFEX maintains higher performance due to the ability that each core can take a divergent control path without hindering the performance of another core and due to the tighter inter-processor integration afforded by EFFEX.

The performance-cost benefits of EFFEX versus other computing solutions can be seen in Figure 5.11. This plot shows the performance of the various designs (in frames processed per second) per unit of cost (either silicon area or power). Clearly the EFFEX solution is capable of providing a higher performance per unit cost than other solutions, making the EFFEX design particularly attractive for cost sensitive embedded targets. This result is due primarily to the efficiency of the vector reduction instructions and the patch based memory.

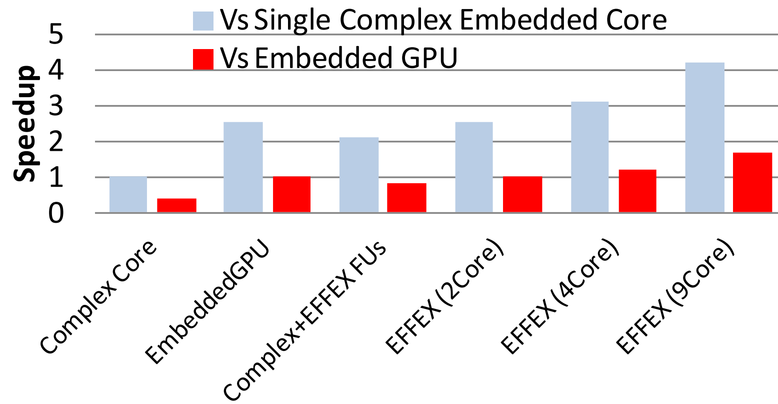


Figure 5.10 Scalability for SIFT Running on EFFEX This graph shows the scalability of the EFFEX cores running SIFT when compared to a complex embedded core (without EFFEX enhancements) and an embedded GPGPU.

We found that the patch memory architecture allows for a decrease in the total number of cycles while also decreasing the total energy. For example for a run of FAST the number of memory cycles decreased from 39 million to 937 thousand while the energy went from 6 J to 12 mJ. This provides a significant performance boost at very low cost.

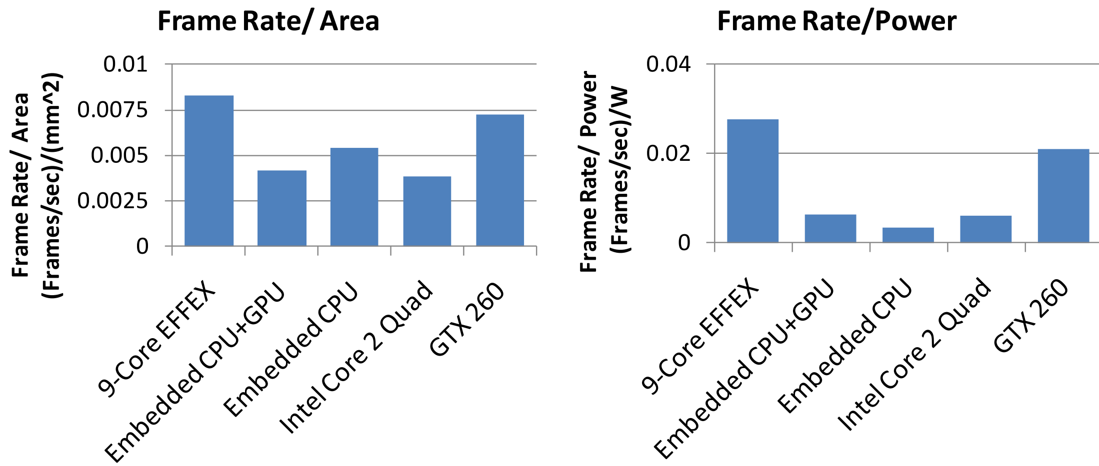


Figure 5.11 EFFEX Normalized Performance Running SIFT This graph shows the normalized performance of EFFEX when compared to other computing solutions. EFFEX outperforms embedded single cores, embedded GPGPUs, desktop CPUs, and desktop GPUs when the performance is normalized by cost. The performance here is measured by the number of 1024x768 frames processed per second.

Figure 5.12 shows a Pareto chart comparison of various computing solutions using the metrics of execution time per frame and overall cost ($area * power$). In the Pareto chart, better designs are in the lower left of the chart, as these designs have faster frame rates

and lower overall cost. $Area * power$ is used for the x axis because it captures two key cost factors into a single metric. The figure shows that while the embedded solutions are not as fast as the desktop GPU and CPU, they are far less expensive in terms of cost. Furthermore, a 9-core EFFEX is cost effective and has the fastest frame rate for the embedded solutions.

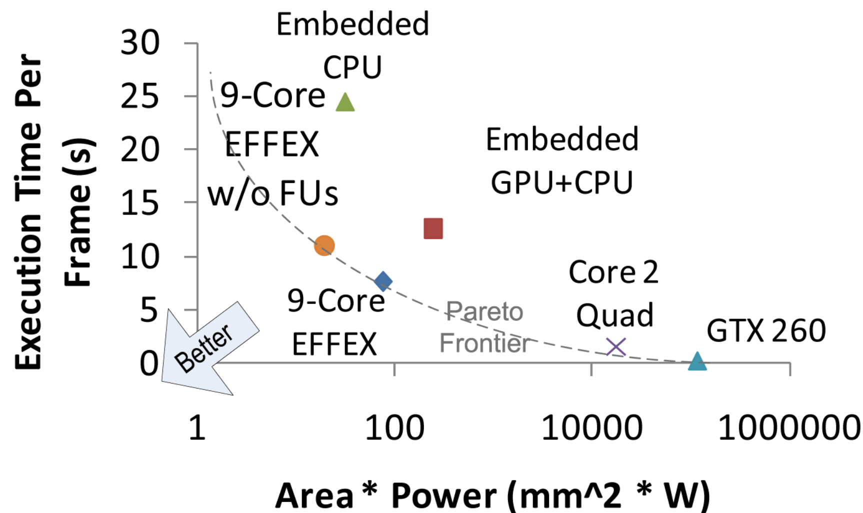


Figure 5.12 Performance versus Overall Cost Running SIFT This graph shows the performance and area and power cost for various computing solutions. The cost is measured in $area * power$ on the x axis while execution time is along the y axis. In this study the desktop CPU and GPU are faster but their power and area make them unattractive for the embedded space. In the embedded space, the 9-core EFFEX has the highest frame rate, making it a very a cost-effective solution.

5.6 Chapter Conclusion

In this chapter, we have analyzed the execution characteristics of three common feature extraction algorithms: FAST, HoG, and SIFT. Using this analysis, we presented EFFEX, an efficient heterogeneous multicore architecture that utilizes specialized functional units and an optimized memory architecture to significantly speedup feature extraction algorithms on embedded mobile platforms. We presented three specialized functional units in our design, including a one-to-many compare, convolution and gradient functional units that can take advantage of our patch memory architecture. Our application-specific heterogenous multicore is capable of improving the performance of embedded feature extraction by over 14x in some cases. We have shown that our architecture is more cost effective than a wide range of alternative design solutions, and it effectively executes a wide variety of feature extraction algorithms.

This chapter explorers the use of a multicore configuration to achieve efficiency. How-

ever, the organization of the cores utilized may not be optimal for the constraints of a given mobile system. This system utilizes only a small number of configurations which limits locating the best configuration in general. There is a need to examine a wider variety of configurations with various constraints to find the optimal configurations. Chapter 6 contains these evaluations.

While feature extraction is a key component of many vision applications, it is not the only component. Furthermore, feature extraction techniques, such as FAST, are not very computationally intensive. In applications that utilize this feature extraction algorithm, the majority of the computation will be in other phases of the vision software pipeline. The insights utilized to develop EFFEX can be extended and utilized to improve the capability of the rest of the vision pipeline. Amdahl's law indicates that we have a limit to the performance improvement possible by optimizing a single component of the vision pipeline [4]. For example, if MVSS was to use a feature extraction algorithm that was 5x faster, then the execution time of the 3D reconstruction becomes a key area for performance improvement. Chapter 6 looks at hardware to improve performance for the entire mobile vision software pipeline.

Chapter 6
The EVA Mobile Vision Platform

Life is like riding a bicycle. To keep your balance, you must keep moving.

Albert Einstein

6.1 Introduction

The previous chapter developed an architecture that is designed for the feature extraction phase of the mobile vision software pipeline based on the analysis of Chapter 3. Chapter 3 also showed that other operations besides feature extraction can benefit from vector reduction operations, heterogeneous parallelism, and 2D spatial locality. To achieve good overall mobile vision performance, the system must focus on all non-trivial aspects of the mobile vision software pipeline. Amdahl's law [4], which states that overall performance is set by the slowest component, teaches us that any other design approach would likely end in failure. Thus, in order to achieve optimal performance, the hardware must perform well on all tasks required by mobile vision systems. In this chapter we look beyond feature extraction to further increase the performance capability of mobile systems.

To meet the limited computation capability of today's mobile processors, computer vision application developer reluctantly sacrifice image resolution, computational precision or application capabilities for lower quality versions of the algorithms. For example, HDR takes seconds to capture an image thus limiting usage of the feature. There is an unsatiable demand for high-performance vision computing in the mobile space to improve the user experience. New applications are being developed such as MVSS, Chapter 2, that require a large increase in processing capability to achieve real-time performance while dealing with the energy constraints imposed by mobile systems. These applications utilize many kernels or modules from the mobile vision space that require efficient computation. To meet the performance, cost and energy demands of mobile vision, we propose EVA, a novel

architecture that utilizes an application-specific design, tailored to the specific application characteristics of mobile vision software.

6.1.1 Contribution of This Chapter

We present EVA, our solution for efficient vision processing in the mobile domain. EVA is a heterogeneous multicore architecture with custom functional units designed to increase processing performance for a wide range of vision applications. Our design leverages a heterogenous multicore architecture, where more powerful cores coordinate the tasks of less powerful, but more energy efficient cores. Both types of cores are enhanced with custom functional units specially designed to increase the performance and energy efficiency of most vision algorithms. Furthermore, EVA develops a novel, flexible, memory organization which enables efficient access to the multidimensional image data utilized in many vision workloads. We also examine the thread-level parallelism available in vision workloads and examine the tradeoff between number of cores, energy and speedup. This work makes three primary contributions:

- We present the EVA application-specific architecture optimized to improve the efficiency of mobile vision computation with unique application-specific accelerators and a flexible memory system capable of increasing the performance of image data accesses.
- Using the MEVBench [33] mobile vision benchmark suite and full-system simulation, we show that the EVA specialized design provides significant energy and performance improvements while being held to the tight cost constraints of mobile systems.
- Finally, we examine the amount of thread-level parallelism available in vision workloads and present the trade off in performance for power and energy in a multicore system.

6.2 Background Review On Mobile Vision Applications

The field of computer vision is a synergy between image processing, artificial intelligence, and machine learning. Computer vision algorithms analyze, process and understand the objects found in image data. For example, Figure 6.1 shows a picture where a computer

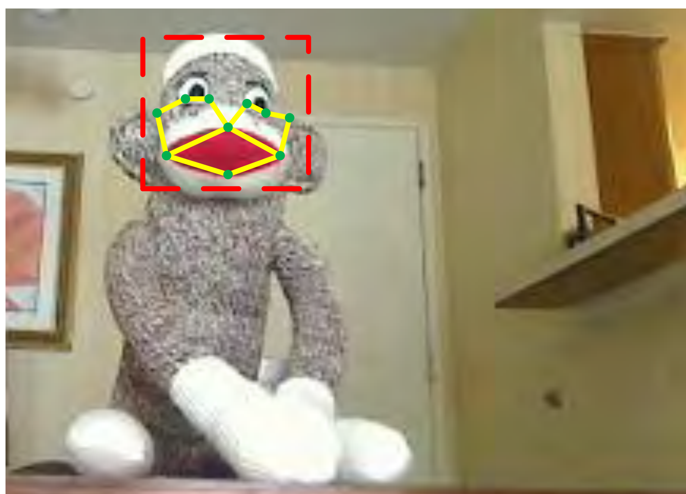


Figure 6.1 Computer Vision Example The figure shows a sock monkey sitting on a table. A computer vision application might be designed to recognize particulars in this image such as the face of the monkey. The algorithm would look for features such as corners, and use their geometric relationship to locate the monkey's face.

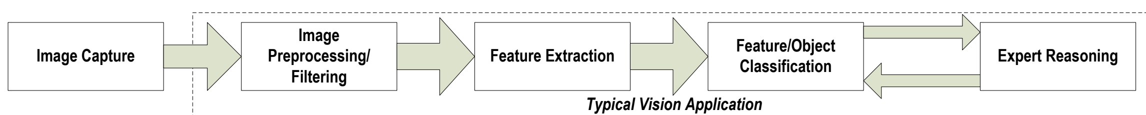


Figure 6.2 Vision Software Pipeline with Image Capture The figure shows a typical computer vision software pipeline with the image capture component. The image is captured using an imaging device such as a camera. The captured image is filtered to eliminate noise. Then, features are extracted from an image. The features are "classified" based on prior knowledge. Expert reasoning is utilized to generate knowledge about the scene.

vision application utilized a feature detection algorithm to locate the monkey's face. There is a wide variety of applications that can benefit from computer vision, such as defense, surveillance, and autonomous vehicles.

A typical computer vision software pipeline can be seen in Figure 6.2. This pipeline consists of five components: image capture, image preprocessing, feature extraction, feature/object classification, and expert reasoning. A challenge in optimizing for the vision pipeline's computation lies in the great variety of computational characteristics within the key kernels.

The first phase (*image capture*) involves capturing the imaging information from a sensor. In mobile vision applications the processor typically retrieves an image from the camera. This step is commonly an API call to the device driver although more capabilities are being exposed to user applications through software libraries such as FCAM [2]. The second phase (*image filtering*) involves applying filtering techniques to the imaging data to increase the discernibility of information in the data. Commonly, this phase is merged

with the third phase (*feature extraction*). Feature extraction consists of the localization of salient image characteristics, such as corners or brightness contrast, and the generation of unique signatures for the located features. These signatures are commonly stored as one dimensional vectors referred to as feature vectors. Feature extraction has been shown to be highly compute intensive [33].

The fourth phase (*classification*) utilizes machine learning algorithms to determine what objects could be represented by the features located in the image. The results of classification are used for semantic and relational processing to better understand the components of the scene. For example, a feature may be classified as belonging to an known object. The final phase (*expert reasoning*) utilizes the information from the previous phases to generate specific knowledge about the original scene. This task may be as simple as recognizing a face in the scene, or as complex as a predicting where a group of people are headed. The expert reasoning phase can iterate with the classification component to refine its outcomes, and this process is ultimately responsible for the output of the vision system. This iterative loop allows the developer to improve the quality of the result given sufficient computation capability.

6.3 Application Traits

To build optimized computing platforms, it is imperative to fully understand the underlying algorithms and the traits of the their computation. Through analysis of the MEVBench mobile vision benchmarks [33], we present three underlying characteristics that can be exploited in hardware to improve the performance of a mobile vision codes. These characteristics are: vector reduction operations, diverse parallel workloads, and memory locality that exists in both one and two dimensions.

6.3.1 Vector Reduction Operations

Vector operations have been a common topic for computer architecture for decades. From vector processors to digital signal processors (DSP) to General Purpose Graphics Processing Units (GPGPUs), there is a lot of work on the processing of vector operations. This work has primarily focused on operation that perform the same operation on a set of data or single instruction multiple data (SIMD) operations. While these operations are quite common within many applications, there is another class for vector operations that is not often considered.

During our investigation of mobile vision applications, we found that a common operation that takes place is a vector reduction operation. While vector operations and vector reduction operations both take vectors as inputs, the former produces a vector result while the latter produces a scalar result. Examples of two common vector reduction operations in vision codes are the dot product and max operations. They are used in preprocessing, image filtering, feature extraction, classification, application reasoning and result filtering. Despite the prevalence of vector reduction operations, most architectures primarily support the classical vector operations. Figure 6.3 shows the frequency of the dot product operations in the benchmarks. The figure also shows the size of the vectors these operations are operating on and the number of floating point multiply-accumulates that result. We examined the run time of the benchmarks to find hot spots and instrumented the calls to these types of operations. We found dot products, monopoly compares, tree compares and max compares to be the most common.

The strict energy and processing performance constraints of mobile systems creates the need to optimize vector reduction operations in hardware. In the mobile vision application space, these operations create opportunities to decrease execution runtime while also reducing energy. EVA provides support to this architecturally underserved class of operations.

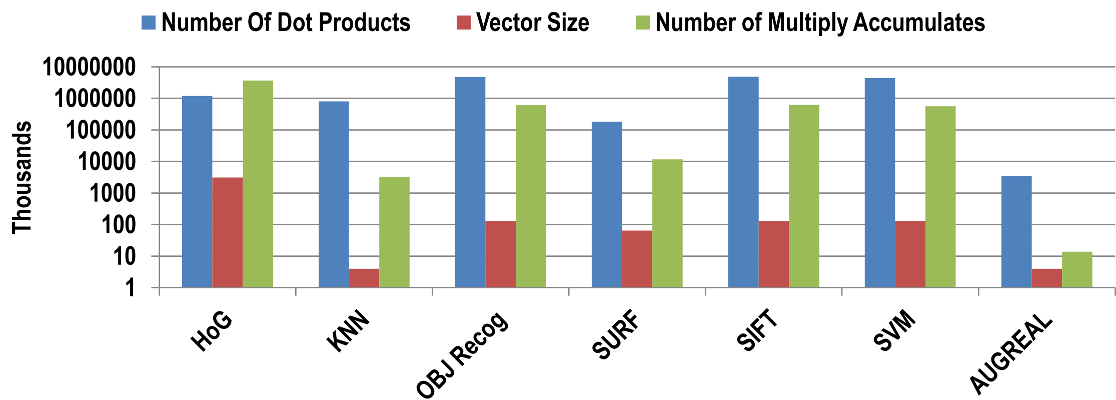


Figure 6.3 Number of Dot Product Operations The figure shows the number of dot product operations in the analyzed benchmarks for a small input size. The floating point multiply operations take a minimum of 5 cycles on ARM and thus each multiply accumulate operation has a large impact on execution time.

6.3.2 Diverse Parallelism

Computer vision workloads have been shown to contain thread-level parallelism by allowing multiple cores to work simultaneously to increase performance [33]. While the parallelism is present in computer vision algorithms, the workloads are not typically equal.

Figure 6.4 shows the performance of the feature extraction and classification benchmarks from MEVBench when running with varied number of threads. This data for this figure was taken using an ARM A9 and timing each thread separately. The timing taken such that only the time a thread is awake and doing work is utilized. The time to complete each benchmark is taken as the maximum time of all the threads in the workload to complete. The average speedup of both types of algorithms is plotted along with their geometric mean for a given number of cores. This figure demonstrates the duality of the workloads in vision applications. On one hand the feature classification workloads scale well with the number of cores, while on the other hand the feature extraction workloads quickly reach an asymptotic limit. Through examining the runtime for each thread, the feature extraction workloads were found to be limited by the performance of the coordination component more so than in the classification application. Most vision applications show similar behavior to feature extraction, resulting in applications having limited thread level parallelism due to coordination.

6.3.3 One and Two Dimensional Locality

Most mobile vision workloads utilize imaging data as input. The initial phase of processing typically involves processing the image data in 2D pixel patches. We examined the source code of MEVBench and found that the feature extraction algorithms often access image data in 2D for computing feature vectors while the classification algorithms work on the one dimensional feature vectors, confirming the result of [33]. The 2D locality does not transfer well into the typical raster scan order of pixel rows. When pixels are stored in raster scan order, two pixels that are vertically adjacent are separated in memory by a step of at least to the width of the image in pixels times the size of a pixel. Thus the typical linear approach to storage can can lead to inefficient accesses. However, the typical next phase of the vision pipeline, classification, utilizes linear vectors. In this phase, 1D spatial locality is ample and readily exploited by current cache architectures. In order to optimize memory accesses in all phases of vision algorithms, the systems need to support both the 1D and 2D locality.

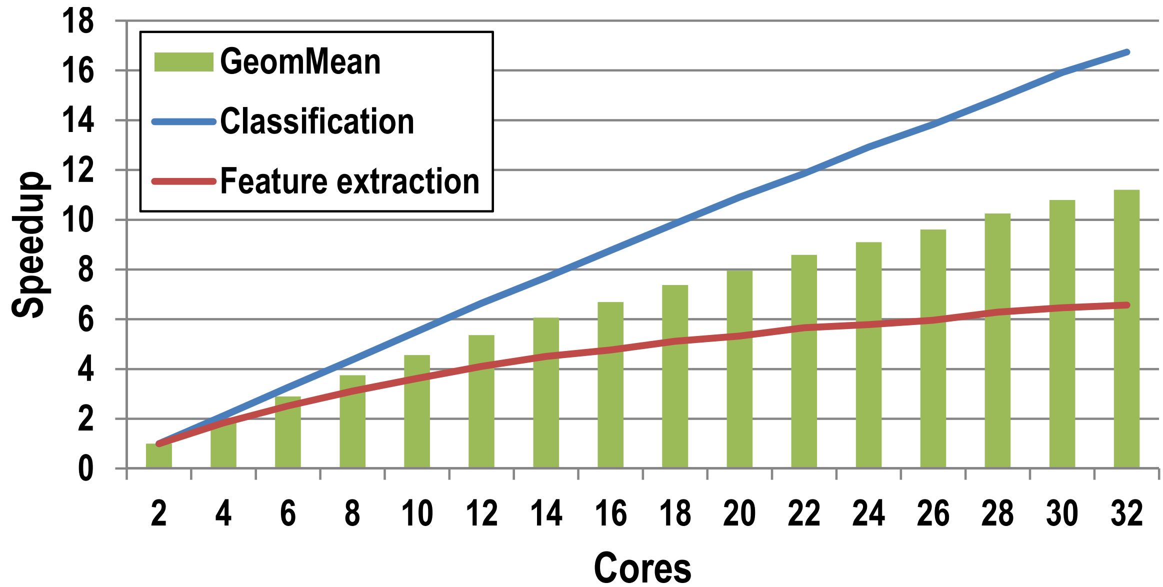


Figure 6.4 Thread Level Parallelism Within Vision Algorithms The figures shows the idealized performance of the feature extraction and feature classification benchmarks from the MEVBench suite. The figure shows the average speedup for each type of benchmark. It also shows the geometric mean of the speedups. Speedups are closer to linear for the feature classification while the feature extraction speedups quickly begin to saturate. Since both are considered part of most vision applications, a system needs to support powerful cores to work through extraction and many cores to work through classification. The limiting factor in the feature extraction is the speed of the coordination element. Thus a heterogeneous multicore can improve overall performance.

6.4 EVA Hardware

We have developed architectural features that provide capabilities based on the characteristics of the mobile vision application space. We add in accelerators for vector reduction operations such as dot product and tree compares. We introduce a software-enabled 2D locality prefetcher called the tile cache. Finally, we exploit diverse parallelism through the introduction of heterogeneous cores.

6.4.1 EVA System Architecture

EVA is designed to efficiently handle mobile vision workloads. An overview of a system with EVA can be seen in Figure 6.5. All cores in EVA contain a set of custom accelerators to handle the common vector reduction operations. In particular, EVA cores have units for performing the dot product, monopoly compare, max compare, and tree compare. These custom accelerators are designed to reduce both energy and latency of their target operations. EVA's cache has been modified to support both 2D and 1D locality in memory accesses.

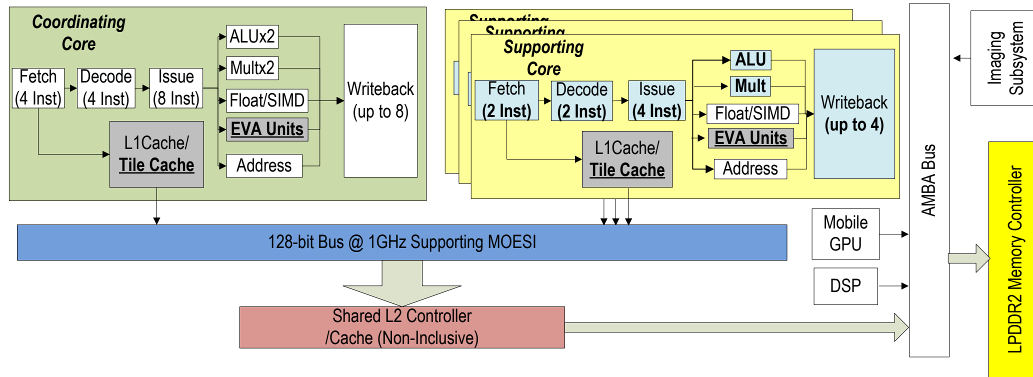


Figure 6.5 An EVA based System Overview The figure shows an example of a mobile SoC with an EVA configuration as the primary processor. EVA contains a set of cores made up of two types: coordinating cores and supporting cores. Coordinating cores are powerful superscalar out-of-order processors who coordinate the work of the coordinating cores. The coordinating cores are more energy efficient but ISA compatible with the coordinating core. Both types of cores contain accelerators that enhance their performance on mobile vision applications. The EVA cores connect together using a 1GHz 128 bit bus system utilizing MOESI cache coherence. The bus connects the cores to a shared L2 cache. The L2 cache connects to a to a LPDDR2 memory controller. EVA can communicate with external computing units using the AMBA bus which also connects to the LPDDR2 memory controller.

This optimization permits improved memory performance when accessing image data.

Our solution takes into account that many mobile vision applications can extract thread-level parallelism, thus EVA is designed as a heterogenous multicore comprised of two types of ISA-compatible cores. The first type of core, called *coordinating core*, is a powerful 4-wide out-of-order core designed to handle sequential code that can not be parallelized effectively, such as that used to coordinate the work of a group of threads. The second type of core, called *supporting core*, is a low-power core that exploits thread-level parallelism by efficiently running the worker threads for workloads. The supporting cores reduce many of the costly architectural features of the coordinating core. In particular, the supporting cores are 2-way superscalar cores as opposed to the 4-way issue that the coordinating core supports. They also have only 2 ALUs instead of the 4 found in the coordinating core. Their physical register file is reduced in size by 25% compared to the larger core. The ratio of coordinating to support cores and the total number of EVA cores can be configured based on constraints of the system and the key application characteristics.

In the example system of Figure 6.5, EVA's cores are connected through a bus with fast snoop controllers. We chose this interconnect strategy due to its common usage in mobile designs such as Tegra 3 [108]. Additionally, this design lends itself to the workload characteristics of mobile vision where most of the time cores work on local data, allowing for efficient use of the bus. EVA's features do not rely on the interconnect as long it does

Table 6.1 EVA Accelerator Instructions The instructions added to utilize the EVA accelerators.

Instruction	Operand A	Operand B	Operand C	Result	Instruction
MONOCMP Monopoly Compare	F[m] Value	F[n] to F[n+15] Vector		R[k] Results	MONOCMP F[m], F[n], R[k] example: MONOCMP F[0], F[16], R[0]
TRECMP Tree Compare	F[m] to F[m+6] Feature Vector	F[n] to F[n+6] Tree Vector		R[k] Node Value	TRECMP F[m], F[n], R[k] example: TRECMP F[0], F[8], R[0]
MXCMP Maximum Compare	F[m] to F[m+7] Vector			R[k] Index of Maximum	MXCMP F[m], R[k] example: MXCMP F[0], R[0]
DOTPROD Dot Product	F[m] to F[m+15] Vector	F[n] to F[n+15] Vector		F[k] Result	DOTPROD F[m], F[n], F[k] example: DOTPROD F[0], F[16], F[32]
PATLOAD Patch Load	R[m] Address	R[n]¡31:16¿ Patch Step	R[n]¡15:8,7:0¿ ¡Width:Height¿	R[k] Loaded Value	EVATCLD R[m], R[n], R[k] example: EVATCLD R[0], R[1], R[2]

not become a bottleneck. For more than eight cores, the bus would need to be replaced with a network-on-chip. The interconnect allows EVA to utilize a shared memory multicore architecture with a shared L2 cache that is non-inclusive. The cache coherency protocol is MOESI. The EVA cores communicate utilizing the Pthreads software library. Our system utilizes memory mapped I/O to access external devices. External subsystems can generate interrupts to the EVA cores and vice versa for coordination. For example, once an image is captured the image subsystem can produce an interrupt in the coordinator core alerting it of the new data. The image can then be retrieved from memory.

6.4.2 Custom Accelerators

The EVA accelerators take advantage of 64 32-bit floating point registers present in mobile SIMD units such as ARM NEON [14]. In typical modern SIMD units for mobile platforms the registers can be accessed individually or in groups of two or four single precision registers. EVA's accelerators require the extension of this ability to groups of up to sixteen registers. EVA assumes the ability to read the registers in groups of eight in one cycle. EVA assumes register operands are aligned on the groups of eight i.e., 0, 8, 16 etc.

Dot Product Accelerator

As shown in Figure 6.3, the dot product occurs frequently in many vision codes. For example, the dot product is used to perform convolution for image filtering and also to normalize vectors during feature extraction. It is also a common operation in the classification phase for comparing various feature vectors. The operation performed by a dot product can be seen in Equation 6.1. The operation works by multiplying corresponding vector entries and summing the result. Figure 6.6 shows both the dot product pseudocode and the operation of

the accelerator.

$$result = \sum_{i=0}^{k-1} A_i * B_i \tag{6.1}$$

EVA supports the dot product with an the DOTPROD instruction seen in Table 6.1. The first operand F[m] indicates the first register in a sequence of 16 registers that will be used as the vector input. For example, F[0] sets floating point registers 0 through 15 as the input to the dot product unit. F[n] is the start index for the second set of sixteen floating point registers to be used as input. F[k] is the register to store the dot product's scalar result. The dot product example in Table 6.1 would result in registers 0 through 15 being the first vector, registers 16 to 31 being the second vector and the scalar result being placed in register 32.

In general, for the EVA accelerator instructions, the vector input sizes have been fixed to a length specified for the instruction in Table 6.1. If the output register overlaps with the input register, the output will overwrite the value in the input register upon instruction completion but the computation will be on the input. In the event of an floating exception, the floating point exception flag is set, and it is handled with the instruction in writeback.

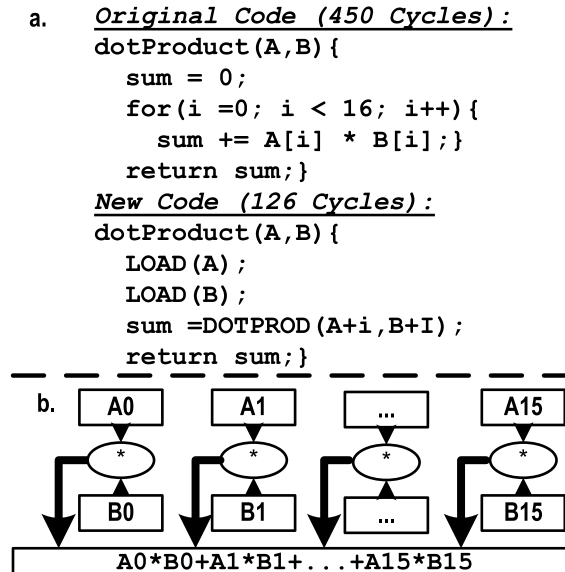


Figure 6.6 Dot Product Accelerator Overview The figure shows the operation of the dot production unit. (a) shows the pseudocode of the dot product operation. (b) shows the operation it performs. The unit performs a dot product operation which is multiplying all the entries of two vectors and adding the results. The add component is done using a tree to add each adjacent result until the final result is computed.

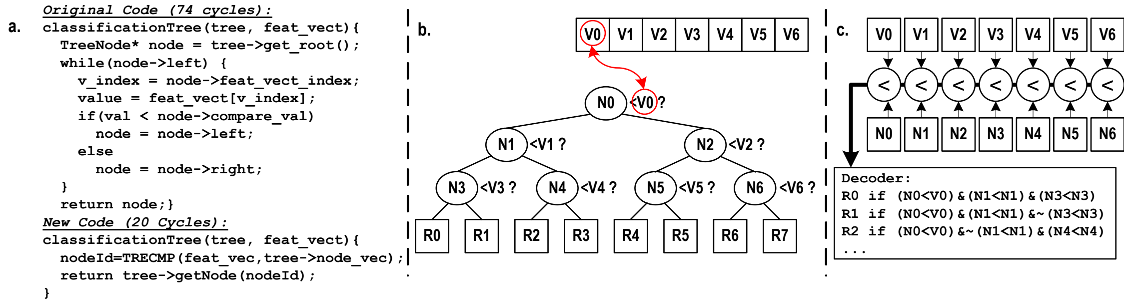


Figure 6.7 Tree Compare Accelerator Overview The figure shows the operation of the tree compare accelerator. (a) shows the typical tree comparison pseudocode used by vision algorithms, in particular classifiers. (b) shows how this looks relative to the tree. (c) shows how the accelerator unrolls the compares and performs them all at once and then decodes the binary bit vector to output the proper index. This operation is common in the many tree-based classification algorithms used in mobile vision.

Tree Accelerator

A commonly used data structure in the classification phase of computer vision is the tree. Trees are used in classification algorithms such as binary decision trees, Adaboost, and k-Nearest Neighbor Classification. They are used to classify feature vectors based on the feature vectors entries. Typically, trees in computer vision are computed offline and are accessed but never modified by vision applications. It has been shown that collections of small trees can be used to produce high quality classification results [20, 22]. Based on this application behavior, we have designed the tree compare accelerator to accommodate a binary tree of depth three.

The tree compare accelerator is based on decision trees. Each node of a decision tree utilizes a value, called a *split*, and compares it to a specific feature entry to determine which child node should be used during the classification phase. The leaf node determines the classification of the feature vector. Figure 6.7a shows the code for a tree compare operation of this type. We assume the feature vector has been compressed to only entries used by the tree. This compressed vector is passed into the tree comparison. The first entry of the vector is compared against the split value at the root node. Then based on the result of the comparison, the left or right child node is evaluated. The applications we considered only require less-than comparison. The feature vector index to compare to is stored with the tree node data structure along with the value to be compared. This traversal will continue until a leaf node is reached. A leaf node is a node with no children.

Figure 6.7b shows how the compressed feature vector and nodes comparisons take place relative to the tree. In this case the seven entries are placed in a vector based on which node they are to be compared with. The feature vector has the entries arranged such that the first

entry is for the comparison to Node 0, the second entry is for comparison to Node 1, and so on. This can be done by taking the indices from the nodes and using them to put the entries in correct order before passing them to the tree compare operation. The node comparison values are also placed in an array in node order. The set of indices, the node values, and their order are computed offline.

Figure 6.7c shows how EVA computes a tree compare with the arrays. The feature vector subset has each value compared to the tree node arrays values in parallel. The result is an 7-bit vector which is decoded to produce the index of the leaf node as the instruction result. The leaf nodes are numbered left to right. The user program uses the produced index to read the result value from an array which is computed offline and is part of the user program.

The instruction for the tree compare can be seen in Table 6.1. The first entry F[m] is the first register in a set of seven that will be used as the first index. For example, F[0] sets floating point registers 0 through 6 as the feature vector input to the tree compare accelerator. F[n] is the start index for the second set of seven floating point registers to be used as the tree values. R[k] is the register to store the result. For the tree compare example in Table 6.1, registers 0 through 7 contain the feature vector, registers 8 to 14 contain the tree vector and the leaf node index result is placed in register 0.

Since a binary tree can be decomposed into subtrees [34], the tree compare accelerator can be used for trees of arbitrary size by using the output value as an index to determine the next subtree to load.

Max Compare

Computing the maximum of a set of numbers is a common operation within mobile vision applications. This operation can be used for performing dilation filtering on an image in preprocessing, or in finding the largest histogram value in a feature extraction algorithm. It is commonly used in an operation called non-maxima suppression to find the best scale/size or location of an object. Non-maxima suppression locates the maximal response within a region or set. This is a key operation in localizing features, objects, and responses. Thus, EVA provides a maximum operation to speed up this common computation.

The operation is utilized through a new instruction in the ISA named Max Compare which can be seen in Table 6.1. The max compare operates on small vectors and returns an index of the maximum value of the vector. The only input, F[m], is the first register in a set of eight that will be used as the vector. For example, F[0] sets floating point registers 0 through 7 as the input to the max compare accelerator. R[k] is the register to store the result. Table 6.1's maximum compare example would result in registers 0 through 7 being the

vector and the index of maximum value in the vector being placed in register 0. Figure 6.8 illustrates the code modifications to utilize the Max Compare.

Given the limited resources in most mobile systems, we have chosen to limit the size of the Max Compare input to a single vector of size eight.

Monopoly Compare

A common operation that takes place in vision applications is the comparison of a single number to a large vector to determine if it is smaller or larger compared to all the numbers in the vector. This operation is used in feature extraction to compare a single pixel value to it's neighbors in an image or to find corners in an image [94] [121]. This operation is quite frequent, and it is often a gating operation to performing more computation [33]. It can also be utilized during feature classification to track the top-N values. Thus, this operation can be used throughout the computer vision software pipeline. Given the potential benefits of speeding up this operation, EVA has an accelerator to support this vector reduction operation.

The monopoly compare accelerator can be seen in Figure 6.9 along with pseudocode of its operation. It supports both less-than and greater-than compares based on a bit in opcode. The basic instruction for the monopoly compare can be seen in Table 6.1. The first entry F[m] is the value that will be compared to the vector. F[n] is the start index for the set of sixteen floating point registers to be compared against. R[k] is the register to store the binary results as a single word. The result can be returned in multiple ways; an anding of the individual results, a binary string representing the individual results and an accumulation

```
Original Code (498 Cycles):
maxCompare(A, AMaxIndx){
  float Amax = A[0];
  *AMaxIndx = 0
  for(int i =0; i < 7; i++){
    if(Amax < A[i]){
      *AMaxIndx = i;
      Amax = A[i];
      }
    return Amax;}}
New Code (143 Cycles):
maxCompare(A,AMaxIndx){
  LOAD(A);
  return  = MXCMP(A,AMaxIndx);}
```

Figure 6.8 Max Compare Accelerator Code The figure shows a comparison of the code for computing the max of a vector with and without the EVA accelerator. The EVA accelerator can decrease the number of cycles for computing the max of an eight entry vector by a factor 2.7.

of the individual results. The result mode is selected by bits in the instruction encoding. The example for the monopoly compare in Table 6.1 would result in the value in register 0 being compared using less-than logic to the values in registers 16 to 31. The result would be placed in register 0.

6.4.3 Tile Memory Architecture

Many vision algorithms has been shown to have 2D spatial locality, in particular the feature extraction algorithms [148]. This characteristic is due primarily to the input of computer vision algorithms being images that are systematically scanned using small 2D windows. However most image data is stored in raster scan order which causes vertically adjacent pixels to be stored at a large distance away. Reordering the data with hardware has been proposed by [32]. However, the extra hardware costs may not be acceptable in mobile designs. Yang et al. [148] proposed the use of software instead of hardware, but a hardware implementation is more efficient and decreases the difficulty of utilization for the developer. Thus, we adopt a novel 2D prefetcher that warms up the cache when the application indicates it is touching 2D data. When the cache is operating with this prefetcher, we refer to this as the tile cache. Unlike other prefetchers, such as a stride prefetcher, that predict the access

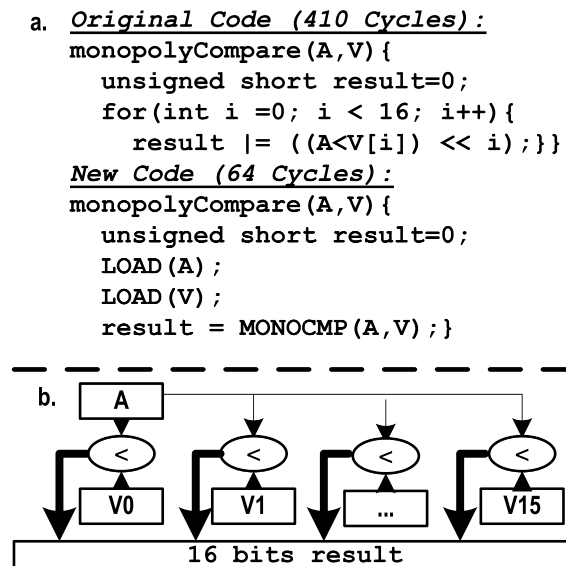


Figure 6.9 Monopoly Compare Accelerator Overview The figure shows the operation of the monopoly compare accelerator. (a) shows the pseudocode for the operation. (b) shows the operation itself. The accelerator compare a single value to the entries in a vector. The results are combined into a single 16bit result. This is useful in vision applications where a minimum value is required for an image sample.

pattern, the tile cache receives the correct step amount from the developer.

The tile cache is activated by a special load instruction that requires three additional pieces of information beyond address. The first additional information is the step between the pixels. The second is the width of a tile. The last is the height of the tile in the number of steps. These are passed using two registers as seen in Table 6.1. In the example in Table 6.1, R[1] would contain an address to load. R[2] would contain the patch step in the upper 16 bits. The lower 16 bits would contain patch width in bits 8 to 15 and the patch height in bits 0 to 7. The width is number of 64 bytes chunks in the tile. The height is the number of steps in the tile. This information is provided directly to the prefetcher. The result would be placed in R[0].

The prefetcher attempts to prefetch the cache lines in a tile. However, the hardware has a limit on the number of prefetch requests to avoid overburdening the memory system. We empirically found a maximum prefetch count of four outstanding 64-byte cache lines to be sufficient. The prefetcher also sorts the requests based on how often a cache line will be accessed. Thus once a slot opens up for prefetching, the line that has been requested the most will be retrieved first. Figure 6.10 shows an example with the tile cache. The pixel values are stored one after the other, and pixels in a tile can span on multiple cache lines (shown in different colors in the figure). The cache will fetch the required data while the tile cache will generate requests for the rest of the tile. The cost of this mechanism is minimal, it requires minor changes to state machine of the prefetcher and the register operands can be general purpose registers. This gives the benefits of 2D locality without modifying the rest of the memory system.

6.4.4 Heterogenous Chip Architecture

As indicated in Section 6.3.2, diverse parallelism is a characteristic of many vision workloads. In particular there are points in many computer algorithms where threads coordinate their efforts and share their results with other threads [33]. Furthermore, there are some components of vision algorithms that do not benefit from thread-level parallelism. For example, when the amount of data being processed is small, the coordination cost may hinder the overall performance. While utilizing more cores can ensure parallelism, the constraints of mobile systems require low energy usage to preserve battery life. Thus a balance must be struck. EVA provides heterogenous cores to deal with this situation. In particular, the EVA architecture is separated into two sets of cores. Each EVA system has one high performance core called a coordinating core and one or more lower performance cores called support cores. All the cores contain the EVA units to improve their energy efficiency and

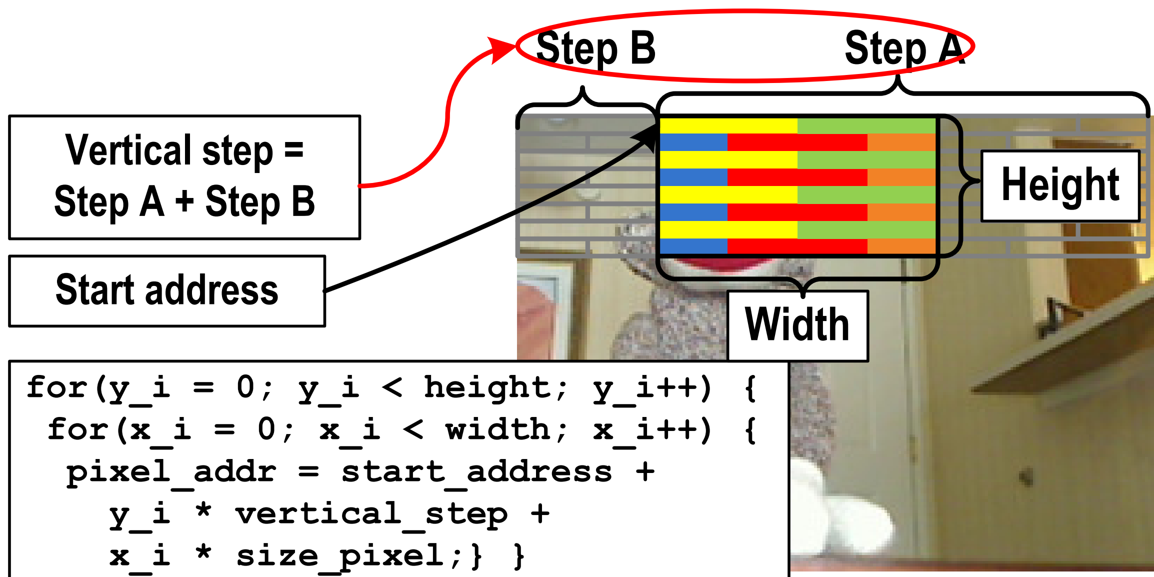


Figure 6.10 Tile Cache Overview The figure shows how the tile cache handles image data. Each different colored segment is a cache line. The tile fetches the current data target cache line but also prefetches consecutive cache lines and caches lines for an address that is certain memory stride or step away. The step is based on the register values that are retrieved with the load instruction.

performance on mobile vision workloads. The EVA architecture can support any processor interconnect; however, for the remainder of this work we assume a bus interconnect. All the cores share an L2 cache while they each have a private L1 cache and tile cache.

Figure 6.5 shows the relationship between the coordinating and supporting cores performance capabilities. In particular the coordinating core has a wider pipeline than the supporting core. The coordinating core also has more integer computation units than the supporting core. They have the same floating point/SIMD engine and EVA units. The coordinating core has a wider writeback stage as well. The supporting core's L1 data cache has half the associativity as the coordinating core. Furthermore, the support cores need their design to be focused primarily on energy more than performance.

During our work we found that support threads take approximately 20% to 40% less time to complete. Based on this information, the support cores in EVA can be as much as forty percent less powerful as the coordinating cores on vision workloads with minimal impact on the overall performance.

Given the two core types, a key design question is what is the proper number of each type. The answer to this design decision lies in the demands of the target applications, combined with the area and cost constraints of the target market. We shall examine this key design decision in the experiments section. We investigate a set of same-sized configurations and evaluate their performance and energy demands. These configurations can be seen in

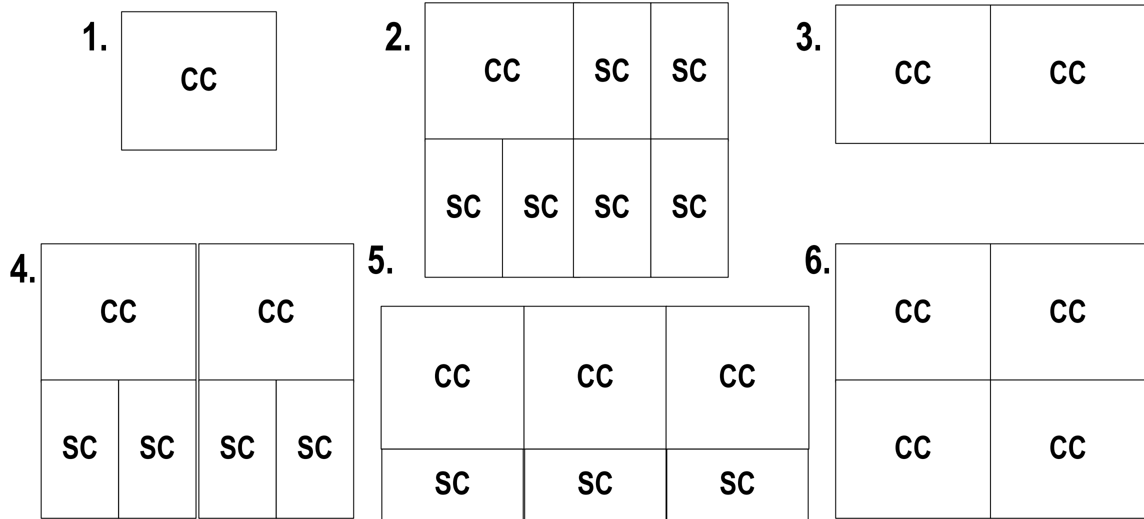


Figure 6.11 EVA Coordinating and Supporting Cores Configurations With Area Constraint The figure shows the possible EVA configurations given the maximum area of four coordinating cores without the EVA accelerators. Configurations (2),(4) and (5) have roughly the same area as a 4 coordinating cores without EVA features. Configuration (6) is slightly larger due to EVA features.

Figure 6.11.

6.5 Experimental Setup

6.5.1 EVA Model

We simulated our system using the gem5 simulator [19] in full system mode. The simulated system ran Ubuntu for ARM Linux with kernel version 3.3. We utilized Linux because it is a common mobile operating system used in Android and Ubuntu for ARM. We utilized the Ubuntu image for compatibility with MEVBench. The Ubuntu for ARM was stripped down to give minimal services. We modified the gem5 ARM model ISA to support the EVA accelerator instructions. We modified the memory system to support the tile cache functionality through a special load instruction. The performance parameters of the gem5 model for both the coordinator and supporting cores can be seen in Table 6.2. The mobile baseline is a coordinating core without EVA accelerators and mobile GPU is a SGX 54x series.

The EVA dot product accelerator is based on the efficient floating point unit design work of Galal and Horowitz [51]. The unit is pipelined and provides its result after 7 cycles. The EVA monopoly compare, tree compare and max compare are based on the compare work

Table 6.2 EVA Configuration

Feature	Configuration
Core Clocks:	1 GHz
Coordinating Core:	32 bit RISC out-of-order, 4-way superscalar
Coordinating Core Pipeline:	8-Stage
Coordinating FUs:	4 integer units, 1 floating point units, 1 SIMD unit, 1 set of EVA accelerators
Vector Registers:	64 32bit Single Precision Registers
Coordinating Core L1 Caches:	32k 4-way assoc. instr. and data (2ns)
Supporting Core:	32 bit RISC out-of-order
Supporting Core Pipeline:	8-Stage, 2-way superscalar
Supporting Core FUs:	2 integer units, 1 floating point units, 1 SIMD unit, 1 set of EVA accelerators
Vector Registers:	64 32bit Single Precision Registers
Supporting Core L1 Cache:	32k 2-way assoc. instr. and data (1ns)
L2 Cache:	1MB unified non-inclusive (12ns)
Cache Coherency:	MOESI
Processor Interconnect:	128-bit Bus@ 1GHz with fast snoop unit
System Memory:	2GB LPDDR2
Instruction Set:	ARM-v7
Technology Node:	45nm

Kim and Yoo [87]. The compare based units are pipelined and provide their results after a 5 cycle delay. The area for each functional unit can be seen in Table 6.3 along with core area estimates. The base estimates for the area of the cores are based on information from [16, 89] and [13]. We estimated the energy of the base cores using Mcpat [93] along with energy models based on the accelerator designs.

Table 6.3 Area estimates for the EVA Cores These estimates assume a 45 nm silicon process.

Module	depth	latency	Area (mm^2)
Monopoly Compare	6	5 cycles	0.0489
Tree Compare	6	5 cycles	0.0215
Max Compare	6	5 cycles	0.0244
Dot Product	8	7 cycles	0.3290
Total for accelerators per core			0.4240
Coordinating Core			7.1200
Supporting Core			1.5839
Baseline Mobile Core w/SIMD + Embedded GPU			15.400

6.6 Experiments

6.6.1 Benchmarks

We utilized the MEVBench mobile vision benchmark suite [33] for our evaluation of the EVA system. We modified the benchmarks to insert the EVA operations where appropriate by finding loops with acceleration opportunities and inserting the new instructions into the code with inline assembly. We used the Code Sourcery ARM cross compiler suite version 4.6.1 [102] to generate static executables. We limited the compiler optimizations to -O1; however, we did enable ARM Neon instructions and their usage by the compiler with auto vectorization. In our simulations the coordinator core was running the coordinating thread of the benchmark.

6.6.2 Single Core Results

Figure 6.12 shows the speedup gained through utilization of the EVA features in benchmarks compared to a coordinating core without the EVA features. The tile cache benefits are primarily seen in the feature extraction benchmarks (e.g., HoG, SIFT, and SURF). These benchmarks all access image data and thus benefit from the 2D caching. The tile cache had little impact on benchmarks without 2D locality such as SVM and BOOST. There is a small amount of improvement with FACEDETECT as well. Overall, the tile cache provides moderate performance improvements (2% to 40%) for programs that exhibit 2D localities. Since use of the tile cache is software controlled its use can be avoided for programs with out 2D locality, thereby limiting the extent to which the tile cache prefetcher negatively impacts program performance. Figure 6.13 shows the usage of the accelerators. The figure demonstrates how the various benchmarks utilize different accelerators. Overall, the most

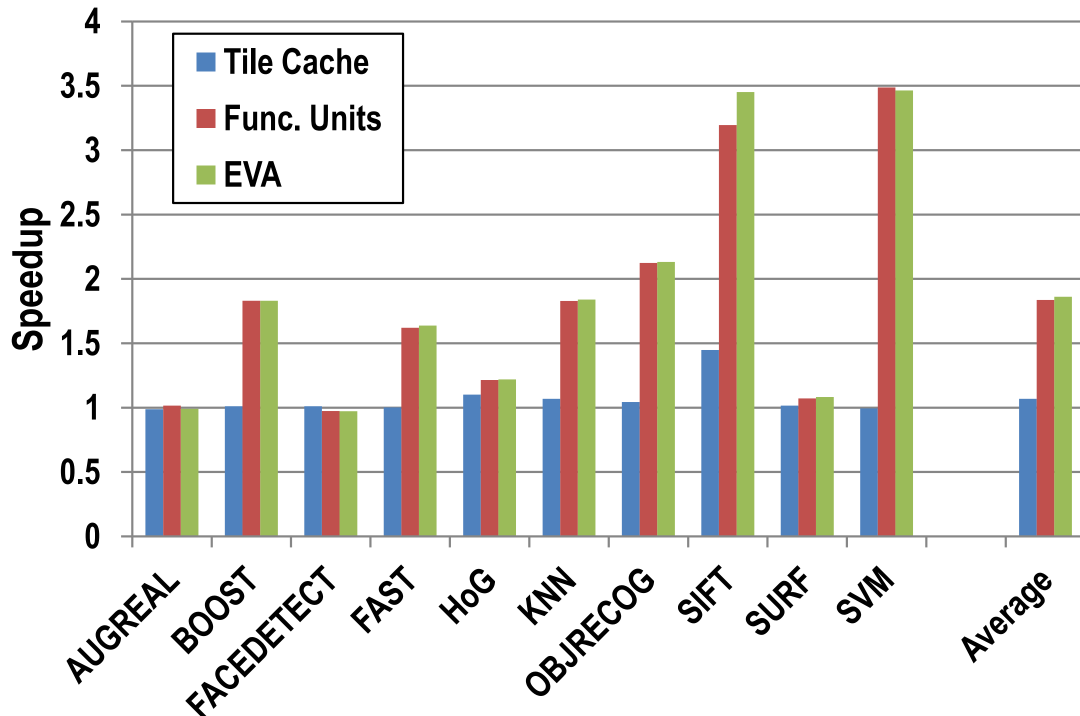


Figure 6.12 EVA Single Coordinating Core Speedup The figure shows the speedup of the an EVA single coordinating core versus a single coordinating core that does not have EVA. The plot shows how both the tile cache and EVA accelerators both contribute to the performance increase.

exercised accelerators are the dot product accelerator followed by the monopoly compare accelerator. The tree compare accelerator is heavily utilized in the tree based benchmarks.

The accelerators of EVA provide a speedup in all the benchmarks except FACEDETECT and AUGREAL. In the case of FACEDETECT, the benefits of the EVA accelerators was not seen fully due to limited use of the accelerators in this benchmark. The same was true in AUGREAL. Overall the complete EVA design provides an average speedup of 1.8x. It peaks near 4x for SVM due to its very heavy use of the dot product operation.

Figure 6.14 shows the normalized energy usage of EVA running the benchmarks. The energy is calculated using the simulation statistics to generate an input model for Mcpat [93]. Mcpat models the common microarchitectural components such as caches and ALUs at our given technology node (45nm) based on runtime activity. A CACTI [133] like framework is used to model memory components in this framework. We combine these results with models for the energy consumption for each custom accelerator to compute the total energy. The graph in Figure 6.14 plots the energy of three EVA designs, normalized to the energy of a coordinating core without EVA enhancements. It is interesting to note that in some cases, the tile cache provides a small amount of energy savings. This is due to not having

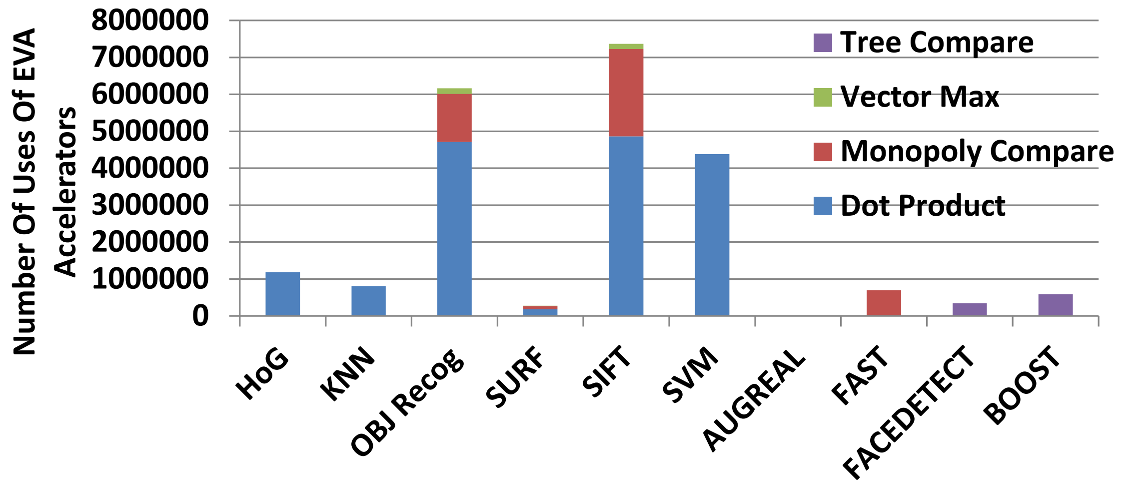


Figure 6.13 EVA Single Coordinating Core Accelerator Usage The figure shows the usage of the EVA single coordinating core accelerators. The plot shows how often the a given accelerator contributes to the performance increase.

the wait for the data to be returned before resuming execution, thus reducing idleness of the accelerators. The tile cache is designed to allow for memory accesses in both a 1D and 2D fashion. It is primarily beneficial in the feature extraction benchmarks as a result. The accelerators show a decrease in energy in all the benchmarks except FACEDETECT. This is due to the same effects that caused the slight slowdown. In the case of SVM, the use of the efficient vector reduction dot product accelerator provides a large amount of energy savings. In terms of energy, EVA provides savings of close to 30% and peaks at 3x decrease in energy while also providing an average speedup. The key savings seem to be attributed to the accelerators although there is modest savings from the tile cache features in a couple of benchmarks as well.

Figure 6.15 shows how the number of committed instructions were affected by the use of EVA components. A large portion of the energy savings come from large reductions in dynamic instruction counts due to use of the vector reduction instructions. In the case of KNN we were able to replace a small number of loads with tile cache loads. The same is true for OBJRECOG. In general, the tile cache should be used in linear mode or optimized efficiently to the vision algorithm.

Overall the single-core EVA provides an average speedup of approximately 1.8x while reducing the energy usage of a core by over 30%.

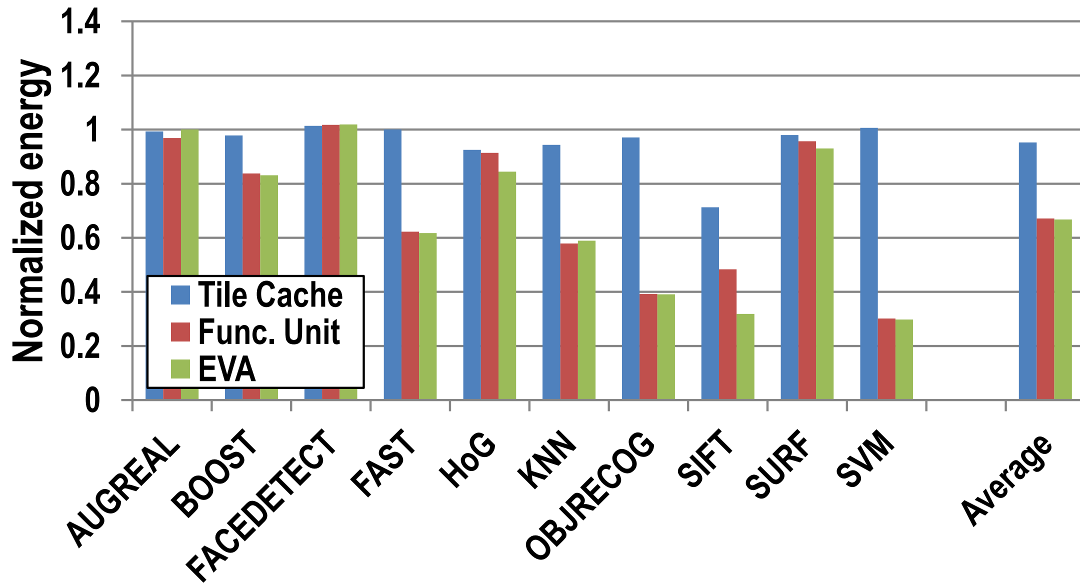


Figure 6.14 EVA Single Coordinating Core Normalized Energy The figure shows the energy per frame for a EVA single coordinating core normalized to a coordinating core without the EVA enhancements. The energy savings using EVA, in general are quite good. The savings primarily come from the use of the EVA accelerators that are specifically designed for the given operation, and the decrease in the committed instructions which decreases the work of the pipeline as a whole.

6.6.3 Multicore Results

We ran the configurations seen in Figure 6.11. We show the average speed up for the benchmark suite for each configuration in Figure 6.16. We constrained our configurations to approximately the area of an embedded quad-core processor utilizing four coordinating cores without EVA features. This represents a modern class of embedded machines and its area constraints. The EVA accelerators were used on all the benchmarks in the figure. The tile cache, which can be utilized in software, was only active on benchmarks that showed a performance benefit during single-threaded execution. Overall, the use of the more cores is beneficial although the benchmark is only completed once all the cores have finished. In some cases, the workload of a supporting core causes slowdown for the entire system. The overall performance shows that having a single coordinator and six supporting core seems to be the best performing configuration.

Figure 6.16 shows the average energy usage to process a 352 x 288 frame for each configuration for the benchmark suite. The energy is normalized to the energy demands of a single coordinating core. The energy requirement drops as the number of cores increases due to the reduced runtime as thread-level parallelism is exploited. An interesting trend is seen when the number of cores is held constant and number of coordinating cores is

reduced, energy decreases. This is due to the imbalance in the parallel workload. At various phases, the work must be coordinated. This can only take place once all the cores have completed their work. Once cores have complete their portion of work, they are sitting idle. The supporting cores are more efficient when waiting.

Figure 6.17 shows the scalability of the EVA coordinating cores given a fixed power budget of 5 Watts as the voltage is scaled down. The plot shows that for a 5W budget, the number of cores reaches a peak of 12 cores with a peak speedup of 3.2x on the MEVBench benchmarking suite with each core running at approximately 800MHz clock. The 24 core configuration is above the power budget slightly but with a slight increase in the energy budget would also be a good platform for vision applications. Thus with a fixed power budget, the performance versus energy tradeoff can be managed to provide a balanced solution. For example, if energy is more important than speed then 14 cores would be a better solution, given this power budget.

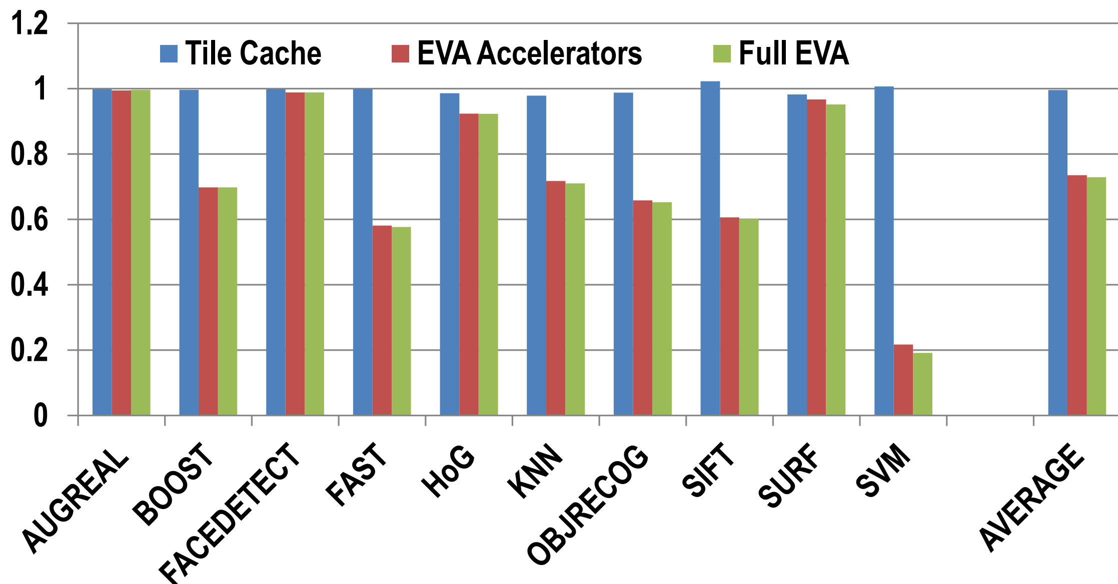


Figure 6.15 EVA Single Coordinating Core Normalized Committed Instructions The figure shows the number of committed instructions by a single EVA coordinating core normalized to a coordinating core without EVA based enhancements. The reduction in instructions accounts for much of the energy savings and speedup. The SVM benchmark shows the greatest decrease due to the linear SVM high utilization of the EVA dot product accelerator.

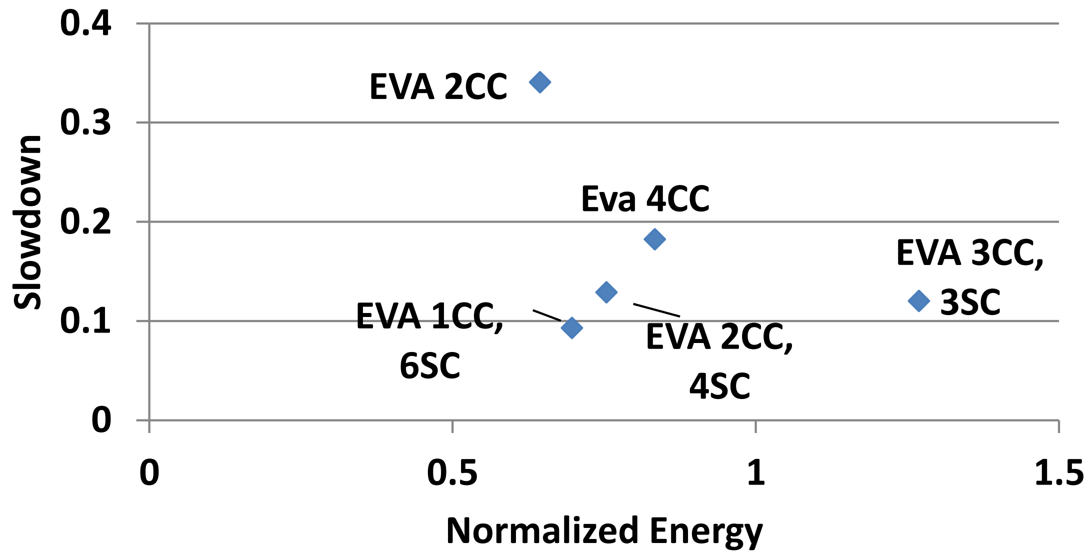


Figure 6.16 Multicore Configuration performance of EVA This figure shows the multicore performance of EVA. The x axis is energy is normalized to a single coordinating core without EVA enhancements. The y axis is 1/speedup such that closer to the origin is better. It shows that utilizing 1 coordinator core and 6 supporting cores is optimal given the quad core area constraint. The energy saving com partially from a more balance working load based on the coordinator. In other configurations, the coordinating cores waste energy waiting. If the supporting cores wait, they waste much less energy than a coordinating core. The tile cache was only utilized in benchmarks that showed it improved performance during the single core tests.

6.6.4 Comparisons To Other Approaches

Figure 6.18 compares the performance of EVA against other solutions. For each experiment, the platform is running the SIFT algorithm optimized for that particular platform. The graph plots the performance of Qualcomm Snapdragon S2 with Adreno [86, 29], Intel i7, GTX 260, EFFEX [32], and EVA on a pareto chart, which indicates the energy demands and performance capabilities of each design solution. The EVA solution is closest to the origin making it a pareto optimal solution, i.e., there are no other solutions with better energy (at this performance), or better performance (at this energy). EVA's accelerators increase the computation performance and decrease the energy usage. The use of heterogeneous multicore takes advantage of thread-level parallelism in an efficient manner. While GPUs are efficient cores for graphics, the comparison of the mobile GPU with the desktop GPGPU shows the large gap in performance in utilizing GPUs on mobile devices [29].

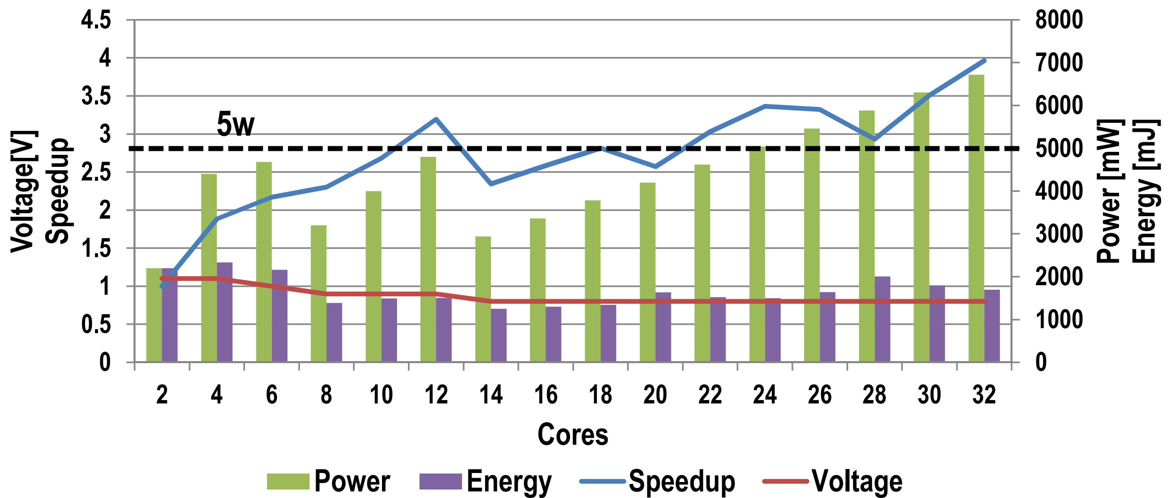


Figure 6.17 EVA Coordinator Cores Scaling The figure shows the energy and speed up for a fixed power budget of 5W as voltage is scaled and the number of cores is increased. This analysis assumes an interconnect that can scale such as an NoC. 12 cores seems to give a good performance with a high speed up and low energy.

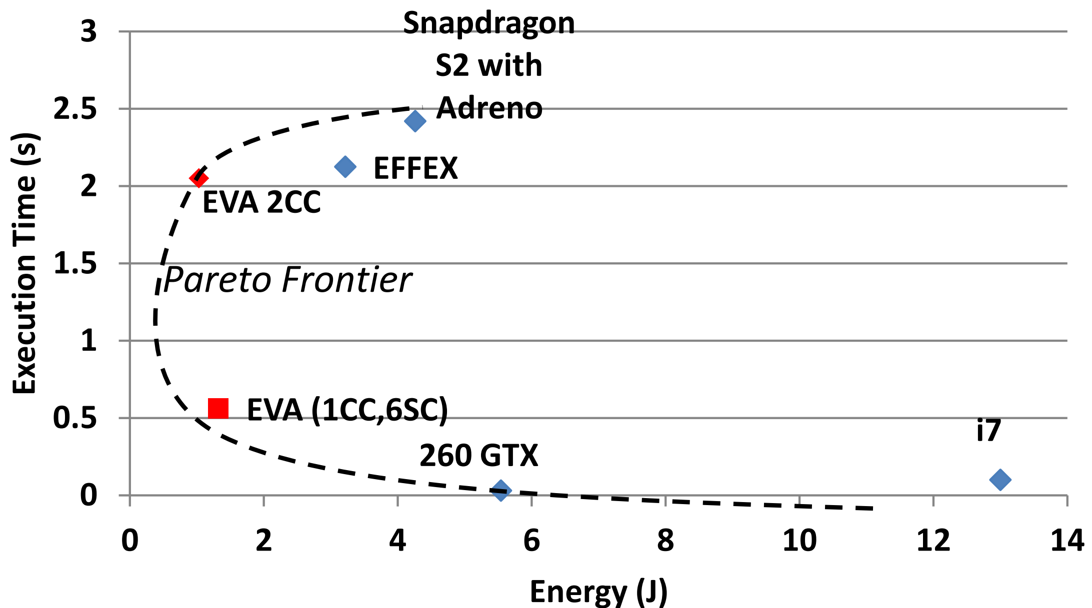


Figure 6.18 Comparison Of EVA with Other Solutions This figure shows a comparison of EVA against other solutions based on [29, 105, 86]. It demonstrates that EVA is currently one of the best options for the SIFT energy/performance tradeoff.

6.7 Related Work

Companies such as Qualcomm and Texas Instruments (TI) have released closed-source computer vision libraries that are optimized for better performance on their SoCs [117, 76]. These libraries typically use compute resources not available to end developers such as small

DSPs and image filtering accelerators. Thus, this approach encourages developers to use only the algorithms the companies choose to expose, and the developers can not modify the operation of the algorithms. New algorithms may not be supported immediately or the API may not quite match the developer's needs. With EVA, the user software has access to the accelerators, thus if a new algorithm is developed or an API needs to be tweaked it can be readily ported to the EVA platform.

Current processors support SIMD instructions that allow computation on vectors of numbers [11] [68]. These solutions perform some operations similar to our accelerators, however they do not typically include the reduction step. In general, the current SIMD instructions allow for multiple compares and multiply accumulates to be performed on vectors but return a vector result. The vector result then has to be accumulated or parsed to be utilized. The EVA accelerators return a scalar result that can be immediately utilized without much extra processing. It should be noted that the EVA accelerators are also wider than the typical mobile SIMD engine such as NEON [11].

Yao et al [149] propose the use of specific hardware for SIFT algorithm. Indeed, other have proposed similar hardware for specific computer vision algorithms such as BOOST [72] or applications such as robotics [150]. Such solutions are capable but inflexible. If an improved algorithm is developed the system must be redesigned. For example, altough ORB [122] was developed based on FAST [120] the algorithms are different enough that any hardware system built for the FAST algorithm would be need to be completely redesigned. EVA's programmability provides flexibility for the developer to evolve the platform's vision software capabilities, while still maintaining an efficient solution.

Clemons et al. [32] developed EFFEX, a processor for the feature extraction phase of mobile vision. EVA provides capabilities for the entire mobile vision pipeline, where as EFFEX focused on only feature extraction. Furthermore, EVA avoids using a special memory controller to exploit 2D locality and instead utilizes a tile cache prefetcher which could be readily integrated into most mobile systems. EVA utilizes less energy and has less latency than EFFEX.

Raghavan et al. has proposed powering up cores for fractions of a second and then powering them down to cool to handle high computational loads such as mobile vision [118]. This technique is very effective for applications that have pauses between computations, however, many vision applications such as person tracking and augmented reality require continuous computation. With EVA, we are able to maintain high performance in mobile vision for extended periods.

ARM has released the big.LITTLE platform for energy efficient heterogeneous computing [13]. This is a general purpose system with two sets of dual core processors of varying

performance capability and energy usage. They are comparable in how they deal with diversely parallel workloads. The major difference between the two is that big.LITTLE cores have no special accelerators for vision. EVA provides heterogeneous computing as well, but also contains accelerators for mobile vision and a tile cache to increase performance. big.Little is designed for general purpose but can run vision applications while EVA is designed for mobile vision but can run general purpose applications.

GPGPUs have been used to increase the performance of computer vision on the desktop with impressive speedups by Prisacariu [115], Fung [50] and OpenCV [110] among others. However, mobile GPUs have been shown to be inefficient at mobile vision due to the energy and cost constraints of mobile systems [86]. Figure 6.18 shows that EVA is a pareto optimal solution when compared to GPGPUs and mobile GPUs. One reason for this is that the computation of vision and graphics are different. In fact, they are inverses. In graphics the system has a scene and is attempting to display an image while in computer vision the system has an image is trying to recreate the scene. There are other solutions within the SoC space, such as the DSP. EVA can be utilized to improve the performance of these solutions as well. EVA is not a mutually exclusive solution as EVA can share the processing with other devices allowing even more heterogeneity in the design.

Chen and Baer proposed the stride prefetcher to accommodate predictable data loads [28]. This method records the program counter of a memory access in a table and calculates the difference or stride of the memory address for each access. When the stride has reached a steady state, the prefetcher will automatically prefetch addresses based on this stride. When accessing data within a 2D window, the stride prefetcher has issues stabilizing because with image data the stride moving in the x direction is one while the stride moving in the y direction is the image step or vice versa. Furthermore, algorithms such as BRIEF can have random steps within a patch further hindering a stride prefetcher's capability to accurately predict the proper address values. The EVA tile cache however has no stabilization requirement because it receives information from the program. Furthermore, the tile cache is capable of dealing with multiple strides for a given memory location by design.

6.8 Chapter Conclusion

This chapter brings together the insights from the previous chapters to develop an architecture with accelerators and memory system optimizations for mobile vision systems. Mobile vision is a complex application space that has an insatiable need for computational power. Some vision applications can utilize large amounts of parallelism while others benefit from

serial performance. There are some vision algorithms that have 2D data locality while others only contain 1D. Most vision algorithms contain vector reduction operations. There is a need for a system that can take advantage of all these traits to increase performance in the mobile space.

Our solution for this space is EVA, an efficient architecture for mobile vision applications. EVA is a heterogeneous multicore with custom accelerators and caching system with both 1D and 2D locality to increase the performance of mobile vision systems. We have shown that the vector reduction accelerators that EVA provides improves the energy usage and computational performance of mobile vision workloads. We have shown how exploiting 2D locality in the cache can improve performance for a number of key mobile vision algorithms and modules.

We have also explored the performance-optimal configuration of EVA given a mobile quadcore area constraint. The single coordinating core with 6 supporting core is the best performing design under this constraint due to having the lowest energy and execution time on the benchmarks. EVA provides up to a 720x performance increase compared to the ARM A8 mobile processor. Additionally, the energy-optimal core configuration given a fixed power budget has also been shown. For a 5W power constraint, the most effective design was a 12 core configuration based running at 680 MHz and .9 volts. EVA has been shown to be a capable solution to mobile vision applications. Chapter 7 summarizes the contributions of this thesis and examines avenues for future work.

Chapter 7
Conclusions And Future Work

The wheel is come full circle.

King Lear
William Shakespeare

7.1 Thesis Summary

This thesis has presented a novel and comprehensive approach to improving the performance computer vision codes in mobile systems. This work begins with presenting a novel mobile vision system and analyzing the computational characteristics of mobile vision. Then it develops a set of novel solutions for improving the performance of most mobile vision systems.

This thesis work has two major components. The first is characterizing and understanding mobile vision applications. We developed two solutions for this. We then utilized this characterization to develop novel solutions to improve mobile vision performance. Our novel solutions that allowed us to characterize the mobile vision space include:

- **MVSS:** The Michigan Visual Sonification System utilizes computer vision to convert a scene into audio signatures that are played using 3D audio rendering to allow the visually impaired to recognize and localize objects. The system utilizes many common components of computer vision including feature extraction, 3D reconstruction, and feature classification to provide this capability. Our experiments show that after proper training, users are capable of accuracies over 70% in recognizing object categories. However, we also demonstrate that real time performance is not possible with current mobile processors.

- **MEVBench:** MEVBench is a multithreaded benchmark suite for mobile vision application. This is the first benchmark suite targeted specifically for mobile vision systems; it provides researchers and developers with multithreaded representative implementations of twelve common applications and algorithms. Our evaluation showed that vector reduction operations and 2D locality are frequent in vision codes.

Our analysis of mobile vision applications allowed us to develop novel software and hardware techniques for improving mobile vision performance. Our solutions for improving the performance of the mobile vision space include:

- **SEVS:** The Singular Eigenvector Solver is an efficient technique for solving for a singular value pair. Solving for singular pairs is common in vision codes that include augmented reality and camera calibration. Experimental results showed this technique reduces computation by over 30% with comparable error and numerical stability to traditional techniques.
- **EFFEX:** The Effecient Fast Feature Extraction architecture is a heterogeneous multicore architecture that utilizes 2D spatial locality and vector reduction operations to improve the performance of the key mobile vision operation of feature extraction. Our results demonstrate a 90x improvement over comparable mobile systems for three common feature extraction algorithms.
- **EVA:** The Efficient Vision Architecture provides vector reduction accelerators for the complete vision software pipeline. It utilizes a heterogeneous multicore topology with accelerators in each core for common operations that take place in feature extraction, feature classification and application reasoning. The architecture also uses a caching system capable of providing both 1D and 2D spatial locality for memory accesses. The hardware features provide an additional improvement of 8x compared to EFFEX.

This work provides an analysis of key techniques for improving mobile vision. Through in-depth analysis, this work has exposed the need for mobile processors to support diverse parallelism and multiple forms of memory access locality. The diverse parallelism present in mobile vision systems ranges from data parallel vector reduction operations to task level parallelism. Furthermore, we have demonstrated that due to the nature of mobile vision, there is a workload imbalance present in mobile vision systems. Mobile systems can exploit

this imbalance in task-level parallelism to efficiently provide support for these applications. We have developed software and hardware techniques to take advantage of our understanding of mobile vision systems. We have shown that our novel solutions provide better mobile vision performance while utilizing less energy than current mobile processor.

7.2 Future Research Directions

While this dissertation has made great strides toward improving the capability of mobile vision systems, it also exposes many new research questions. The exploration of these ideas will improve the capability of current mobile vision systems and enable new systems in the future as well.

Chapter 2 introduces the MVSS. This system is a prototype that opens the door to many future research opportunities. The current system is capable of providing the user with the object category and location but the specifics of the object are lost. For example, a user does not know the color of the object. Can the sonification signal be augmented with object specific information to allow the users to accurately identify unique objects in a given category without hindering general category recognition? The current technique outputs each individual signature in a raster scan order based on the centroid of the region in the frame. Can humans track multiple simultaneous signatures with continuous output for as long as the object exists in the camera stream? There are still questions about how the human brain adapts to the sonification system and what processes are involved. What is this process and can we utilize it to develop optimized audio signature generation techniques based on this process? Also the system locates the objects near the user, but it would be useful to add in structural information such as where walls are or what material the floor is made of. This extension could greatly improve the usability of MVSS, but will require research into how to discern this from the scene and how to convert it to audio.

The analysis performed by MEVBench introduced in Chapter 3 could also be extended for future research directions. One key question that the multithreaded benchmark suite leads to is the optimal method to divide the work among the various threads in a given benchmark. The current implementation splits the input into equal spatial regions and assigns a thread to a region. Based on the underlying processor interconnect and cache configurations, this may not be the optimal split. Can we determine an optimal split based on the memory system to optimize inter-thread communication? The benchmark suite as a whole can be grown to include emerging mobile vision applications such as HDR and image based localization (visual odometry).

Chapter 4 introduced the SEVS technique for optimizing the computation of a small number singular pairs of a matrix. This technique was developed for the use in tasks, such as homography estimation and structure from motion. SEVS can potential be further optimized based on computation platform. What architectural features can SEVS take advantage of? SEVS was developed because we recognized that the libraries and techniques commonly used to compute singular values produce more results than are required, thus we modified the technique. Are there other areas in vision where the computation does not fully match the desires of the developer and can we utilize our understand of what is truly required to increase efficiency? This opens the door for further software and algorithm optimizations in the mobile vision space. This can lead to new features being viable in the low computation environment of mobile systems.

We explore the use of hardware based methods to improve the performance of mobile vision systems in Chapters 5 & 6. These solutions are based on common operations and structures that occur through the mobile vision pipeline. The vector reduction operations are utilized to take advantage of the data parallelism in the application however there is still the question of the optimal size of the various accelerators developed. Both EFFEX and EVA have their sizes based on the size found commonly in the algorithms. What is the the optimal size to balance the static energy usage with the memory bandwidth to keep them full operational? Furthermore, what benefits can an application see when power gating the accelerators? There are also possible uses for the accelerators outside of the mobile vision hardware. Can the accelerators such as the tree comparator be utilized to improve the capability of server based recommender systems? Can these accelerators be applied to DSP and GPGPU systems to improve their performance? The application of these structures to other areas has a large amount of potential.

The improved performance that these works create allow for more efficient mobile vision processing which opens the door to new applications and systems. A common feature of these mobile systems is a wireless data connection that gives access to the cloud for potential server side computing. Given the enhanced performance of the mobile system, what is the best way to partition the workload between the cloud and the mobile system? Even within the system, given the better capability of EVA and EFFEX, what is the best way to partition the workload between the mobile GPU, DSP and main application processor? There is also the question of the optimal interconnect for mobile devices running these workloads and utilizing their structure to create improved coordination between the devices in the system. We have scratched the surface of the research opportunities in the realm of improving and creating new mobile vision systems.

7.3 Conclusion

The analysis and solutions presented here improve the performance of mobile vision applications while lowering the energy consumption. We demonstrated that future mobile processors must taken advantage of the various forms of parallelism present in mobile vision applications in order to efficiently provide adequate performance to applications such as visual sonification, and efficiency is well served by application-specific algorithmic optimizations, hardware accelerators, and memory interfaces. It is our hope that through this effort the increased performance and efficiency will refine visual sonification and new applications will also be developed to improve the lives of us all.

Bibliography

[1] Hewlett-Packard Development Company. HP TouchPad. http://www.hp.com/global/webos/ca/en/shopping-touchpad.html, 2011.

[2] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. The Frankencamera: an experimental platform for computational photography. In *SIGGRAPH*, 2010.

[3] Chloe Albanesius. Apple: Kindle Fire Could Help iPad Sales. *PC Magazine*, December 2011.

[4] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, AFIPS, 1967.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[6] Andrew Rassweiler. iPad 2 Carries Bill of Materials of 326.60 IHS iSuppli Teardown Analysis Shows. http://www.isuppli.com/teardowns/news/Pages/Default.aspx, 2011.

[7] Apple. Apple. http://www.apple.com/, 2011.

[8] ARM. ARM Achieves 10 Billion Processor Milestone. http://www.arm.com/about/newsroom/19720.php, 2008.

[9] ARM. Cortex-A5 Processor, 2010. http://www.arm.com.

[10] ARM. Cortex-A8 Processor, 2010. http://www.arm.com.

[11] ARM. ARM NEON. http://www.arm.com/products/processors/technologies/neon.php, 2011.

[12] ARM. Cortex-A9 Processor. http://arm.com/products/processors/cortex-a/cortex-a9.php, 2011.

[13] ARM. ARM big.Little, 2012.

[14] ARM. Cortex-A9 Processor, 2012. http://www.arm.com.

[15] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *CVIU*, 2008.

[16] Berkeley Design Technology Inc. ARM Announces 2GHz Dual Core Cortex A9. http://www.bdti.com/InsideDSP/2009/09/23/Arm, 2011.

[17] Christian Bienia and Kai Li. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[18] Bill Detwiler. Cracking open the HP TouchPad. http://www.techrepublic.com/photos/cracking-open-the-hp-touchpad/6253940, 2011.

[19] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[20] Dr. Gary Rost Bradski and Adrian Kaehler. *Learning OpenCV*. O'Reilly Media, Inc., 2008.

[21] Christoph Bregler, Aaron Hertzmann, and Henning Biermann. Recovering non-rigid 3d shape from image streams. In *CVPR*, pages 2690–2696. IEEE Computer Society, 2000.

[22] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[23] Brooke Crothers. Getting a look inside the iPhone 4. http://news.cnet.com/8301-13924_3-20008527-64.html, 2010.

[24] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[25] M. Calonder, V. Lepetit, C. Strecha, and P. Fua. BRIEF: Binary Robust Independent Elementary Features. In *European Conference on Computer Vision*, 9 2010.

[26] Leonardo Chang and José Hernández-Palancar. A Hardware Architecture for SIFT Candidate Keypoints Detection. In *CIARP*, 2009.

[27] Jie Chen, Ling-Yu Duan, Rongrong Ji, Hongxun Yao, and Wen Gao. Sorting local descriptors for lowbit rate mobile visual search. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 1029 –1032, may 2011.

[28] Tien-fu Chen and Jean-loup Baer. Effective Hardware-based Data Prefetching for High-performance Processors. *IEEE Transactions on Computers*, 1995.

[29] Kwang-Ting Cheng and Yi-Chu Wang. Using mobile gpu for general-purpose computing: a case study of face recognition on smartphones. In *VLSI-DAT*, 2011.

[30] Roger Cheng. How augmented reality is an opportunity for developers (Inside Apps). *Cnet.com*, October 2011.

[31] J. Clemons, S.Y. Bao, S. Savarese, T. Austin, and V. Sharma. Mvss: Michigan visual sonification system. In *ESPA*, 2012.

[32] Jason Clemons, Andrew Jones, Robert Perricone, Silvio Savarese, and Todd Austin. Effex: an embedded processor for computer vision based feature extraction. In *DAC*, 2011.

[33] Jason Clemons, Haishan Zhu, Silvio Savarese, and Todd Austin. Mevbench: A mobile computer vision benchmarking suite. In *IISWC*, 2011.

[34] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[35] Roberta Cozza. Forecast: Mobile Communications Devices by Open Operating System, Worldwide, 2008-2015. *Gartner*, April 2011.

[36] Creative. OpenAL 1.1, June 2009. `http://www.openal.org`.

[37] G. Csurka et al. Visual categorization with bags of keypoints. In *ECCV*, pages 1–22, 2004.

[38] David E. Culler, Janswinder Pal Singh, and Anoop Gputa. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[39] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *CVPR*, 2005.

[40] Dell Inc. XPS 8300. `http://www.dell.com`, 2011.

[41] Clay Dillow. I am warplane. *Popular Science*, August 2012.

[42] Mark Donovan. The 2010 Mobile Year in Review - U.S. `http://www.comscore.com/Insights/Presentations_and_Whitepapers/2011/2010_Mobile_Year_in_Review_-_U.S`, March 2011.

[43] EEE Journal. ARM11 vs Cortex A8 vs Cortex A9 - Netbooks processors. `http://www.eeejournal.com/2009/12/arm11-vs-cortex-a8-vs-cortex-a9.html`, 2009.

[44] EEMBC. CoreMark: An EEMBC Benchmark. `http://www.coremark.org/home.php`, 2011.

[45] EEMBC. MultiBench 1.0 Multicore Benchmark Software. http://www.eembc.org/benchmark/multi_sl.php, 2011.

[46] Markus Enzweiler and Dariu M. Gavrila. Monocular pedestrian detection: Survey and experiments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31:2179–2195, 2009.

[47] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (voc) challenge. *International Journal of Computer Vision*, 88(2):303–338, June 2010.

[48] L. Fei-Fei and P. Perona. A bayesian hierarchical model for learning natural scene categories. In *CVPR*, 2005.

[49] Yoav Freund and Robert E. Schapire. Experiments with a new boosting algorithm. In *ICML*. Morgan Kaufmann, 1996.

[50] James Fung and Steve Mann. Openvidia: parallel gpu computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia*, MULTIMEDIA '05, New York, NY, USA, 2005. ACM.

[51] Sameh Galal and Mark Horowitz. Energy-efficient floating-point unit design. *IEEE Trans. Comput.*, 60(7):913–922, July 2011.

[52] Geekbench. Geekbench Results Browser. http://browse.geekbench.ca/, 2011.

[53] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial & Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[54] G.H. Golub and C.F. Van Loan. *Matrix computations*, volume 3. Johns Hopkins University Press, 1996.

[55] Google. Google Goggles For Android, September 2010. http://www.google.com/mobile/goggles/#text.

[56] Google. Nexus Galaxy Tech Specs. http://www.google.com/nexus/#/tech-specs, 2011.

[57] Google. Phone Gallery: Nexus One with Google. http://www.google.com/phone/detail/nexus-one, 2011.

[58] Google. Phone Gallery: Nexus S with Google. http://www.google.com/phone/detail/nexus-s, 2011.

[59] Saurabh Goyal. Object Detection using OpenCV II - Calculation of HoG Features, October 2009. http://smsoftdev-solutions.blogspot.com/2009/10/object-detection-using-opencv-ii.html.

[60] P.S. Green, J.W. Hill, J.F. Jensen, and A. Shah. Telepresence surgery. *Engineering in Medicine and Biology Magazine, IEEE*, 14(3):324 –329, may/jun 1995.

[61] Gregg Keizer. iPhone 4S teardowns reveal A5 processor. http://www.computerworld.com/s/article/9220877/iPhone_4S_teardowns_reveal_A5_processor, 2011.

[62] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *MICRO*, 26(2), 2006.

[63] Gumstix Inc. Overo Fire COM. https://www.gumstix.com/store/product_info.php?products_id=227, 2011.

[64] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE International Symposium on Workload Characterization*, 2001.

[65] Matthew Harker and Paul O'Leary. Computation of homographies. In *BMVC*, 2005.

[66] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[67] Randy W. Heiland, M. Pauline Baker, and Danesh K. Tafti. Visbench: A framework for remote data visualization and analysis. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '01, pages 718–727, London, UK, UK, 2001. Springer-Verlag.

[68] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2003.

[69] R. Hess. SIFT feature detector for OpenCV, February 2009. http://web.engr.oregonstate.edu/~hess.

[70] Heiko Hirschmller and Daniel Scharstein. Evaluation of cost functions for stereo matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2007.

[71] L. Hogben. *Handbook of linear algebra*. Chapman & Hall/CRC, 2006.

[72] M. Ibarra-Manzano and D. Almanza-Ojeda. Design and Optimization of Real-Time Boosting for Image Interpretation Based on FPGA Architecture. In *CERMA*, 2011.

[73] ifixit. iPhone 3GS Teardown. http://www.ifixit.com/Teardown, 2011.

[74] Intel. Math kernel library. http://developer.intel.com/software/products/mkl/.

[75] Intel. Intel vtune xe performance analyzer, 2011. http://www.intel.com/VTuneAmplifier.

[76] Texas Intruments. VLIB 2.0: Video Analytics And Vision Library, December 2008. http://www.ti.com/lit/ml/sprt502a/sprt502a.pdf.

[77] Texas Intruments. Blaze Development Platform, September 2011. http://omapworld.com/blaze.html.

[78] Ilse C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Review*, 39:254–291, 1997.

[79] S. Jain, V. Erraguntla, S.R. Vangal, Y. Hoskote, N. Borkar, T. Mandepudi, and V.P. Karthik. A 90mW/GFlop 3.4GHz Reconfigurable Fused/Continuous Multiply-Accumulator for Floating-Point and Integer Operands in 65nm. In *VLSID*, Jan. 2010.

[80] E.R. Jessup and I.C.F. Ipsen. Improving the accuracy of inverse iteration. *SIAM journal on scientific and statistical computing*, 13(2):550–572, 1992.

[81] Luo Juan and Oubong. Gwun. A comparison of sift, pca-sift and surf. *International Journal of Image Processing*, 3(4):143–152, 2010.

[82] R. Kadota, H. Sugano, M. Hiromoto, H. Ochi, R. Miyamoto, and Y. Nakamura. Hardware Architecture for HOG Feature Extraction. In *IIH-MSP*, sep. 2009.

[83] J. M. Kahn, R. H. Katz, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for ”smart dust”, 1999.

[84] Takeo Kanade and Daniel D. Morris. Factorization methods for structure from motion. *Philosophical Transactions of the Royal Society of London, Series A*, 356(1740):1153–1173, 1998.

[85] Hirokazu Kato and Mark Billinghurst. Marker tracking and hmd calibration for a video-based augmented reality conferencing system. In *Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*, pages 85–, Washington, DC, USA, 1999. IEEE Computer Society.

[86] Guy-Richard Kayombya. SIFT feature extraction on a Smartphone GPU using OpenGL ES2.0. Master’s thesis, Massachusetts Institute of Technology, 2010.

[87] Joo-Young Kim and Hoi-Jun Yoo. Bitwise competition logic for compact digital comparator. In *Proceedings of the IEEE Asian Solid States Circuits Conference*, 2007.

[88] Georg Klein and David Murray. Parallel tracking and mapping on a camera phone. In *Proc. Eigth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR’09)*, Orlando, October 2009.

[89] J. Koppanalil, G. Yeung, D. O’Driscoll, S. Householder, and C. Hawkins. A 1.6 GHz dual-core ARM Cortex A9 implementation on a low power high-K metal gate 32nm process. In *VLSI-DAT*, 2011.

[90] Suzanne LaBarre. CES 2013: Lexus Unveils Autonomous Safety Research Car. http://www.popsci.com/cars/article/2013-01/ces-2013-lexus-unveils-autonomous-safety-research-car, January 2013.

[91] Layar. Layar Reality Browser, September 2010. http://www.layar.com/.

[92] K. Levenberg. A method for the solution of certain nonlinear problems in least squares. *Quarterly of Applied Mathematics*, 1944.

[93] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[94] D. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[95] Imagination Technologies Ltd. SGX Graphics IP Core Family, 2010.

[96] BruceD. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision (ijcai). In *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI '81)*, pages 674–679, April 1981.

[97] Steve Mann, Raymond Lo, Jason Huang, Valmiki Rampersad, and Ryan Janzen. Hdrchitecture: real-time stereoscopic hdr imaging for extreme dynamic range. In *SIGGRAPH*, 2012.

[98] John Markoff. Google Cars Drive Themselves, in Traffic. *New York Times*, October 2010.

[99] D. Marr and T. Poggio. *Cooperative computation of stereo disparity*, pages 259–267. MIT Press, Cambridge, MA, USA, 1988.

[100] Kyla A. McMullen. *Interface Design Implications for Recalling the Spatial Configuration of Virtual Auditory Environments*. PhD thesis, University of Michigan, Ann Arbor, 2012.

[101] P.B.L. Meijer. An experimental system for auditory image representations. *IEEE Transactions on Biomedical Engineering*, 1992.

[102] Mentor Graphics. Sourcery CodeBench. http://www.mentor.com/embedded-software/codesourcery, 2011.

[103] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA*, January 2010.

[104] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[105] M. Murphy, K. Keutzer, and Hong Wang. Image feature extraction for mobile processors. In *IISWC*, oct. 2009.

[106] R.A. Newcombe, S. Lovegrove, and A.J. Davison. Dtam: Dense tracking and mapping in real-time. In *Proc. of the Intl. Conf. on Computer Vision (ICCV), Barcelona, Spain*, volume 1, 2011.

[107] NotebookCheck. Intel Core i5 430M. `http://www.notebookcheck.net/Intel-Core-i5-430M-Notebook-Processor.23750.0.html`, 2011.

[108] NVIDIA. Variable SMP A Multi Core CPU Architecture for Low Power and High Performance. `http://www.nvidia.com/object/white-papers.html`.

[109] NVIDIA. Tegra 2, June 2011. `http://www.nvidia.com/-object/tegra-2.html`.

[110] OpenCV.org. Opencv platforms: Cuda, November 2012. `http://opencv.org/platforms/cuda.html`.

[111] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. Marss: A full system simulator for multicore x86 cpus. In *DAC*, 2011.

[112] Conrad Poelman and Takeo Kanade. A paraperspective factorization method for shape and motion recovery. Technical Report CMU-CS-93-219, Computer Science Department, Pittsburgh, PA, December 1993.

[113] Adam Prengler and Ketaki Adi. A Reconfigurable SIMD-MIMD Processor Architecture for Embedded Vision Processing Applications. *SAE World Congress*, 2009.

[114] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[115] Victor Prisacariu and Ian Reid. fastHOG - a real-time GPU implementation of HOG. Technical Report 2310/09, Department of Engineering Science, Oxford University, 2009.

[116] Jingbang Qiu, Ying Lu, Tianci Huang, and Takeshi Ikenaga. An FPGA-Based Real-Time Hardware Accelerator for Orientation Calculation Part in SIFT. *IIH-MSP*, 2009.

[117] Qualcomm. Fastcv. `https://developer.qualcomm.com/mobile-development/mobile-technologies/computer-vision-fastcv`.

[118] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. Computational sprinting. In *HCPA*, 2012.

[119] Michael Rayfield. Tegra Roadmap Revealed: Next Chip To Be Worlds First Quad-Core Mobile Processor. http://blogs.nvidia.com/2011/02/.

[120] Edward Rosten and Tom Drummond. Fusing points and lines for high performance tracking. In *ICCV*, volume 2, October 2005.

[121] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *ECCV*, May 2006.

[122] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: An Efficient Alternative to SIFT or SURF. In *ICCV*, 2011.

[123] Samsung Inc. Samsung Epic 4G Android Smartphone. http://www.samsung.com/us/mobile/cell-phones/SPH-D700ZKASPR, 2011.

[124] S. Savarese and L. Fei-Fei. 3D generic object categorization, localization and pose estimation. In *ICCV*, 2007.

[125] J. Shi and J. Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8):888–905, 2000.

[126] Jianbo Shi and Carlo Tomasi. Good features to track. In *1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, pages 593 – 600, 1994.

[127] D. Shire et al. Development and implantation of a minimally invasive wireless subretinal neurostimulator. *IEEE Transactions on Biomedical Engineering*, 2009.

[128] B.V.N. Silpa, A. Patney, T. Krishna, P.R. Panda, and G.S. Visweswaran. Texture filter memory; a power-efficient and scalable texture memory architecture for mobile graphics processors. In *ICCAD*, 2008.

[129] Kevin Sintumuang. Is the iPhone the Only Camera You Need? http://online.wsj.com/article/SB10001424052702303816504577305702578426084.html, April 2012.

[130] J. Skribanowitz, T. Knobloch, J. Schreiter, and A. Konig. VLSI implementation of an application-specific vision chip for overtake monitoring, real time eye tracking, and automated visual inspection. In *MicroNeuro '99*, 1999.

[131] Noah Snavely, Steven M. Seitz, and Richard Szeliski. Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings*, pages 835–846, New York, NY, USA, 2006. ACM Press.

[132] Richard Szeliski. Image alignment and stitching: a tutorial. *Found. Trends. Comput. Graph. Vis.*, 2:1–104, January 2006.

[133] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *ISCA*, 2008.

[134] Carlo Tomasi. Shape and motion from image streams under orthography: a factorization method. *International Journal of Computer Vision*, 9:137–154, 1992.

[135] Tom's Hardware. ALU Performance: SiSoftware Sandra 2010 Pro (ALU). `http://www.tomshardware.com/charts/desktop-cpu-charts-2010/ALU-Performance-SiSoftware-Sandra-2010-Pro-ALU,2408.html`, 2011.

[136] Lorenzo Torresani, Aaron Hertzmann, and Christoph Bregler. Nonrigid structure-from-motion: Estimating shape and motion with hierarchical priors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(5):878–892, 2008.

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

[138] A. Vedaldi. SIFT++, June 2006. http://www.vlfeat.org/˜vedaldi/code/siftpp.html.

[139] Sravanthi Kota Venkata, Ikkjin Ahn, Donghwan Jeon, Anshuman Gupta, Christopher Louie, Saturnino Garcia, Serge Belongie, and Michael Bedford Taylor. Sd-vbs: The san diego vision benchmark suite. In *IEEE International Symposium on Workload Characterization*, pages 55–64, 2009.

[140] Paul Viola and Michael Jones. Robust real-time object detection. *IJCV*, # 2002.

[141] Paul Viola and Michael Jones. Robust real-time object detection. In *International Journal of Computer Vision*, 2001.

[142] WHO and IAPB. Global Initiative for the Elimination of Avoidable Blindness: Action Plan 2006-2011. Technical report, World Health Organization, 2007.

[143] J. H. Wilkinson, editor. *The algebraic eigenvalue problem*. Oxford University Press, Inc., New York, NY, USA, 1988.

[144] Martyn Williams. Nvidia unveils Tegra 4 processor, first quad-core Cortex-A15 chip. `http://www.infoworld.com/d/mobile-technology/nvidia-unveils-tegra-4-processor-/first-quad-core-cortex-a15-chip-210195`, January 2013.

[145] J. Wilson et al. Swan: System for wearable audio navigation. In *IEEE International Symposium on Wearable Computers*, 2007.

[146] Jenna Wortham. Customers Angered as iPhones Overload ATT. *New York Times*, September 2009.

[147] Changchang Wu. SIFTGPU, September 2010. `http://www.cs.unc.edu/˜ccwu/siftgpu/`.

[148] X. Yang and K.T.T. Cheng. Accelerating surf detector on mobile devices. In *ACM Multimedia Conference*, 2012.

[149] Lifan Yao, Hao Feng, Yiqun Zhu, Zhiguo Jiang, Danpei Zhao, and Wenquan Feng. An architecture of optimised sift feature detection for an fpga implementation of an image matcher. In *FPT*, 2009.

[150] Jones Yudi Mori, Daniel Muñoz Arboleda, Janier Arias Garcia, Carlos Llanos Quintero, and José Motta. Fpga-based image processing for omnidirectional vision on mobile robots. In *SBCCI*, 2011.