ANALYSIS OF PARALLEL SOC ARCHITECTURAL CHARACTERISTICS FOR ACCELERATING FACE IDENTIFICATION

by

Ralph Sprang

B.S.E.E, Ohio State University, 1982

M.S.E.E, The Johns Hopkins University, 1989

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2012

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Ralph Sprang

It was defended on

April 4, 2012

and approved by

Alex P. Jones, PhD, Associate Professor, Department of Electrical and Computer Engineering

Steven P. Levitan, PhD, Professor, Department of Electrical and Computer Engineering

Marlin H. Mickle, PhD, Professor, Department of Electrical and Computer Engineering

Mark P. Mooney, PhD, Professor, Departments of Oral Biology, Anthropology, Surgery-Plastic Surgery, Orthodontics, and Communication and Speech Disorders

Raymond R. Hoare, PhD, CEO, ConcurrentEDA, Inc.

Dissertation Director: James T. Cain, PhD, Professor Emeritus, Department of Electrical and Computer Engineering

ANALYSIS OF PARALLEL SOC ARCHITECTURAL CHARACTERISTICS FOR ACCELERATING FACE IDENTIFICATION

Ralph Sprang, PhD

University of Pittsburgh, 2012

Growing worldwide concerns about terrorism have increased interest in rapidly and accurately identifying individuals such as potential terrorists. The ability to quickly screen an individual against the more than one million entries on the Terrorist Watch List using face identification could significantly improve national security and other security screening applications.

Top accuracy face identification algorithms are not real-time. The top face identification algorithms evaluated in National Institutes of Standards (NIST) testing achieve 95% or greater identification accuracy but require several minutes to complete identification on a 1,196 member gallery set of 100 kilopixel resolution images. Recent testing shows that face identification algorithms are significantly slower for current NIST test sets with a 14,365 member gallery set of 4 megapixel images. Significant performance improvement is needed to match a one million member gallery set.

The International Technology Roadmap for Semiconductors projects Systems on a Chip with more than one thousand processors will be available within ten years. However, it’s not clear how face identification algorithms can use these massively parallel SOCs to improve performance or which architectural characteristics are important for these algorithms.

This research specifies key architectural characteristics for a massively parallel SOC to enable real-time face identification. A set of face identification benchmarks has been created to guide this research and includes small and large image data sets. This research contributes a method to explore the SOC design space to evaluate the final SOC performance. Specifically, this research is focused on the impact of processor instruction set architecture performance, the external memory bandwidth, the quantity of processing cores, the on-chip communication network, and the mapping of the face identification benchmarks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

PREFACE

Words cannot sufficiently express my gratitude for all the help and support provided by my advisors. I have been tremendously blessed to study and research under the careful guidance of truly wise and gifted men, and I am truly grateful for the support, encouragement, insight, and wisdom they shared with me over the many years of research that ultimately led to this dissertation. Thank you, Dr. Cain and Dr. Hoare.

I am also grateful for the knowledge and support shared by my committee members. Dr. Mickle's Computer Architecture class inspired a quest for further knowledge, helped lead to my topic, and his thoughtful discussion and guidance led me to my dissertation advisors. Dr. Levitan's SOC class, direct feedback on my work, and further discussions with him helped focus my topic on SOC applications. Dr. Jones graciously provided office and lab space and support for much of this work and inspired an interest in design automation. Dr. Mooney introduced me to the breadth of applications for Principal Component Analysis and his insights helped me understand that the applications of this work extend beyond the field of face identification. Thank you all for your support, wisdom, and encouragement.

I am grateful to Dr. Pierre LaFrance, my Master's advisor, who nurtured my budding curiosity into a research interest. It has taken many years to complete this journey, but the journey would have never begun without his encouragement and support.

I am deeply grateful to my wife for the patience and support she has extended to me over these many years. It has been a long and challenging journey, and she has been my constant supporter, encourager, and cheerleader. Thank you, my love.

Finally, to the One who set me on this path, opened the door to make all this possible, and is my constant encouragement and hope, I offer my humble praise and gratitude.

1.0 INTRODUCTION

Terrorism and other threats to national security are increasing worldwide [1]. This growth in terrorism motivates efforts to develop effective technologies to identify terrorists before they can attack. The means to quickly and accurately identify potential terrorists posing as airline passengers is one critical requirement to increase the safety of air travel.

Airline passenger safety can be improved if passengers can be quickly screened against lists of terrorists. The Terrorist Watch List contains more than one million names [2] and screening airline passengers against databases of known terrorists could prevent terrorists from boarding airplanes. If the ability to quickly screen an individual against lists of known terrorists can be developed, airline passenger safety can be improved.

Face identification algorithms determine the identity of an individual from a photographic image of their face. Acquiring an image of a person's face is a simple process that requires minimal cooperation from the individual and does not require physical contact with the subject. This face image can be used to identify the individual, perhaps even if they are wearing a disguise or otherwise attempting to avoid identification [3, 4].

Face identification compares a *probe* or unknown face image against a *gallery* set of known face images to determine whether the unknown face is contained in the gallery set and, if so, which gallery member best matches the probe. Face identification is a technology that could be used to screen airline passengers against the Terrorist Watch List.

Current computer-based face identification algorithms can identify a probe with greater accuracy and with larger gallery sets than a human observer. Current face identification algorithms can achieve 87% to 96% accuracy in selecting the best match for the probe face from a large gallery set, while a human observer performing a pair comparison between two images achieves accuracy in the range of 75%, even for “easy” matches [5, 6]. Computer-based face identification algorithms can compare a probe face to a gallery set of thousands of images, while human observers are limited to a gallery with a few tens of images [5]. Furthermore, the accuracy of machine identification is 25% greater than the accuracy humans can achieve for difficult matching tasks [5]. For nearly all security applications, computer-based face identification algorithms provide better accuracy and can perform identification with a larger gallery set than a human observer.

1.1 FACE IDENTIFICATION IS NOT REAL-TIME

Identification time in the range of two minutes is required for real-time identification. *Identification time*, the elapsed time required to compare one probe image to each member in a gallery set and determine the best match, must be in the range of a few minutes for humans to consider it “real time”. A real-time goal of two minutes can be inferred from the airport passenger screening example. The elapsed time for a passenger to complete the screening process is a minimum of three minutes for a small airport during off-peak times [7], and can be much longer for busier airports at peak times. Photographing each passenger as they enter the screening area and identifying them within two minutes would ensure the reporting of results to security personnel before the passenger completed the screening process. The real-time goal is therefore set at two minutes for this research.

Computer-based face identification algorithms that achieve top accuracy in NIST testing are not real-time for one million member gallery sets of high resolution images. NIST Face Recognition Technology (FERET)[1] [8] testing used a relatively small gallery set of 1,196 faces with 98,304 pixel resolution and several algorithms did not achieve real-time performance even for this small gallery set [8]. Furthermore, the top face identification algorithm achieved accuracy in the range of 87% to 96% [8] but required 6.6 minutes to identify a single probe [9]. Identification time increases with both image resolution and gallery size, and top accuracy face identification algorithms are not real-time for large gallery sets such as the one million name Terrorist Watch List [2] with the four megapixel images in current NIST test sets [10].

Face identification algorithms can achieve greater identification accuracy with higher resolution face images but identification time also increases. Increasing the resolution of face images from the 100 kilopixel images used in NIST FERET [8] testing to the four megapixel images in current NIST testing [10] can increase identification accuracy by 20% to 30% [11] but substantially increases the amount of computation required, further compounding the real-time problem.

[1] Portions of the research in this paper use the FERET database of facial images collected under the FERET program, sponsored by the DOD Counterdrug Technology Development Program Office.

1.2 THOUSANDS OF PROCESSORS ON A CHIP WITHIN A DECADE

A *System on a Chip* (SOC) with a thousand cores could be available within the next decade. Within the next ten years, technology advances will allow fabrication of hundreds to thousands of processor cores on a chip according to the Semiconductor Industry Association's *International Technology Roadmap for Semiconductors* (ITRS), an annual report that projects trends in the semiconductor industry (Figure 1) [12]. A thousand cores on an SOC could potentially provide the performance improvement necessary to make face identification real-time for one million member gallery sets of four megapixel images, if techniques to use these processor cores can be developed.

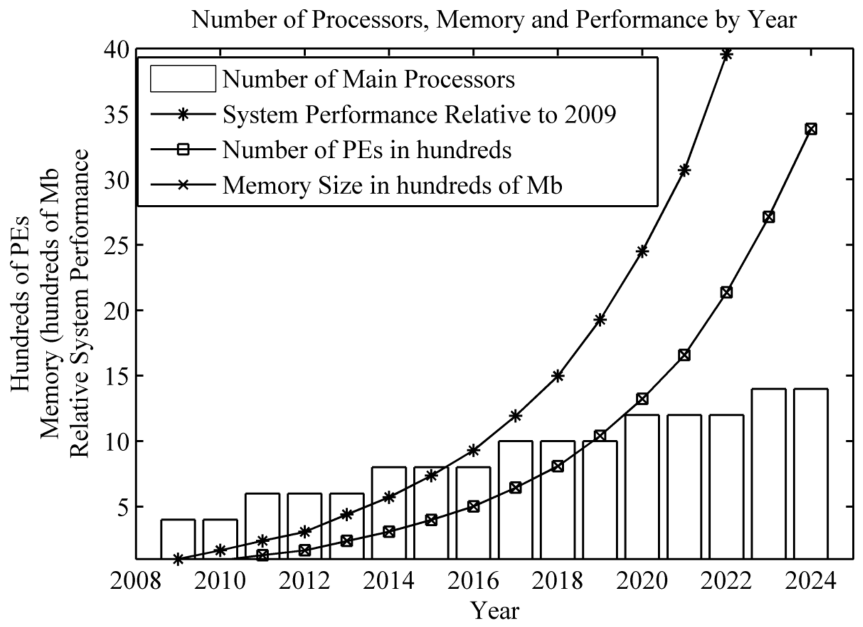


Figure 1: ITRS projects thousands of processors on a chip within ten years.

1.3 DESIGN SPACE FOR MASSIVELY PARALLEL SOCS

To achieve the acceleration required to make face identification real-time, five key questions must be answered:

1. What Instruction Set Architecture will decrease program execution time for face identification?

2. What external memory bandwidth is required to avoid data starvation on the SOC processor cores?

3. How many processor cores can a face identification algorithm use to improve performance?

4. What on-chip communication latency and bandwidth are required to avoid creating bottlenecks or data starving the processor cores?

5. How can algorithms be mapped to thousands of processors?

According to David Patterson, the problem of how to automatically map sequential algorithms to architectures with hundreds or thousands of processors is currently unsolved and may require decades of research to solve [13, 14]. However, some specific problems have been successfully accelerated using parallel architectures and Patterson thinks that other specific problems can be solved in the near term [14]. If design techniques can be developed to map sequential face identification algorithms to thousand core SOCs, real-time face identification for one million member gallery sets of four megapixel images becomes possible.

If parallelism in face identification algorithms can be exposed and exploited, real-time identification may be achievable. A speedup of at least 100 times should be achievable with a thousand processors and a speedup of 512 times or more may be possible for some algorithms [15]. For example, current architectures such as the Graphics Processor Unit (GPU) achieve acceleration of 100 to 1,000 times for certain specific types of algorithms [16, 17], and the Tile64, a research architecture containing 64 processors, achieves a 100 times speedup for some applications [18]. If enough processors are available, real time face identification with a one million member gallery set of four megapixel images may be possible.

2.0 STATEMENT OF THE PROBLEM

Real-time face identification is possible on sequential processors for small gallery sets of low resolution images. When image resolution and gallery size increase, however, real-time performance is not achieved.

The goal of this research is to determine how thousands of processor on a chip can be used to make face identification real-time for high-resolution images and large gallery sets. For many applications such as passenger screening at airports, identification time in the range of two minutes is required. While sequential algorithms achieve identification time of less than a minute for small gallery sets of low resolution images, Table 1 shows these algorithms cannot identify large gallery sets of high resolution images in real-time. To achieve real-time identification for a million member gallery set of four megapixel images requires a speedup ranging from 36 to 748 times.

This research explores the design space for massively parallel processors on a chip for face ID algorithms to answer four key questions:

- Which types of processors are needed?
- How much external memory bandwidth is needed?
- How many processors can be used to accelerate the three top face id algorithms?
- What type of on-chip communication is required?

Table 1: Required Speedup for Real-Time Face Identification.

Benchmark	Image Resolution	Gallery Size	Execution Time Estimate (Minutes)	Required Speedup
Eigenface 100 KP, 1K	100 kilopixel	1,196	0.0015	-
Eigenface 4 MP, 14K	4 megapixel	14,365	0.6800	-
Eigenface 4 MP, 1M	4 megapixel	1,000,000	75.7150	38
Bayesian 100 KP, 1K	100 kilopixel	1,196	0.00146	-
Bayesian 4 MP, 14K	4 megapixel	14,365	0.6791	-
Bayesian 4 MP, 1M	4 megapixel	1,000,000	71.169	36
EBGM 100 KP, 1K	100 kilopixel	1,196	3.70	2
EBGM 4 MP, 14K	4 megapixel	14,365	1,495.16	748
EBGM 4 MP, 1M	4 megapixel	1,000,000	1,496.16	748

Face identification is a computationally intensive process that is not real-time for a one million member gallery set of four megapixel images. Performance estimates show that identification time for four megapixel probe image with a gallery set of one million members can range from 1.18 to 24.9 hours. For many applications, identification time in the range of two minutes is required to be considered real-time and top accuracy sequential face identification algorithms do not achieve this goal for one million member gallery sets of four megapixel images.

Face identification requires two steps, encoding of the probe and comparison of the probe to the gallery set. The probe is first encoded to reduce the data storage requirements and emphasize the data required to differentiate face images, and then compared to each encoded gallery member through a series of pair comparisons.

The pair comparison process is obviously parallel, but the encoding process consumes more than 80% of the identification time for one million member gallery sets of four megapixel images. The probe to gallery comparison process is a series of comparisons between probe-gallery pairs and is obviously parallel. As a result, each probe-gallery pair comparison could be executed in parallel on a different processor to achieve acceleration, but the probe-gallery comparison is only one of the time consuming tasks in face identification. While intuition suggests the probe-gallery comparison would consume most of the execution time, analysis shows probe encoding consumes 80% to 99% of the execution time for one million member gallery sets of four megapixel images. As a result, acceleration of the comparison process alone cannot achieve real-time identification for these gallery sets [19, 20].

Speedup in the range of 36 to 748 times is required to achieve real-time identification as shown in Table 1. Mapping top accuracy face identification algorithms to thousand core parallel SOCs can potentially enable real-time identification with one million member gallery sets of four megapixel images. Five key architectural characteristics significantly impact performance and must be determined to enable real-time face identification:

Processor Instruction Set Architecture (ISA). The ISA defines the *Cycles Per Instruction* (CPI), the number of system clock cycles required to execute a particular instruction [21]. Optimizing the ISA to reduce CPI for heavily used instructions can improve performance.

External memory bandwidth. External memory bandwidth, the rate at which data can be transferred from off-chip System Memory to on-chip Local Memory, impacts computational performance. To achieve maximum computational performance, the memory bandwidth must be sufficient to supply data at the rate needed by the processor. Optimizing the memory bandwidth to protect the processors from data starvation can improve system performance.

Algorithm mapping. The mapping of the algorithm to multiple processors can create or remove performance bottlenecks. An inefficient mapping can create new bottlenecks and degrade performance while an optimal mapping can achieve best use of SOC resources [22].

Number of Processors. The number of processor cores on the SOC directly impacts performance. If all other bottlenecks can be eliminated and sufficient resources are available, speedup proportional to the number of processor cores can be achieved [23].

On-chip communication. The time required to transfer control data between processor cores as well as the time required to communicate data once it is on-chip can add significant overhead to parallel processed [23], and optimizing communication can improve performance.

These architectural characteristics must be optimized to enable real-time face identification. This research shows how to analyze these characteristics to quantify the impact on performance and to develop a mapping that can enable real-time face identification for a one million member gallery of four megapixel face images. The main contributions of this research are performance analysis and benchmark extraction, instruction set architecture improvement, external memory bandwidth improvement, and parallel mapping and computation improvement.

2.1 PERFORMANCE ANALYSIS AND BENCHMARK EXTRACTION

Performance analysis of the sequential algorithms provides a baseline for quantifying performance improvement and provides a way to locate the bottleneck processes that form the benchmarks. Three face identification algorithms that achieved top accuracy in NIST testing were selected for analysis and three data sets were developed from NIST facial recognition data sets. The execution time for each algorithm and each data set was estimated by profiling and counting instructions. Bottleneck processes were selected based on the execution time estimates and extracted to form nine benchmarks. The profiling and instruction counting process was repeated to quantify sequential execution time for the benchmarks. Execution of an implementation of the benchmarks was timed on a current PC architecture with a 1,200 member gallery set of 100 kilopixel images to validate the execution time estimates.

2.2 INSTRUCTION SET ARCHITECTURE IMPROVEMENT

Analysis of the instruction set architecture quantifies the performance improvement resulting from optimizing the instruction set architecture for face identification algorithms. Code implementing the benchmarks was first compiled to produce assembly language code listings. The instruction mnemonics from these code listings were tabulated and counted to quantify the instructions used and the number of times each instruction was executed for each line of high level language code. Execution time was then estimated based on published CPI measurements [24]. Assembly language instructions were sorted by execution count and CPI, and heavily used instructions with CPIs greater than one were selected. Fused instructions were develop to improve performance for these instructions. Execution time was estimated with the ISA improvements to quantify the performance impact.

2.3 EXTERNAL MEMORY BANDWIDTH IMPROVEMENT

The time required to transfer data from off-chip System Memory to on-chip Local Memory was estimated to expose memory bandwidth bottlenecks for the sequential benchmarks. The average data transfer rate for external System Memory was determined from JEDEC specifications [25] and ITRS projections [26]. The volume of data transferred from off-chip System Memory to on-chip Local Memory was calculated from analysis of the algorithm and the average transfer rate was used to estimate memory transfer time. The memory transfer time was then compared to the computational execution time for the algorithm. For memory bound benchmarks the average transfer rate was increased to future values based on ITRS projections [26] to determine if future memory is fast enough to mitigate the bottleneck.

2.4 PARALLEL MAPPING, COMPUTATION, AND COMMUNICATION ANALYSIS AND IMPROVEMENT

The time required for computation and communication for mappings of the face identification algorithms to a 2D mesh parallel SOC architecture was estimated to determine if real-time performance can be achieved for each benchmark and mapping. An initial mapping was developed for each benchmark and expressed as an extended UML Activity diagram. The computation time was estimated using average CPI and cycle counts from the sequential software benchmarks, and the time required for parallel computation was estimated by dividing the sequential computation by the number of processor cores. The benchmark pseudo code was updated to show the data transfer and communication operations and Chan's collective communication model [27] was used to estimate communication time based on communication latency and bandwidth values from the Tile64 architecture [18] and ITRS projections [28].

This research develops techniques to analyze these performance characteristics to develop a high performance mapping of top accuracy face identification algorithms to thousand core 2D Mesh parallel SOC architectures. Chapter 3.0 explores the state of the art in face identification and discusses prior work to accelerate face identification. Chapter 4.0 explains the method used to estimate execution time and develops the benchmarks used to characterize face identification algorithms. Chapter 5.0 explains the analysis of the processor Instruction Set Architecture to determine what changes to the processor ISA can improve performance for face identification. Chapter 6.0 explains the process used to analyze external memory bandwidth and locate and mitigate memory bandwidth bottlenecks. Chapter 7.0 describes the process used to develop a mapping of a sequential face identification algorithm to a parallel 2D mesh SOC architecture and explains how the mapping is analyzed to estimate performance and re-designed to eliminate bottlenecks. Finally, Chapter 8.0 summarizes conclusions and discusses future work.

3.0 BACKGROUND

This chapter explains prior work that provides the background and basis for the novel work in this dissertation. Section 3.1 provides an overview of face identification and explains how the algorithms studied in this dissertation were selected. Section 3.2 reviews the methods used by other researchers to accelerate face identification and discusses the results they achieved.

3.1 FACE IDENTIFICATION

Face identification is a two-step process as shown in Figure 2. The first step encodes the probe face image to prepare it for comparison and the second step compares the encoded probe to each encoded gallery face through a series of pair comparisons.

The probe and gallery face images are encoded to reduce the volume of data required to represent the images, eliminate redundant data, filter out extraneous information and to emphasize the information that differentiates the images from each other. The encoding methods used in top face identification algorithms reduce the data volume to 33% or less of the image resolution with minimal impact on identification accuracy.

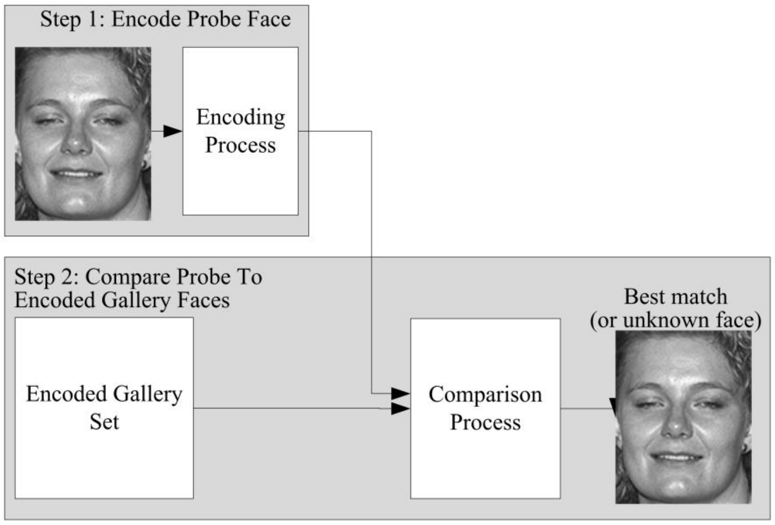

Step 1: Encode Probe Face
Encoding Process
Step 2: Compare Probe To Encoded Gallery Faces
Encoded Gallery Set
Comparison Process
Best match (or unknown face)

Figure 2: A probe image is encoded and compared to the gallery set.

3.1.1 Evaluation of face identification algorithms

The *National Institute of Standards and Technology* (NIST) is devoting considerable resources to sponsoring and supporting a number of face recognition programs [6, 8, 10, 29]. The initial NIST evaluation, the Face Recognition Technology (FERET) program, accepted and tested proposals for face identification and conducted an initial evaluation in 1993 [8, 30]. Following the initial evaluation, NIST funded further research to improve the accuracy of the most promising face identification algorithms. Several tests of the algorithms resulting from this funded research were conducted from 1993 through 1996 to document improvements in accuracy. The final official FERET test in 1996 objectively and independently tested face identification algorithms with a standardized test set and documented the results. These test results are the de facto standard for quantifying accuracy of face identification methods.

Although the last official test was conducted in 1996, NIST has conducted unofficial FERET testing to document accuracy improvements in face identification [31]. Overall, the

FERET test results show that as of 2001, identification accuracy in the range of 82% to 96% is achievable by face identification algorithms tested with the FERET methodology.

Following the FERET tests, NIST shifted focus to face verification. While face identification is a one-to-many comparison of one probe to a gallery set, face verification is a one-to-one comparison of a probe to one gallery member to confirm a claimed identity. This shift to face verification has further emphasized NIST's focus on accuracy rather than performance. As a result of this focus on accuracy, NIST has funded research to improve accuracy with apparently minimal concern about identification time.

The NIST-sponsored *Face Recognition Grand Challenge* (FRGC) funded research and established accuracy goals for a broader range of test scenarios than the FERET test [32]. In 2006, the testing component of this effort, the *Face Recognition Vendor Test* (FRVT), quantified the improvements in verification accuracy resulting from the FRGC and tested both research and commercial systems [6]. The later *Multiple Biometric Grand Challenge* (MBGC) evaluated face verification in combination with other biometric verification approaches [33], and the current *Multiple Biometric Evaluation* (MBE) reported test results in 2010 [10] and is an ongoing evaluation of both still face identification and face verification algorithms with degraded and compressed images .

3.1.2 Face identification algorithm background

Interest in automatic face identification followed the invention of practical electronic computers. While it would take several decades to develop viable identification methods, earlier research established the foundation for this later work. Research prior to the 1970's investigated how the human brain identifies face images [34] and how this process might be automated [35, 36].

However, the complexity of the process and limited understanding of processes in the human brain limited progress.

In the early 1970's, Kanade [37] and Kelly [38] published "seminal works" [39] exploring feature extraction-based face identification methods. Feature extraction methods use particular characteristics of a face, such as eye spacing, nose size, or other observable characteristics of the face for identification.

An automated feature extraction-based system requires a means to automatically locate and measure features. However, this technology had not been developed at the time of this early research. Although the term "automatic face recognition" was used, these early methods required manual intervention and thus were not truly automatic. Reliable methods of locating the face, normalizing it, and extracting the features had to be developed before automatic face identification could progress with feature based approaches. For example, while Bledsoe was the first to claim semi-automated face recognition [35, 40], his method relied on fiducial marks manually hand-drawn on the photographs. These obstacles prompted some researchers to conclude as late as 1989 that face identification by computer was not possible [41, 42].

Feature extraction-based methods are also limited by the tolerance of measurements. It is difficult to set the measurement tolerance large enough to allow for normal photographic variations while still maintaining the precision needed to detect different individuals. For example, to measure eye spacing one might use the center of the pupils as the reference points. However, if the subject is photographed from a slightly different angle or the subject is looking in a different direction, the measurement between the pupils will be different. For individuals with similar faces the variation in pupil distance may be small, so if the measurement tolerance is increased to accommodate for photographic variations, the ability to differentiate between similar individuals may be lost. As a result, the measurement tolerance cannot be set large enough to accommodate photographic variations while maintaining the precision required for accurate identification.

As a result of these issues, much of the work in the late 1970's and early 1980's focused on determining if a face is present in an image, and if so, determining where in the image a face is located [38, 43, 44]. Other researchers continued their work on feature-based methods and proposed alternatives by the late 1980's [45-47]. While some of the limitations of feature based methods had been mitigated, the fundamental issues remained.

Subspace-based methods were developed in response to these issues. Subspace-based methods transform the face image to a subspace and then calculate *similarity*, a scalar number representing how well two images match. Turk proposed one of the earliest subspace-based methods in 1991 [48]. Turk's Eigenface method applies *Principal Component Analysis* (PCA), a technique originally developed to extract statistical information from a data set [49] and later used to reduce a data set to its intrinsic dimensionality [50]. This approach seeks to extract image information that captures the unique differences between images while discarding image information that is duplicative or unnecessary for identification. This approach is known as the Eigenface method and signals the beginning of "modern" face identification methods.

The Eigenface method decomposes a face into whole-image sized components representing a portion of the information contained in the whole image. For the Eigenface method, these components are termed Eigenfaces and are the eigenvectors of the covariance of the training images. A library of Eigenfaces is created during training and a particular face can then be expressed as a weighted linear combination of Eigenfaces.

Subspace-based methods overcame the limitations of feature extraction-based methods and effectively ended most feature extraction-based work. Efforts to improve accuracy of subspace-based methods continued through the 1990's and beyond. In subspace-based methods, the component faces are separated based on the amount of unique information they contain. However, often the most significant differences between images are the result of different lighting and orientation rather than unique differences between images of different individuals [51], and *Linear Discriminant Analysis* (LDA) methods were developed to overcome this sensitivity to lighting [52-54].

LDA based methods alter the training process to construct a different subspace than the Eigenface method, while leaving the Eigenface identification process unchanged. The Eigenface method seeks to maximize the distance between all pairs of training images, including those representing the same individual. In contrast, LDA methods cluster different images of the same individual together to minimize separation between these images while still maximizing the distance between images representing different individuals. This clustering tends to minimize sensitivity to lighting changes, since images of the same individual with different lighting will be clustered together, while images of different individuals are spread farther apart. The component faces determined from these clusters of faces are intended to better represent the differences between individuals.

Other researchers investigated whether selectively removing particular Eigenfaces from the subspace might reduce the bias caused by lighting differences [55]. However, while both LDA-based methods and the selective removal of Eigenfaces improved identification accuracy for some gallery sets, a consistent overall increase in accuracy was not achieved [8].

Some researchers theorized that higher order statistics could improve accuracy over the second order statistics used for the Eigenface method. Kernel-based methods represent the covariance matrix calculation of the Eigenface method as a higher order function, resulting in a higher order mapping into the face space [56, 57]. A similar method, *Independent Component Analysis* (ICA), also maps images to the face space using higher order statistics [58, 59]. Unfortunately, these methods have not resulted in significant improvement in accuracy over the Eigenface method [57].

The basic Eigenface method as defined by Turk has become the de facto baseline for comparison of face identification systems [8]. The identification accuracy of the Eigenface method improves when similarity is calculated as Mahalinobis distance instead of Euclidean distance [8, 60].

Another approach calculates similarity using Bayesian statistics. The Bayesian method differs from the Eigenface method in two primary ways. First, the comparison measure is a Bayesian conditional probability rather than a vector distance calculation and secondly, difference images are used for comparison rather than single images. A difference image is calculated as a pixel-by-pixel subtraction between two images [61-64].

The Bayesian method uses two subspaces. The training set is divided into two groups, the *intrapersonal* group containing different images of the same individual and the *extrapersonal* group containing images of different individuals. Within these two groups, difference images are formed by selecting image pairs and subtracting the second image of the pair from the first. This calculation is repeated for image pairs in both groups to build a set of difference images, one set for each group. Two separate PCA subspaces are then constructed, one for the intrapersonal group and one for the extrapersonal group [61].

For identification, difference images must be calculated between the probe image and each gallery image in the intrapersonal group. These difference images are then mapped into both subspaces. Bayesian conditional statistics are used to calculate the probability that two difference images represent the same pair of individuals. This process is repeated for each probe-gallery difference image and the difference image with the highest probability indicates the best match [61].

While the Bayesian method did increase identification accuracy over the Eigenface and LDA methods, the computational requirements increased significantly due to the difference image calculation. To resolve this issue, Moghaddam proposed a simplification of the method to reduce the computational requirements and reported less than 3% reduction in identification accuracy with the simplified method [61].

This simplified method performs a whitening transformation on the images and then projects the images into two PCA subspaces, one for the extrapersonal set and one for the intrapersonal set. The difference image can then be calculated as a vector distance in the subspace. If the Bayesian Maximum Likelihood (ML) comparison function is used, only one subspace is required and identification is analogous to the Eigenface method with Mahalinobis distance.

A third method was inspired by earlier feature extraction-based research. The *Elastic Bunch Graph Matching* (EBGM) method locates feature points by elastically adjusting a fixed undirected *bunch graph* generated during training. A series of two-dimensional filters are applied to the image region surrounding the feature point and the filter response values at the feature point are used as an encoded representation of that feature. A *face graph* is formed where the edges of the graph contain the distance between features and the vertices of the graph contain the encoded representation of the feature. Vector distance is calculated between the encoded representations of the features and the resulting distances are averaged to calculate a scalar similarity [65, 66].

These three classes of methods, including the subspace, Bayesian, and EBGM methods, remain subjects of current research. Recent research on subspace methods has explored improving subspace projections [67], addressing the image misalignment problem [68], improving discriminant analysis methods [69, 70], and combining methods from different classes [71].

Pang increases the accuracy of subspace projections and therefore identification accuracy by altering the subspace comparison process [67]. Pang's method forms *feature lines* between pairs of projected image points representing the same individual. The distance from the projected probe image to the nearest point on the line is used for comparison.

Conventional subspace methods linearly transform an image vector to a point in the subspace. Pang's process adds the feature lines to the subspace, effectively estimating a continuous set of additional training images and providing some of the benefit of a gallery set with multiple images of each individual. A *nearest feature line* (NFL) comparison then finds the Euclidean distance to the nearest feature line rather than the nearest projected image as in the Eigenface method.

Pang reports a 21% increase in identification accuracy for the NFL comparison with conventional subspaces [72]. In his more recent work, he also changed the subspace generation process to gain an additional 11% increase in identification accuracy [67].

As with the LDA method, Pang's method alters the way the subspace is formed during training and thus forms a different transform matrix for the subspace. However, the process to project an image vector into the subspace is unchanged, so no additional computation is required.

Pang tested his method on the FERET data set by selecting his own set of 1,394 images rather than using one of the standard NIST defined test sets. For his test set, he reports identification accuracy of 45% for the Eigenface method. When the NFL comparison is used alone, identification accuracy is 66%, and identification accuracy increases to 77% when the feature line subspace and NFL comparison are combined.

Recent Bayesian research includes efforts to improve identification accuracy by subdividing the subspace [73] and applying PCA to further reduce computational requirements [74]. Research on EBGM-based methods has explored improving the sampling of image regions [75] and increasing identification accuracy [76, 77]. In addition, research into overcoming orientation and expression differences [4], occlusions [48], and illumination [78] has also continued.

Face identification research has also broadened to include 3D and video-based face identification [6, 79, 80]. Some video-based methods extract still images from the video [39, 79], and some 3D methods form the 3D image from a set of 2D images or a 2D distance from plane representation [39]. For these formats still image-based methods are used for identification and thus further research on still image identification remains relevant.

Table 2 through Table 4 summarizes face identification algorithms by type. Table 2 summarizes PCA-based subspace methods, Table 3 shows methods based on the Gabor wavelet transform, and Table 4 lists methods that fall outside this categorization.

Table 2: Subspace Methods

AUTHOR	SUBSPACE METHODS	DATE	ACCURACY
UMD	Linear Discriminant Analysis (umd_97) LDA [53, 54]	1997	0.96
Moon	Eigenface with Mahalinobis distance [81]	1997	0.96
MIT Media Lab	Bayesian Matching (BM) [82]	1996	0.95
UMD	umd_96 LDA [83]	1996	0.95
MSU	LDA [54, 84]	1996	0.86
Liu	PCA Gabor (PCAG) [85]	2003	0.85
MIT Media Lab	mit_mar_95 [8]	1996	0.83
ARL	arl_ef [48, 81]	1997	0.80
Pang	Feature Line Method [67, 72]	2009	0.77
NIST	ef_hist_dev_ml2 [8]	1997	0.77
NIST	ef_hist_dev_l1[8]	1997	0.77
NIST	ef_hist_deb_anm [8]	1997	0.77
NIST	ef_hist_dev_md [8]	1997	0.74
NIST	ef_hist_dev_ml1 [8]	1997	0.73
NIST	ef_hist_dev_l2 [8]	1997	0.72
NIST	ef_hist_dev_ang [8]	1997	0.70
Yang	Supervised Subspace Learning [68]	2009	0.70

Table 3: Wavelet Methods

AUTHOR	WAVELET METHODS	DATE	ACCURACY
USC	Elastic Bunch Graph Matching [65]	1997	0.95
Du	Nonuniform Gabor [75]	2009	0.94
Shin	Generalized EBGM (GEBGM) [76]	2007	0.91
Tan	Recognition under occlusion [4]	2009	0.88
Kepenekci	Gabor Wavelet (GW) [86]	2002	0.70

Table 4: Uncategorized Methods

AUTHOR	OTHER METHODS	DATE	ACCURACY
Excalibur Inc	Excalibur [8]	1997	0.22
Rutgers	Rutgers [87]	1994	0.18
ARL	arl_cor [8]	1997	0.05

The top accuracy algorithms from each category were selected for this research. The top subspace algorithms include Linear Discriminant Analysis, Eigenface with Mahalinobis distance, and Bayesian Matching. These algorithms are shown as shaded lines in Table 2. The LDA algorithms differ from the Eigenface algorithm only in the training process and perform identification in the same way, so the Eigenface and Bayesian algorithms are selected for this research and the LDA algorithms are not separately analyzed.

The top accuracy wavelet algorithm is the Elastic Bunch Graph Matching algorithm, shown as a shaded line in Table 3. While similar accuracy was reported for the Nonuniform Gabor algorithm, this algorithm was not tested with a standard FERET test set, preventing objective comparison of accuracy. Since a Gabor wavelet-based algorithm is already included in the research set, the Nonuniform Gabor algorithm was not added to the set.

None of the uncategorized algorithms, shown in Table 4, achieved higher than 22% identification accuracy. Given the low accuracy, these algorithms are not considered further.

3.2 ACCELERATION OF FACE IDENTIFICATION

Much of the prior work in the field of face identification investigates how to increase identification accuracy [39, 40, 88-90]. The NIST FERET [8] and FRVT [6] tests evaluated identification accuracy while imposing only minimal constraints on execution time, motivating the emphasis on identification accuracy. As a result of the emphasis on identification accuracy, identification time has not been a primary topic of research until fairly recently [91, 92].

In the last decade, researchers have begun to explore identification time constraints and solutions. Identification accuracy of 95% or better is achievable with the top face identification methods, but these methods are not real-time for large gallery sets of high resolution face images. In effort to accelerate face identification toward the goal of real-time identification, researchers have investigated a range of solutions. Much of the existing research seeks to exploit the obvious parallelism of the comparison process [19, 91, 93-95]. However, the preliminary research for this dissertation shows that the encoding time is significant for most face identification methods and is often the primary constraint on identification time. As a result, research that investigates how to accelerate either the encoding process [96] or both the encoding and comparison processes [92, 97-99] is particularly relevant to the dissertation work.

The research reviewed in this section uses a range of techniques to accelerate a variety of face identification methods. Several researchers used clusters consisting of networked PCs to accelerate the comparison [19, 93] or encoding processes [98]. Other researchers applied special processors such as the GPU [91, 99] or custom hardware optimized for a particular process [92, 94-97] to provide the needed acceleration. Changes to the identification algorithms to reduce computation [95] and reduction of image resolution to reduce the amount of computation required [92, 95, 97, 98] were also investigated.

3.2.1 Networked Computers

Interfacing multiple computers with a common communication network forms a MIMD system termed a cluster [21]. The communication network provides the means for the computers to exchange data and control information, while each computer has a separate CPU and can therefore independently execute a unique program in parallel with other CPUs.

A client-server architecture interfaces one or more *clients* to the network. The client is a device that submits requests to the *server*, the computer or computers that perform or facilitate the actions required to respond to the client request. The server can be a single computer but more commonly is a cluster of networked computers. A *host* computer, a central control node in the cluster, communicates with the clients to receive and respond to queries. The host manages a group of networked *slave* computers, sending requests to the slave computers, receiving results from the slave computers, and then communicating responses back to the client.

A client-server cluster can be used to accelerate the comparison process in face identification methods [19]. The gallery set of known faces is divided into five equal segments and distributed among the Host and the four Slave computers [19]. The Host computer accepts the queries from the Clients, broadcasts the query data to each Slave computer, and combines the results from each Slave to select the gallery face that best matches the probe face.

This system was tested with 100,000 and 200,000 image[2] galleries [19]. The gallery set was subdivided into equal segments that were distributed among five computers and therefore each segment contains either 20,000 or 40,000 faces, depending on the test set. The system accelerates the comparison process four times and compares one encoded probe to a 100,000 member gallery set within 40 seconds. The author does not report either image resolution or the identification method used.

Meng used a similar network configuration but added an additional slave computer [93]. Faster PCs with 2.4 GHz dual core Xeon CPUs and 1 GB of memory were used, and the MMX instructions in the Xeon CPUs provided additional acceleration [93]. Meng used the Multimodal Part PCA (MMP-PCA) face identification method, an extension of the baseline Eigenface method [93]. The baseline Eigenface method converts the entire face image to a vector and

[2] Chunhong states in the introduction that the database contains "1,000 thousands human faces" while the Results section lists comparison times for 100,000 and 200,000 image gallery sets.

projects it into an Eigenvector subspace. In contrast, the MMP-PCA method also projects the entire face image into the subspace but projects an additional four segments of the face image into the subspace. These four segments are extracted regions of the image that contain the eyebrow, eye, nose, and mouth features. The encoded face is therefore represented by five weight vectors, one encoding the entire face as in the baseline Eigenface method and the other four encoding the extracted feature segments of the face.

As in Chunhong's method [19], the encoding of the probe image is performed by the host and the encoded probe image is submitted by a Client to the Host for comparison [93]. Meng does not specify how the database is distributed among the slave computers, but it appears the database is equally distributed in the same manner as Chunhong [19].

In addition to the four times acceleration achieved by performing the comparison process in parallel on the cluster, the MMX instructions in the Xeon CPUs were exploited to provide additional acceleration. These SIMD instructions allow multiple results to be calculated with a single operation and provided an additional acceleration of 14.5 times [93].

Meng used the TH-FACE face database to test his system. The TH-FACE database contains 19,289 images of 750 individuals and these images were cropped to 172 kilopixel resolution. Identification accuracy ranging from 50% to 85% is reported for sequential software implementations of the MMA-PCA method and the TH-FACE database [100], while Meng reports 62.70% accuracy when the five best matching images are selected. Accuracy results for selection of the single best match are not provided, precluding direct comparison of accuracy results with the other methods reviewed [93].

Meng also tested the system with a 2,560,000 face database but does not indicate the source of these images [93]. Only the TH-FACE database is referenced in the paper, so the database may have been formed from the images in that database. For the 2.5 million member gallery set and the six PC cluster, Meng reports a comparison time of 1.094 seconds but does not quantify the encoding time [93]. For a database of this magnitude the encoding time will be significant and will likely preclude real-time encoding.

A cluster where control is distributed among the nodes rather than centralized in a Host computer is termed a peer network. Distributing control among the nodes can improve performance if the process can be partitioned to exploit the autonomy of the nodes. Yang distributed both the encoding and comparison process among the nodes in a peer network [98]. The cluster consists of ten PCs networked together as peers rather than in a host-slave configuration. Each PC contains a 2.2 GHz Core 2 Duo processor and 2 GB of RAM, and the dual cores in the CPUs are configured as separate nodes, providing a 20 node parallel system [98].

Yang used a face identification method derived from the *Local Binary Pattern* (LBP) method [98]. The LBP method encodes local regions of an image by sampling and quantizing characteristics of the pixels in that local region. The samples are combined to form the binary pattern representation for the local region as shown in Figure 3.

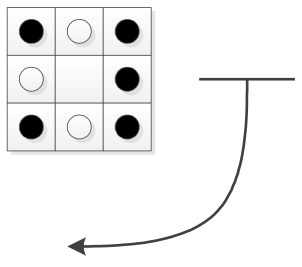


Figure 3: A local binary pattern is formed by sampling pixels in a rotational sequence.

The LBP is generated by sampling characteristics of the pixels in a local region. The image is subdivided into local regions of consistent size as shown in Figure 3. The pixels are then quantized to one or zero based on the relationship with the adjacent pixels. Starting with a specified pixel, a specified path is traversed through the local region to form the binary pattern. In Figure 3, the middle right pixel is the starting location and the pattern is traversed in a clockwise direction, generating the local binary pattern of 11010101 [97]. Combining LBPs that represent the same pattern in rotated and shifted forms allows the total number of LBPs to be reduced to 30 for a nine pixel local region and the LBP can therefore be represented with a five bit binary number [97].

Yang used the *Local Gabor Binary Pattern Histogram Sequence* (LGBPHS) method for face identification. This method enhances the Local Binary Pattern (LBP) method by using Gabor filters to encode the regions of the image rather than the simple magnitude comparison of adjacent pixels used in LBP. The LGBPHS method convolves a set of Gabor filters with the entire image and then stores the magnitude response from the set of convolution results as a set of *Gabor Magnitude Pictures* (GMP). Local binary patterns are then generated for the local regions in each GMP. LBPs corresponding to adjacent local image regions are grouped together and a histogram is calculated for each region. The histograms are combined to form a vector that is the encoded representation of the face [98].

The probe encoding process is performed in parallel on the cluster nodes. The probe image is first broadcast to each of C CPU cores, where C=20 for the test system. The probe image is then filtered with a set of R Gabor filters, each expressed as a filter mask $\mathbf{F}_r$. Each CPU performs the convolution operations required to apply an R/C subset of the R Gabor filters to the probe image, generating an R/C set of GMPs. The LBPs are then calculated for each local region in each GMP to form an R/C set of LBGP maps. Finally, histograms are extracted and concatenated to complete encoding of the probe image [98].

The probe-gallery comparison process is also distributed among the C nodes in the network. The set of pair comparisons is equally distributed among the C CPU cores as in the other cluster-based methods [19, 93, 98]. Each CPU calculates similarity scores for a subset of the gallery. When all CPUs have completed calculation similarity calculations, a reduction operation selects the best similarity score, corresponding to the probe-gallery pair that is the best match [98].

The parallel LGBPHS system was evaluated with both the ORL and FERET databases [98]. For the ORL database, 200 images cropped to 2.5 kilopixels were selected for the gallery set and the identification time of 12.8 seconds includes both encoding and comparison time. The FERET images were cropped to 16 kilopixel resolution and identification time of 561 seconds was achieved with a 1,698 image gallery set. For both the ORL and FERET database testing, the acceleration achieved with ten CPUs is approximately nine times and increased to the range of sixteen times when 20 processors were used [98].

3.2.2 Specialized Processors

General purpose processors trade-off computational performance for the ability to perform complex instructions and execute complex control sequences [16]. Processors optimized for specific types of operations can improve performance by using more chip area for the specialized function while sacrificing the complexity and flexibility of the CPU [16]. As a result, specialized processors can substantially increase performance for the specialized function if the process can be implemented within the constraints and limitations of the specialized processor.

The Graphics Processor Unit (GPU) is a specialized processor designed to improve the performance of mathematical operations [16, 17]. To achieve this performance increase, the GPU restricts communication and data exchange between executing processes and implements only the most fundamental control instructions [17]. If a program can be structured to execute within these constraints, however, a substantial increase in performance is possible.

General Purpose Computing on Graphic Processor Units (GP-GPU) uses the specialized graphics hardware on a GPU to accelerate programs that are not strictly graphical. GPUs have parallel architectures and high memory bandwidth in the range of 40GB/s [91], but many GPUs do not support integer and double-precision arithmetic data types or bitwise logical operations [91]. In addition, GPUs are more difficult to program than a CPU, due the complexity of the model [91]. As a result, some algorithms are better suited to a GPU-based implementation than others [91].

Abate applied GP-GPU techniques to accelerate 3D face identification [91]. A 3D model of a probe image is sampled and projected to form a 2D normal map and a filter or flesh mask, where each matrix is 128 by 128 elements. This preliminary processing is performed on the CPU.

The filter or flesh mask is animated on the GPU to compute the expression mask [91]. The expression mask incorporates multiple facial expressions into the mask and improves identification accuracy by decreasing the sensitivity to expression. The expression mask is combined with the filter mask and normal map to produce the normal planes map, the 128 by 128 element encoded representation of the probe image.

The probe is compared to the gallery set with a series of pair comparisons. The normal planes maps for the probe and one gallery member are subtracted element by element using the

GPU. The GPU then normalizes these results to form the difference map corresponding to that probe-gallery pair. The difference map is then sent to the CPU and the CPU calculates a histogram for the grey levels represented in the difference map. The CPU calculates a weighted sum of the histogram and a Gaussian function to calculate a scalar similarity score for the probe-gallery pair. A difference map, histogram, and similarity score is calculated for each probe-gallery pair. The CPU then ranks the similarity scores for all probe-gallery pairs to select the best similarity score and therefore the gallery member that best matches the probe face.

Abate implemented the difference map calculation on the GPU and used the system CPU to compute the similarity score. This partitioning enabled the GPU to perform the functions that fit well with the GPU architecture while implementing processes requiring significant memory interaction on the CPU. Abate tested his system with a set of 16 kilopixel images consisting of 135 face images of photographed individuals and 10,000 computer-generated face images. The combination of a quad GPU with a CPU achieved 75,000 pair comparisons per second, while a CPU-only implementation achieved 200 pair comparisons per second, showing that the GPU provided a 375 times speedup of the comparison process for 16 kilopixel face images [91].

Other face identification methods can be functionally modeled on a GPU. Optical correlation face identification methods provide high accuracy and performance, but the physical size of the equipment and high cost limit portability and applicability of these methods [99]. To overcome these limitations, Ouerhani applied the functional approach of the optical correlation face identification method to a GP-GPU system [99].

An optical correlation system for face identification contains three planes and two convergent lenses. The first lens generates the Fourier transform of the probe image, and the Fourier transforms of the probe and one gallery member are optically multiplied at the Fourier plane. A second lens generates the inverse Fourier transform to generate the correlation, which is analogous to a similarity measure [99].

Optical correlation methods achieve high performace, but the hardware required is prohibitively expensive and the devices are too large to be portable [99]. Ouerhani proposed a GPU-based correlation method to provide a more cost practical and portable system. In this approach, the Fourier transform of the probe is calculated with an FFT algorithm executing on a GPU. The Fourier transform of the probe is multiplied by the FFT mask for one gallery image and the inverse FFT is computed on the result. Finally, the peak of the correlation is quantified to form the similarity score.

Ouerhani compared a Matlab implementation of this method executing on a 2.4 GHz dual core PC with an NVIDIA GeForce 8400 GS GPU-based implementation of this algorithm. Using an image resolution of 65 kilopixels and a four member gallery set, a pair comparison time of 10 ms was achieved with the GPU, while the Matlab implementation executing on the CPU required 25 ms, a speedup of 2.5 times. This result suggests that this system could perform 100 pair comparisons per second and thus would require 2.7 hours to compare one probe to a one million member gallery set.

3.2.3 Custom Hardware

Custom hardware specifically designed to accelerate a particular task is likely to achieve the best performance. Two approaches dominate the field, full custom transistor level hardware designs and *Field Programmable Gate Arrays* (FPGA). A full custom design offers the best performance but is very expensive to develop and implement. The FPGA, a chip that can be programmed to implement digital logic functions, is more cost-effective but does not achieve the same level of performance as the full custom option.

Lahdenoja used full custom hardware to accelerate face identification using the LBP face identification method previously described (Figure 3) [97]. A face image is encoded by dividing the image into regions, forming the LBP for the sub-regions within each region, and then encoding the face image as an occurrence map. The occurrence map is a matrix that indicates which sub-regions of the image contain a particular LBP, and thirty LBPs will generate 30 occurrence maps [97]. The thirty occurrence maps are then combined to form a feature vector. Using 130 by 150 pixel images (20 kilopixels) requires a 126 by 146 by 5 bit feature vector, a total of 91,980 bits [97]. A nearest neighbor classifier is then used to compare probe-gallery pairs of vectors.

The CSU FaceID software and the FERET database were used for algorithm verification and simulation. The 100 kilopixel FERET images were cropped, scaled, and normalized to produce the 20 kilopixel images used for testing. Several classifiers were evaluated, and identification accuracy of 86% was the best accuracy measured for the FERET FC test set.

A 130 by 150 array of custom parallel processors were used to sample and encode the LBPs and occurrence maps. The processor array encodes the probe image but does not perform the comparison. This implementation achieved identification accuracy up to 80% with the FERET images and identification time of 40 ms with a 100 member gallery set [97].

Arya proposed using associative memories operating in parallel to accelerate the comparison process [94]. An N pixel probe image is encoded by first subdividing the image into $N/256$ sub-regions, each 16 by 16 pixels. The rows of each sub-region are concatenated to form a vector, and the vector for each sub-region is encoded as a one-byte Hebbian weight [101]. The weight bytes are then combined to form one vector, the encoded representation of the probe image.

The probe is then compared to the gallery set through a series of pair comparisons. The encoded vectors for each probe-gallery pair are compared in parallel using one associative memory for each vector element and therefore one set of $N/256$ associative memories for the vector comparison [94]. The outputs from the set of associative memories are combined into a match vector that represents the similarity between probe-gallery pair. After the match vectors are calculated for each probe-gallery pair, these vectors are compared to select the best match [94].

Arya used one image of each of the 40 individuals[3] contained in the ORL face database to train the system and provide the information needed to calculate the Hebbian weights [94]. The remaining nine faces per individual were used for testing, providing both the 390 member gallery set and the probe images. Arya reports identification accuracy of 96.9% for this method and 89.5% for his testing of the baseline Eigenface method with the ORL database, but does not report timing results [94]. Arya plans future work to determine whether improved accuracy improvement can be achieved with large databases and higher resolution images such as the FERET database.

Sotiropoulos [96] noted that multiplication of large matrices constrains identification time in the Eigenface, Linear Discriminant Analysis, and Bayesian methods. He analyzed the CSU Face ID application [102] as a baseline sequential software implementation and determined that

[3] In the introduction to the paper, Arya states he used “six face images of the same person” for training. In the Experimental Results section, he states he used one image of each individual for training and the results are consistent with that interpretation.

80% of the execution time is consumed by matrix multiplication for these methods in this application [96].

Sotiropoulos used a Xilinx Virtex-5 FPGA to implement a Parallel Matrix Multiplication Unit (PMMU) [96]. FPGAs have limited on-chip memory and multiplier resources, and the partitioning of the problem to fit within the FPGA resources can have a significant impact on performance. Sotiropoulos found that the best performance was achieved with 64 by 64 word multipliers [96].

The matrix multiplication is performed by segmenting the input image and basis matrices into 64 by 64 sub-matrices. The matrix multiplication is then performed on each sub-matrix with a series of pipelined multipliers. To perform the matrix multiplication, the first sub-matrix from both the probe image and the matrix of subspace basis vectors is multiplied. The results of this calculation are provided to the next multiplication block, which integrates the results from the first block with the multiplication results for a different set of sub-matrices. This process is repeated until all sub-matrices have been multiplied and combined, generating the encoded probe [96].

Sotiropoulos states he achieved a 50 to 500 times speedup for the matrix multiplication with the PMMU in relation to the software implementation [96]. This research is focused on accelerating the multiplier and therefore overall identification time is not reported. However, from the multiplier configurations provided, we can infer a 19 kilopixel image resolution and a gallery size in the range of 200 images. Combining this information with the author's measurement of 80% of execution time for the matrix multiplication in the encoding process enables estimation of encoding time at 2.83 seconds as shown in Table 5.

Table 5: Calculation of Identification Time

Reported multiplication time for software implementation $\left(T_{MM}\right)$	10.59 s
Estimated encoding time for software implementation $\left(T_{ID} = 0.8/T_{MM}\right)$	13.23 s
Software overhead $\left(T_{OH} = T_{ID} - T_{MM}\right)$	2.64 s
Multiplication time for PMMU $\left(T_{PMMU}\right)$	0.19 s
Estimated encoding time with PMMU $\left(T_{OH} + T_{PMMU}\right)$	2.83 s

Cheong integrates the face detection and recognition process with the goal of creating a "smart camera" capable of detecting and recognizing human faces [92]. Citing the emphasis on identification accuracy rather than computation time in the literature, Cheong selected the Eigenface method and focused on accelerating the entire process of detecting and recognizing a face.

Cheong tested his method with 12 megapixel images but does not report gallery size or identification accuracy [92]. From the results reported and screen captures from the system, it appears the gallery size was in the range of ten images. Identification time for a 12 megapixel image is reported as "more than one minute" [92]. Cheong enables real-time identification by "optimizing" or down-sampling the image to 300 kilopixels and reports an identification time of 22 seconds for the down-sampled image. Identification time for 19 kilopixel video-captured images is reported as 0.5 seconds.

Sajid [95] proposes a different encoding method and a new classifier to reduce comparison time relative to the Eigenface method [48]. Sajid's approach calculates a frequency distribution curve instead of performing the computationally intensive matrix operations required by the Eigenface method [48, 95]. During an off-line training process, the gray level distribution in each known image[4] is calculated and the number of pixels with each of the 256 possible eight-bit gray scale values is tabulated for each training image. The resulting M by 256 element matrix is normalized by N and sorted to form a reference pattern vector for each known image and the Euclidean normalization length for each pattern vector is calculated and stored [95].

For identification, the probe is first encoded as a pattern vector by tabulating the pixel gray scale distribution and normalizing it by the image resolution [95]. The Euclidean normalization length is calculated for the probe pattern vector and a set of difference pairs is then calculated by subtracting the probe normalization length from the normalization length for each known image. The resulting set of vector distances is sorted to select the sequence in which to compare the probe to the known images such that the most probable match will be compared first.

Sajid encodes the probe image on the host PC to avoid transferring and storing the entire image in an FPGA, since the time required to transfer the entire six kilopixel image would exceed the comparison time [95]. After the probe is encoded, the probe pattern vector is transferred to an FPGA, where custom hardware compares the probe pattern vector to the known image pattern vector most likely to match [95]. If the known image pattern vector does not match the probe pattern vector, the next most likely known image pattern vector is compared. This process continues until a match is found or the set of known image pattern vectors is exhausted. Sajid reports that on average, 12 pattern vectors are compared [95].

Sajid tested his system with the ORL database, which contains 400 six kilopixel images of 40 unique individuals, and reported identification accuracy of 99% [95]. A similar

[4] Sajid’s method combines the training and gallery sets so that the set of known images is both the training set and the gallery set.

comparison on a PC-based system required 500 ms per pair comparison [95], while Sajid reports a pair comparison time of 421 ns to 4689 ns, depending on the image and architecture used [95]. These comparison times can be extrapolated to 421 ms and 4.7 s respectively for a one million member gallery set of low resolution images. However, increasing the resolution to the 100 kilopixel resolution of the FERET images will increase the time required substantially and will likely preclude real-time operation.[5]

3.2.4 Conclusion

Real-time face identification with large gallery sets of high resolution face images remains an elusive goal. The research reviewed applies a wide range of techniques and approaches to solving this problem. This research shows that the comparison process can be accelerated relatively easily as a result of the inherently parallel probe-gallery pair comparison process. However, the more complex parallelism of the probe encoding process is more difficult to expose and exploit.

The reviewed research shows that real-time performance can be achieved for small gallery sets of low resolution images. Real-time performance can also be achieved for larger gallery sets if image resolution is reduced or lower identification accuracy is acceptable. However, when the top accuracy face identification methods are used with large gallery sets of high resolution images, identification is not real-time.

[5] Sajid did not explore the effect of image resolution on comparison time and therefore it is unclear how much computation will increase for higher resolution images.

3.3 FACE IMAGE DATA SETS

Standardized tests sets, parameter sets, and architectures are used for this research to provide objective results. The image sets used in official NIST face recognition tests provide a range of image resolutions and gallery sizes. Standard sets of algorithm parameters can be derived from these test sets and provide consistent parameter sets for the research. Selected architectures provide a basis to evaluate both current performance and to project performance ten and twenty years into the future.

Several tests sets of face images are in current use. These test sets consist of photographic images of a subject's head and include the entire head and background. These images are tightly cropped to the sides of the face, the forehead, and the chin to prepare the image for face identification. In these test sets, the cropped image resolutions ranging from 35 kilopixels to 1.25 megapixels and the number of subjects ranges from 1,199 to 6.1 million.

The surveillance camera database SCface contains 4,160 images of 130 subjects. Cropped images of 590 kilopixel resolution are produced from 1.9 megapixel photographic images [103].

The 2006 NIST Face Recognition Vendor Technology High Resolution test set contains 7,192 images of 257 subjects. Cropped images of 1.25 megapixels are produced from 4 megapixel photographic images [6].

The 2006 FRVT High Computational Intensity test set contains 108,000 images of 36,000 subjects. Cropped images of 25 kilopixel resolution are produced from 75 kilopixel photographic images [6]. The FRVT test set is a subset of the images collected for the Visa Database by the Visa Services Directorate, Bureau of Consular Affairs. The full Visa Database contains 6.8 million images of 6.1 million subjects [104].

Four test data sets were developed for the 1997 NIST Face Recognition Technology (FERET) test. The test sets contain either 864 or 1,196 images drawn from a set of 14,126 total images of 1,199 subjects. Cropped images of 35 kilopixel resolution are produced from 100 kilopixel photographic images [30].

Four face identification algorithms achieved highest accuracy in NIST FERET and FRVT testing [8, 105]. The Eigenface algorithm [48] converts the cropped image data to a column vector and treats the vector as a random variable. Principal Component Analysis (PCA) techniques are used to reduce the vector to its intrinsic dimensionality. This process determines the eigenvectors of the covariance matrix of the set of training images, and the Eigenface algorithm draws its name from these eigenvectors, each of which is one face image sized vector that represents a component part of a training image. For comparison, vector distance is calculated between probe-gallery pairs, and the smallest vector distance represents the best match.

The Linear Discriminant Analysis algorithm [52] improves accuracy of the Eigenface algorithm by clustering multiple cropped training images of the same subject to decrease vector distance between images of the same subject while increasing vector distance between images of different subjects. When training is completed and the system is initialized, it processes and compares probe and gallery images in the same way as the Eigenface algorithm.

The Bayesian Matching algorithm [62, 63] forms difference images by subtracting each gallery image from the probe image. To reduce the required computation, PCA techniques are used to reduce vector dimension before the difference images are calculated. The Bayesian maximum likelihood is calculated to quantify the probability that a difference image is a member of the class of difference images representing the same subject, and the greatest probability represents the best match.

The Elastic Bunch Graph Matching algorithm (EBGM) [65] overlays one bunch graph representing all the training images on the probe face, elastically adjusts the vertices of the graph to overlay features of the probe face, and then constructs a face graph for the probe face. Each node of the face graph is labeled with a jet, a sampled wavelet transform of the corresponding facial feature. Two images are compared by calculating a the similarity, a scalar measure of how well two jets match, between each pair of jets at each node in the graph. The node similarities are then averaged over the graph to find the graph similarity, and the greatest similarity represents the best match.

Table 6: Face Identification Data Sets and Parameters

		TEST SET							
	PARAMETER	WISKOTT	FERET 1996	FRVT 2002		FRVT 2006 AND MBE 2010			RESEARCH DATA
				HCINT	MCINT	NOTRE DAME	SANDIA	DEPT. OF STATE	
COMMON PARAMETERS	Image Resolution N (pixels)	16,384	98,304	75,600	76,800	6×10^6	4×10^6	75,600	6×10^6
	Distance Between Eye Centers O (pixels)	24	77	75	45	400	350	75	400
	Gallery Size G (images)	250	1,196	121,589	7,722	7,496	14,365	108,000	1×10^6
EIGENFACE ALGORITHM	Number of Training images [M=G] M (images)	250	1,196	121,589	7,722	7,496	14,365	108,000	1×10^6
	Number of Retained Eigenvectors M' (Eigenvectors)	150	718	72,953	4,633	4,498	8,619	64,800	600,000
EBGM ALGORITHM	Number of Training images M (images)	250	1,196	121,589	7,722	7,496	14,365	108,000	1×10^6
	Mask Dimension W (pixels)	16	51	50	30	267	233	50	267
	Search Region Dimension R (pixels)	16	51	50	30	267	233	50	267
BAYESIAN ALGORITHM	Number of Training images M (images)	250	1,196	121,589	7,722	7,496	14,365	108,000	1×10^6
	Number of Retained Eigenvectors M' (Eigenvectors)	150	718	72,953	4,633	4,498	8,619	64,800	600,000

4.0 PERFORMANCE ANALYSIS AND BENCHMARK EXTRACTION

A performance analysis method will be developed to provide a means to estimate execution time and the percentage of time consumed by each line of a C code program for a face identification algorithm. Execution time will be estimated to quantify the time consumed by each code segment and calculating the percentage of total time consumed by a code segment will provide a means to expose the bottleneck processes that constrain identification time.

The performance analysis method will integrate complexity analysis and profiling techniques to estimate execution time. A technique based on complexity analysis will be used to determine the number of times each part of the algorithm will be executed. The algorithm will then be implemented in C code to verify functional correctness and to generate code for profiling. The C code will be profiled to determine the number of clock cycles required for execution of each line of C code and execution time will be estimated by dividing the cycle count by the system clock frequency.

The performance analysis method will be applied to the sequential face identification algorithms to provide a baseline for quantifying performance improvement and to provide a way to locate the bottleneck processes that limit performance in the algorithms. The execution time estimate for each C code line will be divided by the total execution time and expressed as a percentage to quantify the portion of execution time consumed by each code line. Code lines that consume a majority of the execution time represent the bottlenecks or processes that limit computational performance in the algorithm and therefore are the processes that must be accelerated to make the algorithm real-time.

4.1 FACE IMAGE DATA SETS

The three data sets selected to represent the range of data used in NIST testing are summarized inTable 7. The 100 KP, 1K data set is a NIST FERET [106, 107] test set, a de facto standard for algorithm comparison in the face identification field. This data set includes a gallery G of 1,196 face images with 384 rows and 256 columns for a resolution N of 98,304 pixels as shown in Table 7. The entire gallery set is used for training so the training set size is also 1,196 images. The number of retained eigenvectors M' is set to 60% of G to maximize identification accuracy [55]. The face images in the 100 KP, 1K data set contain 77 pixels between the centers of the eyes, and the EBGM mask dimension and search region dimension are calculated as 2/3 of the spacing between the eyes or 52 pixels.

The 4 MP, 14K data set represents the current NIST MBE2010 test set [10]. This data set includes a gallery of 14,365 face images with 2,272 rows and 1,704 columns for a resolution of 3,871,488 pixels as shown in Table 7. The training set contains 14,365 images and the retained eigenvectors are set to 8,620 which is 60% of the gallery set. The face images in the 4 MP, 14K data set contain 350 pixels between the centers of the eyes and the EBGM mask dimension and search region dimension are calculated as 2/3 of the spacing between the eyes or 234 pixels.

Table 7: Face Data Set Parameters

Benchmark	Image Size	Image Resolution	Gallery Size	Subspace Vectors	Mask and Search Region Dimension
Eigenface 100 KP, 1K	384 x 256	100 kilopixel	1,196	718	52
Eigenface 4 MP, 14K	2,272 x 1,704	4 megapixel	14,365	8,620	234
Eigenface 4 MP, 1M	2,272 x 1,704	4 megapixel	1,000,000	600,000	234
Bayesian 100 KP, 1K	384 x 256	100 kilopixel	1,196	718	52
Bayesian 4 MP, 14K	2,272 x 1,704	4 megapixel	14,365	8,620	234
Bayesian 4 MP, 1M	2,272 x 1,704	4 megapixel	1,000,000	600,000	234
EBGM 100 KP, 1K	384 x 256	100 kilopixel	1,196	718	52
EBGM 4 MP, 14K	2,272 x 1,704	4 megapixel	14,365	8,620	234
EBGM 4 MP, 1M	2,272 x 1,704	4 megapixel	1,000,000	600,000	234

The 4 MP, 1M data set represents a larger gallery set such as the Terrorist Watch List, with image resolution in the range of current NIST test sets. This data set includes a gallery of one million face images with the same resolution as the 4 MP, 14K data set as shown in Table 7. The training set contains one million images and the retained eigenvectors are set to 60% of the gallery set or 600,000. The EBGM mask dimension and search region are based on image resolution and spacing between the eyes and are therefore set to 234 pixels as in the 4 MP, 14K data set.

4.2 PERFORMANCE ANALYSIS METHOD SELECTION

The performance analysis method will combine complexity analysis and code profiling techniques to estimate execution time. Computational analysis techniques based on complexity analysis will determine the number of iterations for each code line, and code profiling will be used to estimate the time required for a single execution of each line of code. Multiplying the number of iterations for each code line by the single execution time for each line and totaling the results will provide a means to estimate execution time.

4.2.1 Complexity analysis

Complexity analysis is an objective analysis technique that can be performed on an algorithm or program to determine the rate of computational growth of the algorithm [108]. Complexity analysis expresses the order of magnitude of the computation required for the algorithm as a function of variables in the algorithm such as the image resolution or gallery size. Since the purpose of complexity analysis is to determine the relative rate of growth for an algorithm rather than the execution time or the exact number of loop iterations for an algorithm, constants are neglected so that characteristics of the algorithm that change as a result of variables are included in the complexity analysis, while characteristics based on constants are discarded.

Complexity analysis implicitly assumes unit instruction execution time for groups of instructions. If execution time is estimated based only on complexity analysis, the unit time assumption introduces estimation error, since the time required for execution of assembly language instructions for contemporary processors varies with the particular instruction and processor. For example, the average Cycles Per Instruction (CPI) for the instruction set of the AMD 10H processor ranges from 0.33 to 150 cycles [24]. As a result, the error in execution time estimates based only on complexity analysis for this processor could understate the execution time by up to 150 times, and therefore a more accurate estimation method is required to develop usable execution time estimates.

As noted, complexity analysis ignores constant-valued loop iterations and, therefore, tends to under-estimate the number of iterations. For example, the feature search process for the EBGM face identification algorithm iterates KVR^2W^2 times. R and W are calculated from variable algorithm parameters derived from the image resolution but K=40 and V=25 and these values are constants that are independent of algorithm parameters. As a result, in Big-O complexity analysis the constant terms K and V would be dropped and the complexity would be expressed as $O(R^2W^2)$. For the 100 KP, 1K data set, the number of loop iterations based on the complexity result is therefore $(52^2)(52^2)$ or 7.3E6, while the actual number of loop iterations is $(40)(25)(52^2)(52^2)$=7.3E9, one thousand times the value from the complexity analysis. As a result, estimates of execution time based solely on complexity analysis would contain significant error due to difference in the number of loop iterations.

Retaining the constant valued loop iterations as well as the parameter-based loop iterations provides a more accurate iteration count and therefore a more accurate execution time estimate. Rather than dropping constant valued loop iterations, the computational analysis method developed for this research will keep these constant values, resulting in a more accurate estimate of the actual number of loop iterations. The computational analysis method will use a C code implementation of the algorithm to quantify the number of iterations for each line of code. Each and every loop will be analyzed whether the number of iterations is based on algorithm parameters or constants, resulting in an accurate iteration count for each line of code.

4.2.2 Code Profiling

Existing code profiling tools provide some but not all of the information needed to estimate execution time based on instruction execution counts. Gprof [109] is a statistical profiler that instruments the code and counts the number of times each code line is executed, then expresses the execution counts in terms of time and percentage of time. As a statistical tool, Gprof produces more accurate results when more measurements are available and therefore provides more accurate measurements expressed in smaller time resolution units for code that iterates many times. However, Gprof only provides time measurements for functions within a program and does not generate information per line of code or on the relative time required for execution of different assembly language instructions. The Valgrind Cachegrind tool [110] provides cycle execution counts but assumes each instruction requires one CPI. As noted previously, actual CPI per instruction varies widely for contemporary processors and assuming unit CPI will not produce an accurate execution time estimate. Cachegrind does not expose the specific instructions or provide a way to scale the CPI for different instructions, so there is no practical way to incorporate the actual CPI into the Cachegrind results. The ConcurrentAnalytics tool [111] does provide both cycle and execution counts, but expresses these results for groups of instructions and does not break it down to the level of individual assembly language instructions. This tool provides the needed information for groups of instructions, but information per instruction is needed to analyze the ISA. Each tool provides a piece of the needed information, but none of these tools provide all of the information needed to accurately estimate execution time from instruction counts and to relate the execution counts to individual assembly language instructions. The computational analysis method described in the next section was therefore developed to provide the information needed to estimate execution time.

4.2.3 Computational Analysis Method

The execution time for a face identification algorithm will be estimated with the following method:

<u>Step 1: Code and verify the algorithm</u>. The face identification algorithm is coded in C and compiled using the gcc compiler. Functionality for the C code is verified by running the program with the 100 KP, 1K data set and comparing the identification results with the results of the CSU FaceID software [60], a research software application that implements verified versions of the Eigenface, Bayesian, and EBGM algorithms.

<u>Step 2: Evaluate code optimizations</u>. The C code is separately compiled three times with the GNU gcc compiler [112] with the optimization level set to O0, O1, and O2. Code listings showing the assembly language code interleaved with the C code are produced for each compilation using the Objdump application [113]. The code produced for each line of C code is then compared across the optimization levels and the assembly language code segment that uses the fewest instructions for each line of C code is selected.

<u>Step 3: Tabulate assembly instructions.</u> Each line of the C code and the assembly language code generated from that C code line is pasted into a spreadsheet. The assembly language instructions are counted to determine the total number of executions for each instruction type for each line of C code line as shown in the light gray shaded area in Figure 4. The number of times each instruction is executed is shown in the *Execution Count* row in Figure 4.

	addsd	mulsd	movsd	Totals	Percentage Time
p_curl[k]=p_curl[k]+U[j][k]*p_bar[j]					
400652 f2 0f 10 8d 28 ff ff movsd -0xd8(%rbp),%xmm1 400659 ff			1	1	25.0%
40065a f2 0f 10 95 e8 fc ff movsd -0x318(%rbp),%xmm2 400661 ff			1	1	25.0%
400662 f2 0f 10 85 90 fe ff movsd -0x170(%rbp),%xmm0 400669 ff			1	1	25.0%
40066a f2 0f 59 c2 mulsd %xmm2,%xmm0		1		1	25.0%
40066e f2 0f 58 c1 addsd %xmm1,%xmm0	1			1	25.0%
400672 f2 0f 11 85 28 ff ff movsd %xmm0,-0xd8(%rbp)			1	1	25.0%
400679 ff					
Execution Count:	1	1	4		
AMD 10H CPI:	1.0	1.0	0.5		
Total Cycles:	1.00	1.00	2.00	4.00	
			Execution Time (minutes)	2.02E-11	100%

Figure 4: Instruction tabulation example

<u>Step 4: Use CPI measurements to estimate total cycles per line and instruction.</u> The number of cycles required to execute a C code line is calculated from the Execution Counts and the average *Cycles Per Instruction* (CPI) for each instruction as shown in Equation (4.1) and the light gray shaded area in Figure 4.

$$C_{Exec} = \sum_{k \in \text{All Instructions}} \left(ExecutionCount_k\right)\left(CPI_k\right) \tag{4.1}$$

The CPI for a given instruction is determined by the *Instruction Set Architecture* (ISA) and hardware architecture of the processor. However, CPI for a particular instruction can vary for different executions of that same instruction as a result of processor resource sharing, pipeline operation, and data dependencies [21]. For example, two successive instructions that do not share resources or data can be pipelined, reducing the number of cycles for the executions of those instructions and therefore the total execution time for those two instructions. However, if an input to the second instruction is an output of the first instruction, the first instruction must produce the data before the second instruction can use it, increasing the number of cycles and therefore the total execution time for the two instructions.

Accurate values for CPI are required to estimate execution time, but the CPI values provided by the manufacturer are the latency values [114, 115] and are the worst case number of cycles required to execute a particular instruction. For example, AMD uses the term static execution latency, defined as "the number of clock cycles it takes to execute the serially dependent sequence of micro-ops that comprise the instruction" [114]. This definition expresses that the latency is quantified for instruction execution that does not take advantage of resources and optimizations to reduce the execution time. As a result, the CPI specified as static execution latency is higher than the average CPI expected when a code segment is executed. Therefore, an execution time estimation based on the static execution latency would significantly over-estimate execution time.

More accurate CPI values for executing programs can be determined experimentally. Agner Fog performed experiments to quantify average CPI for several contemporary processors [24]. Fog wrote sets of benchmarks and timed execution of these benchmarks to quantify average CPI values for each instruction. Instructions that operate on input data require two benchmarks to measure both the data dependent and data independent average CPI. The benchmarks were executed on the target architecture and timed using the hardware clock cycle counters in the CPU to measure the actual number of clock cycles required for execution of each benchmark. These experiments were repeated multiple times and the results averaged to calculate an average data dependent and data independent CPI for each instruction.

Fog reports two values for each instruction, which he terms *latency* and *reciprocal throughput*. Latency is “the number of clock cycles it takes to execute the serially dependent sequence of micro-ops that comprise the instruction” [24] while reciprocal throughput is “the maximum number of instructions of the same kind that can be executed per clock cycle when the operands of each instruction are independent of the preceding instructions” [24]. Fog’s latency is therefore the CPI for data dependent execution of an instruction and represents the largest average CPI for an instruction. The reciprocal throughput is the CPI for data independent execution of an instruction and represents the lowest average CPI and therefore the best performance for an instruction.

The average CPI values used to estimate execution time in this analysis method are the reciprocal throughput values reported by Fog [24]. The purpose of estimating execution time for this research is to determine the best case performance to quantify whether a face identification algorithm could execute in real-time. Using the reciprocal throughput values for the estimate will estimate best case performance and therefore will accomplish this goal. As noted previously, using static latency values would overstate actual execution time and therefore would not a good basis for an accurate estimate. The best case average CPI may tend to understate execution time in some cases, but contemporary compilers are expected to generate efficient code that can approach the best case execution time, so actual execution time should approach the estimate. This assumption was validated by estimating execution time for the face identification processes used for this research using Fog’s reciprocal throughput values and then timing execution of the same processes on a reference architecture, a PC with an AMD 10H processor. This procedure showed that the estimated execution time was within 1% of the measured execution time.

The number of cycles required to execute a line of code is estimated based on the execution count and the CPI. The total number of cycles required for a single execution of one line of C code is calculated by multiplying the Execution Count by the average CPI for each instruction as shown in the lightest gray shaded area in Figure 4 and Equation (4.1) and summing for all instructions.

Step 5: Estimate the number of loop iterations. The complexity of the C code is analyzed to quantify the number of iterations for each line of code for each data set. Constant iteration counts as well as variable iteration counts are retained, and the iteration counting process is repeated for each data set.

Step 6: Estimate execution time and percentage time for each code line. The total number of cycles required for execution of each C code line is calculated by multiplying the single iteration cycle count by the number of iterations for that code line. The number of cycles required for each code line is summed over the algorithm to determine the number of cycles for the algorithm.

The execution time for the algorithm $T_{ExecAllLines}$ is estimated by dividing the total number of cycles required to execute all lines of code $C_{AllLines}$ by the system clock frequency f_{SysClk} as shown in Equation (4.2) and on the bottom line of the spreadsheet in Figure 4, where it is listed as "Execution Time" and expressed in minutes.

$$T_{ExecAllLines} = C_{AllLines} / f_{SysClk} \qquad (4.2)$$

The percentage of time required for execution of each line is calculated as the ratio of the cycles per line divided by the total number of cycles for the entire algorithm Equation (4.3) as shown in the line labeled Total Cycles in Figure 4. The system clock period is common to both the numerator and denominator and therefore drops out of the Equation as shown in Equation (4.3), allowing calculation of percentage of time as the ratio of the clock counts.

$$T_{Percentage} = \frac{C_{OneLine} T_{SysClk}}{C_{AllLines} T_{SysClk}} = \frac{C_{OneLine}}{C_{AllLines}} \tag{4.3}$$

<u>Step 7: Extract benchmarks.</u> The percentage time results are analyzed to determine which lines of high level code consume the majority of the execution time and are therefore the bottleneck code segments, and the bottleneck code segments are extracted to form the benchmarks. Code lines with an execution time of less than 0.1 second for all data sets are removed to simplify the benchmarks.

4.3 EIGENFACE ALGORITHM ANALYSIS

The Eigenface algorithm [48, 116] applies Principal Component Analysis (PCA) techniques [50] to encode the probe image and then compares the encoded probe to a set of previously encoded gallery images to find the best match as shown in Figure 5. The probe image, **I**, is first converted to a probe vector, **p**, by appending the columns. The mean image vector, $\bar{\mathbf{a}}$, contains a mean value for each pixel calculated during the training process by averaging the pixel over all the training images. The mean subtracted probe vector, $\bar{\mathbf{p}}$, is then projected into the Eigenvector subspace by multiplying the probe vector by the transpose of the subspace matrix, $\mathbf{U}^{\mathbf{T}}$. The resulting vector, $\breve{\mathbf{p}}$, is the encoded probe vector.

Comparison is performed as a series of pair comparisons between encoded probe vector, $\breve{\mathbf{p}}$, and the matrix of encoded gallery members, **Z**. The pair comparison metric is a vector distance calculation such as Euclidean vector distance. While Turk used Euclidean vector distance in the original Eigenface paper [48], subsequent research showed that using Mahalinobis vector distance significantly improved identification accuracy [55] and Mahalinobis distance will therefore be used for this research.

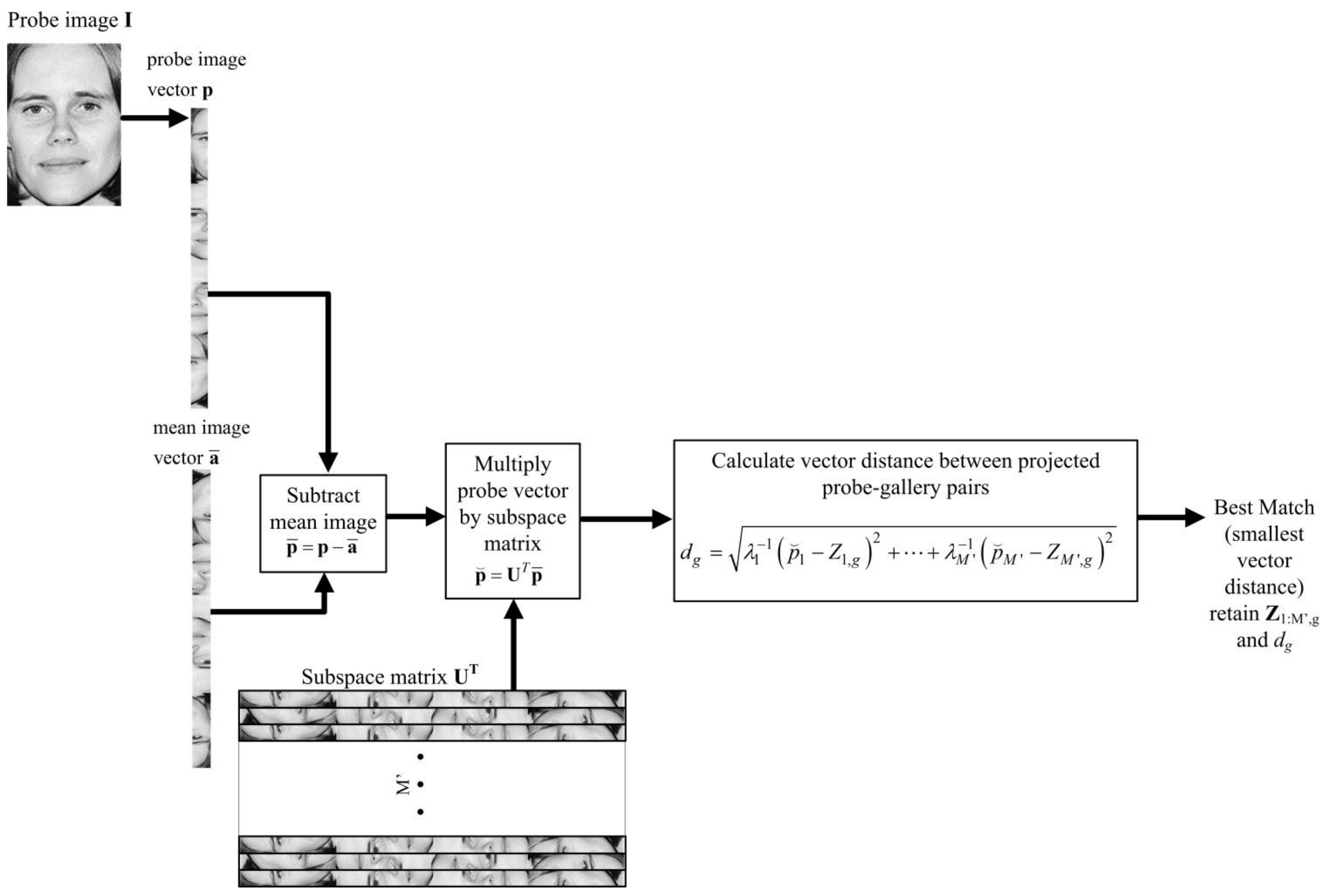

Probe image $\mathbf{I}$
probe image vector $\mathbf{p}$
mean image vector $\bar{\mathbf{a}}$
Subtract mean image $\bar{\mathbf{p}} = \mathbf{p} - \bar{\mathbf{a}}$
Multiply probe vector by subspace matrix $\breve{\mathbf{p}} = \mathbf{U}^T \bar{\mathbf{p}}$
Calculate vector distance between projected probe-gallery pairs
$d_g = \sqrt{\lambda_1^{-1}\left(\breve{p}_1 - Z_{1,g}\right)^2 + \cdots + \lambda_{M'}^{-1}\left(\breve{p}_{M'} - Z_{M',g}\right)^2}$
Best Match (smallest vector distance) retain $\mathbf{Z}_{1:M',g}$ and d_g
Subspace matrix $\mathbf{U}^{\mathbf{T}}$
M'

Figure 5: Eigenface algorithm overview.

The Eigenface algorithm is shown in equation form in Figure 6. The conversion of an N pixel probe image, **I**, to an N element probe vector, **p**, is shown as a function named ConvertImageToVector. The subtraction of the mean vector, $\bar{\mathbf{a}}$, from probe vector, **p**, to form the mean-subtracted probe vector, $\bar{\mathbf{p}}$, is shown in the second line in Figure 6, and the projection of $\bar{\mathbf{p}}$ into the Eigenvector subspace is shown on line 3.

```
$BestMatch = \text{EFId}(\mathbf{I},\mathbf{Z})$
1 $\mathbf{p}=\text{ConvertImageToVector}(\mathbf{I})$
2 $\bar{\mathbf{p}}=(\mathbf{p}-\bar{\mathbf{a}})$   // Subtract mean from probe pixels
3 $\breve{\mathbf{p}}=\mathbf{U}^T\bar{\mathbf{p}}$   // Project probe into subspace
4 For each encoded gallery face $\mathbf{Z}_{1:M',g}$ in $\mathbf{Z}$
5     $d_g = \sqrt{\lambda_1^{-1}\left(\breve{p}_1 - Z_{1,g}\right)^2 + \cdots + \lambda_{M'}^{-1}\left(\breve{p}_{M'} - Z_{M',g}\right)^2}$
6     Keep best match $d_g$ and corresponding $\mathbf{Z}_{1:M',g}$
```

Figure 6: Eigenface face identification algorithm.

The subspace matrix $\mathbf{U^T}$ contains M' rows, where each row is an Eigenvector calculated from the training images and M' represents the number of retained Eigenvectors. During training, Eigenvalues and Eigenvectors are calculated in the same quantity as the number of training images. The Eigenvalue corresponds to the amount of information the Eigenvector contributes to the image, and the Eigenvectors corresponding to smaller valued Eigenvalues can be discarded with minimal loss of information. Given M training images, M' is the number of Eigenvectors that are retained from the original M member set of Eigenvectors and is typically set to 60% to maintain the highest identification accuracy [55].

The comparison of the encoded probe vector to each member of gallery set, **Z**, is shown as lines 4 through 6 in Figure 6. The loop on line 4 iterates through each encoded gallery vector, $\mathbf{Z}_{1:M',g}$, and calculates Mahalinobis distance, d_g, between the encoded probe and gallery vectors. The Mahalinobis distance calculation is similar to a Euclidean vector distance and quantifies the difference between the two vectors, and the probe-gallery pair with the smallest Mahalinobis distance is the best match.

4.3.1 Eigenface Algorithm Computational Analysis

Computational analysis of the Eigenface algorithm is shown in Figure 7. The first column numbers each code line and lists the corresponding C code. Three sets of computational analysis results are shown, one for each data set. For each data set, the Computation Cycles column lists the total number of cycles for each line of code for all assembly language instructions and iterations for that line. The Percent column lists the percentage of total execution time required for that code line.

The Total Cycles row in Figure 7 lists the number of cycles required for the algorithm, and the Total Time row shows the total execution time in minutes. The computational analysis shown in Figure 7 indicates that the probe projection process, line 3 through line 6, iterates M'N times and consumes 62.0% to 99.1% of the execution time, depending on the data set. This code segment is therefore the primary bottleneck. However, the comparison process, line 9 through line 12, consumes 0.9% to 38.0% of the execution time and therefore must also be included in the benchmark.

The Eigenface benchmark is shown in Figure 8. The computational analysis of the Eigenface benchmark shows that the difference between the benchmark execution time and the execution time for the full algorithm (Figure 7) ranges from 1.6 x 10^{-6} minute to 0.00034 minute, or 0.09 ms to 20.35 ms, and can therefore be neglected when evaluating real-time performance.

		100 KP, 1K		4 MP, 14K		4 MP, 1M	
		N=1E5, G=1E3, M'=6E2		N=4E6, G=14E3, M'=9E3		N=4E6, G=1E6, M'=6E5	
		Computation cycles	**Percent**	**Computation cycles**	**Percent**	**Computation cycles**	**Percent**
Eigenface Algorithm							
1	for each probe element index j in N						
2	p_bar[j] = p[j] - a[j]	2.5E+05	<0.1%	9.7E+06	<0.1%	9.7E+06	<0.1%
3	for each Eigenvector index k in M'						
4	p_curl[k]=0	3.6E+02	<0.1%	4.3E+03	<0.1%	3.0E+05	<0.1%
5	for each probe element index j in N						
6	p_curl[k] = p_curl[k] + U[j][k] * p_bar[j]	2.8E+08	97.1%	1.3E+11	99.1%	9.3E+12	62.0%
7	d_save=9999	5.0E-01	<0.1%	5.0E-01	<0.1%	5.0E-01	<0.1%
8	bestmatch=0	5.0E-01	<0.1%	5.0E-01	<0.1%	5.0E-01	<0.1%
9	for each gallery member index g in G						
10	d_sum=0	6.0E+02	<0.1%	7.2E+03	<0.1%	5.0E+05	<0.1%
11	for each coefficient index k in M'						
12	d_sum=d_sum + (Z[k][g] - p_curl[k])^2 * (const[k])	8.2E+06	2.8%	1.2E+09	0.9%	5.7E+12	38.0%
13	d=sqrt(d_sum)	6.1E+04	<0.1%	7.3E+05	<0.1%	5.1E+07	<0.1%
14	if d<d_save then d_save=d; bestmatch=g	6.8E+03	<0.1%	8.1E+04	<0.1%	5.7E+06	<0.1%
	Total Cycles	2.9E+08		1.3E+11		1.5E+13	
	Total Time (minutes)	0.0015		0.6801		75.7153	

Figure 7: Eigenface algorithm computational analysis.

	Eigenface Benchmark	100 KP, 1K		4 MP, 14K		4 MP, 1M	
		N=1E5, G=1E3, M'=6E2		N=4E6, G=14E3, M'=9E3		N=4E6, G=1E6, M'=6E5	
		Computation cycles	**Percent**	**Computation cycles**	**Percent**	**Computation cycles**	**Percent**
1	for each Eigenvector index k in M'						
2	for each probe element index j in N						
3	p_curl[k] = p_curl[k] + U[j][k] * p_bar[j]	2.8E+08	97.2%	1.3E+11	99.1%	9.3E+12	62.0%
4	for each gallery member index g in G						
5	for each coefficient index k in M'						
6	d_sum=d_sum+(Z[k][g] - p_curl[k])^2 * (const[k])	8.2E+06	2.8%	1.2E+09	0.9%	5.7E+12	38.0%
	Total Cycles	2.9E+08		1.3E+11		1.5E+13	
	Total Time (minutes)	0.0015		0.6800		75.7150	

Figure 8: Eigenface benchmark and computational analysis.

Figure 8 further shows that the execution time for the Eigenface benchmark is 0.0015 minute for the 100 KP, 1K and 0.68 minute for the 4 MP, 14K data set and is therefore real-time for both data sets. However, the Eigenface 4 MP, 1M benchmark requires 75.7 minutes for execution and is therefore not real-time, but a speedup of 38 times would reduce execution time to 1.99 minutes and enable real-time identification.

4.4 BAYESIAN ALGORITHM ANALYSIS

The original Bayesian face identification algorithm used a Bayesian probability to determine how well a difference vector represents two images of the same face. Two image vectors such as the probe and one gallery member were subtracted pixel by pixel to form the difference vector, which therefore represents the differences between the two images [61]. During training, difference vectors were calculated between all pair combinations of the training images and the difference vectors were grouped into an interpersonal set and an extrapersonal set. The intrapersonal set consisted of difference vectors calculated from two images of the same person while the extrapersonal set contained difference vectors calculated from two images representing different people. For identification, difference vectors were calculated for all probe-gallery pairs and the Bayesian Maximum a posteriori (MAP) probability was calculated to determine the probability that a given probe-gallery difference vector was in the intrapersonal set and therefore represented a gallery image that matches the probe image. The MAP probability was calculated for all probe-gallery pairs, and the highest probability indicated the best match.

The “efficient” Bayesian algorithm mitigates the computational constraints of the original Bayesian algorithm. The original Bayesian algorithm requires significant computation to calculate the difference vectors for the entire gallery set. Furthermore, estimation of the prior probabilities needed for the Bayesian conditional probability calculations is problematic. These issues motivated application of PCA techniques and the development of the “efficient” Bayesian algorithm [63]. This algorithm projects the image vector into a PCA subspace and calculates the Bayesian Maximum Likelihood (ML) probability to quantify the likelihood that the probe image matches the gallery image. The ML probability requires less calculation because only the intrapersonal set is required, eliminating the need for the calculations involving the extrapersonal set. The PCA process reduces the required computation and provides a way to estimate the prior probabilities, and using only the intrapersonal set further reduces the required computation.

The efficient Bayesian algorithm is shown in Figure 9. The probe image is first converted to a vector, mean-subtracted, and projected into the PCA subspace as in the Eigenface algorithm. However, the Bayesian subspace matrix, $\mathbf{U}^{\mathbf{T}}$, differs from the Eigenface subspace matrix in that during training, the Eigenvectors in the subspace matrix are divided by the square root of the corresponding Eigenvalue. This division during training avoids the need to perform the division during identification.

For comparison, the Bayesian ML probability that the probe-gallery pair represents the same individual, $\text{prob}\left(\breve{\mathbf{p}} = \mathbf{Z}_{:,g}\right)$, is calculated as an exponential function of the Euclidean distance between the probe-gallery vectors scaled by S_I, a subspace constant calculated during training. A probe-gallery pair is formed for each gallery member and the probe-gallery pair with the highest probability represents the best match.

This process is expressed in algorithm form in Figure 10. Lines 1 and 2 convert the probe to a vector and subtract the mean in the same way as the Eigenface algorithm. Line 3 multiplies Bayesian intrapersonal subspace matrix $\mathbf{U}^{\mathbf{T}}$ by the mean-subtracted probe vector $\bar{\mathbf{p}}$ and

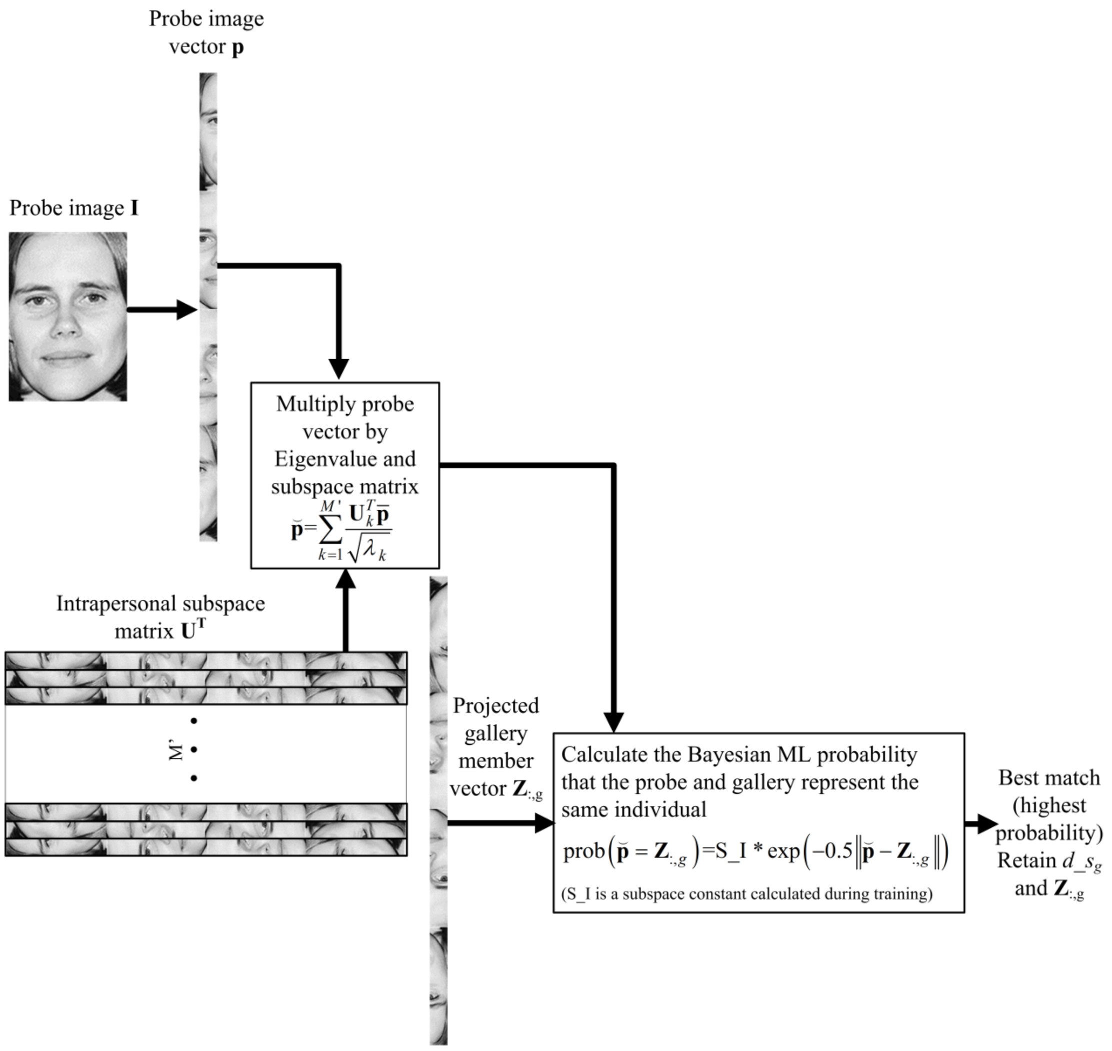

Probe image vector $\mathbf{p}$
Probe image $\mathbf{I}$
Multiply probe vector by Eigenvalue and subspace matrix
$\breve{\mathbf{p}} = \sum_{k=1}^{M'} \frac{\mathbf{U}_k^T \bar{\mathbf{p}}}{\sqrt{\lambda_k}}$
Intrapersonal subspace matrix $\mathbf{U}^T$
M'
Projected gallery member vector $\mathbf{Z}_{:,g}$
Calculate the Bayesian ML probability that the probe and gallery represent the same individual
$\text{prob}\left(\breve{\mathbf{p}} = \mathbf{Z}_{:,g}\right) = \text{S\_I} * \exp\left(-0.5\left\|\breve{\mathbf{p}} - \mathbf{Z}_{:,g}\right\|\right)$
(S_I is a subspace constant calculated during training)
Best match (highest probability) Retain d_s_g and $\mathbf{Z}_{:,g}$

Figure 9: Bayesian face identification algorithm overview.

divides by the square root of the Eigenvalue. This division by the Eigenvalue is shown explicitly in Figure 10, but in practice the division will be performed during training and subspace matrix $\mathbf{U^T}$ will contain pre-divided values to reduce computation by avoiding the need to perform this division at run time.

```
$BestMatch = \mathrm{BAYId}(\mathbf{I}, \mathbf{Z})$
1 $\mathbf{p}$=ConvertImageToVector$(\mathbf{I})$
2 $\bar{\mathbf{p}} = (\mathbf{p} - \bar{\mathbf{a}})$   // Subtract mean from probe pixels
3 $\breve{\mathbf{p}} = \sum_{k=1}^{M'} \frac{\mathbf{U}_k^T \bar{\mathbf{p}}}{\sqrt{\lambda_k}}$, $\mathbf{U}_k^T = \{U_{k,1}, \cdots, U_{k,N}\}$
4 For each encoded gallery face $\mathbf{Z}_{:,g}$ in $\mathbf{Z}$
5     // S_I is a subspace constant calculated during training
6     $\mathrm{prob}(\breve{\mathbf{p}} = \mathbf{Z}_{:,g})$=S_I * $\exp(-0.5\|\breve{\mathbf{p}} - \mathbf{Z}_{:,g}\|)$
7     Keep best match indicated by highest probability
8     Retain probability and $\mathbf{Z}_{:,g}$
```

Figure 10: Bayesian face identification algorithm.

The loop on line 4 in Figure 10 iterates through each of the gallery vectors in gallery set, **Z**, calculating an ML probability $\mathrm{prob}(\breve{\mathbf{p}} = \mathbf{Z}_{:,g})$ for each probe gallery pair. However, the comparison process only needs to determine the relative similarity between the various probe-gallery pairs and this relative value can be calculated with fewer operations than required to calculate the numerical probability value. The comparison between two probabilities $\mathrm{prob}(\breve{\mathbf{p}} = \mathbf{Z}_{:,1}) / \mathrm{prob}(\breve{\mathbf{p}} = \mathbf{Z}_{:,2})$ can be expressed as the ratio of the two exponential terms and, in addition, the constant divides out for both terms, simplifying the probability calculation to $\mathrm{prob}(\breve{\mathbf{p}} = \mathbf{Z}_{:,1}) / \mathrm{prob}(\breve{\mathbf{p}} = \mathbf{Z}_{:,2}) = \exp(-0.5\|\breve{\mathbf{p}} - \mathbf{Z}_{:,1}\|) / \exp(-0.5\|\breve{\mathbf{p}} - \mathbf{Z}_{:,2}\|)$. Since the exponential function is performed on both the numerator and divisor, this expression can be further simplified by removing the exponential which simplifies the expression to

$\text{prob}\left(\breve{\mathbf{p}} = \mathbf{Z}_{:,1}\right) / \text{prob}\left(\breve{\mathbf{p}} = \mathbf{Z}_{:,2}\right) \propto -0.5\left\|\breve{\mathbf{p}} - \mathbf{Z}_{:,1}\right\| / -0.5\left\|\breve{\mathbf{p}} - \mathbf{Z}_{:,2}\right\|$. This expression can be further rewritten as $\dfrac{\text{prob}\left(\breve{\mathbf{p}} = \mathbf{Z}_{:,1}\right)}{\text{prob}\left(\breve{\mathbf{p}} = \mathbf{Z}_{:,2}\right)} \propto \dfrac{\sqrt{\left(\breve{p}_1 - Z_{1,1}\right)^2 + \cdots + \left(\breve{p}_{M'} - Z_{M',1}\right)^2}}{\sqrt{\left(\breve{p}_1 - Z_{1,2}\right)^2 + \cdots + \left(\breve{p}_{M'} - Z_{M',2}\right)^2}}$, which is the ratio of the Euclidean distances between the probe and gallery vectors. Thus, Euclidean distance can be used a comparison metric for the efficient Bayesian algorithm.

4.4.1 Bayesian Algorithm Computational Analysis

The computational analysis of the Bayesian algorithm is shown in Figure 11. As with the Eigenface analysis in Figure 7, the first column numbers each code line and lists the C code and three sets of computational analysis results are shown, one for each data set. For each data set, the Computation Cycles column lists the total number of cycles for each line of code for all instructions and iterations for that line. The Percent column lists the percentage of total execution time required for that code line.

The computational analysis in Figure 11 shows that the probe projection, line 3 through line 6, iterates M'N times and consumes 65.9% to 99.3% of the execution time depending on the data set. This code segment is therefore the primary bottleneck. However, the comparison

	Bayesian Algorithm	100 KP, 1K N=1E5, G=1E3, M'=6E2 Computation Cycles	 Percent	4 MP, 14K N=4E6, G=14E3, M'=9E3 Computation Cycles	 Percent	4 MP, 1M N=4E6, G=1E6, M'=6E5 Computation Cycles	 Percent
1	for each probe element index j in N						
2	p_bar[j]=p[j]-a[j]	2.46E+05	<0.1%	9.68E+06	<0.1%	9.68E+06	<0.1%
3	for each Eigenvector index k in M'						
4	p_curl[k]=0	3.59E+02	<0.1%	4.31E+03	<0.1%	3.00E+05	<0.1%

5	for each probe element index j in N
	// subspace matrix U is divided by sqrt(eigval[k])
	// during training so no division is needed here						

6	p_curl[k] = p_curl[k] + U[j][k] * p_bar[j]	2.82E+08	97.5%	1.33E+11	99.3%	9.29E+12	65.9%
7	d_save=0	5.00E-01	<0.1%	5.00E-01	<0.1%	5.00E-01	<0.1%
8	bestmatch=0	5.00E-01	<0.1%	5.00E-01	<0.1%	5.00E-01	<0.1%
9	for each gallery member index g in G						
10	d_sum=0	5.98E+02	<0.1%	7.18E+03	<0.1%	5.00E+05	<0.1%
11	for each coefficient index k in M'						
12	d_sum = d_sum + (Z[k][g] - p_curl[k])^2	6.87E+06	2.4%	9.90E+08	0.7%	4.80E+12	34.1%
13	d = exp(-1/2 * sqrt(d_sum))	2.09E+05	<0.1%	2.51E+06	<0.1%	1.75E+08	<0.1%
14	if d>d_save then d_save=d; bestmatch=g	6.78E+03	<0.1%	8.14E+04	<0.1%	5.67E+06	<0.1%
	Total Cycles	2.9E+08		1.3E+11		1.4E+13	
	Total Time (minutes)	0.00146		0.67917		71.17049	

Figure 11: Bayesian computational analysis.

process, line 9 through line 12, consumes 0.7% to 34.1% of the execution time and therefore must also be included in the benchmark.

The computational analysis of the Bayesian benchmark shown in Figure 12 indicates that the difference between the benchmark execution time and the execution time for the full algorithm (Figure 7) ranges from 2.3×10^{-6} minute to 0.00094 minute or 0.14 ms to 56.10 ms and therefore can be neglected when evaluating real-time performance. Figure 12 further shows that the Bayesian benchmark achieves identification time of 0.00146 minute for the 100 KP, 1K data set and 0.6791 minute for the 4 MP, 14K data set, both real-time. The Bayesian 4 MP, 1M benchmark requires 71.169 minutes and is therefore not real-time, but a speedup of 36 times would reduce execution time to 1.98 minutes and make the benchmark real-time.

	100 KP, 1K		4 MP, 14K		4 MP, 1M	
	N=1E5, G=1E3, M'=6E2		N=4E6, G=14E3, M'=9E3		N=4E6, G=1E6, M'=6E5	
Bayesian Algorithm	**Computation Cycles**	**Percent**	**Computation Cycles**	**Percent**	**Computation Cycles**	**Percent**
1 for each Eigenvector index k in M'						
2 for each probe element index j in N						
3 p_curl[k] = p_curl[k] + U[j][k] * p_bar[j]	2.82E+08	97.6%	1.33E+11	99.3%	9.29E+12	65.9%
4 for each gallery member index g in G						
5 for each coefficient index k in M'						
6 d_sum = d_sum + (Z[k][g] - p_curl[k])^2	6.87E+06	2.4%	9.90E+08	0.7%	4.80E+12	34.1%
Total Cycles	2.9E+08		1.3E+11		1.4E+13	
Total Time (minutes)	0.00146		0.67911		71.16955	

Figure 12: Bayesian benchmark and computational analysis.

4.5 EIGENFACE AND BAYESIAN ALGORITHM COMPARISON

The use of PCA techniques in the efficient Bayesian algorithm results in an algorithm that is quite similar to the Eigenface algorithm [117]. In the efficient Bayesian algorithm, each probe or gallery image is divided by the square root of the Eigenvalue as part of the PCA projection process. However, as noted previously the elements of the subspace matrix, $\mathbf{U}^T$, can be divided by the square roots of the Eigenvalues during training, resulting in a probe projection calculation that is identical to the Eigenface algorithm. As a result, the only difference between the Eigenface and Bayesian probe projection processes is the numerical value of the coefficients stored in the subspace matrix, $\mathbf{U}^T$. The probe projection process is therefore computationally identical for the Eigenface and Bayesian algorithms as shown in the summary of the computational analysis results in Figure 13.

	100 KP, 1K		4 MP, 14K		0	
	N=1E5, G=1E3, M'=6E2		N=4E6, G=14E3, M'=9E3		N=4E6, G=1E6, M'=6E5	
	Eigenface Computation cycles	**Bayesian Computation cycles**	**Eigenface Computation cycles**	**Bayesian Computation cycles**	**Eigenface Computation cycles**	**Bayesian Computation cycles**
Projection	2.82E+08	2.82E+08	1.33E+11	1.33E+11	9.29E+12	9.29E+12
Comparison	8.16E+06	6.87E+06	1.18E+09	9.90E+08	5.70E+12	4.80E+12

Figure 13: Eigenface and Bayesian computation comparison.

The comparison process differs slightly for the Eigenface and Bayesian algorithms. When the Mahalinobis distance is used as the comparison metric in the Eigenface algorithm, the difference terms in the Euclidean norm calculated between the projected probe and gallery vectors is divided by the corresponding Eigenvalue as shown for the calculation of d_g in Figure 6. This calculation is computationally equivalent to the pre-scaling by the square roots of the Eigenvalues in the Bayesian algorithm, but in this case the terms of the Euclidean norm calculation for the probe-gallery difference vector are divided by the Eigenvector, so the division cannot be factored out and performed during training. However, to reduce computation time the division can be reduced to a multiplication as shown by multiplying by the reciprocal of the Eigenvalues instead of dividing by the Eigenvalues.

As noted, the comparison of the Bayesian ML probability is proportional to $\sqrt{\left(\breve{p}_1 - Z_{1,g}\right)^2 + \cdots + \left(\breve{p}_{M'} - Z_{M',g}\right)^2}$, the Euclidean vector distance. The Eigenface comparison metric is Mahalinobis distance $d_g = \sqrt{\lambda_1^{-1}\left(Z_{1,g} - \breve{p}_1\right)^2 + \cdots + \lambda_{M'}^{-1}\left(Z_{M',g} - \breve{p}_{M'}\right)^2}$, which multiples each squared Euclidean vector norm term by the reciprocal of the corresponding Eigenvalue. Therefore, the computational for the Eigenvalue benchmark requires one additional multiplication per vector term in comparison to the Bayesian benchmark. As the computational summary in Figure 13 shows, this additional multiplication in the Eigenface comparison process increases the number of cycles required for execution by 18% over the execution cycle count for the Bayesian comparison.

Improvements that increase performance for the Eigenface benchmark will also improve performance for the Bayesian benchmark. Since the probe projection is the same for both benchmarks and the Eigenface comparison process requires 18% more cycles than the Bayesian comparison process, accelerating the Eigenface benchmark to achieve real-time performance will also accelerate the slightly less computationally intensive Bayesian benchmark to achieve real-time performance. The analysis in the following chapters will therefore be performed on the Eigenface benchmark and the Bayesian results will be shown in summary form with the application of the same performance improvements as the Eigenface benchmark.

4.6 ELASTIC BUNCH GRAPH MATCHING ALGORITHM ANALYSIS

The Elastic Bunch Graph Matching (EBGM) face identification algorithm encodes both the probe and gallery images as graphs where the vertices represent facial features. The probe and gallery face graph pairs are then compared to select the gallery face that best matches the probe face [65, 66].

The vertices of the face graph are labeled with the coordinates of the facial feature such as the center of the eye or tip of the nose in addition to an encoded representation of the feature called a *jet*. The jet is formed by convolving an image region surrounding a feature with a set of 40 Gabor wavelet filters, implemented as a set of 80 real Gabor wavelet filter masks representing the real and imaginary parts of the filter [118, 119], and the response is a function of the coordinates of the feature of the pixel values as well as the filter frequency, phase, and orientation [65]. The filter responses are sampled at the feature coordinates and one complex value from each of the 40 filters is stored in the jet as the real and imaginary coefficients and the jet is thus a 40 by two element vector.

The jet captures the feature and texture information needed for identification of a face. As shown in Figure 14, a region of the face image is first filtered with the set of 40 Gabor filters and the center points of these filter responses are combined into a vector to form the jet.

The coefficients of the real and imaginary parts of the filter response are calculated as real operations using two masks, $\mathbf{F}_k$ to calculate the real coefficient for the k^{th} filter and $\mathbf{H}_k$ to calculate the imaginary coefficient. The filter responses are then sampled at the center points and combined to form a vector of complex numbers in polar form.

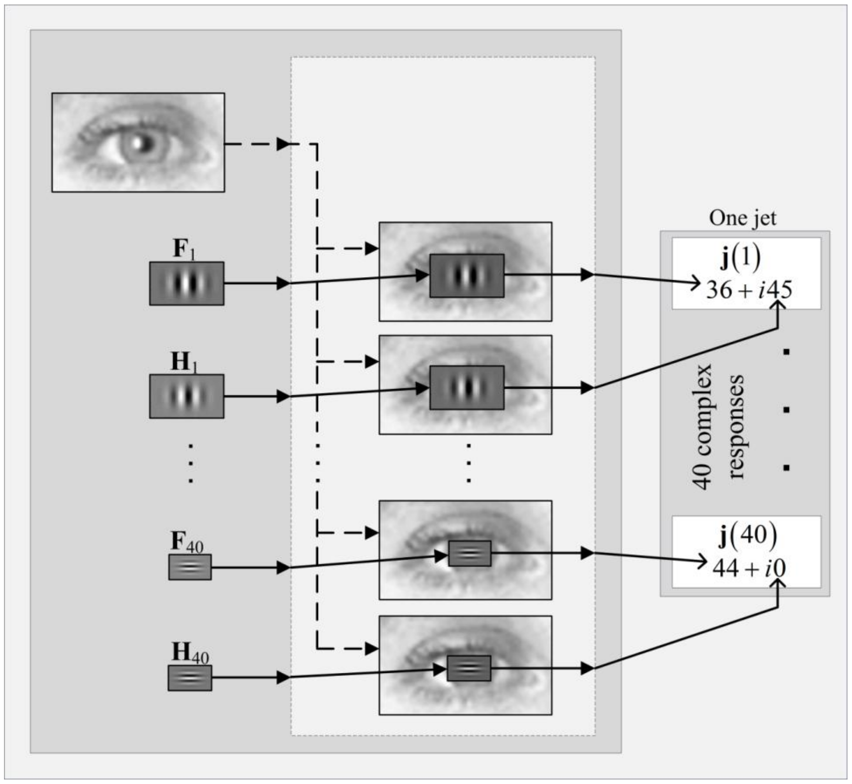


Figure 14: The jet is the encoded representations of one image region.

A single "bunch" graph combining face graphs from all of the training images is overlaid on the probe face and the vertices are elastically adjusted to locate the probe features as shown in Figure 15. The vertices in the bunch graph are labeled with a bunch, a set of jets representing models of the corresponding feature, one from each training image. The bunch graph vertices are also labeled with a single coordinate pair representing the average of the coordinates for the feature from all the training images. These average coordinates are used to make an initial estimate of the feature coordinates in a new face image and a model for the feature is selected from the bunch and used to determine the feature coordinates in the new face.

Identification is performed through a series of pair comparisons of the probe face graph with each encoded gallery face graph. The similarity, a scalar measure of the how well two jets match, is calculated between the probe and gallery jets at each vertex in the pair of face graphs. The similarities for all the vertices are averaged to calculate a scalar similarity for the graph, and the probe-gallery pair with the best similarity is the best match.

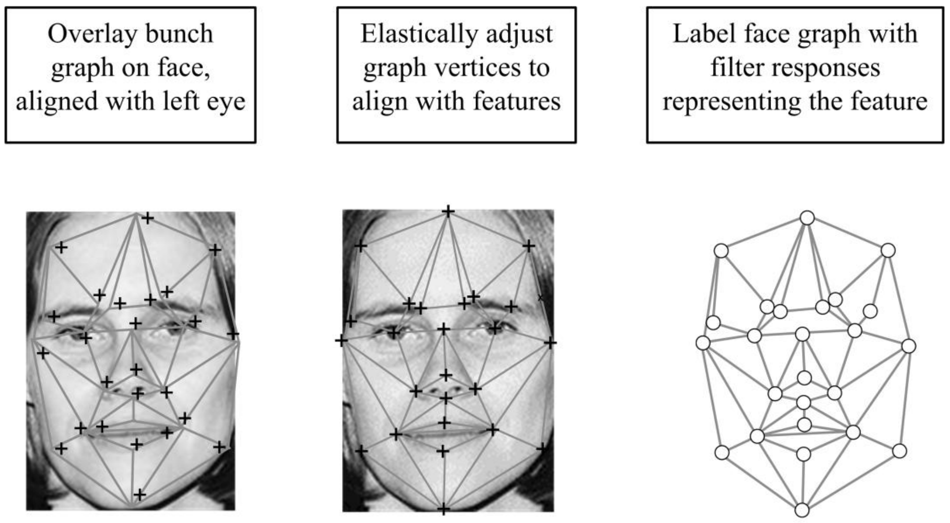

Overlay bunch graph on face, aligned with left eye
Elastically adjust graph vertices to align with features
Label face graph with filter responses representing the feature

Figure 15: EBGM overlays a bunch graph on the face image and extracts a face graph.

The EBGM algorithm performs face identification with the three-step process shown as pseudo code in Figure 16. The LocateFeatures process overlays the bunch graph on the face graph, aligning the coordinates of the left eye with the left eye coordinates in the probe image. The left eye coordinates for the probe image are determined during image preprocessing. The coordinates of the probe features are iteratively estimated using the average coordinates in the bunch graph to elastically adjust the bunch graph to overlay the probe features. A model for each feature is then selected from the bunch, and jets are calculated at each point in the image region surrounding the estimated feature coordinates and compared to the model jet to find the true feature coordinates.

```
$\mathrm{S}_{BEST} = \mathrm{EBGMid}\left(\mathbf{I}, \mathbf{B}, \left(x_e, y_e\right)\right)$
    $\mathbf{LocateFeatures}\left(\mathbf{I}, \mathbf{B}, x_{LeftEye}, y_{LeftEye}\right)$
    $\mathbf{MakeFaceGraph}\left(\mathbf{I}, \{\mathbf{F}\}, \{\mathbf{H}\}, \mathbf{x}, \mathbf{y}\right)$
    $\mathbf{Comparison}\left(\mathbf{J_P}, \{\mathbf{J_G}\}\right)$
```

Figure 16: Top level EBGM pseudo code.

A jet is calculated at the coordinates of each feature and at the midpoints of selected graph edges and the face graph contains a total of 80 jets representing 25 features and 55 edge midpoints. The probe face graph is represented as a matrix $\mathbf{J}_P$ that contains the coordinates of each feature and edge vertex and the jet calculated at those coordinates.

The probe face graph $\mathbf{J}_P$ is compared to each gallery face graph in a set of gallery graphs $\{\mathbf{J}_G\}$ to find the best match. Similarity scores analogous to a vector distance are calculated between pairs of feature vertices, one from the probe graph and one from the gallery graph. The vertex similarities are averaged over the graph to calculate a scalar similarity for the graph pair. The graph pair with the best similarity indicates the gallery graph that is most like the probe graph.

4.6.1 EBGM Algorithm Computational Analysis

The C code implementation for the EBGM algorithm is shown in Figure 17. Only the LocateFeatures code is shown in detail as the computational analysis shown in Figure 18 indicates that this is the computation intensive section.

The computational analysis in Figure 18 shows that the feature search process, line 7 through line 8.9, consumes 98.4% to nearly 100% of the execution time, depending on the data set. Note that these lines are within the vertex iteration loop on line 2 and that lines 8.4 and 8.5 iterate KVR^2W^2 times, where K=40 and V=25 so that KV=1,000 and the number of iterations is $1000R^2W^2$.

The computational analysis of the EBGM benchmark shown in Figure 19 shows that the benchmark execution time is within 0.02 minute or 1.006 second of the execution time for the full algorithm shown in Figure 18. Figure 19 further shows that the EBGM benchmark is not real-time for any of the data sets. The 100 KP, 1K data set requires 3.70 minutes and a speedup of two times would reduce execution time to a real-time 1.85 minutes. The 4 MP, 14K data set requires 1,495.16 minutes and the 4 MP, 1M data set requires 1,496.16 minutes. A speedup of 748 times would reduce execution time to 1.99 minutes and 2.00 minutes respectively and make the EBGM benchmark real-time with both data sets.

EBGM Algorithm

```
    //LocateFeatures
    // Estimate image feature coordinates from the left eye and bunch graph coordinates
 1  ({x},{y})=EstimateCoordinates(B, l_eye_x, l_eye_y)
 2  for each feature vertex v, v=1 to 25
        // Calculate an image jet
 3      j=CalcJet({F}, {H}, I, dx, dy)
        // Select the best model from the bunch for the image jet
 4      for each jet b in the bunch, b=1 to M
            // compare bunch jet b to the image jet j
 5          s=CalcMagnitudeSimilarity(j, b)
 6          keep model jet b with best similarity
        // Search for the coordinates of the feature in the image
 7      for each half-point (dx, dy) in an R by R region centered at (x,y)
 8          // j=CalcJet({F}, {H}, I, dx, dy)
            Sample 40 filter responses centered at one coordinate pair to calculate one jet j
            8.1  for each filter k in the set of wavelet filters, k=1 to 40
                     // Calculate filter response at integer coordinates close to the decimal coordinates (x,y)
            8.2      for each row r in the image region centered at (int[x],int[y]), r=1 to W
            8.3          for each column c in the image region centered at (int[x],int[y]), c=1 to W
                             // Multiply the mask coefficient by the image pixel and accumulate the result
            8.4              real_part = real_part + FaceImage[r][c] * RealMask[k][r][c]
            8.5              imag_part = imag_part + FaceImage[r][c] * ImagMask[k][r][c]
                     // Adjust the phase to move the response to the decimal coordinates
            8.6      dx = x - int(x)
            8.7      dy = y - int(y)
                     // fx[k] and fy[k] are precalculated constants for the kth filter
            8.8      real_part =  sqrt(real_part^2 + imag_part^2) * cos(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)
            8.9      imag_part =  sqrt(real_part^2 + imag_part^2] * sin(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)

 9          s=CalcPhaseSimilarity(j, b)
10          keep coordinates (x, y) of the best match
11  MakeFaceGraph
12  Comparison
```

Figure 17: C code for the EBGM algorithm.

		100 KP, 1K		4 MP, 14K		4 MP, 1M	
		N=1E5, G=1E3, M'=6E2		N=4E6, G=14E3, M'=9E3		N=4E6, G=1E6, M'=6E5	
EBGM Algorithm		**Computation Cycles**	**Percent**	**Computation Cycles**	**Percent**	**Computation Cycles**	**Percent**
	//LocateFeatures						
1	({x},{y})=EstimateCoordinates(B, l_eye_x, l_eye_y)	1.47E+05	<0.01%	1.47E+05	<0.01%	1.47E+05	<0.01%
2	for each feature vertex v, v=1 to 25						
	// Calculate an image jet						
3	j=CalcJet({F}, {H}, I, dx, dy)	2.22E+06	<0.01%	3.43E+07	<0.01%	3.43E+07	<0.01%
	// Select the best model from the bunch for the image jet						
4	for each jet b in the bunch, b=1 to M						
	// compare bunch jet b to the image jet j						
5	s=CalcMagnitudeSimilarity(j, b)	2.39E+08	0.03%	2.87E+09	<0.01%	2.00E+11	0.07%
6	keep model jet b with best similarity						
	// Search for the coordinates of the feature in the image						
7	for each half-point (dx, dy) in an R by R region						
8	// j=CalcJet({F}, {H}, I, dx, dy)						
8.1	for each filter k in the set of wavelet filters						
8.2	for each row r in the image region centered at (int[x],int[y]), r=1 to W						
8.3	for each column c in the image region centered at (int[x],int[y]), c=1 to W						
8.4	real_part + = FaceImage[r][c] * RealMask[k][r][c]	3.61E+11	49.2%	1.48E+14	50.0%	1.48E+14	49.9%
8.5	imag_part + = FaceImage[r][c] * ImagMask[k][r][c]	3.61E+11	49.2%	1.48E+14	50.0%	1.48E+14	49.9%
	// Adjust the phase						
8.6	dx = x - int(x)	8.11E+07	0.0%	1.64E+09	<0.01%	1.64E+09	<0.01%
8.7	dy = y - int(y)	8.11E+07	0.0%	1.64E+09	<0.01%	1.64E+09	<0.01%
8.8	real_part = sqrt(real_part^2 + imag_part^2) * cos(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	2.93E+09	0.4%	5.94E+10	0.02%	5.94E+10	0.02%
8.9	imag_part = sqrt(real_part^2 + imag_part^2] * sin(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	2.93E+09	0.4%	5.94E+10	0.02%	5.94E+10	0.02%
9	s=CalcPhaseSimilarity(j, b)	4.76E+09	0.6%	9.63E+10	0.03%	9.63E+10	0.03%
10	keep coordinates (x, y) of the best match						
11	**MakeFaceGraph**	4.90E+04	<0.01%	4.90E+04	<0.01%	4.90E+04	<0.01%
12	**Comparison**	4.92E+07	<0.01%	5.91E+08	<0.01%	4.11E+10	0.0%
	Total Cycles	7.3E+11		3.0E+14		3.0E+14	
	Total Time (minutes)	3.70		1,495.18		1,496.18	

Figure 18: EBGM algorithm computational analysis.

		100 KP, 1K		4 MP, 14K		4 MP, 1M	
		N=1E5, G=1E3, M'=6E2		N=4E6, G=14E3, M'=9E3		N=4E6, G=1E6, M'=6E5	
	EBGM Benchmark	**Computation Cycles**	**Percent**	**Computation Cycles**	**Percent**	**Computation Cycles**	**Percent**
1	for each feature vertex v, v=1 to 25						
2	for each jet b in the bunch, b=1 to M						
3	s=CalcMagnitudeSimilarity(j, b)	2.39E+08	0.03%	2.87E+09	<0.01%	2.00E+11	0.07%
4	for each half-point (dx, dy) in an R by R region centered at (x,y)						
4.1	for each filter k in the set of wavelet filters, k=1 to 40						
4.2	for each row r in the image region centered at (int[x],int[y]), r=1 to W						
4.3	for each column c in the image region centered at (int[x],int[y]), c=1 to W						
4.4	real_part + = FaceImage[r][c]* RealMask[k][r][c]	3.61E+11	49.3%	1.48E+14	50.0%	1.48E+14	49.9%
4.5	imag_part + = FaceImage[r][c]* ImageMask[k][r][c]	3.61E+11	49.3%	1.48E+14	50.0%	1.48E+14	49.9%
4.6	real_part = sqrt(real_part^2 + imag_part$^{2)}$ * cos(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	2.93E+09	0.4%	5.94E+10	0.02%	5.94E+10	0.02%
4.7	imag_part = sqrt(real_part^2 + imag_part^2] * sin(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	2.93E+09	0.4%	5.94E+10	0.02%	5.94E+10	0.02%
5	s=CalcPhaseSimilarity(j, b)	4.76E+09	0.6%	9.63E+10	0.03%	9.63E+10	0.03%
	Total Cycles	7.3E+11		3.0E+14		3.0E+14	
	Total Time (minutes)	3.70		1,495.16		1,496.16	

Figure 19: EBGM benchmark and computational analysis.

4.7 CONCLUSION

Three data sets were used to develop three benchmarks for each algorithm. Table 8 shows the benchmarks, the estimated execution time, and the speedup required for real-time identification. The “100 KP, 1K” data set represents the 1,196 member gallery set of NIST 100 kilopixel FERET images, a de facto standard. The “4 MP, 14K” data set represents the NIST MBE2010 data set and contains a 14,365 member gallery set of four megapixel images. The “4 MP, 1M” data set extends the NIST MBE2010 data set to a one million member gallery set of four megapixel images to simulate a large gallery set such as the Terrorist Watch List.

Table 8: Performance Estimates and Required Speedup.

Benchmark	**Image Resolution**	**Gallery Size**	**Execution Time Estimate (Minutes)**	**Required Speedup**
Eigenface 100 KP, 1K	100 kilopixel	1,196	0.0015	-
Eigenface 4 MP, 14K	4 megapixel	14,365	0.6800	-
Eigenface 4 MP, 1M	4 megapixel	1,000,000	75.7150	38
Bayesian 100 KP, 1K	100 kilopixel	1,196	0.00146	-
Bayesian 4 MP, 14K	4 megapixel	14,365	0.6791	-
Bayesian 4 MP, 1M	4 megapixel	1,000,000	71.169	36
EBGM 100 KP, 1K	100 kilopixel	1,196	3.70	2
EBGM 4 MP, 14K	4 megapixel	14,365	1,495.16	748
EBGM 4 MP, 1M	4 megapixel	1,000,000	1,496.16	748

Each algorithm has a primary bottleneck. For the Eigenface and Bayesian algorithms the process of projecting the probe into the vector subspace is the bottleneck, but the comparison process is a secondary bottleneck for some data sets. For the EBGM algorithm, the convolution in the filtering process of the feature search is the bottleneck as a result of the large number of iterations.

Table 8 shows that the Eigenface 100 KP, 1K, Eigenface 4 MP, 14K, and Bayesian 100 KP, 1K data sets and algorithms complete identification within the two minute real-time goal and are therefore real-time. However, the Eigenface 4 MP, 1M data benchmark requires a 38 times speedup for real-time identification, while the Bayesian 4 M, 1M benchmark requires a 36 times speedup to become real-time. As noted, the Eigenface and Bayesian benchmarks are similar and the computation required differs only in the comparison process within the benchmark.

The EBGM 100 KP, 1K benchmark requires 3.7 minutes for identification and a two times speedup to meet the time goal, and both the EBGM 4 MP, 14K and EBGM 4 MP, 1M data sets require 1,495.16 to 1,496.16 minutes for identification and a speedup of 748 to become real-time.

5.0 INSTRUCTION SET ARCHITECTURE ANALYSIS

The computational analysis method described in Chapter 4 estimated execution time based on the average CPI for assembly language instructions on the reference architecture. This analysis showed which code segments consume the largest percentage of the total execution time and therefore must be accelerated to enable real-time face identification.

The computational bottlenecks exposed by the Chapter 4 analysis will be investigated in this chapter. Given a computational bottleneck that prevents real-time performance, accelerating the computation may provide enough speedup to enable real-time performance. This speedup can potentially achieved by either increasing processor performance or by using multiple processors. This chapter will explore how to increase the performance of a single processor and Chapter 7 will explore how to improve performance using multiple processors.

The increased performance of one processor will be modeled by increasing the performance of the instructions the processor executes. The Instruction Set Architecture determines the average Cycles Per Instruction (CPI) or number of system clock cycles required to execute each instruction [21]. The execution time is therefore directly affected by the ISA, and improvements to the ISA can improve algorithm performance. If average CPI can be reduced for particular instructions, code segments that use those instructions can be accelerated. The analysis method developed in this chapter will provide a way to determine which instructions have greater impact on performance and therefore will contribute the most speedup if average CPI for those instructions can be reduced.

The analysis method developed in this chapter will refine the computational analysis method to guide selection of instructions for acceleration and will develop methods to reduce average CPI for these instructions. The computational analysis results will be summarized by instruction type to determine which instructions consume a larger percentage of the total execution time for the benchmark. These instructions will then be accelerated to reduce average CPI and the execution time will be estimated for a system incorporating the change to quantify the speedup achieved.

5.1 ISA ANALYSIS METHOD

The following process will be used to analyze the ISA to determine which are heavily used and the performance improvement if those instructions are accelerated:

Step 1: Analyze the benchmarks to locate heavily used instructions. In Chapter 4, benchmarks were developed and the execution time for the benchmarks was estimated. This execution time estimate exposed the code segments within the benchmarks that limit performance of the benchmarks and therefore are the bottlenecks for the particular benchmark. These code segments can be analyzed at the assembly instruction level to determine which particular instructions are used in the bottleneck code segments and the percentage of execution time consumed by each type of instruction.

The computational analysis method developed in Chapter 4 counted the number of times each instruction was executed and multiplied the execution count by the average CPI to estimate the number of cycles required for execution. These results were calculated in a spreadsheet and were reported in summary form to quantify the percentage of time required to execute each line of code. Although the Chapter 4 analysis was based on assembly language instructions, the details of those instructions were not exposed. For the ISA analysis, detailed assembly language instruction level information is needed and the computational analysis results are expanded to express percentage of execution time and the number of cycles consumed for each type of assembly instruction.

Step 2: Analyze instruction usage. Amdahl's law shows that accelerating instructions that consume more of the total execution time will provide a better payback for the investment in resources and chip area required to accelerate the instructions. Furthermore, accelerating instructions that consume less of the total execution time will provide minimal reduction in execution time and is therefore not an efficient use of resources.

The instruction level computational analysis results from Step 1 will be sorted in descending order by the total number of cycles required for all executions of each instruction in the benchmark. The total number of cycles will be divided by the system clock frequency to estimate execution time, and instructions with an execution time greater than 0.1 second will be selected for further analysis. For the instructions selected, the usage per line of code will be analyzed to determine which lines of code use those instructions more heavily.

Step 3: Accelerate heavily used instructions. The heavily used instructions exposed in Step 2 will then be analyzed to determine how these instructions can be accelerated. A data flow graph will be generated for each line of C code in the bottleneck processes to show the data and control flow among the assembly language instructions.

This analysis will graphically show sequences of instructions that operate on the same set of data. These instructions that operate on common data can potentially be accelerated by combining or *fusing* multiple instructions. For example, the process of multiplying a vector by a row of a matrix performs a sequence of multiplications and additions. Adding a single fused multiply-accumulate instruction that performs both the multiplication and the addition could improve performance. Instruction sequences exposed by the data flow graph analysis will be analyzed to determine if performance can be improved by fusing the instructions.

The latency for fused instructions will be several cycles. Instructions can be implemented in combinational logic. but the propagation delay of combinational logic is in the range of the system clock period. As a result, the propagation delay of the combinational logic gates limits performance to one level of logic or one gate delay per system clock cycle. However, since fused instructions combine simpler instructions they require multiple levels of combinational logic and therefore the latency for the fused instructions will be a minimum of two clock cycles.

The average CPI for fused instructions can be reduced to one cycle with pipelining techniques. Pipelining divides a multiple cycle instruction such as a fused instruction into a set of simpler, faster operations or stages that can complete within one clock cycle [21]. The pipeline stages are then executed in sequence to perform the fused instruction. The latency or time required to complete the first execution of the fused instruction still requires multiple cycles, but the throughput or time to complete successive repeated operations is reduced to one cycle and therefore average CPI can be reduced to one cycle, given enough repetitions of the instruction. As a result, average CPI can be reduced to a single cycle if enough pipeline stages are added and therefore, pipelining can be used to reduce the average CPI for a fused instruction to a single clock cycle.

If greater acceleration is required, the average CPI for the fused instruction can be reduced to less than one cycle by using multiple function units. Given a pipelined function unit that can complete a fused instruction within one clock cycle on average, adding a second function unit operating in parallel can reduce the average CPI for that operation to (1 CPI)/(2 function units) or 0.5 CPI. Adding more function units proportionally reduces average CPI, provided all function units can execute in parallel and the code allows the function units to be used in this way. For example, adding one pipelined multiply-accumulate instruction for a vector multiplication reduces average CPI to one cycle, but adding two multiply-accumulate instructions reduces average CPI to 0.5 cycles.

As fused instructions and multiple function units are added, the data pathway to the register file can become a bottleneck that limits performance. The processor register file is configured to source two input operands and store one result within one cycle in a common processor architectures such as the X86 [21]. If the added fused instruction requires three input operands rather than the two operands required by each of the original instructions, the number of register file ports must be increased to provide the additional input operand. In the same way, if a second function unit is added, the register file must then provide six operands rather than the original two. As a result, the resources required for the register file increase as function units are added and therefore the resource cost increases. For this analysis, the number of register file ports will be used as a metric to quantify the resource cost and evaluate the cost/performance tradeoff for increasing the number of function units.

The average CPI for the accelerated instructions is used to estimate the performance improvement. To estimate the execution time with the ISA changes, average CPI is calculated for each instruction. The execution counts for each instruction are multiplied by the new average CPI to calculate the execution time with the accelerated instructions, in the same way execution time was calculated for the original instructions and CPI. This calculation is repeated for the addition of one, two, four, and eight function units for the fused instructions to quantify the cost performance tradeoff for adding function units.

The average CPI for data movement instructions will also be adjusted to accommodate the increase in the number of register file ports. The registers in the register file are loaded from Local Memory with explicit move instructions. Increasing the number of register file ports will potentially cause the move instructions to become a bottleneck and limit performance. To avoid this bottleneck, the average CPI for the move instructions will be scaled proportionally to adjust for the increase in the number of register file ports to avoid this bottleneck. For example, if one register port is added for a total of three ports, the move instruction average CPI will be reduced by 1/3.

The CPI reduction for move instructions could be implemented by increasing the memory bandwidth. Memory bandwidth can be increased by widening the data bus. DRAM modules increase net memory bandwidth by combining several DRAM chips to form a module with a wider data bus [120]. Given a DRAM chip with a 3.6 GB/s data rate, combining eight chips on a module with a 64 bit parallel data bus provides net bandwidth of 28.8 GB/s. This principle can be applied to the processor architecture to widen the data bus between on-chip memory and the register file to increase net bandwidth. Increasing the bandwidth overcomes the memory to register file bottleneck and reduces the time required to transfer a constant sized set of data. In this way, the average CPI for move instructions can be reduced, and this memory bandwidth effect will be modeled in this analysis as a reduction in average CPI for the move instructions.

5.2 EIGENFACE BENCHMARK ISA ANALYSIS

The analysis in Chapter 4 showed that the Eigenface 100 KP, 1K and 4 MP, 14K benchmarks achieve real-time performance on sequential architectures and do not require additional acceleration. However, the Eigenface 4 MP, 1M benchmark requires a 38 times acceleration to achieve real-time performance.

The Chapter 4 analysis also showed that the Eigenface and Bayesian benchmarks are quite similar and that applying techniques that achieve real-time performance for the Eigenface benchmarks will also ensure real-time performance for the Bayesian benchmarks. Therefore, analysis of the Eigenface 4 MP, 1M benchmark is sufficient, and this section will analyze the Eigenface 4 MP, 1M benchmark.

The instruction level analysis for the Eigenface 4 MP, 1M benchmark is shown in Figure 20. This figure shows that five different instructions are needed to execute the bottleneck code in the benchmark, sorted by the number of cycles required for execution on the reference architecture. The number of cycles required for all five instructions is within an order of magnitude and all five instructions will therefore be analyzed.

	4 MP, 1M					
	N=4E6, G=1E6, M'=6E5					
	movsd	mulsd	addsd	subsd	movapd	**Totals**
Eigenface Benchmark						
1 for each Eigenvector index k in M'						
2 for each probe element index j in N						
3 p_curl[k] = p_curl[k] + U[j][k] * p_bar[j]	9.29E+12	2.32E+12	2.32E+12			1.39E+13
4 for each gallery member index g in G						
5 for each coefficient index k in M'						
6 d_sum=d_sum + (Z[k][g] - p_curl[k])^2 * (const[k])	4.2E+12	1.2E+12	6E+11	1.2E+12	1.2E+12	8.40E+12
Number of executions:	1.35E+13	3.52E+12	2.92E+12	1.20E+12	1.20E+12	2.23E+13
AMD 10H CPI:	0.5	1.0	1.0	1.0	0.5	
Total Cycles	6.75E+12	3.52E+12	2.92E+12	1.20E+12	6.00E+11	1.5E+13
Execution Time (minutes):						75.72

Figure 20: Eigenface benchmark instruction level analysis.

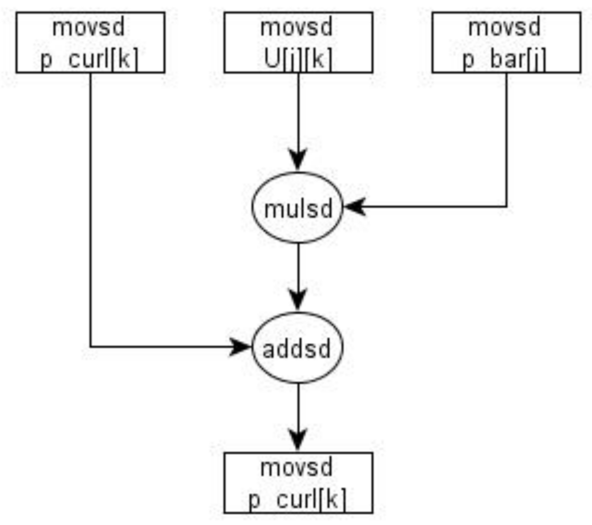


Figure 21: Eigenface benchmark line 3 data flow graph.

The data flow graph for the probe projection (line 3 in Figure 20) is shown in Figure 21. This process loads U[j][k] and p_bar[j] from local memory into the register file with movsd instructions, multiplies the two registers with a mulsd instruction, loads p_curl[k] with a movsd instruction and adds it to the result of the mulsd with an addsd instruction, and then stores the result back to local memory with a movsd instruction. Figure 21 shows that three data values are loaded, operated upon, and then a single result is returned.

A pipelined fused multiply and add instruction could be added to accelerate the probe projection as shown in Figure 22 with mnemonic MADD. This fused instruction would transfer three register values from the register file, perform the multiply and add operation, and return one result to the register file. Since the standard register file can transfer only two input registers and store one result in an instruction cycle, the register file would need to be modified to add one port for a total of three ports. If the register file change can be implemented, the fused multiply and add instruction could provide a two times speedup for the computation in this operation.

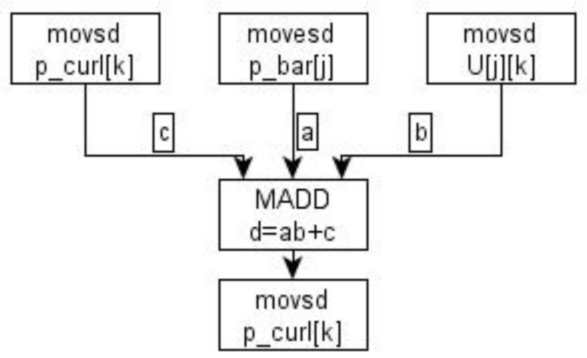


Figure 22: Eigenface benchmark fused multiply-add MADD instruction for line 3.

The data flow graph for the comparison calculation (line 6 in Figure 20) is shown in Figure 23. This process loads Z[k][g] and p_curl[k] from local memory with movsd instructions, subtracts the registers with a subsd instruction, squares the result with a mulsd instruction, multiplies that result by the constant with another mulsd instruction, loads d_sum with a movsd instruction and accumulates the result with a addsd instruction, then stores the result with a movapd instruction in aligned packed double precision format.

A new difference and square fused instruction DFSQ can be added to calculate the squared difference of Z[k][g] and p_curl[k] as shown in Figure 24. The MADD instruction can then be used to multiply by the constant and the DFSQ result.

The instruction analysis is updated to show the speedup achieved with the fused instructions in Figure 25. In this table the instruction columns have been rearranged to group instructions by function, including data movement and computation instructions. This analysis shows a speedup of 1.38 times is obtained with the fused instructions.

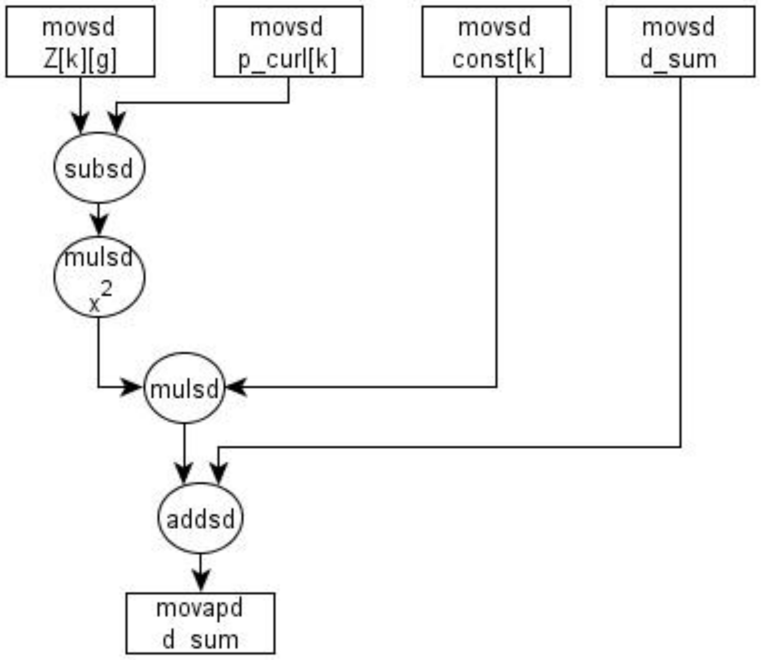


Figure 23: Eigenface benchmark line 6 data flow graph.

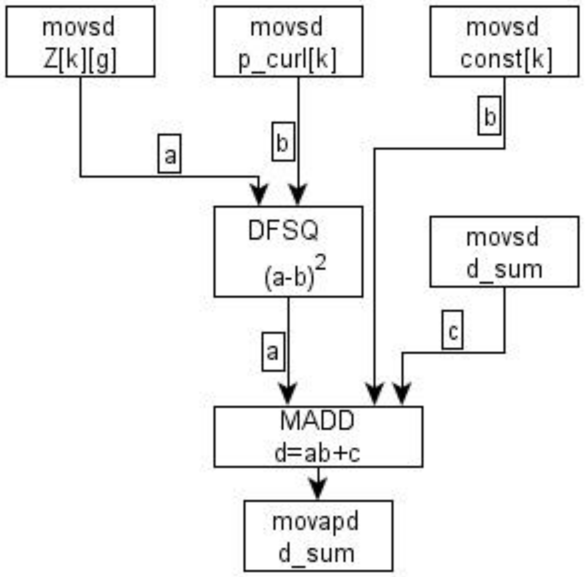


Figure 24: Eigenface benchmark fused diff-squared instruction DFSQ for line 6.

	4 MP, 1M							
	N=4E6, G=1E6, M'=6E5							
	Data Movement		**Computation**					
	movsd	movapd	mulsd	addsd	subsd	MADD	DFSQ	**Totals**
Eigenface Benchmark								
1 for each Eigenvector index k in M'								
2 for each probe element index j in N								
3 p_curl[k] = p_curl[k] + U[j][k] * p_bar[j]	9.29E+12					2.32E+12		1.16E+13
4 for each gallery member index g in G								
5 for each coefficient index k in M'								
6 d_sum=d_sum + (Z[k][g] - p_curl[k])^2 * (const[k])	4.20E+12	1.20E+12				6.00E+11	6.00E+11	6.60E+12
Number of executions:	1.35E+13	1.20E+12				2.92E+12	6.00E+11	1.82E+13
Total Cycles:	6.75E+12	6.00E+11				2.92E+12	6.00E+11	1.09E+13
							Execution Time (minutes):	54.89
							ISA Speedup:	1.38

Figure 25: Eigenface ISA speedup with fused instructions.

The modest 1.38 times speedup achieved with the fused instructions as shown in Figure 25 reflects the local memory bottleneck caused by the move instructions. The move instructions require 2.09 times the number of cycles required for the computation and therefore limit performance. To overcome this bottleneck, additional local memory channels could be added. Adding three channels to the data path between local memory and the register file would allow the three input move instructions to be executed in 0.5 cycles, the number of cycles required for one move in the original ISA. Adding this local memory bandwidth increase results in a total speedup of 2.51 times for the 4 MP, 1M data set. This speedup reduces benchmark execution time to 30.16 minutes but real-time performance is not achieved.

Figure 26 shows the speedup achieved with multiple function units and Table 9 shows the number of register file ports required to achieve that speedup. The graph in Figure 27 shows the relationship between the number of register ports and the speedup achieve. Adding two function units for both fused instructions increases speedup to 5.02 but requires six register ports. Execution time is still not real-time at 15.08 minutes. Four function units increases speedup to 10.25 and requires 12 register ports an execution time drops to 7.39 minutes, still not real-time. Eight function units increases speedup to 20.08 and achieves execution time of 3.77 minutes, but real-time performance is still not achieved.

	4 MP, 1M							
	N=4E6, G=1E6, M'=6E5							
	Data Movement		**Computation**					
	movsd	movapd	mulsd	addsd	subsd	MADD	DFSQ	**Totals**
Eigenface Benchmark								
1 for each Eigenvector index k in M'								
2 for each probe element index j in N								
3 p_curl[k] = p_curl[k] + U[j][k] * p_bar[j]	9.29E+12					2.32E+12		1.16E+13
4 for each gallery member index g in G								
5 for each coefficient index k in M'								
6 d_sum=d_sum + $(Z[k][g] - p\_curl[k])^2$ * (const[k])	4.20E+12	1.20E+12				6.00E+11	6.00E+11	6.60E+12
Number of executions:	1.35E+13	1.20E+12				2.92E+12	6.00E+11	1.82E+13
New CPI, one function unit:	0.17	0.17		1.00		1.00	1.00	
Total Cycles:	2.25E+12	2.00E+11				2.92E+12	6.00E+11	5.97E+12
	Execution Time with ISA Speedup (minutes)							30.16
	ISA Speedup:							2.51
New CPI, two function units:	0.08	0.08		1.00		0.50	0.50	
Total Cycles:	1.12E+12	1.00E+11				1.46E+12	3.00E+11	2.99E+12
	Execution Time with ISA Speedup (minutes)							15.08
	ISA Speedup:							5.02
New CPI, four function units:	0.04	0.04		1.00		0.25	0.20	
Total Cycles:	5.62E+11	5.00E+10				7.31E+11	1.20E+11	1.46E+12
	Execution Time with ISA Speedup (minutes)							7.39
	ISA Speedup:							10.25
New CPI, eight function units:	0.02	0.02		1.00		0.13	0.13	
Total Cycles:	2.81E+11	2.50E+10				3.65E+11	7.50E+10	7.46E+11
	Execution Time with ISA Speedup (minutes)							3.77
	ISA Speedup:							20.08

Figure 26: Eigenface benchmark speedup with multiple function units.

Table 9: Eigenface and Bayesian Speedup Summary.

Benchmark	FUSED INSTRUCTIONS	NUMBER OF FUNCTION UNITS	NUMBER OF REGISTER FILE PORTS	SPEEDUP	IDENTIFICATION TIME
Eigenface 4 MP, 1M	2	1	3	2.51	30.16
Eigenface 4 MP, 1M	2	2	6	5.02	15.08
Eigenface 4 MP, 1M	2	4	12	10.25	7.39
Eigenface 4 MP, 1M	2	8	24	20.08	3.77
Bayesian 4 MP, 1M	2	1	3	2.21	32.18
Bayesian 4 MP, 1M	2	2	6	4.04	17.60
Bayesian 4 MP, 1M	2	4	12	6.90	10.32
Bayesian 4 MP, 1M	2	8	24	10.66	6.67

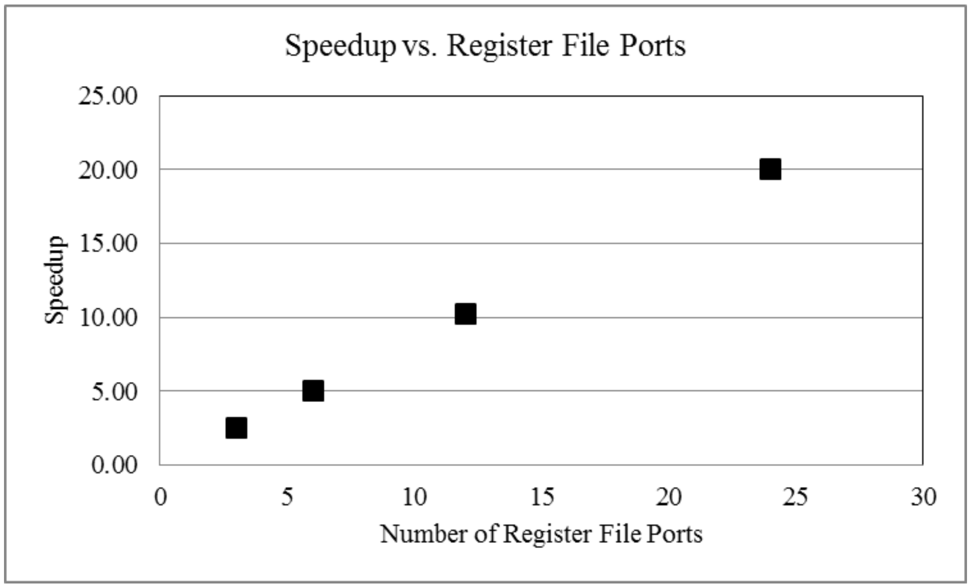


Figure 27: Eigenface ISA speedup and register file ports.

5.3 EBGM BENCHMARK ISA ANALYSIS

The computational analysis in Chapter 4 showed that the EBGM benchmark requires acceleration for all three data sets. The EBGM 100K, 1K benchmark requires a two times speedup, but the EBGM 4 MP, 14K and 4 MP, 1M benchmarks require a 748 times speedup. Given the 0.06% difference in execution time and the common 748 speedup required, the two EBGM benchmarks with 4 MP images will not need to be analyzed separately. Therefore, in this section the EBGM 100K, 1K and EBGM 4 MP, 1M benchmarks will be analyzed.

The instruction level analysis for the EBGM 100 KP, 1K benchmark is shown in Figure 28. Thirty-one different instructions are used for this benchmark and the columns in the table in Figure 28 are sorted by the number of cycles required for execution on the reference architecture.

The integer multiply instruction IMUL consumes the largest number of cycles, and only the nine instructions shown in the table consume within 100 times the cycles required by IMUL. As a result, only the nine instructions shown in the table will impact performance by 1% or more and can provide a sufficient cost-performance tradeoff to warrant analysis. Furthermore, only the primary bottleneck process, lines 4.4 and 4.5, consume enough total cycles to warrant analysis.

The data flow graph for the real filter multiplication in line 4.4 is shown in Figure 29. While line 4.5 multiplies the imaginary mask, the instructions used are the same and thus do not need to be analyzed separately. The data flow graph shows that calculating the memory address for the matrix requires multiple instructions and consumes a significant number of cycles. The row and column indices r and c are loaded from local memory into registers edx and eax with mov instructions. The cltq instruction sign extends eax to rax, and movslq instruction moves and sign extends edx to rdx. The imul instruction calculates the offset and the lea instruction calculates the memory address. A movsd instruction then moves the FaceImage[r][c] data

	100 KP, 1K								
	N=1E5, G=1E3,M'=6E5								
	imul	movsd	mov	mulsd	addsd	movslq	cltq	lea	**Totals**
EBGM Benchmark									
1 for each feature vertex v, v=1 to 25									
2 for each jet b in the bunch, b=1 to M									
3 s=CalcMagnitudeSimilarity(j, b)		2.5E+07	1.7E+07	8.4E+06	6.0E+06		9.6E+06	4.8E+06	1.3E+08
4 for each half-point (dx, dy) in an R by R region									
4.1 for each filter k in the set of wavelet filters									
4.2 for each row r in the image region centered at (int[x],int[y]), r=1 to W									
4.3 for each column c in the image region centered at (int[x],int[y]), c=1 to W									
4.4 real_part + = FaceImage[r][c]* RealMask[k][r][c]	5.8E+10	1.2E+11	1.2E+11	2.9E+10	2.9E+10	5.8E+10	5.8E+10	5.8E+10	5.3E+11
4.5 imag_part + = FaceImage[r][c]* ImagMask[k][r][c]	5.8E+10	1.2E+11	1.2E+11	2.9E+10	2.9E+10	5.8E+10	5.8E+10	5.8E+10	5.3E+11
4.6 real_part = sqrt[real_part^2 + imag_part^2] * cos(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	0	8.7E+07	3.2E+07	5.4E+07	3.2E+07	0	0	0	5.6E+08
4.7 imag_part = sqrt[real_part^2 + imag_part^2] * sin(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	0	8.7E+07	3.2E+07	5.4E+07	3.2E+07	0	0	0	5.6E+08
5 s=CalcPhaseSimilarity(j, b)	0	3.1E+08	2.6E+08	9.8E+07	6.5E+07	0	1.3E+08	6.5E+07	1.9E+09
Number of executions:	1.2E+11	2.3E+11	2.3E+11	5.9E+10	5.9E+10	1.2E+11	1.2E+11	1.2E+11	1.1E+12
AMD 10H CPI:	2.0	0.5	0.5	1.0	1.0	0.5	0.3	0.3	
Total Cycles	2.3E+11	1.2E+11	1.2E+11	5.9E+10	5.9E+10	5.8E+10	3.9E+10	3.9E+10	7.3E+11
								ISA Speedup:	2.88

Figure 28: EBGM benchmark instruction level analysis.

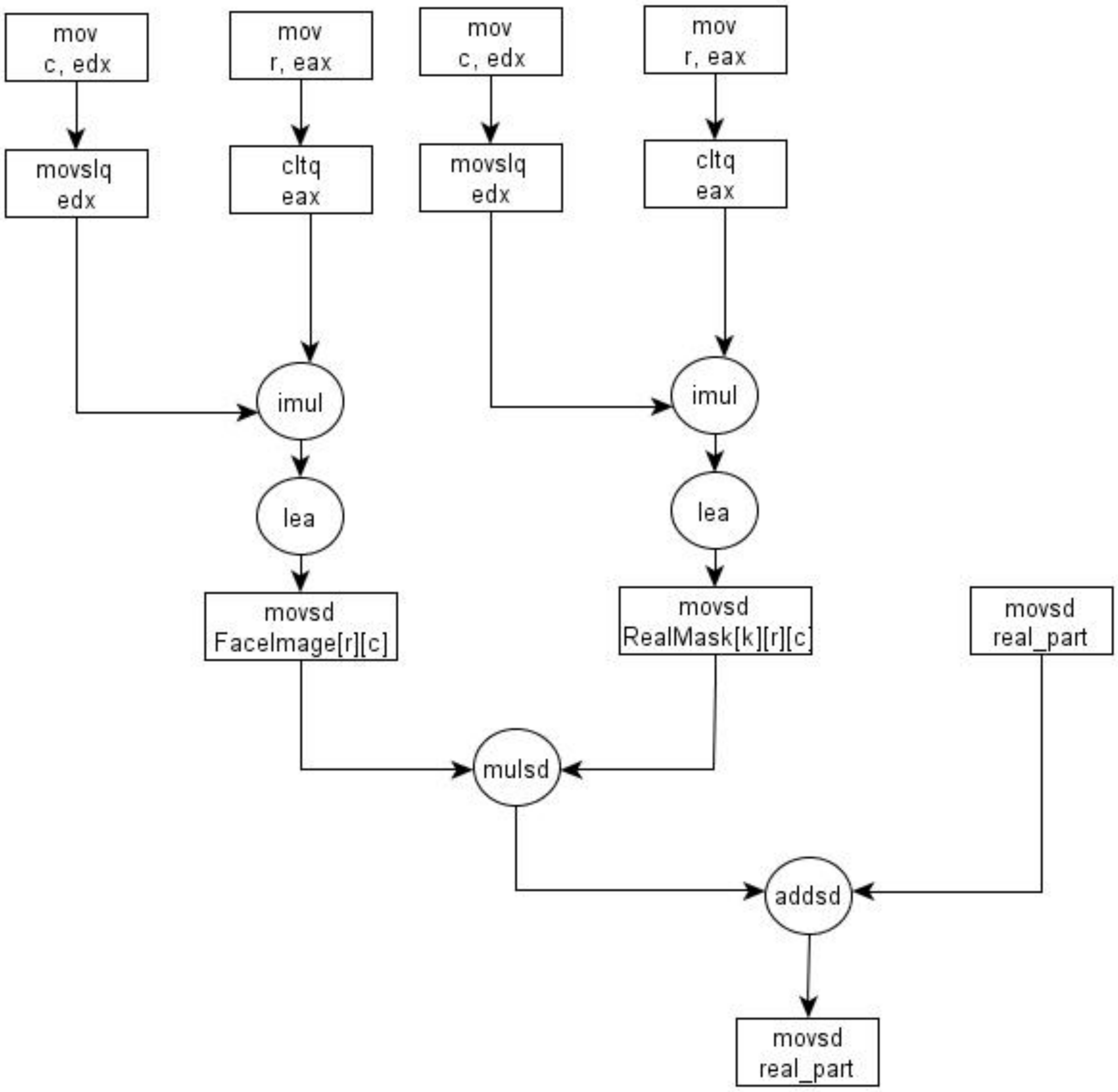


Figure 29: EBGM benchmark line 4.4 data flow graph.

element from the calculated address to the register. This process is repeated to calculate the index for the RealMask and to load the mask data element. A mulsd instruction multiplies the image pixel and mask coefficient, and an addsd instruction accumulates the real_part result. Finally, a movsd instruction transfers the data back to local memory.

A fused instruction to calculate the index and move the data to the register file could be added to accelerate the filter process as shown in Figure 30. This fused MOVIDX could calculate the index and load the data within one cycle. This calculation could be implemented with combinational logic to complete within in 0.5 cycle, and the 0.5 cycle time for other mov instructions suggests that the move could be completed within 0.5 cycle, reducing the total to one cycle for the fused instruction.

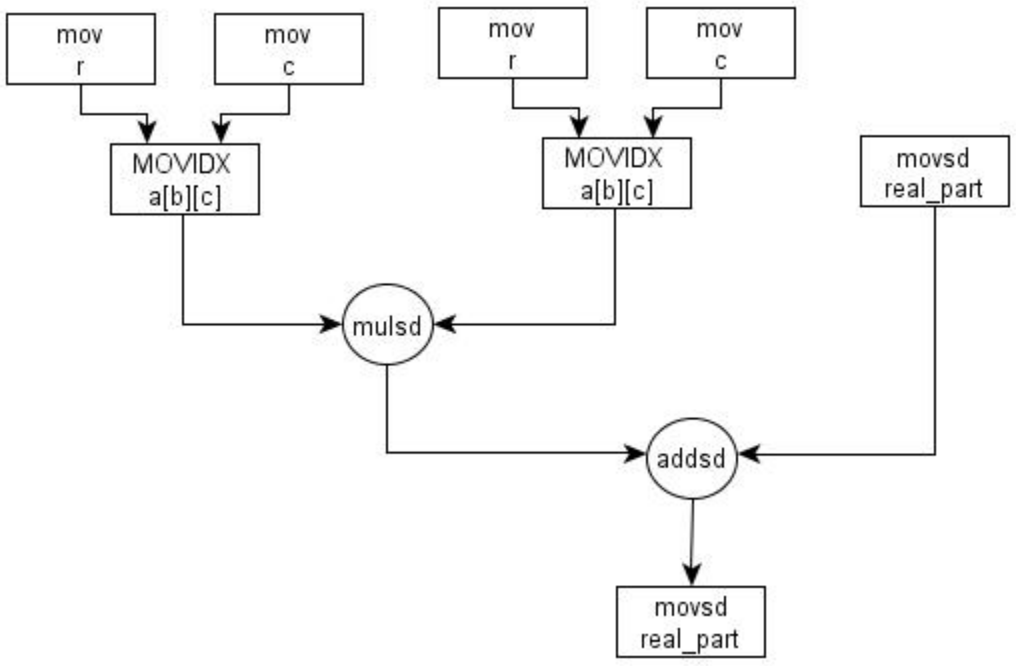


Figure 30: EBGM benchmark fused move-index MOVIDX for line 4.4.

The instruction analysis is updated to show the speedup achieved with the fused instructions in Figure 31 for the EBGM 100 KP, 1K benchmark. The graph in Figure 32shows the relationship between the number of register ports required and the speedup. Figure 31 shows the MOVIDX instruction in the next to the last column and uses this instruction for lines 4.4 and 4.5. This analysis shows a speedup of 1.70 is achieved for the 100K, 1K benchmark, reducing execution time to a near real-time 2.18 minutes. Adding two function units increases speedup to 3.24 and achieves real-time performance of 1.14 minutes, but requires four register file ports as shown Figure 31.

Figure 33 shows the analysis for the 4 MP, 1M benchmark and the graph in Figure 34 shows the resource cost for the speedup achieved. The 4 M, 1M benchmark speedup is 2.46 with one function unit but performance is not real-time at 607.80 minutes. Figure 33 shows that identification time is 77.61 minutes with an unrealistic eight function units, but even the addition of eight function units does not enable real-time performance.

Table 10 summarizes the EBGM speedup results and shows that the ISA speedup enables real-time performance only for the EBGM 100 KP, 1K benchmark.

	100 KP, 1K									
	N=1E5, G=1E3, M'=6E2									
	imul	movsd	mov	mulsd	addsd	movslq	cltq	lea	MOVIDX	**Totals**
EBGM Benchmark										
1 for each feature vertex v, v=1 to 25										
2 for each jet b in the bunch, b=1 to M										
3 s=CalcMagnitudeSimilarity(j, b)		2.5E+07	1.7E+07	8.4E+06	6.0E+06		9.6E+06	4.8E+06		1.3E+08
4 for each half-point (dx, dy) in an R by R region										
4.1 for each filter k in the set of wavelet filters										
4.2 for each row r in the image region centered										
4.3 for each column c in the image region centered at (int[x],int[y]), c=1 to W										
4.4 real_part + FaceImage[r][c]* RealMask[k][r][c]		5.8E+10	1.2E+11	2.9E+10	2.9E+10				5.8E+10	2.9E+11
4.5 imag_part + = FaceImage[r][c]* ImagMask[k][r][c]		5.8E+10	1.2E+11	2.9E+10	2.9E+10				5.8E+10	2.9E+11
4.6 real_part = sqrt[real_part^2 + imag_part^2] * cos(atan(imag_part / real_part) +		8.7E+07	3.2E+07	5.4E+07	3.2E+07					5.6E+08
4.7 imag_part = sqrt[real_part^2 + imag_part^2] * sin(atan(imag_part / real_part) +		8.7E+07	3.2E+07	5.4E+07	3.2E+07					5.6E+08
5 s=CalcPhaseSimilarity(j, b)		3.1E+08	2.6E+08	9.8E+07	6.5E+07	0	1.3E+08	6.5E+07		1.9E+09
Number of executions:	0	1.2E+11	2.3E+11	5.9E+10	5.9E+10	0	1.4E+08	7.0E+07	1.2E+11	5.9E+11
New CPI, one function unit:	2.0	0.5	0.5	1.0	1.0	0.5	0.3	0.3	1.0	
Total Cycles	0	5.9E+10	1.2E+11	5.9E+10	5.9E+10	0	4.6E+07	2.3E+07	1.2E+11	4.3E+11
									Execution Time (minutes):	2.18
									ISA Speedup:	1.70
New CPI, two function units:	2.00	0.25	0.25	0.50	0.50	0.50	0.33	0.33	0.50	
Total Cycles	0	2.9E+10	5.9E+10	2.9E+10	2.9E+10	0	4.6E+07	2.3E+07	5.8E+10	2.3E+11
									Execution Time (minutes):	1.14
									ISA Speedup:	3.24
New CPI, four function units:	2.00	0.13	0.13	0.25	0.25	0.50	0.33	0.33	0.25	
Total Cycles	0	1.5E+10	2.9E+10	1.5E+10	1.5E+10	0	4.6E+07	2.3E+07	2.9E+10	1.2E+11
									Execution Time (minutes):	0.62
									ISA Speedup:	5.94
New CPI, eight function units:	2.00	0.06	0.06	0.13	0.13	0.50	0.33	0.33	0.13	
Total Cycles	0	7.3E+09	1.5E+10	7.3E+09	7.3E+09	0	4.6E+07	2.3E+07	1.5E+10	7.2E+10
									Execution Time (minutes):	0.36
									ISA Speedup:	10.18

Figure 31: EBGM 100K, 1K benchmark ISA speedup with fused instruction.

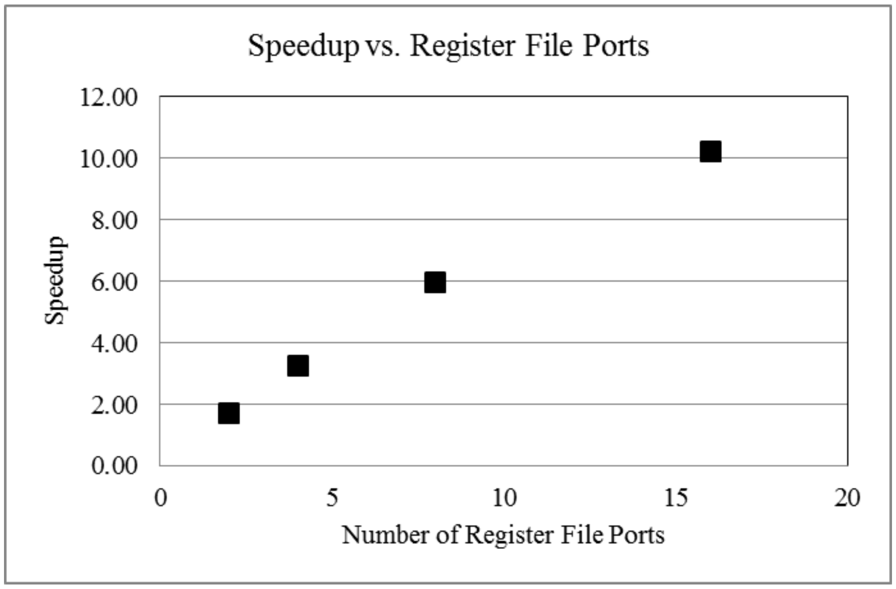


Figure 32: EBGM 100 KP, 1K speedup and register file ports.

	imul	movsd	mov	mulsd	addsd	movslq	cltq	lea	MOVIDX	Totals
	4 MP 1M									
	N=4E6, G=1E6, M'=6E5									
EBGM Benchmark										
1 for each feature vertex v, v=1 to 25										
2 for each jet b in the bunch, b=1 to M										
3 s=CalcMagnitudeSimilarity(j, b)	0	2.1E+10	1.4E+10	7.0E+09	5.0E+09	0	8.0E+09	4.0E+09		1.1E+11
4 for each half-point (dx, dy) in an R by R region										
4.1 for each filter k in the set of wavelet filters										
4.2 for each row r in the image region centered at (int[x],int[y]), r=1 to W										
4.3 for each column c in the image region centered at (int[x],int[y]), c=1 to W										
4.4 real_part + FaceImage[r][c]* RealMask[k][r][c]		2.4E+13		1.2E+13	1.2E+13				2.4E+13	7.2E+13
4.5 imag_part + = FaceImage[r][c]* ImagMask[k][r][c]		2.4E+13		1.2E+13	1.2E+13				2.4E+13	7.2E+13
4.6 real_part = sqrt[real_part^2 + imag_part^2] * cos(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	0	1.8E+09	6.6E+08	1.1E+09	6.6E+08	0	0	0		1.1E+10
4.7 imag_part = sqrt[real_part^2 + imag_part^2] * sin(atan(imag_part / real_part) + fx[k] * dx+fy[k] * dy)	0	1.8E+09	6.6E+08	1.1E+09	6.6E+08	0	0	0		1.1E+10
5 s=CalcPhaseSimilarity(j, b)	0	6.4E+09	5.3E+09	2.0E+09	1.3E+09	0	2.6E+09	1.3E+09		3.8E+10
Number of executions:	0	4.8E+13	2.1E+10	2.4E+13	2.4E+13	0	1.1E+10	5.3E+09	4.8E+13	1.4E+14
New CPI, one function unit:	2.0	0.5	0.5	1.0	1.0	0.5	0.3	0.3	1.0	
Total Cycles	0	2.4E+13	1.0E+10	2.4E+13	2.4E+13	0	3.5E+09	1.8E+09	4.8E+13	1.2E+14
							Execution Time (minutes):			607.80
							ISA Speedup:			2.46
New CPI, two function units:	2.00	0.25	0.25	0.50	0.50	0.50	0.33	0.33	0.50	
Total Cycles	0	1.2E+13	5.2E+09	1.2E+13	1.2E+13	0	3.5E+09	1.8E+09	2.4E+13	6.0E+13
							Execution Time (minutes):			304.83
							ISA Speedup:			4.91
New CPI:, four function units	2.00	0.13	0.13	0.25	0.25	0.50	0.33	0.33	0.25	
Total Cycles	0	6.0E+12	2.6E+09	6.0E+12	6.0E+12	0	3.5E+09	1.8E+09	1.2E+13	3.0E+13
							Execution Time (minutes):			153.35
							ISA Speedup:			9.76
New CPI, eight function units:	2.00	0.06	0.06	0.13	0.13	0.50	0.33	0.33	0.13	
Total Cycles	0	3.0E+12	1.3E+09	3.0E+12	3.0E+12	0	3.5E+09	1.8E+09	6.0E+12	1.5E+13
							Execution Time (minutes):			77.61
							ISA Speedup:			19.28

Figure 33: EBGM 4 MP, 1M benchmark ISA speedup with fused instruction.

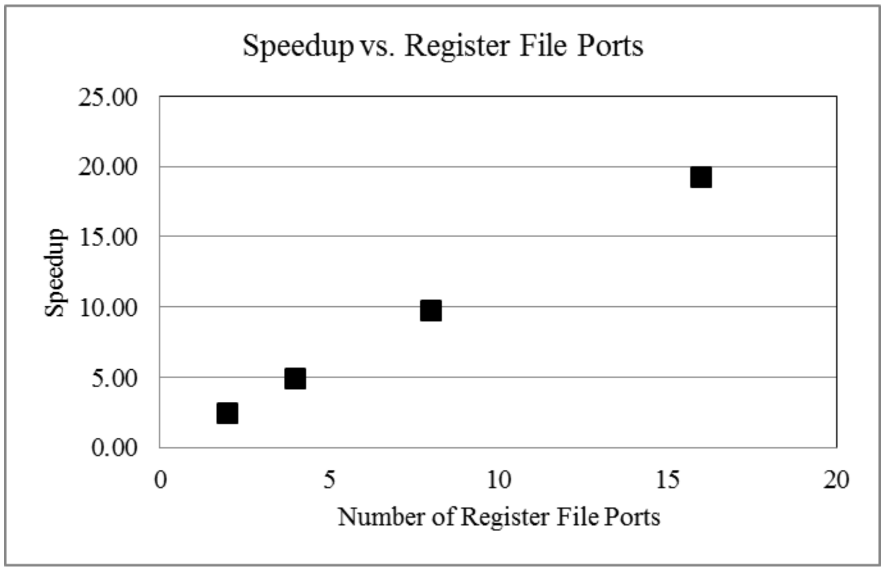


Figure 34: EBGM 4 MP, 1M speedup and register file ports.

Table 10: EBGM ISA Speedup Summary.

BENCHMARK	FUSED INSTRUCTIONS	NUMBER OF FUNCTION UNITS	NUMBER OF REGISTER FILE PORTS	SPEEDUP	IDENTIFICATION TIME
EBGM 100 KP, 1K	1	1	2	1.70	2.18
EBGM 100 KP, 1K	1	2	4	3.24	1.14
EBGM 100 KP, 1K	1	4	8	5.94	0.62
EBGM 100 KP, 1K	1	8	16	10.18	0.36
EBGM 4 MP, 14K	1	1	2	2.46	606.80
EBGM 4 MP, 14K	1	2	4	4.92	303.91
EBGM 4 MP, 14K	1	4	8	9.81	152.47
EBGM 4 MP, 14K	1	8	16	19.48	76.74
EBGM 4 MP, 1M	1	1	2	2.46	607.80
EBGM 4 MP, 1M	1	2	4	4.91	304.83
EBGM 4 MP, 1M	1	4	8	9.76	153.35
EBGM 4 MP, 1M	1	8	16	19.28	77.61

5.4 CONCLUSION

The ISA analysis results in this chapter are summarized in Table 11. This analysis shows that while a 2.51 times ISA speedup can be achieved for the Eigenface and Bayesian 4 MP, 1M benchmarks, it is insufficient to make them real-time. Even the addition of eight function units and the resulting 20 times speedup is insufficient to enable real-time performance, even if an unrealistic 24 register file ports could be implemented. For further analysis in the later chapters, a single function unit and a 2.51 ISA speedup will be used.

The ISA analysis further shows that real-time performance can be achieved for the EBGM 100K, 1K data set by adding two function units to implement a fused index instruction. Adding two function units to the EBGM benchmark allows the two indexes to be calculated in parallel. Both index results are needed for the next operation and adding two function units will therefore provide a good performance tradeoff.

For the EBGM benchmark with the 4 MP, 14K and 4 MP, 1M benchmarks, the two function units provide a 4.9 speedup as shown in Table 11. However, this speedup is insufficient to achieve real-time performance. Therefore, for further analysis in the later chapters, two function units and a 4.9 times ISA speedup will be used.

Table 11: ISA Speedup.

Benchmark	Fused Instructions	Number of Function Units	Number of Register File Ports	Speedup	Identification Time
Eigenface 4 MP, 1M	2	1	3	2.51	30.16
Eigenface 4 MP, 1M	2	2	6	5.02	15.08
Eigenface 4 MP, 1M	2	4	12	10.25	7.39
Eigenface 4 MP, 1M	2	8	24	20.08	3.77
Bayesian 4 MP, 1M	2	1	3	2.21	32.18
Bayesian 4 MP, 1M	2	2	6	4.04	17.60
Bayesian 4 MP, 1M	2	4	12	6.90	10.32
Bayesian 4 MP, 1M	2	8	24	10.66	6.67
EBGM 100 KP, 1K	1	1	2	1.70	2.18
EBGM 100 KP, 1K	1	2	4	3.24	1.14
EBGM 100 KP, 1K	1	4	8	5.94	0.62
EBGM 100 KP, 1K	1	8	16	10.18	0.36
EBGM 4 MP, 14K	1	1	2	2.46	606.80
EBGM 4 MP, 14K	1	2	4	4.92	303.91
EBGM 4 MP, 14K	1	4	8	9.81	152.47
EBGM 4 MP, 14K	1	8	16	19.48	76.74
EBGM 4 MP, 1M	1	1	2	2.46	607.80
EBGM 4 MP, 1M	1	2	4	4.91	304.83
EBGM 4 MP, 1M	1	4	8	9.76	153.35
EBGM 4 MP, 1M	1	8	16	19.28	77.61

6.0 MEMORY ANALYSIS

The analysis methods developed in Chapters 4 and 5 analyzed computational performance limits without regard for limitations of external memory. Simplifying assumptions were made for the computational analysis method to enable determination of computational constraints. Data movement instructions such as the "mov" instruction were assumed to reference on-chip memory. The access time for on-chip memory was assumed to be less than the experimentally determined average CPI for the data movement instruction, allowing the total time for an instruction to be estimated with the average CPI. The on-chip memory capacity was assumed to be sufficient to store the required data. Finally, off-chip memory transfer time was assumed to be less than total computation time, so that the sum of the average CPIs for the instructions represented the net time required for the operations and the external memory transfers were masked by the computation time. These assumptions provided a means to analyze the computational performance limitations on the face identification algorithms.

The time required for transferring data between external memory and the SOC can limit performance. Data must be transferred from off-chip to on-chip memory before it can be accessed by the processor, and the bandwidth for off-chip memory is less than the bandwidth of on-chip memory. If the bandwidth of off-chip memory is insufficient to meet processor demands, the processor may stall while waiting for data, resulting in reduced performance. As a result, the external memory bandwidth may limit SOC performance.

6.1 EXTERNAL OR OFF-CHIP MEMORY BANDWIDTH ANALYSIS

An analysis method will be developed in this chapter to explore whether performance of a perfectly mapped program is constrained by external memory bandwidth. This analysis will estimate the total time required to transfer data between external memory and the SOC to determine if the transfer time prevents real-time identification. This analysis will provide a means to expose the performance limitations of external memory bandwidth as well as evaluate whether a particular mapping is not feasible due to memory constraints.

The time required for transferring data between external and on-chip memory is a function of the external memory bandwidth. Future memory bandwidth specifications must therefore be determined to support this analysis. Bandwidth specifications for the 2019 and 2022 technology points will be developed from JEDEC specifications and projections. JEDEC memory bandwidth projections for DDR4L DRAM [121] show 2,932 Mbps transfer rates in 2019 and project an average increase per year of 200 Mbps. The JEDEC data rates are transfer rates, the bandwidth for a transfer from a single RAM chip with an eight bit wide data path. DRAM modules combine eight chips to form a 64 bit word, so the data rate in bytes for the word transfer is eight times the data rate per module. Multiplying the 2,932 Mbps module data rate by eight and scaling to GB/s shows the 2019 bandwidth based on JEDEC projections is 23.5 GB/s.

Projecting into the future from this point with the 200 Mbps growth rate estimates 2022 bandwidth as 3,600 Mbps per chip or 28.8 GB/s per module.

The JEDEC projected transfer rates are maximum data rates for continuous streaming of data from consecutive memory addresses at the maximum burst rate. DRAM chips support a burst mode where eight bytes at consecutive addresses are read from the chip. This burst mode infers addressing rather than requiring each address to be set on the DRAM chip and therefore the time required to transfer multiple address words is eliminated [25]. In addition, accessing sequential data in the DRAM requires less time than a random access as a result of the structure of the DRAM [25], further increasing performance.

Face identification algorithms consist of large blocks of vector and matrix data, which the C language stores in consecutive memory locations. Face identification algorithms therefore require transfer of large blocks of data from consecutive memory locations and meet the requirements for sustained maximum length burst mode transfers.

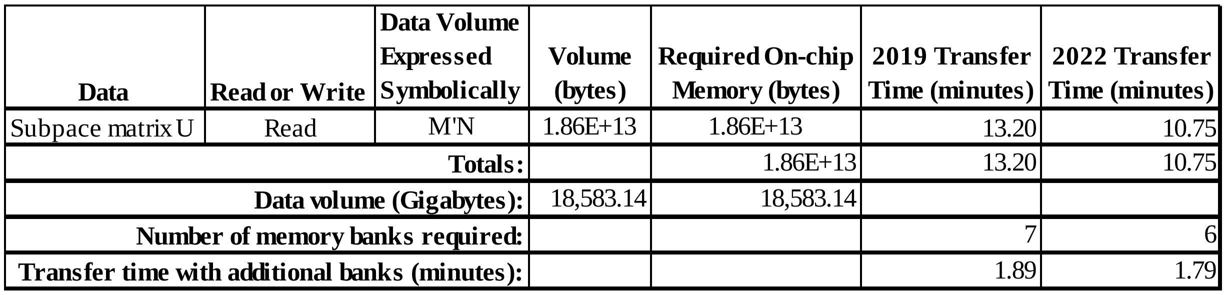
The memory analysis will analyze the sequential benchmarks to expose data movement bottlenecks. Each face identification benchmark and data set requires a particular set of data containing the probe image, gallery set, and information needed for the encoding process. The size of these data sets is independent of the on-chip implementation of the algorithm and will be the same for both sequential and parallel implementations of an algorithm, so performing the analysis on the sequential implementation will expose memory performance constraints that will impact both sequential and parallel implementations.

A single memory channel is provided to move data from external memory to on-chip memory and this analysis will initially estimate performance for that one memory channel. Once the data is on-chip, it must be transferred to the parallel processors, and this aspect of performance is a function of the mapping and will be analyzed in Chapter 7, while this chapter will analyze only the time required to move data across the single memory channel and onto the SOC chip.

6.2 MEMORY BANDWIDTH ANALYSIS METHOD

The impact of external memory bandwidth on execution time will be analyzed with the following process:

Step 1: Determine the required volume of data. The volume of data that must be transferred to and from external memory is calculated based on the benchmark code. Each data element is counted one time when first loaded from external memory and subsequent references to the same data are assumed to be accessed from on-chip memory. For example, Figure 35 shows the analysis for transferring the Eigenface subspace matrix U from off-chip memory. The data transferred is listed under the "Data" heading and the "Read or Write" column indicates data direction. The symbolic data size is listed in the next column and is shown as M'N for this example, indicating that the M' Eigenface vectors each have N elements.



Data	Read or Write	Data Volume Expressed Symbolically	Volume (bytes)	Required On-chip Memory (bytes)	2019 Transfer Time (minutes)	2022 Transfer Time (minutes)
Subpace matrix U	Read	M'N	1.86E+13	1.86E+13	13.20	10.75
		Totals:		1.86E+13	13.20	10.75
		Data volume (Gigabytes):	18,583.14	18,583.14		
		Number of memory banks required:			7	6
		Transfer time with additional banks (minutes):			1.89	1.79

Figure 35: Memory bandwidth analysis for the Eigenface subspace matrix.

The volume of data transferred is calculated by multiplying the number of data words transferred by the word size. In this example, M'=600,000, N=3,871,488, and 64 bit words are used, so the volume of data is 600,000*3,871,488*8 or 1.86 x 10^{13} bytes.

Step 2: Estimate the transfer time. The time required to transfer the data is calculated by dividing the volume of data transferred by the memory bandwidth as shown in Equation (6.1).

$$T_{MemoryTransfer} = \frac{NumberOfBytesTransferred}{MemoryBandwidth} \tag{6.1}$$

The subspace matrix example shown in Figure 35 requires 1.86 x 10^{13}/23.5 x 10^{9} seconds, which totals 791.5 seconds or 13.20, minutes to transfer the data with the 2019 bandwidth.

Step 3: Calculate required number of memory banks. If the total transfer time exceeds the real-time goal of two minutes, memory bandwidth must be increased to enable real-time identification. The increase in memory bandwidth will be modeled by adding additional banks of memory, separate blocks of external memory that are accessed through parallel data channels.

For example, adding a second memory bank will double the bandwidth and therefore halve the time required to transfer a fixed size block of data. The analysis assumes that memory coherence between the banks is separately maintained and does not require additional time or resources.

The number of banks required is calculated by dividing the data transfer time by the real-time goal and rounding up to an integer value as shown in Equation (6.2).

$$NumberOfBanks = ceiling\left(\frac{DataTransferTime}{TwoMinutes}\right) \tag{6.2}$$

The new memory bandwidth can then be calculated by multiplying the single bank memory bandwidth by the number of banks as shown in Equation (6.3) and the new data transfer time can be calculated with Equation (6.1).

$$TotalMemoryBandwidth = (NumberOfBanks)(MemoryBandwidth) \tag{6.3}$$

6.3 EIGENFACE MEMORY BANDWIDTH ANALYSIS

The memory bandwidth analysis for the 100 KP, 1K data set is shown in Figure 36. The Eigenface benchmark data includes an N element probe image, the N element mean image, the M’ by N element subspace matrix, the M’ element projected probe image p_curl, and the GM’ element encoded gallery set. These variables are listed in the left column in Figure 36, and the data direction is indicated by the "Read" or "Write" notation in the column to the right of the variable name. The third column in Figure 36 expresses the volume of data symbolically and the fourth column shows the numeric value. For example, the probe image p shown in the first row has N elements and N is 98,304 for the 100 KP, 1K data set. Each data word consists of eight bytes, so 8N bytes or 786 KB of data are required as shown. The 2019 data transfer time is calculated by dividing by the module bandwidth of 23.5 GB/s and then dividing by 60 to express the result in minutes as shown in the figure. These calculations show that 5.59 x 10^{-7} minutes are required to transfer one probe as shown in Figure 36. The analysis shows that total data transfer time is 0.00041 minutes in 2019 and 0.00033 minutes in 2022 and therefore memory bandwidth does not limit performance for this benchmark and data set.

Eigenface 100 KP, 1K					
Data	**Read or Write**	**Data Volume Expressed Symbolically**	**Volume (bytes)**	**2019 Transfer Time (minutes)**	**2022 Transfer Time (minutes)**
Encoding					
Probe image p	Read	N	7.86E+05	5.59E-07	4.55E-07
Mean image a	Read	N	7.86E+05	5.59E-07	4.55E-07
Subpace matrix U^T	Read	M'N	5.65E+08	4.01E-04	3.27E-04
Projected probe image p_curl	Write	M'	5.74E+03	4.08E-09	3.32E-09
Comparison					
Encoded gallery set	Read	GM'	6.87E+06	4.88E-06	3.98E-06
Projected probe image p_curl	Read	M'	5.74E+03	4.08E-09	3.32E-09
Eigenvalue constant vector eigvalue[ ]	Read	M'	5.74E+03	4.08E-09	3.32E-09
		Totals:		0.00041	0.00033

Figure 36: Eigenface 100 KP, 1K memory bandwidth analysis.

The memory bandwidth analysis for the 4 MP, 14K data set is shown in Figure 37. The transfer time is again real-time at 0.16 minutes and no additional memory banks are required.

The 4 MP, 1M data set, however, is constrained by memory bandwidth. Figure 38 shows the data transfer time is 16.61 minutes in 2019 and 13.53 minutes in 2022 and therefore memory bandwidth prevents real-time identification for both technology points.

Memory banks can be added to overcome the memory bandwidth constraints on the Eigenface 4 MP, 1M mapping as shown in Figure 38. Adding nine memory banks for the 2019 technology point reduces transfer time to 1.85 minutes and resolves the memory bandwidth constraint. Given the greater bandwidth available for the 2022 technology point, seven memory banks are enough to reduce the data transfer time to 1.93 minutes and remove the memory constraint on real-time identification.

As discussed in Chapter 4, the data volume and therefor data transfer times for the Bayesian benchmarks are approximately the same as for the Eigenface benchmarks. Therefore adding nine memory banks in 2019 and seven memory banks in 2022 will also resolve the memory bandwidth limitations for the Bayesian benchmarks.

Eigenface 4 MP, 14K					
Data	**Read or Write**	**Data Volume Expressed Symbolically**	**Volume (bytes)**	**2019 Transfer Time (minutes)**	**2022 Transfer Time (minutes)**
Encoding					
Probe image p	Read	N	3.10E+07	2.20E-05	1.79E-05
Mean image a	Read	N	3.10E+07	2.20E-05	1.79E-05
Subpace matrix U^T	Read	M'N	2.67E+11	1.90E-01	1.54E-01
Projected probe image p_curl	Write	M'	6.90E+04	4.90E-08	3.99E-08
Comparison					
Encoded gallery set	Read	GM'	9.90E+08	7.04E-04	5.73E-04
Projected probe image p_curl	Read	M'	6.90E+04	4.90E-08	3.99E-08
Eigenvalue constant vector eigvalue[ ]	Read	M'	6.90E+04	4.90E-08	3.99E-08
		Totals:		0.19	0.16

Figure 37: Eigenface 4 MP, 14K memory bandwidth analysis.

Eigenface 4 MP, 1M					
Data	**Read or Write**	**Data Volume Expressed Symbolically**	**Volume (bytes)**	**2019 Transfer Time (minutes)**	**2022 Transfer Time (minutes)**
Encoding					
Probe image p	Read	N	3.10E+07	2.20E-05	1.79E-05
Mean image a	Read	N	3.10E+07	2.20E-05	1.79E-05
Subpace matrix U^T	Read	M'N	1.86E+13	1.32E+01	1.08E+01
Projected probe image p_curl	Write	M'	4.80E+06	3.41E-06	2.78E-06
Comparison					
Encoded gallery set	Read	GM'	4.80E+12	3.41E+00	2.78E+00
Projected probe image p_curl	Read	M'	4.80E+06	3.41E-06	2.78E-06
Eigenvalue constant vector eigvalue[ ]	Read	M'	4.80E+06	3.41E-06	2.78E-06
		Totals:		16.61	13.53
		Number of memory banks required:		9	7
		Transfer time with additional banks (minutes):		1.85	1.93

Figure 38: Eigenface 4 MP, 1M memory bandwidth analysis.

6.4 EBGM MEMORY BANDWIDTH ANALYSIS

The memory bandwidth analysis for the EBGM 100 KP, 1K benchmark is shown in Figure 39. The encoding process loads bunch B containing one jet for each of 25 features vertices and each of M training images or 25M. Each jet contains 80 elements and the bunch therefore totals 2,000M words or 16,000M bytes. The probe jet adds another 2000 words or 16,000 bytes, one jet at each of 25 feature vertices. Each probe image region is W by W pixels, and one region is needed for each of the 25 vertices, requiring 25WW words or 2000WW bytes. Eighty filter masks are required, each W by W, requiring 80WW words or 6400 bytes. The resulting face graph has one jet at each of 25 feature vertices and 55 edge midpoints and therefore totals 6400 words or 512,000 bytes. For comparison, the 6,400 word probe face graph is required as well a G gallery face graphs of the same dimensions.

Figure 39 shows the transfer time for the 100KP, 1K data set is 5.36 x 10^{-5} minutes for the 2019 technology point and 4.36 x 10^{-5} minutes for the 2022 technology point. Both times meet the real-time data transfer requirement and therefore memory bandwidth is not a performance limit for the EBGM 100 KP, 1K data set.

The analysis also shows that memory bandwidth is not a performance constraint for the EBGM 4 MP, 14K and 4 MP, 1M benchmarks. Figure 40 shows the 4 MP, 14K data set requires transfer time of 6.57 x 10^{-4} minutes in 2019 and 5.35 x 10^{-4} minutes in 2022, and Figure 41 shows the 4 MP, 1M data set requires transfer time of 4.32 x 10^{-2} minutes in 2019 and 3.52 x 10^{-2} minutes in 2022, all well within the two minute real-time goal.

EBGM 100 KP, 1K					
Data	**Read or Write**	**Data Volume Expressed Symbolically**	**Volume (bytes)**	**2019 Transfer Time (minutes)**	**2022 Transfer Time (minutes)**
Encoding					
bunch B	Read	2000M	1.15E+07	8.16E-06	6.65E-06
probe jets	Read	2000	1.60E+04	1.14E-08	9.26E-09
probe image region	Read	25WW	8.11E+05	5.76E-07	4.69E-07
filter masks	Read	80WW	1.73E+06	1.23E-06	1.00E-06
probe face graph	Write	6400	5.12E+04	3.64E-08	2.96E-08
Comparison					
probe face graph	Read	6400	5.12E+04	3.64E-08	2.96E-08
gallery face graphs	Read	6400G	6.12E+07	4.35E-05	3.54E-05
		Totals:		5.36E-05	4.36E-05

Figure 39: EBGM 100 KP, 1K memory bandwidth analysis.

EBGM 4 MP, 14K					
Data	Read or Write	Data Volume Expressed Symbolically	Volume (bytes)	2019 Transfer Time (minutes)	2022 Transfer Time (minutes)
Encoding					
bunch B	Read	2000M	1.38E+08	9.80E-05	7.98E-05
probe jets	Read	2000	1.60E+04	1.14E-08	9.26E-09
probe image region	Read	25WW	1.64E+07	1.17E-05	9.51E-06
filter masks	Read	80WW	3.50E+07	2.49E-05	2.03E-05
probe face graph	Write	6400	5.12E+04	3.64E-08	2.96E-08
Comparison					
probe face graph	Read	6400	5.12E+04	3.64E-08	2.96E-08
gallery face graphs	Read	6400G	7.35E+08	5.23E-04	4.26E-04
		Totals:		6.57E-04	5.35E-04

Figure 40: EBGM 4 MP, 14K memory bandwidth analysis.

EBGM 4 MP, 1M					
Data	Read or Write	Data Volume Expressed Symbolically	Volume (bytes)	2019 Transfer Time (minutes)	2022 Transfer Time (minutes)
Encoding					
bunch B	Read	2000M	9.60E+09	6.82E-03	5.56E-03
probe jets	Read	2000	1.60E+04	1.14E-08	9.26E-09
probe image region	Read	25WW	1.64E+07	1.17E-05	9.51E-06
filter masks	Read	80WW	3.50E+07	2.49E-05	2.03E-05
probe face graph	Write	6400	5.12E+04	3.64E-08	2.96E-08
Comparison					
probe face graph	Read	6400	5.12E+04	3.64E-08	2.96E-08
gallery face graphs	Read	6400G	5.12E+10	3.64E-02	2.96E-02
		Totals:		4.32E-02	3.52E-02

Figure 41: EBGM 4 MP, 1M memory bandwidth analysis.

6.5 CONCLUSION

The results of the memory bandwidth analysis are summarized in Table 12 and show data transfer is real-time for the Eigenface and Bayesian 100 KP, 1K and 4 MP, 14K data sets. The Eigenface 4 MP, 1M and Bayesian 4 MP, 1M benchmarks require nine memory banks in 2019 and seven memory banks in 2022 to remove the memory bandwidth bottleneck as a constraint on real-time performance. Memory bandwidth does not constrain real-time performance for any of the EBGM benchmarks.

Table 12: Memory Bandwidth Analysis Summary.

Benchmark	Number of Banks in 2019	Number of Banks in 2022	Transfer Time 2019 (minutes)	Transfer Time 2022 (minutes)
Eigenface 100 KP, 1K	1	1	<0.001	<0.001
Eigenface 4 MP, 14K	1	1	0.19	0.16
Eigenface 4 MP, 1M	9	7	1.85	1.93
Bayesian 100 KP, 1K	1	1	<0.001	<0.001
Bayesian 4 MP, 14K	1	1	0.19	.016
Bayesian 4 MP, 1M	9	7	1.85	1.93
EBGM 100 KP, 1K	1	1	<0.001	<0.001
EBGM 4 MP, 14K	1	1	0.001	0.001
EBGM 4 MP, 1M	1	1	0.043	0.035

7.0 BENCHMARK MAPPING AND ANALYSIS

Chapter 4 analyzed performance of the Eigenface, Bayesian, and EBGM face identification algorithms. This analysis showed that the Eigenface and Bayesian algorithms are very similar and that if the Eigenface algorithm can be made real-time, the Bayesian algorithm would be real-time if implemented in the same way. The bottleneck code was extracted to form benchmarks and the performance of the benchmarks was estimated. The performance estimates showed that Eigenface and Bayesian 4 MP, 1M benchmarks were not real-time, and that all three EBGM benchmarks were not real-time.

Chapter 5 analyzed performance of the ISA and developed fused instructions to improve single PE performance. This analysis showed that the EBGM 100 KP, 1K benchmark could be made real-time with ISA speedup alone as shown in Table 13. However, for the other benchmarks that require computational speedup, the ISA speedup was insufficient to completely resolve the bottleneck. The Chapter 5 analysis summarized in Table 13 showed that the Eigenface and Bayesian benchmarks with the 4 MP, 1M data set could be accelerated 2.2 to 2.5 times by adding a fused multiply and add instruction MDADD and a fused difference squared instruction DFSQ. The EBGM benchmarks with the 4 MP, 14K and 4 MP, 1M data sets can be accelerated 2.46 times by adding a fused indexed move instruction MOVIDX, and adding two MOVIDX function units made the EBGM 100 KP, 1K benchmark real-time. This resulted in an improved PE for the Eigenface and Bayesian benchmarks that includes one function unit that implements the MDADD instruction and one function unit for the DFSQ instruction. The improved PE for the EBGM benchmark includes two function units that implement the MOVIDX instruction to meet the input data requirements of the multiply instruction that follows the index instructions.

Table 13: Required Computational Speedup

Benchmark	Sequential Speedup Required	ISA Speedup	Additional Speedup Required
Eigenface 100 KP, 1K	-	-	-
Eigenface 4 MP, 14K	-	-	-
Eigenface 4 MP, 1M	38	2.51	15.14
Bayesian 100 KP, 1K	-	-	-
Bayesian 4 MP, 14K	-	-	-
Bayesian 4 MP, 1M	38	2.21	17.19
EBGM 100 KP, 1K	2	3.24	-
EBGM 4 MP, 14K	748	4.92	151.96
EBGM 4 MP, 1M	748	4.91	152.42

External memory bandwidth was analyzed in Chapter 6. ITRS and JEDEC project memory bandwidth of 23.46 GB/s in 2019 and 28.80 GB/s in 2022. The analysis in Chapter 6 showed that memory bandwidth did not create a bottleneck for the Eigenface and Bayesian 100 KP, 1K and 4 MP, 14K data sets, but that the 4 MP, 1M data set is memory bound. However, using 9 memory banks in 2019 and 7 in 2022 was sufficient to reduce the data transfer time below the two minute real-time goal and these memory banks are used for subsequent analysis. The EBGM benchmark was not memory bound for any of the data sets and no additional memory banks are required.

The prior chapters independently analyzed key components that can limit performance of the face identification benchmarks and did not consider interaction between the components. In this chapter, the components will be integrated into an SOC system architecture and the performance of this system architecture will be analyzed to determine if real-time performance can be achieved.

Two system models for SOC system architectures will be developed. Both models will consist of an SOC and an external memory block as shown in Figure 42, but the interface between external memory and the SOC will be different for the two architectures. The first or "basic" architecture will contain a single channel between external memory and the SOC. In this model, only one processor will have direct access to external memory. In the second or "multichannel" model, multiple external memory banks will be provided, and each memory bank will be interfaced to one processor. In this model, given *n* memory banks, *n* processors will each have direct access to memory. The multichannel model will be used for the benchmarks that require multiple memory banks based on the Chapter 6 analysis, and the basic model will be used for benchmarks that do not require multiple memory banks.

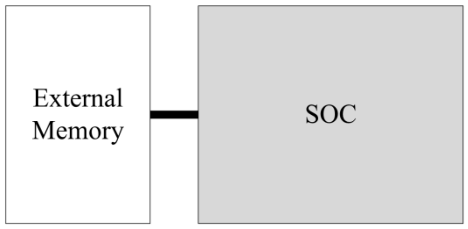


Figure 42: System Architecture Model.

Given an SOC model, this chapter will explore how the benchmarks can be mapped to the SOC in a way that improves performance while keeping within the constraints of on-chip memory capacity, number of processing elements, and limitations of on-chip communication. A benchmark mapping analysis method will be developed to quantify the impact of mapping, on-chip memory capacity, memory transfer time, the number of PEs, and on-chip communication on performance. An analysis method to evaluate performance will be developed and this analysis method will include a graphical representation of the mapping that will expose communication operations and transfers between external memory and the SOC. An estimation method to quantify the time required for each component of system performance will also be developed. This estimation method will be used in conjunction with the previously developed computational analysis and memory bandwidth analysis methods to estimate performance for the benchmark mapping to the SOC system model and therefore provide a method to estimate overall system performance for the parallel SOC.

7.1 PARALLEL SOC SYSTEM MODEL

System models for a parallel SOC architecture are needed to support development of a method to estimate system performance. Performance of a benchmark mapped to a parallel SOC is impacted by the time required for data movement, computation, and communication. Estimation of the time required for each of these components requires definition of the characteristics of the components as well as the architecture. For example, to estimate communication performance, the message type, network topology, routing method, latency, and bandwidth must be determined and a set of equations to estimate the communication time based on these characteristics is required. Two system models for parallel SOC architectures will be developed in this section to specify the characteristics required to estimate performance for the SOCs.

Two parallel SOC models will be defined. The first or "basic" system architecture consists of one SOC and one external memory system as shown in Figure 43. The external memory provides main storage for large data sets and is characterized by memory bandwidth and storage capacity. The external memory bandwidth was analyzed in Chapter 6 and the bandwidth estimates and time estimation method from that chapter will be used to estimate external memory transfer time for the system model.

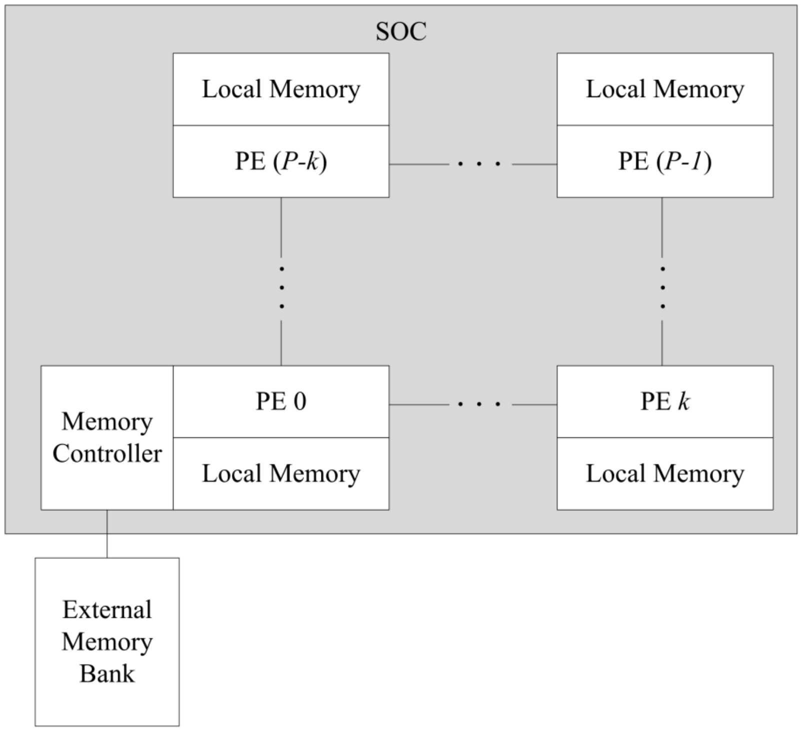


Figure 43: Basic parallel SOC system model.

The "multichannel" system architecture provides multiple channels to access multiple memory banks as shown in Figure 44. The Chapter 6 analysis showed that some Eigenface and

Bayesian benchmarks are memory bandwidth limited and require more than one external memory bank to avoid a bottleneck. The multichannel model incorporates the multiple memory banks by connecting each external memory bank to a different processor. As in the basic model, only one processor can directly access a particular external memory bank and therefore each processor can transfer only one data word at a time. However, the several processors can operation in parallel and transfer multiple data words simultaneously, on word per processor, and thus increase the total memory bandwidth for the SOC.

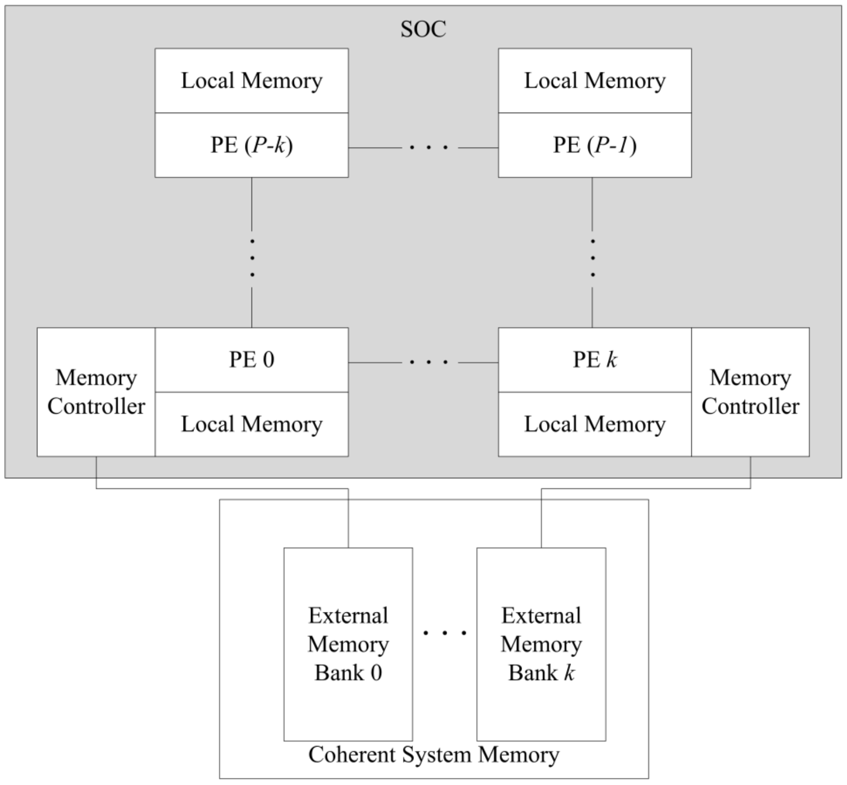


Figure 44: Multichannel parallel SOC system model.

7.1.1 Processor Model

Both the basic and multichannel SOC models contain multiple processors as shown in Figure 43 and Figure 44. Each processor block contains one *Processing Element* (PE), a hardware element that executes a stream of instructions [122], and one Local Memory that can be directly accessed only by the one PE within the block. Each PE is an improved x86 processor that executes the instructions defined by the improved ISA developed in Chapter 5 and the average CPI quantifies the number of cycles required to execute each instruction. The execution time estimation method described in Chapter 4 will be used to estimate execution time for each PE in the SOC model.

The minimum number of cycles required to execute a process in parallel on multiple PEs, $C_{Parallel}$, is calculated by dividing the single PE execution cycle count, $C_{Sequential}$, by the number of parallel PEs, *P*, expressed in equation form as $C_{Parallel} = C_{Sequential} / P$. This calculation assumes that the register file and Local Memory can provide data to the PE as required to avoid stalling.

The processor blocks in both the basic and multichannel SOC models contain one Local Memory connected to only one PE. This memory configuration in the SOC models therefore forms a distributed memory model [21], where each PE has access its own Local Memory but cannot directly access the Local Memory of any other PE.

The storage capacity of Local Memory is a function of the technology used for the SOC. One model in the ITRS projects a constant per PE storage capacity for Local Memory for the range of future technology projections [123]. This model accounts for increasing transistor density by increasing the number of PEs on the SOC rather than increasing the Local Memory size for each PE. As a result, the memory per PE is constant but the total on-chip memory grows from 1.45 GB of on-chip memory in the 2019 thousand PE SOC to 2.9 GB in the 2022 two thousand PE SOC. For the basic and multichannel SOC models, Local Memory capacity is set to 1.45 GB for the 2019 technology point and 2.9 GB for the 2022 technology point as projected by the ITRS [123] and as shown in Table 14.

Table 14: Local Memory Capacity.

TECHNOLOGY POINT	NUMBER OF PEs	LOCAL MEMORY PER PE	TOTAL SOC MEMORY
2019	1,000	1.45 MB	1.45 GB
2022	2,000	1.45 MB	2.90 GB
2022	20	356.86 MB	7.14 GB

The parallel SOC system models being specified for this research assumes PEs can be traded for memory. If a benchmark mapping does not require all 1,000 PEs for the 2019 technology point or 2,000 PEs for the 2022 technology point, the unused PEs can be exchanged for increased Local Memory storage capacity as shown in Table 14. The ITRS estimates 40 million transistors are required to implement one general purpose PE and that dynamic memory requires chip area equivalent to two transistors per bit [123]. Therefore the chip area required for one PE can be exchanged for (40 x 10^6)/2=20 x 10^6 bits of dynamic memory or 2.5 MB. In addition, the 1.45 MB of local memory previously allocated to the PE that was exchanged is no longer used and therefore is also available for allocation to other PEs. The total additional memory provided by exchanging one PE for memory is therefore 1.45 MB+2.5 MB or 3.95 MB.

Dividing the total on-chip memory capacity by the number of PEs on the chip calculates the Local Memory storage capacity per PE. For example, if a mapping requires 1,000 PEs and the 2,000 PE SOC projected for 2022 is used, the 1,000 PEs on the 2022 SOC that are not needed for the mapping could be exchanged for 1,000*3.95 MB=3,950 MB of total additional memory on the SOC, adding 3,950 MB/1,000 = 3.95 MB of Local Memory for each PE for a total Local Memory of 5.4 MB per PE.

7.1.2 External Memory Model

The basic SOC model has a single external memory bank that interfaces only to PE0 as shown in Figure 43. When another PE needs to move data between its own Local Memory and external memory, it sends a message to PE0 to request the transfer. If the request is an external memory read, PE0 first transfers the data to its own Local Memory, then sends a message containing the data to the destination PE. If the request is an external memory write, PE0 receives a message containing the data from the source PE, stores the data in its own Local Memory, and then moves the data to the external memory.

The time to transfer data between PE0 and the external memory bank is characterized by the memory bandwidth and can be estimated with the analysis method developed in Chapter 6. Memory bandwidth is assumed to increase with technology and time and the 2019 bandwidth of 23.46 GB/s and 2022 bandwidth of 28.80 GB/s determined in Chapter 6 from ITRS projections [123] is used to estimate data transfer time.

A single memory channel creates a bottleneck for some benchmarks. The Chapter 6 analysis showed that the Eigenface and Bayesian benchmarks are memory bound for some data sets and require additional memory banks. Multiple memory banks could be connected to PE0 to increase the average memory bandwidth. As in the basic SOC model, in this model PE0 would move data between external memory and its own Local Memory and communicate the data with other PEs over the communication network. However, the average communication bandwidth would be less than the accelerated memory bandwidth and would likely create a communication bottleneck. While the communication performance could be increased to mitigate the new bottleneck, the additional resources required in addition to the resources required to improve the memory bandwidth suggests this approach would be inefficient.

Interfacing each memory bank to a separate PE could improve performance while minimizing the requirements for additional resources. If the added memory banks are interfaced to separate PEs, those PEs can access the external memory directly without the need to communicate messages over the network, thus avoiding the bottleneck of the communication bandwidth. Figure 44 shows the multichannel SOC architecture with multiple memory banks and multiple channels. This example shows the addition of (k+1) memory banks, and each bank interfaces to one PE in the range of PE0 to PEk. Although the memory banks are shown as separate blocks of memory, this model assumes that coherence is maintained independently and does not increase the workload of the PEs.

The performance analysis method will model the multiple memory banks as a single higher bandwidth channel. As in the Chapter 6 analysis method, the additional memory channels are modeled by multiplying the memory bandwidth for a single channel by the total number of channels to calculate the total memory bandwidth. The data transfer time can then be estimated by dividing the number of data words transferred by the total memory bandwidth, effectively modeling the multiple memory banks as a single, higher bandwidth external memory. The bandwidth for each benchmark and data set as calculated in Chapter 6 is summarized in Table 15.

Table 15: Benchmark Memory Bandwidth Requirements.

Benchmark	Number of Banks in 2019	Number of Banks in 2022	2019 Bandwidth (GB/s)	2022 Bandwidth (GB/s)
Eigenface 100 KP, 1K	1	1	23.5	28.8
Eigenface 4 MP, 14K	1	1	23.5	28.8
Eigenface 4 MP, 1M	9	7	211.1	201.6
Bayesian 100 KP, 1K	1	1	23.5	28.8
Bayesian 4 MP, 14K	1	1	23.5	28.8
Bayesian 4 MP, 1M	9	7	211.1	201.6
EBGM 100 KP, 1K	1	1	23.5	28.8
EBGM 4 MP, 14K	1	1	23.5	28.8
EBGM 4 MP, 1M	1	1	23.5	28.8

Two operations are defined to access external memory, *Localize* and *Globalize*. The Localize operation transfers data from the external memory to the Local Memory of the PE connected to that external memory bank. Conversely, the Globalize operations transfers data from the Local Memory of the PE connected to the memory bank to the off-chip memory bank [122]. These two operations are the only operations that can be used to directly access external memory.

The size of the Local Memory associated with PE0 in the basic model can impose a performance constraint. Since only PE0 has access to external memory in the basic system model, all data must first be loaded to the Local Memory associated with PE0, then transferred to other PEs with communication operations. If a large block of data such as the U^T matrix for the Eigenface benchmark is Localized on PE0 first and then scattered to the other PEs, PE0 will require greater Local Memory capacity than the other PEs since it has to store the entire matrix. Alternatively, if one segment of U^T is loaded, then transferred to the target PE, and this process is repeated for each PE, the memory requirement for PE0 is the same as the other PEs. Thus, the benchmark mapping for the basic SOC system model will affect the required capacity for PE0 Local Memory, and Local Memory capacity must be considered when designing a mapping.

7.1.3 Communication Model

For both the basic and multichannel SOC models, the communication network is defined as a 2D mesh topology interconnecting the PEs. The PEs are configured as a square grid with $\sqrt{P}$ rows and $\sqrt{P}$ columns, where P is the total number of PEs on the SOC. The 2D mesh is selected as the network topology because it maps well to the two dimensions of the SOC and is considered to be one of the most efficient topologies for on-chip networks [124].

The PEs communicate over the network using standard MPI collective communication messages [125]. The MPI collective communication standard defines a complete set of messages that efficiently support communication of data and control information between PEs. These messages can be used to transfer data between PEs with external memory access and the other PEs as well as to exchange synchronization and control information between PEs.

The time required to communicate a message on the network is estimated using Chan's model for collective on-chip communication [27]. The time required to communicate a message is based on the *latency*, the time required to initiate a message, and the *bandwidth*, the rate at which each data word is communicated. In Chan's model, the symbol α represents the latency while the symbol β represents the time to transfer one byte, the inverse of bandwidth [27].

Latency and bandwidth were estimated for the 2019 and 2022 SOCs based on the Tile64 SOC [18] and ITRS projections [123] as shown in Table 16. The 2D mesh communication network on the Tile64 has a latency of 45.5 ns and a bandwidth of 325 MB/s and was produced in 2009. The ITRS projects an increase of 1.154 times per year, and scaling the 2009 latency of the TILE64 by the ITRS growth rate projects latency of 10.85 ns in 2019 and 7.06 ns in 2022. In the same way, bandwidth is projected as 1.36 GB/s in 2019 and 5.70 GB/s in 2022.

Table 16: On-chip Communication Latency and Bandwidth.

Communication	Tile64 2009	ITRS Projection 2019	ITRS Projection 2022	
Latency α	7.58E-10	1.81E-10	1.18E-10	minutes
Data Time β	5.13E-11	1.22E-11	2.92E-12	minutes/B
Data Rate (Bandwidth)	0.33	1.36	5.70	GB/s

Six types of messages are defined for the SOC system models as shown in Figure 45. The BCAST message sends the same block of data to multiple destination PEs. The SCATTER message divides the data block into equal size segments and sends one segment to each destination PE. The GATHER message receives equal size segments from each PE and combines the segments to form one block of data. The REDUCE message receives a data word from each PE and combines the data with a specified operation such as SUM, MIN, or MAX. The SUM operation adds the data words, the MIN operation selects the smallest data word, and the MAX operation selects the largest data word. The SEND message transmits a point-to-point message from the transmitting PE and the RECV message receives the message on the destination PE.

		Collective Communication Equations [27]			
		Small Message		Long Message	
Communication Operations [125]	Description	Startup Time (Latency)	Transfer Time for N words (bandwidth)	Startup Time (Latency)	Transfer Time for N words (bandwidth)
BCAST(data, size, number PE)	Send data to each PE	Ceiling[logP]3α	Ceiling[logP](nβ)	2α+ Ceiling[logP](3α)	[2(P-1)/P]nβ
SCATTER(data, size, number PEs)	Send data segment to each PE	Ceiling[logP](3α)	[(P-1)/P]nβ	3(P-1)α	[(P-1)/P]nβ
GATHER(data, segment size, number PEs)	Retrieve data segment from each PE	Ceiling[logP](3α)	[(P-1)/P]nβ	3(P-1)α	[(P-1)/P]nβ
REDUCE(OP, data, segment size, number PEs)	Retrieve data segment from each PE, reduce to one output	Ceiling[logP](3α)	Ceiling[logP](nβ)	2α+ Ceiling[logP](3α)	[2(P-1)/P]nβ
SEND(data, size, receiving PE)	Send one message	α	nβ	α	nβ
RECV(data, size, send PE)	Receive message	α	nβ	α	nβ

Figure 45: Collective on-chip communication messages.

7.2 MAPPING ANALYSIS METHOD

Given a mapping of a face identification benchmark to either the basic or multichannel parallel SOC architectures, the performance of the mapping and the benchmark will be analyzed with the following process:

Step 1: Express the parallel mapping. The analysis method described in this chapter analyzes the performance of a particular mapping of a particular face identification benchmark to the improved parallel SOC architecture. The first step in the analysis process is to express the mapping as an extended UML diagram and then use the UML diagram to amend the benchmark code to include memory transfers and on-chip communications.

The standard UML Activity diagram [126, 127] expresses control flow of a process as shown in Figure 46. The black solid circle at the top of the diagram represents the entry point of the process and the similar circle enclosed within a second circle shown at the bottom of the figure represents the exit point of the process. Operations are represented by Action blocks, rectangles with rounded corners containing a text description of the operation or task performed. The bold horizontal bars represent *Fork*, a control flow operation to separate one process into multiple processes, and *Join*, a control flow operation that combines multiple processes into one process. The thin lines with arrows drawn between Action symbols show the control flow between Actions. The diamond symbol represents a decision block and therefore shows two control flow lines leaving the symbol, one for each binary decision outcome. In this example, the control flow from Action D goes to the decision block, which either continues with Action A or exits the process based on the outcome of the decision.

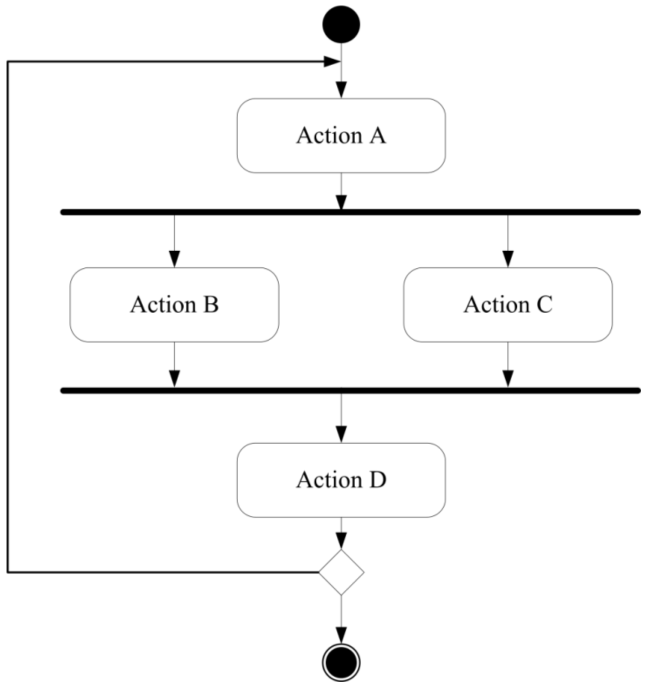


Figure 46: UML Activity diagram.

The standard UML diagram can be extended to explicitly show external memory transfers and on-chip communication as shown in Figure 47. Separate Action blocks are added to show the external memory transfers as Localize and Globalize operations. For example, in Figure 47 the first block labeled “Localize(vector x[N]” indicates that N elements of vector x will be transferred from external memory to on-chip Local Memory. The last block labeled “Globalize vector y_local[N]” indicates that the N elements of the vector y_local will be transferred from on-chip Local Memory to external memory.

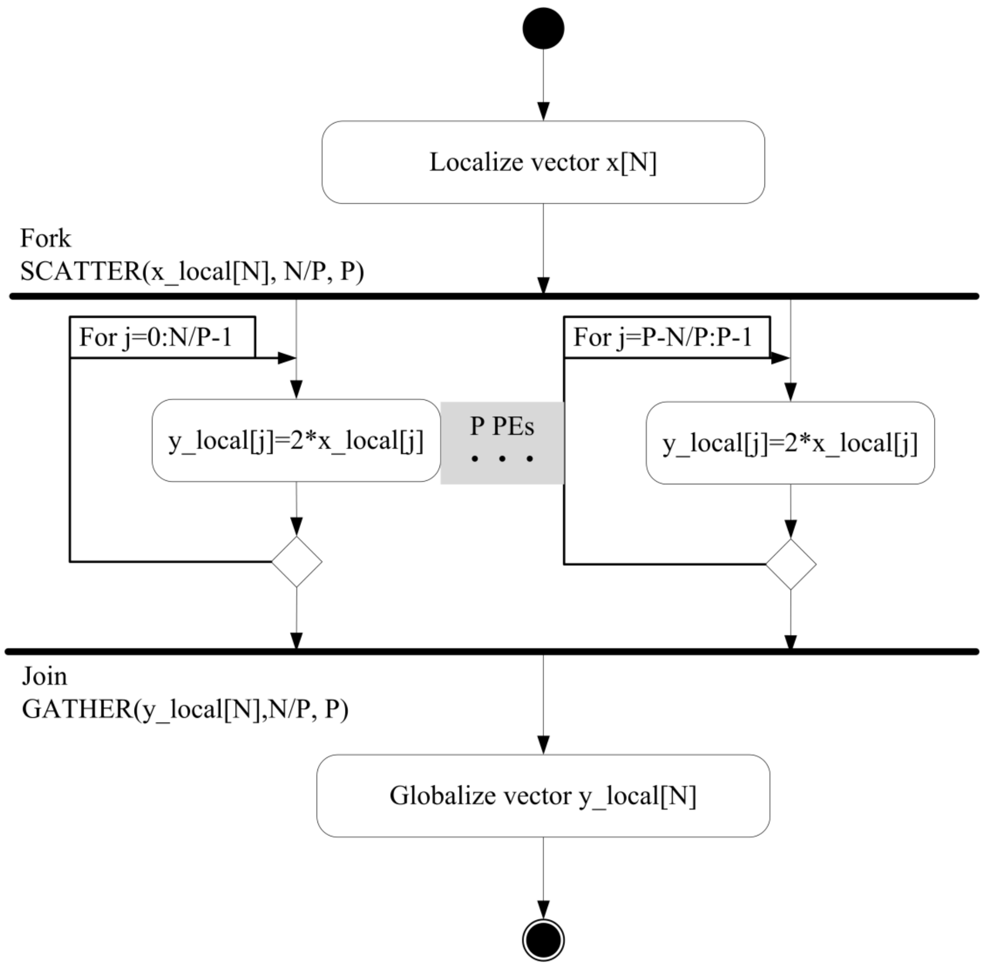


Figure 47: The extended UML Activity diagram.

The extended UML diagram adds labels to show on-chip communication operations to the Fork and Join bars defined in the standard UML Activity diagram. Figure 47 shows that the fork bar near the top of the diagram is labeled “Fork” and “SCATTER(x_local[N],N/P,P)” on the left side. The “Fork” label indicates the process will split into multiple processes, and “SCATTER” label indicates that [N] elements of the vector x_local will be segmented into N/P element groups and each of P PEs will receive on segment.

The join bar at the bottom of Figure 47 is similarly labeled to show the “Join” and also “GATHER(y_local[N],N/P,P)” to indicate that P segments, each with N/P elements, of the [N] element vector y_local are collected from P PEs.

The extended UML diagram shows parallel processes as a horizontal row of Action blocks as shown in Figure 47. Standard UML does not provide a notation to indicate a large number of parallel processes in an Activity diagram, so the ellipsis notation is added as an obvious extension to indicate many parallel processes as shown in Figure 47. Action blocks for the first and last parallel process are explicitly shown, and an ellipsis labeled “P PEs” between the first and last Action blocks indicates that there are P Action blocks and that these processes are performed in parallel on P PEs.

The explicit data movement and on-chip communication operations shown in the extended UML diagram are then added to the benchmark code. Figure 48 shows example pseudo code for the extended UML diagram shown in Figure 47. Line 1 indicates that lines 2 through 4 are executed only on PE0. The Localize operation in the first block in the extended UML diagram is shown on line 3 in the pseudo code, and line 4 shows the SCATTER operation. The PARDO on line 5 indicates that lines 6 through 8 are executed in parallel on P PEs. Line 5 therefore corresponds to the fork bar in Figure 47 and lines 7 and 8 correspond to the parallel Action blocks in the UML diagram. Line 7 iterates through the elements of x_local, which have indices 1 through N/P on each PE. While these indexes are the same on each PE, the segment of x_local that was scattered to each processor is a different segment and therefore the data accessed by the index on each PE is different. Line 8 multiplies the element of x_local by two and stores the result to y_local.

The Join is shown implicitly in Figure 48 by the end of the PARDO loop. Line 9 causes only PE0 to execute lines 10 and 11. Line 10 corresponds to the GATHER label on the join bar in Figure 47 and combines the segments of y_local received from each PE into the vector y_local[]. Line 10 then transfers y_local back to external memory as the vector y.

```
1  If PE0
2      // transfer x to PE0 Local Memory
3      x_local[0:N-1]=Localize(x[0:N-1])
4      SCATTER(x_local[0:N-1], N/P, P)
5  for p=1 to P PARDO
6      // iterate through the N/P elements on processor p
7      for each element j in N/P
8          y_local[j]=2*x_local[j]
9  If PE0
10     GATHER(y_local[0:N-1],N/P, P)
11     y[0:N-1]=Globalize(y_local[0:N-1])
```

Figure 48: Data movement and communication operations added to the code.

<u>Step 2: Estimate required memory capacity.</u> The required on-chip memory capacity is calculated based on the largest set of data that must reside on-chip simultaneously. The benchmarks use multiple data structures but a particular mapping may not require all the data to be in on-chip memory simultaneously. For example, to subtract the mean image from the probe image in the Eigenface benchmark requires both the probe image and the mean image input data to be on-chip simultaneously, and the mean-subtracted probe vector that will be generated as an output will be stored on-chip as well. However, the subspace matrix, $\mathbf{U}^{\mathbf{T}}$, is not required for the mean subtraction process and therefore does not have to be on-chip simultaneously.

The basic SOC model requires all data to be moved through PE0 and PE0 may therefore require larger Local Memory storage capacity than the other PEs. The storage capacity for PE0 must therefore be sufficient to temporarily store all the data that will be transferred to other PEs as well as the data required for PE0. For example, if PE0 is loading the Eigenface matrix U^T from external memory in preparation for distributing it to the other PEs, it will be necessary to store the entire Eigenface matrix on PE0. If the all of the data will be broadcast to the other PEs, every PE will receive the same volume of data and the storage capacity requirement for PE0 does not increase. However, if the matrix is scattered to the other PEs, each PE will receive a segment of the matrix but PE0 will require enough Local Memory capacity to store the entire matrix. Therefore, the code developed in Step 1 must be analyzed to determine the largest volume of data that must be on-chip simultaneously for both PE0 and the other PEs, and these data volumes set the requirement for Local Memory capacity for the SOC.

Step 3: Estimate data movement time. The data movement time can then be estimated using the memory analysis method developed in Chapter 6 and the amended benchmark code. For each Localize and Globalize operation added to the code, the total number of bytes transferred can be calculated based on the size of the data structure. Given a data structure containing B bytes of data and a memory bank with bandwidth $1/\beta$, the time required to transfer that data structure is the number of bytes divided by the bandwidth, $T_{Transfer} = B\beta$ as shown in Chapter 6.

Step 4: Estimate computation time. The number of cycles required for parallel computation is estimated based on the sequential computation estimation results developed in Chapter 4 and improved with the ISA speedup developed in Chapter 5. As discussed in Section 7.1.1, the number of cycles required for parallel execution is the number of cycles for sequential execution on one improved PE divided by the number of PEs.

Step 5: Estimate on-chip communication time. The number of cycles required for each communication operation is estimated using Chan's communication model as described in Section 7.1.3.

Step 6: Estimate performance. The components of execution time were calculated in the prior steps. The required storage capacity for Local Memory was estimated in Step 2, and Step 3 estimated the time required to move data on and off the chip. Step 4 calculated the time for both sequential and parallel computation and Step 5 estimated the time for the required communication operations. These results are accumulated in this step to calculate the execution time for the current mapping of the algorithm.

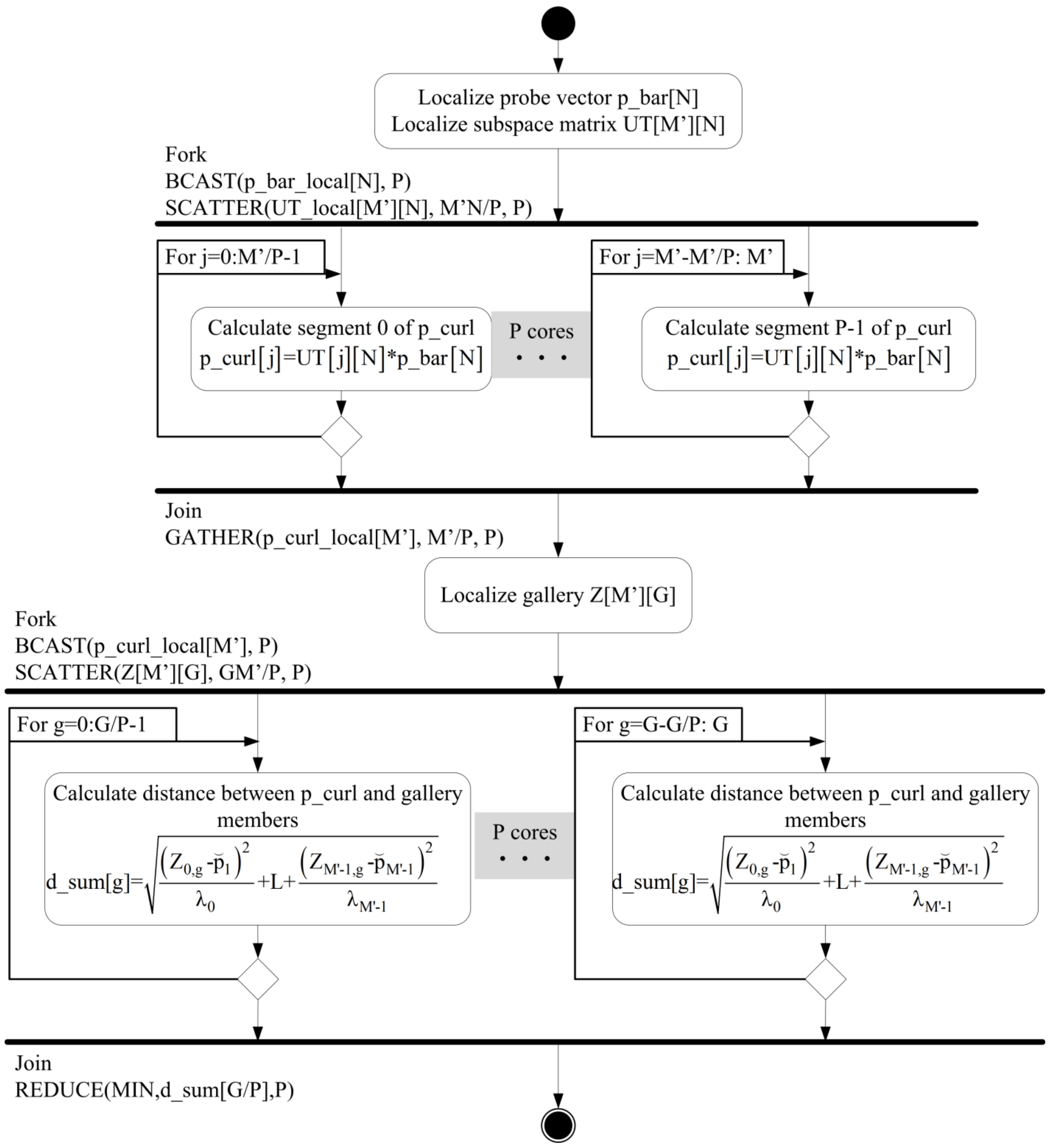
The number of cycles required for external memory transfers, computation, and on-chip communication are summed to estimate the total number of cycles for the parallel mapping. The percentage of total cycles required for data movement, computation, and on-chip communication is then calculated and expressed as a percentage per line of pseudo code and as a percentage for each component operation, including external memory transfers, computation, and communication.

The number of cycles is converted to minutes to evaluate whether the performance is real-time. If real-time identification is not achieved with a particular mapping, the benchmark can be re-mapped to mitigate the bottleneck operation and the performance can be re-estimated.

7.3 EIGENFACE MAPPING ANALYSIS

The analysis in Chapter 4 showed that an overall speedup of 38 times is required to make the Eigenface 4 MP, 1M benchmark real-time. The ISA speedup achieved with a single set of function units provided a speedup of 2.5 times, reducing the additional speedup requirement to 15.2 times for this data set. The analysis in Chapter 6 showed that the bandwidth of external memory also constrains performance for this data set, so nine memory banks were added for the 2019 technology point and seven memory banks were added for the 2022 technology point to mitigate this bottleneck. The goal in this section is therefore to determine a mapping that will provide the remaining 15.2 times speedup required to make this benchmark real-time.

The initial mapping for the Eigenface benchmark is shown in Figure 49. This mapping broadcasts the projected probe vector, p_bar, to each PE core and scatters M'/P rows of subspace matrix, U^T, across the PE cores, where P is the number of PEs on the SOC. Each PE calculates M'/P elements of the projected probe vector p_curl in parallel, and the elements are gathered on one PE to form the p_curl vector.



Localize probe vector p_bar[N]
Localize subspace matrix UT[M’][N]
Fork
BCAST(p_bar_local[N], P)
SCATTER(UT_local[M’][N], M’N/P, P)
For j=0:M’/P-1
For j=M’-M’/P: M’
Calculate segment 0 of p_curl
p_curl[j]=UT[j][N]*p_bar[N]
P cores
Calculate segment P-1 of p_curl
p_curl[j]=UT[j][N]*p_bar[N]
Join
GATHER(p_curl_local[M’], M’/P, P)
Localize gallery Z[M’][G]
Fork
BCAST(p_curl_local[M’], P)
SCATTER(Z[M’][G], GM’/P, P)
For g=0:G/P-1
For g=G-G/P: G
Calculate distance between p_curl and gallery members
$d\_sum[g]=\sqrt{\frac{(Z_{0,g}-\breve{p}_1)^2}{\lambda_0}+L+\frac{(Z_{M'-1,g}-\breve{p}_{M'-1})^2}{\lambda_{M'-1}}}$
P cores
Calculate distance between p_curl and gallery members
$d\_sum[g]=\sqrt{\frac{(Z_{0,g}-\breve{p}_1)^2}{\lambda_0}+L+\frac{(Z_{M'-1,g}-\breve{p}_{M'-1})^2}{\lambda_{M'-1}}}$
Join
REDUCE(MIN,d_sum[G/P],P)

Figure 49: Eigenface benchmark row-column mapping.

The projected probe vector, p_curl, is then broadcast to the P PEs and the matrix of projected gallery members, Z, is scattered to the P PEs. Each PE calculates the vector distance between the probe and each of G/P gallery members, producing G/P distance values d_sum_g. The minimum d_sum value from all of the PE is calculated on one PE with a Reduce operation and this d_sum value is the gallery member that best matches the probe.

Given this mapping, the next step is to determine the required Local Memory capacity. A 15.2 times computational speedup is required, so 20 PEs should be sufficient to provide this speedup and the additional PEs can be traded for memory. Estimating for the 2,000 PE 2022 technology point shows the total on-chip memory capacity is 7,137.2 MB as shown in Figure 50. However, this mapping requires on-chip memory capacity of 18,583.15 MB, orders of magnitude greater than the SOC memory capacity. Therefore, this mapping is not viable and another mapping must be developed. To be feasible, the new mapping will have to segment the data into smaller units that can fit in the on-chip memory capacity.

4 MP, 1M				
Data	**Read or Write**	**Data Volume Expressed Symbolically**	**Volume (bytes)**	**Required On-chip Memory (bytes)**
		Encoding		
Probe image p	Read	N	3.10E+07	
Mean image a	Read	N	3.10E+07	
Subpace matrix U	Read	M'N	1.86E+13	
Projected probe image p_curl	Write	M'	4.80E+06	1.86E+13
		Comparison		
Encoded gallery set	Read	GM'	4.80E+12	
Projected probe image p_curl	Read	M'	4.80E+06	
Eigenvalue constant vector eigvalue[ ]	Read	M'	4.80E+06	4.80E+12
	Required SOC Memory Capacity			1.86E+07 MB
	Available SOC Memory Capacity			356.86 MB

Figure 50: Memory capacity analysis, Eigenface row-column mapping.

A mapping that uses the on-chip memory resources efficiently is needed to practically map the Eigenface benchmark to the parallel SOC. The revised mapping shown in Figure 51 segments the subspace matrix and gallery set matrix to enable them to fit within the on-chip memory. The projection process divides U^T into S segments by rows and distributes groups of R rows across the PEs. Each PE calculates the elements of p_curl that correspond to the rows of U^T stored in the Local Memory for that PE, then globalizes the result to consolidate the p_curl segments in external memory. The comparison process is performed in a similar way, with C columns of Z distributed to each PE and one set of C Mahalinobis distances calculated on each PE.

The memory capacity analysis for this mapping is shown in Figure 52. This analysis confirms that this mapping will fit in the on-chip memory when 20 PEs are used. In addition, this mapping uses the memory efficiently and uses 96% of the available memory, with the 4% unused resulting from defining the segments in terms of multiples of the dimensions of U^T.

The benchmark is next modified in a series of incremental revisions to add the explicit data movement instructions and to parallelize the sequential code. Figure 53 shows the Eigenface algorithm in equation form, as previously described in Chapter 4. The benchmark pseudo code is shown in Figure 54 with loop index variables simplified to expose the numerical values.

Lines 1 through 3 in Figure 54 express the probe projection process. Line 1 iterates variable k through the M' Eigenfaces, and M' is 600,000 for the 4 MP, 1M benchmark data set. Line 2 iterates variable j through the $N=3.87 \times 10^6$ image pixels and line 3 multiplies and accumulates one probe projection coefficient, p_curl[k].

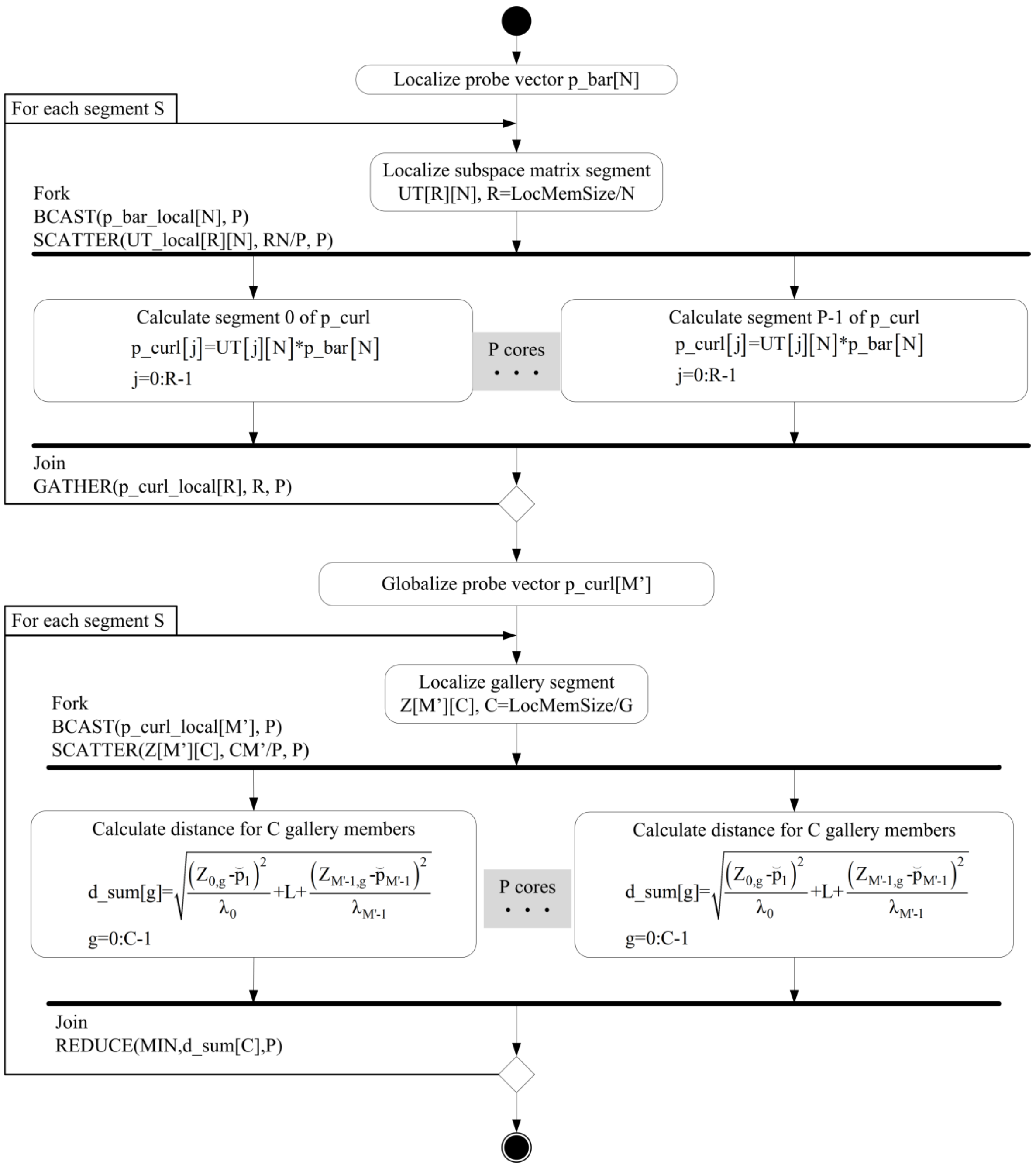

Localize probe vector p_bar[N]
For each segment S
Localize subspace matrix segment UT[R][N], R=LocMemSize/N
Fork
BCAST(p_bar_local[N], P)
SCATTER(UT_local[R][N], RN/P, P)
Calculate segment 0 of p_curl
p_curl[j]=UT[j][N]*p_bar[N]
j=0:R-1
P cores
Calculate segment P-1 of p_curl
p_curl[j]=UT[j][N]*p_bar[N]
j=0:R-1
Join
GATHER(p_curl_local[R], R, P)
Globalize probe vector p_curl[M']
For each segment S
Localize gallery segment Z[M'][C], C=LocMemSize/G
Fork
BCAST(p_curl_local[M'], P)
SCATTER(Z[M'][C], CM'/P, P)
Calculate distance for C gallery members
d_sum[g]=√((Z_{0,g} - p̆_1)²/λ_0 + L + (Z_{M'-1,g} - p̆_{M'-1})²/λ_{M'-1})
g=0:C-1
P cores
Calculate distance for C gallery members
d_sum[g]=√((Z_{0,g} - p̆_1)²/λ_0 + L + (Z_{M'-1,g} - p̆_{M'-1})²/λ_{M'-1})
g=0:C-1
Join
REDUCE(MIN,d_sum[C],P)

Figure 51: Eigenface benchmark segment mapping.

4 MP, 1M				
Number of PEs	20			
Projection Segment Size R	10			
Gallery Segment Size C	72			
Data	**Read or Write**	**Data Volume Expressed Symbolically**	**Volume (bytes)**	**Required On-chip Memory (bytes)**
Encoding				
Mean subtracted probe p_bar	Read	M'	3.10E+07	
Subpace matrix U segment	Read	RNP	6.19E+09	
Projected probe image p_curl	Write	M'	4.80E+06	6.23E+09
Comparison				
Gallery Segment	Read	CM'P	6.91E+09	
Projected probe image p_curl	Read	M'	4.80E+06	
Eigenvalue constant vector eigvalue[ ]	Read	M'	4.80E+06	6.92E+09
	Required SOC Memory Capacity		6,921.6	MB
	Available SOC Memory Capacity		7,137.2	MB

Figure 52: Memory requirements for Eigenface segment mapping.

```
$BestMatch = \mathrm{EFId}(\mathbf{I}, \mathbf{Z})$
1 $\mathbf{p}$=ConvertImageToVector$(\mathbf{I})$
2 $\bar{\mathbf{p}} = (\mathbf{p} - \bar{\mathbf{a}})$   // Subtract mean from probe pixels
3 $\breve{\mathbf{p}} = \mathbf{U}^T \bar{\mathbf{p}}$        // Project probe into subspace
4 For each encoded gallery face $\mathbf{Z}_{1:M',g}$ in $\mathbf{Z}$
5     $d_g = \sqrt{\lambda_1^{-1}\left(\breve{p}_1 - Z_{1,g}\right)^2 + \cdots + \lambda_{M'}^{-1}\left(\breve{p}_{M'} - Z_{M',g}\right)^2}$
6     Keep best match $d_g$ and corresponding $\mathbf{Z}_{1:M',g}$
```

Figure 53: Eigenface algorithm in equation form.

```
1 for k= 0 to M' -1
2    for j=0 to (N-1)
3          p_curl[k]=p_curl[k] + UT[k][j] * p_bar[j]
4 for g=0 to G-1
5    for k=0 to (M’-1)
6          d_sum[g]=d_sum + (Z[k][g] - p_curl[k]) * (Z[k][g] - p_curl[k]) *  (const[k])
```

Figure 54: Eigenface benchmark code.

Lines 4 through 6 in Figure 54 express the comparison process. Line 4 iterates variable g through the G gallery members, and G is one million for the 4 MP, 1M benchmark data set. Line 5 iterates through the M’ elements of the encoded probe vector and M’ is 600,000. Line 6 calculates and accumulates the Mahalinobis distance to generate a scalar difference value d_sum for each probe gallery pair. As line 6 shows, the distance value is stored in a vector d_sum which has G elements, representing one distance result for each probe-gallery pair.

As the initial mapping for the Eigenface benchmark showed, the on-chip local memory is not large enough to store the entire subspace matrix, U^T, or the entire gallery set, Z, on-chip at one time. As a result, these matrices must be segmented into pieces that can fit in on-chip memory. Figure 55 shows the sequential pseudo code revisions to perform the probe projection and comparison processes on S segments of U^T and Z. Note that the lines in the pseudo code are renumbered for clarity and do not directly match up with the line numbers in the prior figures.

The pseudo code in Figure 55 includes additional loops on lines 3 and 11 to iterate loop index s through the S segments of the data structures. Lines 1 through 8 contain the probe projection process and correspond to lines 1 through 3 in Figure 54. Line 1 calculates the number of columns of UT that will fit in memory at one time, R. The number of segments S is then the total number of rows in UT or M’ divided by the number of segments as shown on line 2. Line 3 is the newly added loop that iterates through the S segments. Line 4 calculates the lower index for the k iteration loop in line 6, k_min. The loop limits k_min and k_max will differ for each segment S, since each segment contains a different range of data from the UT. For example, if UT had five segments of 20 lines, the first segment would have index range 0 to 19, the indices for the second segment would range from 20 to 39, and so forth. The lower index k_min and the upper index k_max, calculated in line 7, represent the index range for the s th segment. The range of the "for" loop in 6 is expressed with k_min and k_max, but the code within the loop on lines 7 and 8 is identical to the code in Figure 54.

```
// UT has M' rows and N columns
// Segment Size = number of rows of UT that can be stored in local memory
1  R = (LocalMemSize-M'-N)/(N)
// S=Number of segments = total number of rows/rows in segment
2  S = M' / R
3  for s= 0 to (S-1)
4     k_min       = s *(R)
5     k_max      = (s+1)*(R) – 1
6     for k= k_min to k_max
7          for j=0 to (N-1)
8               p_curl[k] = p_curl[k] + UT[k][j] * p_bar[j];
// Z has M' rows and G columns, segment across columns
// C=Number of columns of Z that can be stored in local memory
9   C = (LocalMemSize-2M')/(M')
// S=Number of segments = number of C columns
10  S = G / C
11  for s= 0 to S
12       g_min  = s *(C)
13       g_max = (s+1)*(C) - 1
14   for g= g_min to g_max
15   for k=0 to (M'-1)
16       d_sum[g]=d_sum[g] + (Z[k][g] - p_curl[k]) * (Z[k][g] - p_curl[k]) * (const[k])
```

Figure 55: Sequential code with S segments.

The comparison process shown in lines 9 through 16 of Figure 55 corresponds to lines 4 through 6 in Figure 54. Line 11 iterates index variable s through the segments in the same manner as line 3 in the projection process. The segments of Z represent columns, and line 9 calculates segment size as C columns of Z, and line 10 calculates the number of segments, S. Line 11 iterate through the segments, and lines 12 and 13 calculate g_min and g_max, the index range for the current segment. Line 14 iterates k through this range, and the lines 15 and 16 are again identical to the sequential code in Figure 54.

Another loop with index variable p is added to the probe projection process as line 3 in Figure 56. This loop represents the forking of the process into P separate threads that could be executed on P PEs. Lines 1 and 2 again calculate the segment size and number and are identical to lines 1 and 2 in Figure 55. Line 3 shows the added loop that iterates P times. The number of iterations in the next loop is now the produce of S and P, so the loop index range must be calculated differently. Line 4 expresses the loop with index variable q, which iterates through the number of segments on one PE. Line 5 calculates the new value of index s from the PE number and segment index q. The remaining lines 6 through 10 are identical to the prior version.

Another loop is added to represent the forking of the comparison process as shown in Figure 56. Lines 11 and 12 are again the same as the corresponding lines in the prior figure, and line 13 indexes through the P PEs. Line 14 iterates through the segments on a particular PE, and line 15 calculates index s in terms of the processor number p and segment index q. As with the projection process, the remaining lines 16 through 20 are unchanged.

The last step in the parallelization process adds explicit data movement operations, represented as Localize and Globalize operations in Figure 57. The code is again renumbered for clarity, and the only changes between Figure 56 and Figure 57 are the addition of Localize, Globalize, and Initialize instructions. The Initialize operation in line 8 reserves memory for the variable p_curl_local. Line 9 Localizes the p_bar variable, transferring it from external memory to on-chip memory. As discussed previously, this benchmark is memory bound and memory banks were added to overcome this bottleneck. Each added memory bank interfaces to a different PE and each PE therefore has access to its own external memory bank.

```
// UT has M' rows and N columns
// Segment Size = number of rows of UT that can be stored in local memory
1  R = (LocalMemSize-M'-N)/(N)
// S=Number of segments = total number of rows/rows in segment
2  S = M' / R
3  for p=0 to P-1 PARDO
4    for q= 0 to (S/P – 1)
5          s = p*(S/P) + q
// s = 0 to S
6          k_min  = s *(R)
7          k_max = (s+1)*(R) – 1
8          for k= k_min to k_max
9               for j=0 to (N-1)
10                   p_curl[k] = p_curl[k] + UT[k][j] * p_bar[j];
// Z has M' rows and G columns, segment across columns
// C=Number of columns of Z that can be stored in local memory
11 C = (LocalMemSize-2M')/(M')
// S=Number of segments = number of C columns
12 S = G / C
13 for p=0 to (P-1)
14  for q= 0 to (S/P-1)
// s = 0 to S
15         s = p*(S/P) + q
16         g_min  = s *(C)
17         g_max = (s+1)*(C) – 1
18         for g=g_min to g_max
19              for k=0 to (M'-1)
20                   d_sum[g]=d_sum[g] + (Z[k][g] - p_curl[k]) * (Z[k][g] - p_curl[k]) * (const[kj])
```

Figure 56: Sequential pseudo code with P PEs and S segments.

This architectural configuration achieves higher performance by allowing memory accesses to be performed in parallel and also eliminates the communication time that would be required to move the data from PE0 if there were only one memory bank. After a segment of p_curl_local is calculated, line 14 in Figure 57 performs a Globalize operation to transfer the p_curl results back to the external system memory. In the same way, lines 22 through 25 initialize the result variable and localize the input variables for the comparison process, and line 29 globalizes the result back to external memory.

The computational analysis for the Eigenface segmented mapping is shown in Figure 58. The memory analysis in Chapter 6 showed that this benchmark is memory bound and required seven additional memory banks for the 2022 technology point. As discussed earlier, these memory banks will be interfaced to separate PEs, and the mapping will therefore only support one PE per memory bank and therefore seven PEs were selected for the initial analysis.

The computational analysis is Figure 58 shows that the benchmark with these changes achieves identification time of 2.60 minutes. While this is near real-time, the real-time goal has not been achieved. The analysis further shows that data movement still consumes 74.3% of the total time and remains a bottleneck, but the computation time is still significant at 25.7%.

The Eigenface segment mapping does not achieve real-time performance with seven memory banks as shown in Figure 58. However, the analysis shows that data movement requires 74.3% of the execution time and this mapping is still memory bound with seven memory banks. Therefore, increasing the number of memory banks should improve performance to achieve rea-time operation.

```
// UT has M’ rows and N columns
// Segment Size = number of rows of UT that can be stored in local memory
1  R = (LocalMemSize-M’-N)/(N)
// S=Number of segments = total number of rows/rows in segment
2  S = M’ / R
3  for p=0 to P-1 PARDO
4    for q= 0 to (S/P – 1)
5          s = p*(S/P) + q;
// s = 0 to S
6          k_min  = s *(R)
7          k_max = (s+1)*(R) – 1
8          Initialize(p_curl_local[0:M’/P-1])
9          p_bar_local [0:N-1]= Localize (p_bar[0:N-1]);
10         UT_local[k_min: k_max] [0:N-1]  = Localize (UT[k_min: k_max] [0:N-1]);
11         for k= k_min to k_max
12              for j=0 to (N-1)
13                   p_curl_local[k] = p_curl_local[k] + UT_local[k][j] * p_bar_local[j];
14         p_curl[k_min:k_max]=Globalize(p_curl_local[k_min:k_max]
// Z has M’ rows and G columns, segment across columns
// C=Number of columns of Z that can be stored in local memory

15 C = (LocalMemSize-2M’)/(M’)
// S=Number of segments = number of C columns
16  S = G / C
17  for p=0 to (P-1)
18         for q= 0 to (S/P-1)
// s = 0 to S
19              s = p*(S/P) + q
20              g_min  = s *(C)
21              g_max = (s+1)*(C) – 1
22              Initialize(d_sum_local[1:G/P])
23              p_curl_local [0:M’-1]= Localize (p_curl[0:M’-1]);
24              const [0:M’-1]= Localize (const[0:M’-1]);
25              Z_local[0:M’-1][g_min:g_max]=Localize(Z[0:M’-1][g_min:g_max]
26              for g=g_min to g_max
27                   for k=0 to (M’-1)
28                        d_sum[g]=d_sum[g]+ (Z[k][g] - p_curl[k]) * (Z[k][g] - p_curl[k]) * const[kj])
29              d_sum[g_min:g_max]=Globalize(d_sum_local[g_min:g_max])
```

Figure 57: Parallel code with explicit data movement.

	Computation Cycles Per line	Total Number of Bytes	Number of Bytes per PE	Number of PEs	2022 Computation Time (minutes)	2022 Memory Time (minutes)	2D Mesh Topology Communication		Total	Percentage
							Latency	Bandwidth		
R1=(LocalMemSize-M'-N)/N										
S=M'/R1										
for p=0 to P-1 PARDO										
for q = 0 to S/P-1										
s=p*(S/P)+q										
k_min =s*(R1)										
k_max = (s+1)*(R1)-1										
Initialize (p_curl_local[1:N]);										
p_bar_local [0:N-1]= Localize (p_bar_local [0:N-1]);		3.10E+07	3.10E+07	7		2.56E-06			0.00	<0.01%
UT_local[k_min: k_max][0:N-1]= Localize (UT[k_min: k_max][0:N-1]);		1.86E+13	9.60E+08	7		1.54E+00			1.54	59.0%
for k= k_min to k_max										
for j=0 to (N-1)										
p_curl_local[k] = p_curl_local[k] + U[k][j] * p_bar_local[j];	3.87E+12			7	4.34E-01				0.43	16.7%
p_curl[k_min:k_max]=Globalize(p_curl_local[k_min:k_max]		4.80E+06		7		3.97E-07			0.00	<0.01%
R2 = (LocalMemSize-2M')/(M')										
S = G/R2										
SegSize = R2										
for p=0 to P-1 PARDO										
for q= 0 to S/P-1										
s = p*(S/P) + q;										
g_min=s*R2										
g_max=(s+1)*R2-1										
Initialize(d_sum_local[1:G/P])										
p_curl_local [0:M'-1]= Localize (p_curl[0:M'-1])		4.80E+06	4.80E+06	7		3.97E-07			0.00	<0.01%
const [0:M'-1]= Localize (const[0:M'-1]);		4.80E+06	4.80E+06	7		3.97E-07			0.00	<0.01%
Z_local[1:M'][g_min:g_max]=Localize(Z[1:M'][g_min:g_max]		4.80E+12	1.01E+09	7		3.97E-01			0.40	15.2%
for g=g_min to g_max										
for j=0 to (M'-1)										
d_sum_local[g]=d_sum_local[g] +									0.24	
(Z_local[j][g] - p_curl_local[j])2 * (const[j])	2.10E+12			7	2.35E-01					9.0%
d_sum[g_min:g_max]=Globalize(d_sum_local[g_min:g_max])		8.00E+06		7		6.61E-07			0.00	<0.01%
Totals (minutes)					0.67	1.93			2.60	100.0%
Percentage Time					25.7%	74.3%			100.0%	

Figure 58: Eigenface benchmark segment mapping analysis.

Figure 58 shows that execution time is 2.6 minutes with seven memory banks and that an additional 1.3 times speedup is required. Given that the mapping is memory bound, increasing the number of memory banks should provide a proportional speedup. Scaling the seven memory banks by the 1.3 speedup required shows that 9.1 memory banks are required, and rounding up to an integer value shows that 10 memory banks should resolved this bottleneck.

The analysis of the segmented mapping with ten memory banks and ten PEs is shown in Figure 59. As expected, this mapping improves performance enough to achieve real-time identification of 1.83 minutes.

	Computation Cycles Per line	Total Number of Bytes	Number of Bytes per PE	Number of PEs	2022 Computation Time (minutes)	2022 Memory Time (minutes)	2D Mesh Topology Communication		Total	Percentage
							Latency	Bandwidth		
R1=(LocalMemSize-M'-N)/N										
S=M'/R1										
for p=0 to P-1 PARDO										
for q = 0 to S/P-1										
s=p*(S/P)+q										
k_min =s*(R1)										
k_max = (s+1)*(R1)-1										
Initialize (p_curl_local[1:N]);										
p_bar_local [0:N-1]= Localize (p_bar_local [0:N-1]);		3.10E+07	3.10E+07	10		1.79E-06			0.00	<0.01%
UT_local[k_min: k_max][0:N-1]= Localize (UT[k_min: k_max][0:N-1]);		1.86E+13	6.50E+08	10		1.08E+00			1.08	59.0%
for k= k_min to k_max										
for j=0 to (N-1)										
p_curl_local[k] = p_curl_local[k] + U[k][j] * p_bar_local[j];	3.87E+12			10	3.04E-01				0.30	16.7%
p_curl[k_min:k_max]=Globalize(p_curl_local[k_min:k_max]		4.80E+06		10		2.78E-07			0.00	<0.01%
R2 = (LocalMemSize-2M')/(M')										
S = G/R2										
SegSize = R2										
for p=0 to P-1 PARDO										
for q= 0 to S/P-1										
s = p*(S/P) + q;										
g_min=s*R2										
g_max=(s+1)*R2-1										
Initialize(d_sum_local[1:G/P])										
p_curl_local [0:M'-1]= Localize (p_curl[0:M'-1])		4.80E+06	4.80E+06	10		2.78E-07			0.00	<0.01%
const [0:M'-1]= Localize (const[0:M'-1]);		4.80E+06	4.80E+06	10		2.78E-07			0.00	<0.01%
Z_local[1:M'][g_min:g_max]=Localize(Z[1:M'][g_min:g_max]		4.80E+12	7.06E+08	10		2.78E-01			0.28	15.2%
for g=g_min to g_max										
for j=0 to (M'-1)										
d_sum_local[g]=d_sum_local[g] + (Z_local[j][g] - p_curl_local[j])2 * (const[j])	2.10E+12			10	1.65E-01				0.16	9.0%
d_sum[g_min:g_max]=Globalize(d_sum_local[g_min:g_max])		8.00E+06		10		4.63E-07			0.00	<0.01%
Totals (minutes)					0.47	1.35			1.82	100.0%
Percentage Time					25.7%	74.3%			100.0%	

Figure 59: Eigenface segment mapping analysis with ten memory banks and ten PEs.

7.4 EBGM MAPPING ANALYSIS

The computational analysis in Chapter 4 showed that a speedup of two times is required to make the EBGM 100 KP, 1K benchmark real-time, and a speedup of 748 times is required to make the EBGM 4 MP, 14K and 4 MP, 1M benchmarks real-time. Chapter 5 showed that adding two function units improved performance for the 100 KP, 1K benchmark to a real-time 1.14 minutes. The 4.92 times speedup achieved with the two function units improved performance for the 4 MP, 14K and 4 MP, 1M benchmarks to 304.83 minutes or less, a significant improvement but still not real-time, and showed that an additional 152.4 times speedup is required for real-time performance with these two data sets. The analysis in Chapter 6 showed that memory bandwidth does not limit performance for the EBGM benchmarks, and that computational performance is the primary performance limit for real-time identification with the EBGM benchmark. The goal in this section is to find a mapping that will provide the 152.4 times speedup required to make the two EBGM 4 MP benchmarks real-time. Given that the identification time for the 4 MP, 14K and 4 MP, 1M benchmarks differs by 0.92 minutes and that this difference would reduce to an insignificant 0.006 minute if the 152.4 times acceleration can be achieved, the following analysis will use the 4 MP, 1M data set.

The initial mapping for the EBGM benchmark is shown in Figure 60. This mapping distributes the 40 filters across the PEs to calculate one filter response on each PE. The filter masks are first localized and scattered in an initialization step that executes one time. An outer loop iterates through the 25 vertices to find the coordinates of each vertex. Within the vertex loop, the image is localized and compared to the bunch graph to select the model jet, B_kept. The image is broadcast to the PEs and convolved with the filter masks in parallel on P PEs with

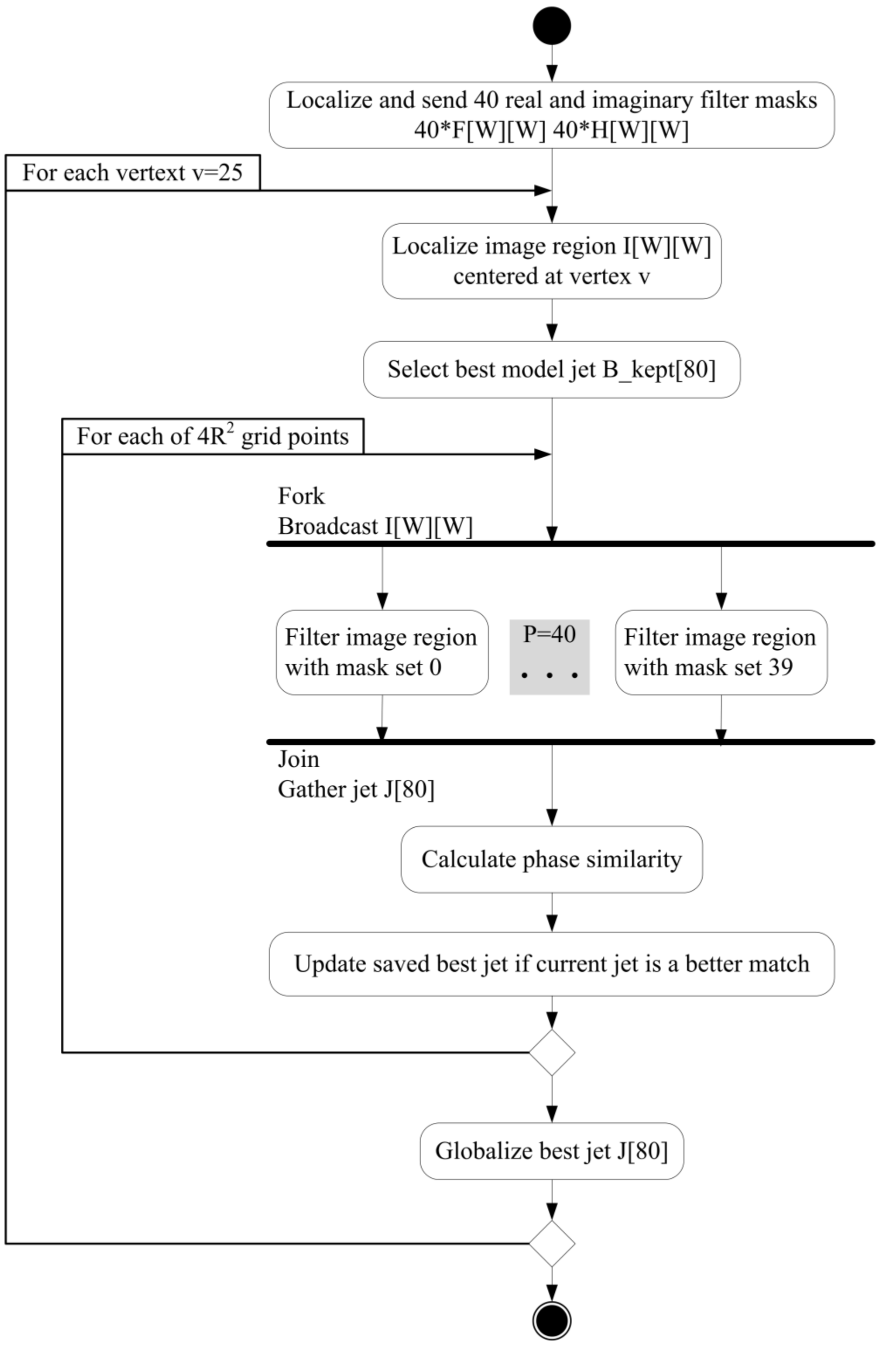
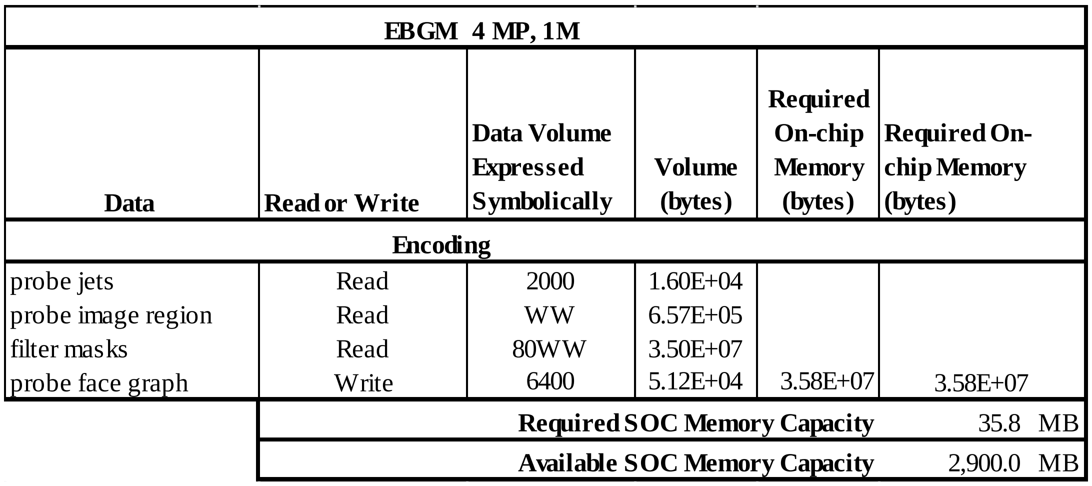

Localize and send 40 real and imaginary filter masks
40*F[W][W] 40*H[W][W]
For each vertext v=25
Localize image region I[W][W] centered at vertex v
Select best model jet B_kept[80]
For each of 4R² grid points
Fork
Broadcast I[W][W]
Filter image region with mask set 0
P=40
. . .
Filter image region with mask set 39
Join
Gather jet J[80]
Calculate phase similarity
Update saved best jet if current jet is a better match
Globalize best jet J[80]

Figure 60: EBGM filter mask mapping.

each PE calculating one jet element. The resulting jet is gathered on PE0 and compared to the model jet, and the best jet and coordinates are updated if the phase similarity with the newly calculated jet is better. This process is repeated for all points in the search region and then the best jet found is retained on PE0.

Given this mapping, the next step is to determine the required Local Memory capacity. Figure 61 shows the memory capacity analysis for the EBGM 4 MP, 1 M benchmark. This analysis shows that 35.8 MB of on-chip memory is required and 2.9 GB is available on-chip in 2022 if 2,000 PEs are used. Therefore on-chip memory capacity is not a constraint for this mapping.



EBGM 4 MP, 1M					
Data	**Read or Write**	**Data Volume Expressed Symbolically**	**Volume (bytes)**	**Required On-chip Memory (bytes)**	**Required On-chip Memory (bytes)**
Encoding					
probe jets	Read	2000	1.60E+04		
probe image region	Read	WW	6.57E+05		
filter masks	Read	80WW	3.50E+07		
probe face graph	Write	6400	5.12E+04	3.58E+07	3.58E+07
	Required SOC Memory Capacity				35.8 MB
	Available SOC Memory Capacity				2,900.0 MB

Figure 61: EBGM 4 MP, 1M Memory Capacity Requirements.

As with the Eigenface benchmark, the EBGM benchmark code can incrementally revised to incorporate the parallelization and data movement operations. The top level pseudo code for the EBGM benchmark, originally described in Chapter 4, is included for reference as Figure 62. The detailed pseudo code for the EBGM benchmark is shown in Figure 63. As in the previous examples, the pseudo code is renumbered for clarity.

```
$S_{BEST} = \text{EBGMid}(\mathbf{I}, \mathbf{B}, (x_e, y_e))$

    $\mathbf{LocateFeatures}(\mathbf{I}, \mathbf{B}, x_{LeftEye}, y_{LeftEye})$
    $\mathbf{MakeFaceGraph}(\mathbf{I}, \{\mathbf{F}\}, \{\mathbf{H}\}, \mathbf{x}, \mathbf{y})$
    $\mathbf{Comparison}(\mathbf{J_P}, \{\mathbf{J_G}\})$
```

Figure 62: EBGM benchmark top level pseudo code.

The pseudo code in Figure 63 describes the bottleneck process within the EBGM algorithm but includes additional code to place the bottleneck in context. Line 1 iterates index variable v through the 25 feature vertices of the face image. Line 2 initializes variable s_kept, the similarity for the best model jet found so far. Line 3 iterates bunch jet index b through the M bunch graph jets associated with vertex v, and for the 4 MP, 1M data set M=600,000. Line 4 compares the b^{th} bunch graph jet with J[0:79], a jet previously calculated at the probe image feature location, to calculate the similarity between the bunch jet and the probe jet. If the similarity between the jets is greater than s_kept, s_kept as well as model jet B_kept is updated in line 5.

When the loop in lines 2 through 5 in Figure 63 completes, B_kept[0:79] contains the model jet, the jet from the bunch graph that best matches the probe jet, J[0:79], calculated at the estimated feature coordinates. Lines 6 through 17 then search an R by R region of the probe image in half-pixel steps to find the jet and coordinates in the probe image that best match the model jet B_kept[0:79]. Lines 6 and 7 iterate the coordinate pair (dx, dy) through the R by R pixel search region in half-pixel steps, where R is 234 for the 4 MP, 1M data set, so dx and dy each iterate 468 times. Line 8 iterates through the K=40 Gabor wavelet filters. Lines 9 and 10 iterate through the rows r and columns c of the W by W filter masks and probe image region, and W=234 for the 4 MP, 1M data set. Line 11 multiplies and accumulates the real filter mask response at the integer coordinate pair, (int(dx), int(dy)), and stores the result in an even-numbered index of vector filt_part[], thus convolving the real filter mask and image region. Line 12 repeats the convolution for the imaginary filter mask and stores the results in the odd-numbered indexes of filt_part[].

```
1 for v=1 to 25
2    s_kept=0;
3    for b=1 to M         // 600k
4        s=CalcMagnitudeSimilarity(J[0:79], B[b] [0:79])
5        if s>s_kept then s=s_kept, B_kept[0:79]=B[b] [0:79]
6    for dx=0 to (R-1) in 0.5 steps    //468
7        for dy= 0 to  (R-1) in 0.5 steps    //468
8            for k=0 to K-1  //40
9                for r=1 to W
10                   for c=1 to W
11                       filt_part[2k]      =filt_part[2k]     +Image[r][c]*Mask[2*k][r][c]
12                       filt_part[2k+1] =filt_part[2k+1] +Image[r][c]*Mask[2*k+1][r][c]
13               J[2k]=sqrt(filt_part[2k]$^2$+filt_part[2k+1]$^2$)*cos(atan(filt_part[2k+1] /
                         filt_part[2k]) + fx[k] * dx+fy[k] * dy)
14               J[2k+1]=sqrt(filt_part[2k]$^2$ + filt_part[2k+1]$^2$)*sin(atan(filt_part[2k+1] /
                         filt_part[2k]) +fx[k] * dx+fy[k] * dy)
15           s=CalcPhaseSimilarity(J[0:79], B_kept[0:79])
16           keep coordinates (dx, dy) and jet J[0:79] for the best match
```

Figure 63: EBGM simplified pseudo code.

Lines 13 and 14 in Figure 63 adjust the filter response calculated at the integer coordinates to estimate the filter response at the real coordinates, (dx, dy). Line 13 converts the real and imaginary values to polar form, offsets the phase response by scaling constants fx and fy with dx and dy respectively, then converts the response back to real form. Line 14 similarly adjusts the imaginary coefficient of the complex filter response.

Line 15 in Figure 63 calculates the phase similarity between the model jet, B_kept[0:79], and the jet calculated at the current coordinates (dx, dy). Line 16 retains the probe jet that best matched the model jet in the R by R image region along with the coordinates.

The EBGM pseudo code is further simplified in Figure 64. This pseudo code is functionally equivalent to the code in Figure 63, but variable names are changed for clarity and readability. The filt_part variable name is changed to fp to shorten the lines and avoid word wrap, and the names of constant variables fx[ ] and fy[ ] are changed to cx[ ] and cy[ ] to better reflect that these variables are constants calculated during training.

```
1  for v=1 to 25
2    s_kept=0;
3    for b=1 to M        // 600k
4        s=CalcMagnitudeSimilarity(J[0:79], B[b] [0:79])
5        if s>s_kept then s=s_kept, B_kept[0:79]=B[b] [0:79]
6    for dx=0 to (R-1) in 0.5 steps    //468
7        for dy= 0 to  (R-1) in 0.5 steps    //468
8            for k=0 to K-1  //40
9                for r=1 to W
10                   for c=1 to W
11                       fp[2k]   =fp[2k]  +Image[r][c]*Mask[2*k][r][c]
12                       fp[2k+1] =fp[2k+1] +Image[r][c]*Mask[2*k+1][r][c]
13               J[2k]=sqrt($fp[2k]^2$+ $fp[2k+1]^2$)*cos(atan(fp[2k+1] / fp[2k]) +cx[k]*dx+cy[k]*dy)
14               J[2k+1]=sqrt($fp[2k]^2$+$fp[2k+1]^2$)*sin(atan(fp[2k+1]/ fp[2k])+cx[k]*dx+cy[k]*dy)
15           s=CalcPhaseSimilarity(J[0:79], B_kept[0:79])
16           keep coordinates (dx, dy) and jet J[0:79] for the best match
```

Figure 64: EBGM further simplified pseudo code.

A function is then defined to represent the calculation of the response for one Gabor filter as shown in Figure 65. The function CalcJetPart contains the code from lines 9 through 14 in Figure 64 and returns the response to the real filter mask, J[2k], and the response to the imaginary filter mask, J[2k+1].

```
[J[2k], J[2k+1] ] =  CalcJetPart (image, dx, dy, k) =apply one filter to image at dx, dy
  1   for r=1 to W
  2       for c=1 to W
  3           fp[2k]     =fp[2k]     +Image[r][c]*Mask[2*k][r][c]
  4           fp[2k+1] =fp[2k+1] +Image[r][c]*Mask[2*k+1][r][c]
  5   J[2k]=sqrt($fp[2k]^2 + fp[2k+1]^2$)*cos(atan(fp[2k+1] / fp[2k]) +  cx[k] * dx+cy[k] * dy)
  6   J[2k+1]=sqrt($fp[2k]^2 + fp[2k+1]^2$)*sin(atan(fp[2k+1] / fp[2k]) +  cx[k] * dx+cy[k] * dy)
```

Figure 65: Calcjet function definition.

The EBGM pseudo code is shown with the CalcJetPart function call in Figure 66. Lines 9 through 14 in Figure 64 are replaced with the function call and the lines are then renumbered as shown in Figure 66.

```
1 for v=1 to 25
2    s_kept=0;
3    for b=1 to M         // 600k
4        s=CalcMagnitudeSimilarity(J[0:79], B[b] [0:79])
5        if s>s_kept then s=s_kept, B_kept[0:79]=B[b] [0:79]
6    for dx=0 to  (R-1) in 0.5 steps    //463
7        for dy= 0 to  (R-1) in 0.5 steps    //468
8            for k=0 to K-1  //40
9                [J[2k], J[2k+1] ]= CalcJetPart ( image, dx, dy, k)
10           s=CalcPhaseSimilarity(J[0:79], B_kept[0:79])
11           keep coordinates (dx, dy) and jet J[0:79] for the best match
```

Figure 66: CalcJetPart function integrated into benchmark pseudo code.

The computational analysis in Chapter 4 showed that the convolution required to filter the image region is the primary bottleneck in the EBGM algorithm, and the CalcJetPart function encapsulates the filtering process. The K iteration loop can be forked into 40 threads to execute on 40 different PEs and would be expected to provide speedup in the range of 40 times. Figure

67 shows the pseudo code with the loop variable on line 8 changed from k to p to reflect the change to P PEs and the addition of the PARDO notation to indicate parallel execution.

```
1 for v=1 to 25
2   s_kept=0;
3   for b=1 to M        // 600k
4       s=CalcMagnitudeSimilarity(J[0:79], B[b] [0:79])
5       if s>s_kept then s=s_kept, B_kept[0:79]=B[b] [0:79]
6   for dx=0 to  (R-1) in 0.5 steps   //463
7       for dy= 0 to  (R-1) in 0.5 steps   //468
8           for p=0 to P-1  PARDO
9               [J[2p], J[2p+1] ]= CalcJetPart ( image, dx, dy, p)
10          s=CalcPhaseSimilarity(J[0:79], B_kept[0:79])
11          keep coordinates (dx, dy) and jet J[0:79] for the best match
```

Figure 67: EBGM benchmark pseudo code with filter loop parallelization.

The pseudo code shown in Figure 68 adds the explicit data movement and communication operations. Line 6 causes lines 7 and 8 to execute only on PE0. Line 7 transfers the image from external memory into the local memory of PE0, and line 8 broadcasts the image to all of the PEs. Lines 11 and 12 execute the filtering process in parallel on P PEs. The filter masks are constants for each PE and are assumed to already reside in Local Memory on each PE. Line 13 gathers the two probe jet elements from each PE to build the jet J[0:79]. Line 14 limits execution of lines 15 and 16 to PE0. Line 15 calculates the similarity and line 16 retains the jet and coordinates of the best matching jet.

The analysis for the EBGM filter mask mapping is shown for the 4 MP, 1M data set in Figure 69. The analysis shows identification time of 9.0 minutes is achieved with this mapping and the ISA speedup with the addition of two function units for the fused instructions. The analysis shows that nearly 100% of the execution time is spent on computation and this mapping is therefore computationally bound. Performance should improve if more PEs can be used.

```
for v=1 to 25
  s_kept=0;
  for b=1 to M        // 600k
      s=CalcMagnitudeSimilarity(J[0:79], B[b] [0:79])
      if s>s_kept then s=s_kept, B_kept[0:79]=B[b] [0:79]
  if PE0
      image_local[0:233][0:233]=Localize(image[0:233][0:233])
      BCAST(image_local[0:233][0:233])
  for dx=0 to  (R-1) in 0.5 steps    //463
      for dy= 0 to  (R-1) in 0.5 steps    //468
          for p=0 to P-1  PARDO
              [J_local[2p], J_local[2p+1] ]= CalcJetPart ( image, dx, dy, p)
          J[2p:2p+1]=GATHER(J_local[2p:2p+1])
      if PE0
          s=CalcPhaseSimilarity(J[0:79], B_kept[0:79])
          keep coordinates (dx, dy) and jet J_group[p][0:79] for the best match
```

Figure 68: EBGM benchmark pseudo code with explicit data movement operations.

One way to use more PEs is to convolve each filter mask on a separate PE. The EBGM algorithm uses 40 filters and each filter is represented by a real and an imaginary mask. The 40 PE mapping convolved both masks for one filter on one PE, but the convolution could be performed separately on two PEs. If each mask convolution were performed on a separate PE, 80 PEs could be used. Given that essentially all of the execution time is computation, as speedup of approximately two times would be achieved with this mapping. However, given that the execution time for the 40 PE mapping was nine minutes, an additional two times speedup would reduce the execution time to 4.5 minutes but would still not achieve real-time performance. A mapping that provides better speedup is required.

EBGM 4MP, 1M										
		Computation Cycles Per line	Number of Bytes	Number of PEs	Computation Time (minutes)	One bank 2022 Memory Time (minutes)	2D Mesh Topology Communication 2022		Total	Percentage
							Latency	Bandwidth		
	Localize({F})		1.75E+07	1		0.00001			1.01E-05	<0.01%
	Localize({H})		1.75E+07	1		0.00001			1.01E-05	<0.01%
	SCATTER({H})		4.38E+05	40			2.4E-09	6.8E-06	6.81E-06	<0.01%
	SCATTER({F})		4.38E+05	40			2.4E-09	6.8E-06	6.81E-06	<0.01%
1	for each feature vertex v, v=1 to 25									
2	s_kept=0									
3	for each jet b in the bunch, b=1 to M									
4	s=CalcMagnitudeSimilarity(j, b)	1.85E+11		1	0.9354				0.94	10.41%
7	Localize(I[])		3.10E+07	1		0.00002			1.79E-05	<0.01%
8	BCAST(I[])		4.38E+05	40			2.4E-09	6.81E-06	6.81E-06	<0.01%
9,10	for each half-point (dx, dy) in an R by R region									
11	for p=0 to P-1 PARDO			40						
12	for each row r			40						
12	for each column c			40						
12	accumulate real part	3.00E+13		40	3.7856				3.79	42.13%
12	accumulate imaginary part	3.00E+13		40	3.7856				3.79	42.13%
12	adjust real part	5.79E+10		40	0.0073				7.31E-03	0.08%
12	adjust imaginary part	5.79E+10		40	0.0073				7.31E-03	0.08%
13	GATHER(j)		1.60E+04	40			1.38E-08	2.49E-07	2.62E-07	<0.01%
15	s=CalcPhaseSimilarity(j, b)	9.18E+10		1	0.4635				0.46	5.16%
16	keep coordinates (x, y) of the best match			1						
				Totals (minutes)	9.0	3.82E-05	2.08E-08	2.07E-05	9.0	100.000%
				Percentage Time	100.0%	<0.01%	<0.01%	<0.01%	100.0%	

Figure 69: EBGM filter mapping analysis.

More PEs can be used to further reduce computation time. Since distributing the 80 filter mask convolutions across 80 PEs does not provide enough speedup, the next loop level must be parallelized to allow more PEs to be used. The mapping shown in Figure 70 separates each column of dx, dy coordinate pairs in the search region into 24 coordinate segments. Each coordinate pair within each segment is then distributed to a different set of 80 PEs for filtering. This mapping therefore uses 80*24 PEs, and the 1,920 PEs will use nearly all the PEs on the two thousand PE 2022 technology point SOC.

The computational analysis for two distributed loop mapping is shown in Figure 71. This analysis shows that identification time with this mapping is a real-time 1.6 minutes. The analysis further shows that the benchmark remains computationally bound. Given that the search region loop is not fully parallelized in this mapping, further speedup could be achieved with more PEs, suggesting real-time identification could be achieved for larger gallery sets if SOCs with more than 2,000 PEs become available.

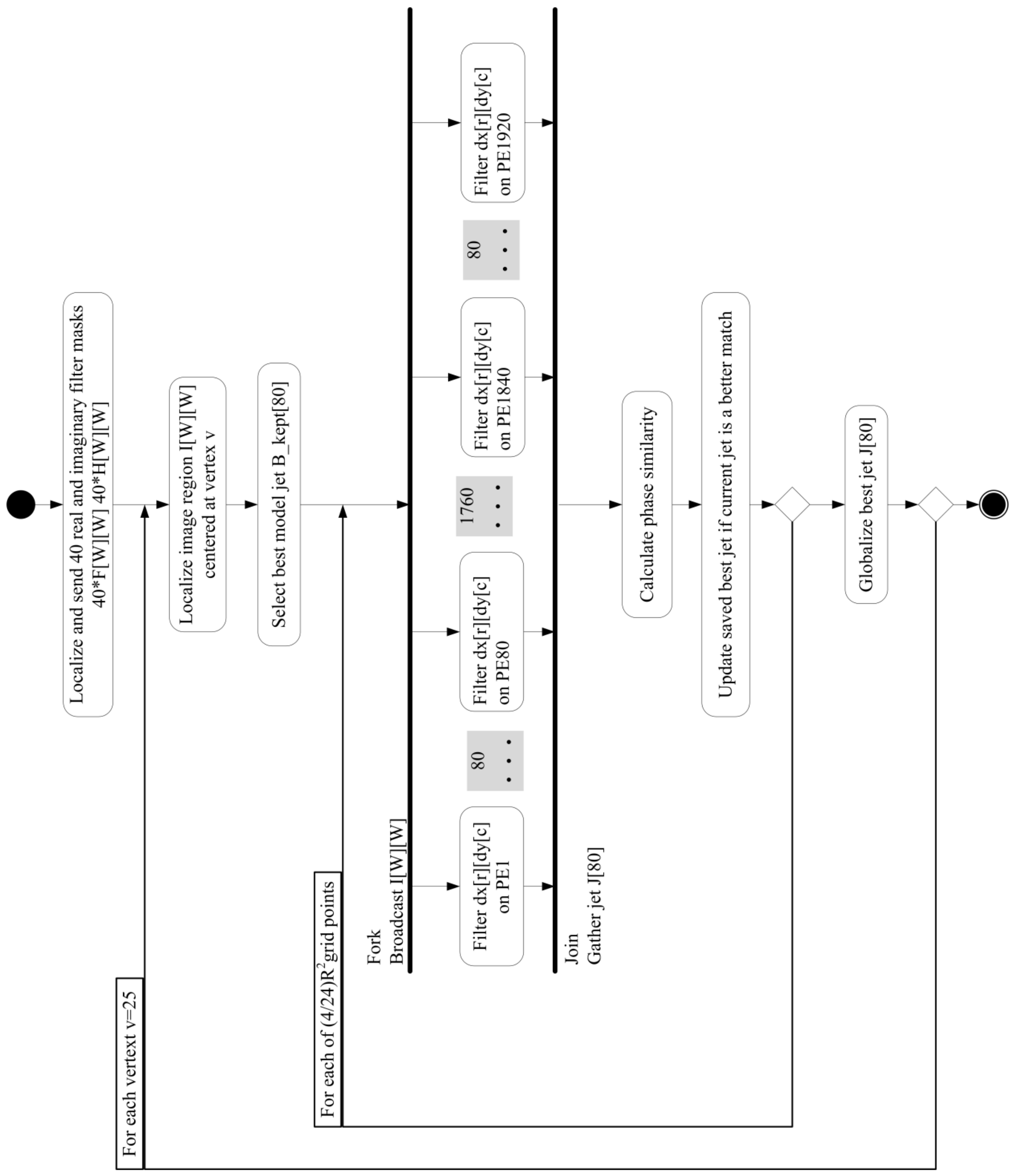

Localize and send 40 real and imaginary filter masks 40*F[W][W] 40*H[W][W]
Localize image region I[W][W] centered at vertex v
Select best model jet B_kept[80]
Fork
Broadcast I[W][W]
Filter dx[r][dy[c] on PE1
80
Filter dx[r][dy[c] on PE80
1760
Filter dx[r][dy[c] on PE1840
80
Filter dx[r][dy[c] on PE1920
Join
Gather jet J[80]
Calculate phase similarity
Update saved best jet if current jet is a better match
Globalize best jet J[80]
For each of (4/24)R^{2}grid points
For each vertext v=25

Figure 70: EBGM mapping with two distributed loops.

EBGM 4MP, 1M							2D Mesh Topology Communication 2022			
		Computation Cycles Per line	Number of Bytes	Number of PEs	Computation Time (minutes)	One bank 2022 Memory Time (minutes)	Latency	Bandwidth	Total	Percentage
	Localize({F})		1.75E+07	1		0.00001			1.01E-05	<0.01%
	Localize({H})		1.75E+07	1		0.00001			1.01E-05	<0.01%
	SCATTER({H})		4.38E+05	960			3.8E-09	7.0E-06	6.98E-06	<0.01%
	SCATTER({F})		4.38E+05	960			3.8E-09	7.0E-06	6.98E-06	<0.01%
1	for each feature vertex v, v=1 to 25									
2	s_kept=0									
3	for each jet b in the bunch, b=1 to M									
4	s=CalcMagnitudeSimilarity(j, b)	1.85E+11		1	0.9354				0.94	60.08%
7	Localize(I[])		3.10E+07	1		0.00002			1.79E-05	<0.01%
8	BCAST([I])		4.38E+05	1920			4.1E-09	6.98E-06	6.98E-06	<0.01%
9,10	for r=0 to (2/24)*R, for c=0 to 2*R									
11	for p=0 to P-1 PARDO			1920						
12	accumulate real part	3.00E+13		1920	0.0789				0.08	5.07%
12	accumulated imaginary part	3.00E+13		1920	0.0789				0.08	5.07%
12	adjust real part	5.79E+10		1920	0.0002				1.52E-04	0.01%
12	adjust imaginary part	5.79E+10		1920	0.0002				1.52E-04	0.01%
13	GATHER(j)		1.60E+04	1920			6.78E-07	2.55E-07	9.32E-07	<0.01%
15	s=CalcPhaseSimilarity(j, b)	9.18E+10		1	0.4635				0.46	29.77%
16	keep coordinates (x, y) of the best match			1						
				Totals (minutes)	1.6	3.82E-05	6.89E-07	2.12E-05	1.6	100.000%
				Percentage Time	100.0%	<0.01%	<0.01%	<0.01%	100.0%	

Figure 71: EBGM analysis for the two distributed loops mapping.

7.5 CONCLUSION

The mapping analysis shows that real-time performance can be achieved for all three face identification algorithms with all three data sets as shown in Table 17. The Eigenface and Bayesian 100 KP, 1K and 4 MP, 14K benchmarks are real-time on a sequential processor and no acceleration is required. The Eigenface and Bayesian 4 MP, 1M benchmarks can be made real-time with an SOC based on the 2022 technology point. Ten banks of external memory are required and each bank is interfaced to one PE. This multiple memory bank architecture avoids creating communication bottlenecks and maximizes performance for the memory interface.

Table 17: Requirements for Real-Time Face Identification.

BENCHMARK	ISA SPEEDUP	MEMORY BANKS	MAPPING	EXECUTION TIME (MINUTES)
Eigenface 100 KP, 1K	-	1	Sequential	0.0015
Eigenface 4 MP, 14K	-	1	Sequential	0.68
Eigenface 4 MP, 1M	2.51	10	Parallel Memory	1.82
Bayesian 100 KP, 1K	-	1	Sequential	0.0015
Bayesian 4 MP, 14K	-	1	Sequential	0.68
Bayesian 4 MP, 1M	2.21	10	Parallel memory	1.82
EBGM 100 KP, 1K	3.24	1	Sequential	1.14
EBGM 4 MP, 14K	4.92	1	Two distributed loops	1.56
EBGM 4 MP, 1M	4.91	1	Two distributed loops	1.56

All of the EBGM benchmarks require acceleration to achieve real-time performance. The EBGM 100 KP, 1K benchmark can be made real-time by adding two function units to implement a fused index instruction, but the 4 MP, 14K and 4 MP, 1M benchmarks require a parallel architecture to achieve real-time performances. A parallel architecture based on the

2022 technology point can provided the 1,920 PEs needed to distribute two algorithm loops across these PEs to accelerate computation to a real-time 1.56 minutes.

The analysis of on-chip communication shows that Broadcast, Scatter, Gather, and Reduce operations are used in the face identification benchmarks. The analysis further shows that on-chip communication over the 2D mesh architecture is not needed for the Eigenface or Bayesian benchmarks. Communication consumes 0.1% or less of execution time for all of the EBGM data sets and is therefore not a constraint.

8.0 CONCLUSIONS AND FUTURE WORK

This research analyzed performance for top accuracy face identification algorithms to determine how real-time performance can be achieved using a parallel System-on-a-Chip in 2019 or 2022 for a one million member gallery set of four megapixel images.

Three top accuracy face identification algorithms were selected for study. The published results from objective NIST FERET tests [8] were reviewed to select the top accuracy face identification algorithms. The algorithms selected for study were the Eigenface algorithm, the Bayesian algorithm, and the Elastic Bunch Graph Matching algorithm.

Sequential computational performance for each algorithm was first analyzed to expose bottleneck processes that constrain execution time and these bottlenecks were extracted to form benchmarks. Three data sets were developed, including a 1,200 member gallery set of 100 kilopixel images developed from the NIST FERET test set [8], a 14,365 member gallery set of four megapixel images developed from the Sandia test set used in the NIST MBE2010 evaluation [10] and a one million member gallery set of four megapixel images developed to represent the Terrorist Watch List [2].

Three critical architectural components that can significantly impact performance were analyzed separately, and then evaluated together with for each of the three algorithms. The components analyzed include:

<u>Computational Performance.</u> The computation required for each benchmark and each data set were analyzed to establish baseline performance. This analysis showed that the Eigenface and Bayesian 100 KP, 1K and 4 MP, 14K benchmarks achieved real-time performance on sequential architectures and required no speedup. Speedup of 38 times is required to make the Eigenface and Bayesian 4 MP, 1M benchmarks real-time. The EBGM 100 KP, 1K benchmark requires a two time speedup for real-time performance, and the EBGM 4 MP, 14K and 4 MP, 1M benchmarks require a 748 times speedup for real-time performance.

<u>Processor Instruction Set Architecture (ISA).</u> The ISA for the processors on the SOC was analyzed to determine whether instructions could be accelerated or fused to improve performance. An improved ISA was developed that included fused instructions to accelerate the Eigenface, Bayesian, and EBGM benchmarks.. The ISA changes provided a speedup of 2.21 to 2.51 times for the Eigenface and Bayesian benchmarks, but an additional 15 times speedup is required for real-time performance. The 3.24 times speedup was sufficient to make the EBGM 100 KP, 1K benchmark real-time, but an additional speedup of 152 time is required for real-time performance for the 4 MP, 14K and 4 MP, 1M benchmarks.

<u>External memory bandwidth.</u> External memory bandwidth was analyzed to determine whether this bandwidth is a constraint on real-time performance. Additional memory banks can mitigate this constraint, provided the number of banks required is within practical limits. The analysis showed that nine memory banks are required for real-time identification for the sequential Eigenface and Bayesian 4 MP, 1M benchmarks in 2019, but the required number of memory banks drops to 7 by 2022. However, none of the EBGM benchmarks are constrained by memory bandwidth.

Algorithm mapping. How an algorithm is mapped to a parallel SOC architecture significantly impacts performance, and real-time mappings were developed for all the benchmarks as shown in Table 18. Two mappings were evaluated for the Eigenface and EBGM benchmarks and one mapping for each benchmark was shown to be higher performance. For the Eigenface and Bayesian 4 MP, 1M benchmarks, a mapping that segments the subspace matrix by rows and scatters them across the PEs was needed to fit within the capacity of on-chip memory. The mapping analysis showed that these benchmarks are memory bound, and an architecture with multiple external memory banks was required to overcome this bottleneck. Real-time performance was achieved with 10 PEs, but 10 external memory banks and an architecture that connected the 10 PEs directly to the 10 external memory banks was required.

For the EBGM 4 MP, 14K and 4 MP, 1M algorithms, a mapping that scattered the filter masks across sets of 80 PEs and also filtered multiple search points in parallel used 1,920 PEs and achieved real-time performance. This mapping remains computationally bound, suggesting that higher resolution images could be identified in real-time if SOCs with more PEs become available.

Communication between PEs is not required for the Eigenface and Bayesian benchmarks, and consumes less than 0.01% of execution time for the EBGM mappings.

The Broadcast, Scatter, Gather, and Reduce messages are the only messages used in these mappings of face identification algorithms.

Table 18: Real Time Benchmark Mappings

Benchmark	ISA Speedup	Memory Banks	Processors	Mapping	Sequential Execution Time (minutes)	Parallel Execution Time (minutes)
Eigenface 100 KP, 1K	-	1	1	Sequential	0.0015	0.0015
Eigenface 4 MP, 14K	-	1	1	Sequential	0.6800	0.68
Eigenface 4 MP, 1M	2.51	10	10	Parallel Memory	75.7150	1.82
Bayesian 100 KP, 1K	-	1	1	Sequential	0.00146	0.0015
Bayesian 4 MP, 14K	-	1	1	Sequential	0.6791	0.68
Bayesian 4 MP, 1M	2.21	10	10	Parallel memory	71.169	1.82
EBGM 100 KP, 1K	3.24	1	1	Sequential	3.70	1.14
EBGM 4 MP, 14K	4.92	1	1920	Two distributed loops	1,495.16	1.56
EBGM 4 MP, 1M	4.91	1	1920	Two distributed loops	1,496.16	1.56

8.1 CONTRIBUTIONS

The research described in this dissertation makes the following contributions:

1. **Analysis methods to estimate performance for sequential and parallel mappings of face identification algorithms.** This research developed methods to estimate performance for computation, data movement, and communication on a 2D mesh SOC with multiple processors.

2. **Computational analysis of top accuracy face identification algorithms and a set of benchmarks.** Top accuracy face identification algorithms were selected and analysis methods were used to estimate performance for these algorithms and this analysis exposed the bottleneck processes in these algorithms.

3. **Determination that probe encoding rather than comparison is the primary bottleneck for face identification algorithms.** The analysis shows that the probe encoding process is the primary computational bottleneck for the three top face identification algorithms. Within the face identification community, the comparison process is generally considered to be the bottleneck. As a result, most of the prior research to accelerate face identification algorithms has explored various ways to accelerate the obvious parallelism of the pair comparison process. However, this analysis shows that even if the execution time of the comparison process could be reduced to zero, the algorithms would not achieve real-time performance due to the time required to encode the probe. This analysis further shows that while the encoding process does contain parallelism, the parallelism is not obvious and requires considerable effort to expose and exploit.

4. **Determination that communication is not a primary bottleneck for face identification algorithms.** The analysis shows that communication is not a constraint for the EBGM algorithm and that the Eigenface and Bayesian algorithms do not require communication for real-time performance. The EBGM benchmark mappings use c communication to transfer data segments from processors with direct memory access to

processors that do not have direct access to external memory, and no communication is required between processors while executing the computationally intensive code segments. As a result, face identification algorithms would map well to architectures that provide increased computational performance but restrict communication during execution such as the Graphics Processor Unit (GPU) and stream processors.

5. **Determination that on-chip memory capacity is a significant limitation for the Eigenface and Bayesian face identification algorithms.** The analysis shows that the volume of data required for large gallery sets is huge and will not fit within on-chip memory in the SOC models, thus requiring segmentation of the data and multiple memory banks. However, algorithm parameters such as the number of retained Eigenvectors have a significant impact on the size of the data set, and changing the parameter values could help mitigate this issue. The parameter values have been correlated to accuracy for current data sets, but it is unclear whether the parameters can be changed for larger gallery sets without negatively impacting accuracy, and this remains a topic for further study.

6. **Real-time mappings for top accuracy face identification algorithms.** Mappings that achieve real-time performance were developed for each of the top accuracy face identification algorithms for data sets that are not real-time on sequential processors. These mappings show a set of architectural requirements for face identification algorithms to achieve real-time performance on future parallel SOC architectures.

8.2 FUTURE WORK

8.2.1 Algorithm Research

The large data storage requirement for the Eigenface and Bayesian algorithms tend to make large gallery sets impractical. Published research suggests that 60% of the Eigenvectors must be retained to achieve the best identification accuracy, but the data sets studied were relatively small and were not in the range of one million members. Further research to determine whether accuracy can be maintained for a smaller percentage of retained Eigenvectors when the gallery set is very large could provide a way to reduce the data storage requirements without sacrificing accuracy.

Reduction of the filter mask and feature search region dimensions for the EBGM algorithm may be possible and could significantly improve performance, but the impact on identification accuracy is undetermined. Further research to determine whether these dimensions can be reduced without negatively impacting identification accuracy could improve performance.

8.2.2 Develop Real-Time Mappings for GPU Architectures

The analysis results show that the face identification algorithms do not require communication during the execution of computationally intensive processes and therefore should map well to GPU architectures. A high performance mapping to a GPU architecture has the potential to achieve real-time performance, but these mappings need to be developed. In addition, mapping these algorithms to a GPU may expose other performance constraints, and further research is needed to explore these issues.

8.2.3 Develop Real-Time Mappings for 3D Face Identification

Still image face identification remains an important and active research area, but interest in 3D facial models is increasing. Some 3D face identification algorithms are extensions of the 2D algorithms while others use completely. The techniques developed in this research could be extended to analyze 3D face identification algorithms to enable real-time 3D face identification.

BIBLIOGRAPHY

[1] H. E. Reser, "Airline Terrorism: The Effect of Tightened Security on the Right to Travel," *Journal of Air Law and Commerce,* vol. 63, 1997.

[2] U. S. Government, "Terrorist Watch List Screening," United States Government Accountability Office, Washington, DC GAO-08-110, 2007.

[3] N. Ramanathan*, et al.*, "Facial similarity across age, disguise, illumination and pose," in *International Conference on Image Processing*, 2004, pp. 1999-2002 Vol. 3.

[4] X. Tan*, et al.*, "Face Recognition Under Occlusions and Variant Expressions With Partial Similarity," *Information Forensics and Security, IEEE Transactions on,* vol. 4, pp. 217-230, 2009.

[5] A. J. O'Toole*, et al.*, "Face Recognition Algorithms Surpass Humans Matching Faces Over Changes in Illumination," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* vol. 29, pp. 1642-1646, 2007.

[6] P. J. Phillips*, et al.*, "FRVT 2006 and ICE 2006 Large-Scale Results," NIST NISTIR 7408, 2007.

[7] Transportation Security Administration. (May 23, 2009). *TSA Travel Assistant.* [Online]. Available: http://www.tsa.gov/travelers/airtravel/screening/index.shtm

[8] P. J. Phillips*, et al.*, "FERET (Face Recognition Technology) Recognition Algorithm Development and Test Results," Army Research Laboratory, Adelphi, MD ARL-TR-995, 1996.

[9] D. S. Bolme*, et al.*, "FacePerf: Benchmarks for Face Recognition Algorithms," in *Workload Characterization, 2007. IISWC 2007. IEEE 10th International Symposium on*, Boston, MA, 2007, pp. 114-119.

[10] P. Grother*, et al.*, "Multiple-Biometric Evaluation (MBE) 2010, Report on the Evaluation of 2D Still-Image Face Recognition Algorithms," NIST, Washington, DC NIST Interagency Report 7709, 2011.

[11] J. R. Beveridge, *et al.*, "FRVT 2006: Quo Vadis face quality," *Image and Vision Computing*, vol. 28, pp. 732-743, 2009.

[12] Semiconductor Industry Association. (2010). *System Driver Chapter 2010 Updates* [Online]. Available: http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_SysDrivers.pdf

[13] K. Asanovic, *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California at Berkeley, Berkeley, CA UCB/EECS-2006-183, 2006.

[14] D. A. Patterson, "The Trouble With Multicore," *IEEE Spectr.*, July 2010.

[15] S. P. Levitan and D. M. Chiarulli, "Massively parallel processing: It's Deja Vu all over again," in *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, San Francisco, CA, 2009, pp. 534-538.

[16] J. D. Owens, *et al.*, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879-899, 2008.

[17] M. Pharr, *GPU Gems 2: Programming Techniques for High-performance Graphics and General-purpose Computation*. Boston, MA: Addison Wesley Professional, 2005.

[18] D. Wentzlaff, *et al.*, "On-Chip Interconnection Architecture of the Tile Processor," *IEEE Micro*, vol. 27, pp. 15-31, 2007.

[19] J. Chunhong, *et al.*, "A distributed parallel system for face recognition," in *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, 2003, pp. 797-800.

[20] T. C. Deepak Shekhar and K. Varaganti, "Parallelization of Face Detection Engine," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, 2010, pp. 113-117.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Third ed. San Francisco: Morgan Kaufmann, 2003.

[22] J. Gunnels, *et al.*, "A flexible class of parallel matrix multiplication algorithms," in *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, 1998, pp. 110-116.

[23] C. Lin and L. Snyder, *Principles of Parallel Programming*. Boston, MA: Pearson Education, Inc., 2009.

[24] A. Fog. (2011). *Instruction tables: Lists of instruction latencies, throughputs and micro-operations for Intel, AMD and VIA CPUs* [Online]. Available: http://www.agner.org/optimize/instruction_tables.pdf

[25] JEDEC Solid State Technology Association, "DDR3 SDRAM Specification," JEDEC JESD79-3E, 2010.

[26] Semiconductor Industry Association. (2010). *Test and Test Equipment, 2010 Tables* [Online]. Available: http://www.itrs.net/Links/2010ITRS/2010Update/ToPost/2010Tables_Test_FOCUS_A_ITRS.xls

[27] E. Chan, *et al.*, "Collective Communication: Theory, Practice, and Experience," *Concurrency and Computation: Practice and Experience*, vol. 19, pp. 1749-1783, 2007.

[28] Semiconductor Industry Association. (2009). *Test and Test Equipment* [Online]. Available: http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_Test.pdf

[29] P. J. Phillips, *et al.*, "Face Recognition Vendor Test 2002 " NIST Washington, DC NISTIR 6965, 2003.

[30] P. J. Phillips, *et al.*, "The FERET evaluation methodology for face-recognition algorithms," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 22, pp. 1090-1104, 2000.

[31] NIST. (February 2, 2010). *FERET Evaluation* [Online]. Available: http://www.itl.nist.gov/iad/humanid/feret/perf/eval.html

[32] P. J. Phillips, *et al.*, "Overview of the Face Recognition Grand Challenge," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2005.

[33] NIST. (May 16, 2009). *NIST Multiple Biometric Grand Challenge* [Online]. Available: http://face.nist.gov/mbgc/

[34] I. S. Bruner and R. Tagiuri, "The perception of people," in *Handbook of Social Psychology*. vol. 2, G. Lindzey, Ed., ed Reading, MA: Addison-Wesley, 1954, pp. 634-654.

[35] W. W. Bledsoe, "Man-Machine Face Recognition," Panoramic Research Inc., Palo Alto, CA Technical Report PRI 22, 1966.

[36] W. W. Bledsoe, "The model method in facial recognition," *Technical Report*, vol. PRI:15, Panoramic Research Inc, Palo Alto, CA, 1964.

[37] T. Kanade, *Computer Recognition of Human Faces*: Birkhauser, 1973.

[38] M. Kelly, "Visual identification of people by computer," Stanford, CA AI 130, 1970.

[39] W. Zhao, *et al.*, "Face Recognition: a literature survey," *ACM Computing Surveys,* vol. 35, pp. 399-458, December 2003.

[40] W. Zhao and R. Chellappa, *Face Processing: Advanced Modeling and Methods*. Burlington: Elsevier, 2006.

[41] M. Minsky, *The Society of Mind*. New York: Simon and Schuster, 1986.

[42] D. Hubel, *Eye, Brain, and Vision* vol. No 22: W.H. Freeman and Company, 1989.

[43] T. Sakai, *et al.*, "Processing of multilevel pictures by computer - the case of photographs of human faces," *Systems, Computers, Controls 2,* vol. No. 3, pp. 47-53, 1971.

[44] L. D. Harmon, *et al.*, "Machine Identification of human faces," *Pattern Recognition,* vol. 13, pp. 97-110, 1981.

[45] S. R. Cannon, *et al.*, "A computer vision system for identification of individuals," *Proceedings of IECON,* vol. 1, pp. 347-351, 1986.

[46] I. Craw, *et al.*, "Automatic extraction of face features," *Pattern Recognition Letters,* vol. 5, pp. 183-187, 1987 1987.

[47] K. Wong, *et al.*, "A system for recognising human faces," *Proceedings ICASSP,* pp. 1638-1642, May 1989 1989.

[48] M. Turk and A. Pentland, "Eigenfaces for recognition," *Journal of Cognitive Neuroscience,* vol. 3, pp. 71-86, 1991.

[49] R. Fisher, "The Statistical Utilization of Multiple Measurements," *Annals of Eugenics,* vol. 8, pp. 376-386, 1938.

[50] I. T. Jolliffe, *Principal Component Analysis*, 2nd ed. New York: Springer, 2002.

[51] Y. Adini, *et al.*, "Face recognition: The problem of compensating for changes in illumination direction," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 19, pp. 721-732, 1997 1997.

[52] P. N. Belhumeur, *et al.*, "Eigenfaces vs. Fisherfaces: recognition using class specific linear projection," *IEEE Trans. Pattern Anal. Mach. Intell.,* vol. 19, pp. 711-720, 1997.

[53] K. Etemad and R. Chellappa, "Discriminant analysis for recognition of human face images," *Journal of the Optical Society,* pp. 1724-1733, 1997.

[54] W. Zhao, *et al.*, "Discriminant analysis of principal components for face recognition," in *Third IEEE International Conference on Automatic Face and Gesture Recognition*, 1998, pp. 336-341.

[55] W. S. Yambor, *et al.*, "Analyzing PCA-based Face Recognition Algorithms: Eigenvector Selection and Distance Measures," Colorado State University, Fort Collins, CO July 1, 2000.

[56] M. H. Yang, *et al.*, "Face recognition using kernel eigenfaces," in *International Conference on Image Processing*, Vancouver, BC, 2000, pp. 37-40 vol.1.

[57] M. H. Yang, "Kernel Eigenfaces vs. Kernel Fisherfaces: Face Recognition Using Kernel Methods," in *Fifth IEEE International Conference on Automatic Face and Gesture Recognition*, Washington, D.C., 2002, pp. 215-220.

[58] M. S. Bartlett, *et al.*, "Independent component representation for face recognition," in *SPIE Symposium on Electronic Imaging: Science and Technology*, 1998, pp. 528-539.

[59] C. Liu, "Enhanced independent component analysis and its application to content based face image retrieval," *IEEE Trans. Syst. Man Cybern. B, Cybern*, vol. 34, pp. 1117-1127, 2004.

[60] R. Beveridge, *et al.*, "The CSU face identification evaluation system: Its purpose, features, and structure," *Machine Visions and Applications*, January 2005.

[61] B. Moghaddam and A. Pentland, "Probabilistic matching for face recognition," in *IEEE Southwest Symposium on Image Analysis and Interpretation*, 1998, pp. 186-191.

[62] B. Moghaddam, *et al.*, "Beyond eigenfaces: probabilistic matching for face recognition," in *Proceedings, Third IEEE International Conference on Automatic Face and Gesture Recognition*, 1998, pp. 30-35.

[63] B. Moghaddam, *et al.*, "Bayesian Face Recognition," *Pattern Recognition*, vol. 33, pp. 1771-1782, November 2000.

[64] B. Moghaddam, *et al.*, "Bayesian Face Recognition," Mitsubishi Electric Research Laboratories TR2000-42, 2002.

[65] L. Wiskott, *et al.*, "Face recognition by elastic bunch graph matching," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 19, pp. 775-779, 1997.

[66] L. Wiskott, *et al.*, "Face Recognition by Elastic Bunch Graph Matching," in *Intelligent Biometric Techniques in Fingerprint and Face Recognition*, L. C. Jain, Ed., ed: CRC Press, 1999, pp. 355-396.

[67] Y. Pang, *et al.*, "Iterative Subspace Analysis Based on Feature Line Distance," *IEEE Trans. Image Process.*, vol. 18, pp. 903-907, 2009.

[68] J. Yang, *et al.*, "Ubiquitously Supervised Subspace Learning," *IEEE Trans. Image Process.*, vol. 18, pp. 241-249, 2009.

[69] Z. Li, *et al.*, "Nonparametric Discriminant Analysis for Face Recognition," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, pp. 755-761, 2009.

[70] I. Kotsia, *et al.*, "Novel Multiclass Classifiers Based on the Minimization of the Within-Class Variance," *IEEE Trans. Neural Netw.*, vol. 20, pp. 14-34, 2009.

[71] L. Zhang and Y. Zhang, "Face Recognition Based on the Statistics Methods," in *3rd International Conference on Bioinformatics and Biomedical Engineering ICBBE 2009.* , 2009, pp. 1-4.

[72] Y. Pang, *et al.*, "Generalised nearest feature line for subspace learning," *Electronics Letters*, vol. 43, pp. 1079-1080, 2007.

[73] Y. Zhang and T. Zhang, "Combining Variation in the Bayesian Face Recognition," in *2nd International Congress on Image and Signal Processing, CISP '09.* , 2009, pp. 1-4.

[74] L. Chunming, *et al.*, "A Statistical PCA Method for Face Recognition," in *Second International Symposium on Intelligent Information Technology Application, IITA '08* 2008, pp. 376-380.

[75] S. Du and R. K. Ward, "Improved Face Representation by Nonuniform Multilevel Selection of Gabor Convolution Features," *IEEE Trans. Syst. Man Cybern. B, Cybern*, vol. 39, pp. 1408-1419, 2009.

[76] H. Shin, *et al.*, "Generalized elastic graph matching for face recognition," *Pattern Recognition Letters*, pp. 1077-1082, 2007.

[77] R. Senaratne and S. Halgamuge, "Optimised landmark model matching for face recognition," in *7th International Conference on Automatic Face and Gesture Recognition, FGR 2006* 2006, pp. 6 pp.-125.

[78] T. Zhang, *et al.*, "Face Recognition Under Varying Illumination Using Gradientfaces," *IEEE Trans. Image Process.*, vol. 18, pp. 2599-2606, 2009.

[79] B. Li and R. Chellappa, "Face verification through tracking facial features," *Journal of the Optical Society of America*, vol. 18, 2001.

[80] L. Hongliang, *et al.*, "FaceSeg: Automatic Face Segmentation for Real-Time Video," *IEEE Trans. Multimedia*, vol. 11, pp. 77-88, 2009.

[81] H. Moon and P. J. Phillips, "Analysis of pca-based face recognition algorithms," in *Empirical Evaluation Techniques in Computer Vision*, K. Boyer and P. J. Phillips, Eds., ed: IEEE Computer Society Press, 1998.

[82] B. Moghaddam, *et al.*, "Bayesian face recognition using deformable intensity surfaces," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition,CVPR '96* 1996, pp. 638-645.

[83] W. Zhao, *et al.*, "Subspace linear discriminant analysis for face recognition," Center for Automation Research University of Maryland, College Park, MD. CAR-TR-914, 1999.

[84] D. L. Swets and J. J. Weng, "Using discriminant eigenfeatures for image retrieval," *IEEE Trans. Pattern Anal. Mach. Intell.,* vol. 18, pp. 831-836, 1996.

[85] C. Liu and H. Wechsler, "Independent component analysis of Gabor features for face recognition," *IEEE Trans. Neural Netw.,* vol. 14, pp. 919-928, 2003.

[86] B. Kepenekci, *et al.*, "Occluded face recognition based on Gabor wavelets," in *International Conference on Image Processing* 2002, pp. 293-296.

[87] J. Wilder, "Face Recognition Using Transform Coding of Grayscale Projections and the Neural Tree Network," in *Artificial Neural Networks with Applications in Speech and Vision*, R. J. Mammone, Ed., ed: Chapman Hall, 1994, pp. 520-536.

[88] R. Chellappa, *et al.*, "Human and machine recognition of faces: a survey," *Proceedings of the IEEE,* vol. 83, pp. 705-741, 1995.

[89] K. W. Bowyer, *et al.*, "A survey of approaches and challenges in 3D and multi-modal 3D+2D face recognition," *Computer Vision and Image Understanding,* vol. 101, pp. 1-15, 2006.

[90] A. F. Abate, *et al.*, "2D and 3D face recognition: A survey," *Pattern Recognition Letters,* vol. 28, pp. 1885-1906, 2007.

[91] A. F. Abate, *et al.*, "Ultra fast GPU assisted face recognition based on 3D geometry and texture data," in *Image Analysis and Recognition. Third International Conference, ICIAR 2006. Proceedings, Part II (Lecture Notes in Computer Science Vol. 4142)*, Povoa de Varzim, Portugal, 2006, pp. 353-64.

[92] W. L. Cheong, *et al.*, "Building a computation savings real-time face detection and recognition system," in *2nd International Conference on Signal Processing Systems, ICSPS* pp. V1-815-V1-819.

[93] K. Meng, *et al.*, "A high performance face recognition system based on a huge face database," in *Proceedings of 2005 International Conference on Machine Learning and Cybernetics* 2005, pp. 5159-5164 Vol. 8.

[94] K. V. Arya, *et al.*, "Face recognition using Parallel Associative Memory," in *IEEE International Conference on Systems, Man and Cybernetics, SMC* 2008, pp. 1332-1336.

[95] I. Sajid, *et al.*, "Hardware-Based Speed Up of Face Recognition Towards Real-Time Performance," in *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, DSD* pp. 763-770.

[96] I. Sotiropoulos and I. Papaefstathiou, "A fast parallel matrix multiplication reconfigurable unit utilized in face recognitions systems," in *International Conference on Field Programmable Logic and Applications, FPL* Prague, Czech Republic, 2009, pp. 276-281.

[97] O. Lahdenoja, *et al.*, "A massively parallel face recognition system," *EURASIP J. Embedded Syst.,* vol. 2007, pp. 31-31, 2007.

[98] Q. Yang and C. Guo, "Parallel face recognition approach based on LGBPHS with homogeneous PC cluster," *Journal of Information and Computational Science,* vol. 7, pp. 637-648, March 2010.

[99] Y. Ouerhani, *et al.*, "Fast Face Recognition Approach Using a Graphical Processing Unit "GPU"," in *2010 IEEE International Conference on Imaging Systems and Techniques (IST 2010)*, Piscataway, NJ, USA, 2010, p. 5 pp.

[100] D. Zhang, *et al.*, "Technology Evaluations on the TH-FACE Recognition System," in *Advances in Biometrics*. vol. 3832, ed: Springer Berlin / Heidelberg, 2005, pp. 589-597.

[101] D. O. Hebb, *The organization of behavior*. New York: Wiley & Sons, 1949.

[102] R. Beveridge, *et al.*, "The CSU face identification evaluation system: Its purpose, features, and structure," *Machine Visions and Applications*, January 2005.

[103] M. Grgic, *et al.*, "SCface - surveillance cameras face database," *Multimedia Tools and Applications*, vol. 51, pp. 863-879.

[104] P. J. Phillips, *et al.*, "Face Recognition Vendor Test 2002, Technical Appendices," National Institute of Standards and Technology (NIST) NISTIR 6965, March 2003.

[105] P. J. Phillips, *et al.*, "FRVT 2006 and ICE 2006 Large-Scale Experimental Results," *IEEE Trans. Pattern Anal. Mach. Intell.,* vol. 32, pp. 831-846, 2010.

[106] P. J. Phillips, *et al.*, "The FERET evaluation methodology for face-recognition algorithms," *Pattern Analysis and Machine Intelligence, IEEE Transactions on,* vol. 22, pp. 1090-1104, 2000.

[107] P. J. Phillips, *et al.*, "The FERET database and evaluation procedure for face recognition algorithms," *Image and Vision Computing J,* vol. 16, pp. 295-306, 1998.

[108] D. Knuth, *The Art of Computer Programming* vol. Vol. 3 Reading, Massachusetts: Addison-Wesley, 1998.

[109] J. Osier. (Published 1993, Accessed November 19, 2011). *GNU gprof*. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html

[110] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, California, 2007.

[111] ConcurrentEDA Inc. *Concurrent Analytics*. Available: http://www.concurrenteda.com/index.php?option=com_content&view=category&layout=blog&id=47&Itemid=83

[112] R. M. Stallman and GCC Developer Community, *Using the GNU Compiler Collection*. Boston, MA: GNU Press, a division of the Free Software Foundation, 2003.

[113] Free Software Foundation Inc. (2011, February 10, 2012). *GNU Binutils*. Available: http://www.gnu.org/software/binutils/

[114] Advanced Micro Devices, "Software Optimization Guide for AMD Family 10h Processors, Publication Number 40456, Revision 3.13," February 2011.

[115] Intel Corporation, "Intel 64 and IA-32 Architectures Optimization Reference Manual," Document number: 248966-025, 2011.

[116] M. Turk and A. Pentland, "Face Recognition Using Eigenfaces," in *IEEE Conference on Computer Vision and Pattern Recognition*, Maui, Hawaii, 1991, pp. 586-591.

[117] S. Z. Li and A. K. Jain, *Handbook of Face Recognition*. New York: Springer Science+Business Media, Inc., 2005.

[118] D. Gabor, "Theory of Communication," *J. IEE*, vol. 93, pp. 429-459, 1946.

[119] J. Daugman, "Complete Discrete 2D Gabor Transform by Neural Networks for Image Analysis and Compression," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 36, pp. 1169-1179, July 1988.

[120] JEDEC, "DDR2 SDRAM Specification," JEDEC Solid State Technology Association2009.

[121] R. D. Williams, *et al.*, "Server Memory Road Map," presented at the Memory Forum, Shenzhen, China, 2012.

[122] H. F. Jordan and G. Alaghband, *Fundamentals of Parallel Processing*. Upper Saddle River, NJ: Pearson Education, Inc., 2003.

[123] Semiconductor Industry Association. (2009). *2009 Update, System Drivers* [Online]. Available: http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_SysDrivers.pdf

[124] N. D. Enright Jerger, *On-chip Networks*. San Rafael, CA: Morgan & Claypool Publishers, 2009.

[125] M. Snir*, et al.*, *MPI-The complete reference* Second ed. vol. 1: The MPI Core. Cambridge, MA: The MIT Press, 1998.

[126] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third ed. Boston, MA: Pearson Education, Inc., 2004.

[127] Object Management Group, "OMG Unified Modeling Language (OMG UML), Infrastructure," 2010.