Energy-Efficient Circuits and Systems for Computational Imaging and Vision on Mobile Devices

by

Priyanka Raina

B.Tech., Indian Institute of Technology Delhi (2011)
S.M., Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Author ..... Signature redacted
Department of Electrical Engineering and Computer Science
January 31, 2018

Certified by ..... Signature redacted ...
Anantha Chandrakasan
Vannevar Bush Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ..... Signature redacted
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

Energy-Efficient Circuits and Systems for Computational Imaging and Vision on Mobile Devices

by

Priyanka Raina

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Eighty five percent of images today are taken by cell phones. These images are not merely projections of light from the scene onto the camera sensor but result from a deep calculation. This calculation involves a number of computational imaging algorithms such as high dynamic range (HDR) imaging, panorama stitching, image deblurring and low-light imaging that compensate for camera limitations, and a number of deep learning based vision algorithms such as face recognition, object recognition and scene understanding that make inference on these images for a variety of emerging applications. However, because of their high computational complexity, mobile CPU or GPU based implementations of these algorithms do not achieve real-time performance. Moreover, offloading these algorithms to the cloud is not a viable solution because wirelessly transmitting large amounts of image data results in long latency and high energy consumption, making them unsuitable for mobile devices.

This work solves these problems by designing energy-efficient hardware accelerators targeted at these applications. It presents the architecture of two complete computational imaging systems for energy-constrained mobile environments: (1) an energy-scalable accelerator for blind image deblurring, with an on-chip implementation and (2) a low-power processor for real-time motion magnification in videos, with an FPGA implementation. It also presents a 3D imaging platform and image processing workflow for 3D surface area assessment of dermatologic lesions. It demonstrates that such accelerator-based systems can enable energy-efficient integration of computational imaging and vision algorithms into mobile and wearable devices.

Thesis Supervisor: Anantha Chandrakasan
Title: Vannevar Bush Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Professor Anantha Chandrakasan, for being a great advisor and a source of inspiration to me, and for guiding and supporting me in my research. Despite all of his numerous responsibilities, he always found time to talk to me. Not only did he guide me in my research, he continuously helped me grow as a teacher and a mentor by giving me the opportunity to teach 6.374 and work with a number of undergraduate students during my time at MIT. I want to thank him for the collaborative and supportive culture that he has fostered in *ananthagroup*; my colleagues in the lab became some of my best friends. I have some big shoes to fill as I start my academic career.

I would like to thank my thesis committee members, Professor Vivienne Sze and Professor Arvind, for giving me feedback on my research and thesis and for supporting me in my professional endeavors.

I would like to thank Professor William T. Freeman and Professor Frédo Durand for collaborating with me on the image deblurring and the motion magnification projects, and for their feedback during the course of the two projects.

I would like to thank Dr. Victor Huang for collaborating with me on the vitiligo project, for collecting the clinical data to test our platform and for providing valuable clinical insights. I would like to acknowledge and thank Jiarui (Gary) Huang for working with me to implement and test the lesion surface area measurement algorithm for vitiligo.

I would like to thank Foxconn Technology Group for funding and supporting the image deblurring project and Intel Corporation for funding the motion magnification project. In particular, I would like to thank Yihui Qiu for her valuable feedback and suggestions on the deblurring project and Mondira Pant for her feedback on the motion magnification project. I would like to thank TSMC University Shuttle Program for chip fabrication.

I would like to thank all the members of *ananthagroup* for making the lab such an exciting place to work at. In particular, I would like to thank Mehul Tikekar, for

being a great teacher, friend and mentor. I would like to thank Arun Paidimarri, Chiraag Juvekar and Phillip Nadeau for all the help and feedback they gave me on my projects and papers. I would like to thank Avishek Biswas for being a wonderful TA and for making teaching 6.374 a great experience. I would like to thank Preet Garcha and Tugce Yazicigil for their friendship and support when I needed it the most.

I would like to thank Anasuya Mandal for being a wonderful roommate and for all the good times we have spent together.

Finally, I would like to thank my father, Romesh Kumar, my mother, Archana Raina, my brother, Pranav Raina, and my fiancé, Raghu Mahajan, for their love, support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

It is projected that 1.2 trillion photos will be taken in 2017 [14]. 85% of these photos will be taken by cell phones and only 10% by traditional digital cameras. Moreover, the imaging that we do with cell phones isn't what it used to be five years ago. The final image that we see today is not merely a projection of light from the scene onto a sensor but results from a deep calculation. A number of computational imaging and vision algorithms are involved in this deep calculation, and based on their characteristics, we can classify them into three categories, ones that (1) correct for camera limitations and extend its capabilities such as high dynamic range (HDR) imaging, panorama stitching, image deblurring and low-light imaging (2) reveal hidden information from a scene or add new information to a scene such as motion magnification, video super-resolution and augmented reality and (3) make inference on images such as face recognition, object detection, scene classification and image captioning. Even though for most of these applications the algorithms have matured to a large extent, only a few of them such as HDR imaging, panorama stitching and face recognition have been implemented widely on mobile devices, while most others remain entirely academic. The main reason for this is that most of these algorithms are extremely computationally complex, and existing implementations on CPU or GPU platforms are neither energy-efficient, nor high-performance enough to

support these applications in real-time. Figure 1-1 plots the energy efficiency and execution time (for the same frame size) for different modern computational imaging and vision applications when run on different platforms: CPU, GPU and dedicated hardware accelerator. A low value for both energy and execution time is desirable for mobile devices. For example, estimating the blur kernel for deblurring a single image takes 2.23 minutes on a desktop CPU and 13.60 minutes on a mobile CPU. It consumes 225 microjoules of energy per pixel on a desktop CPU and 1102 microjoules of energy per pixel on a mobile CPU. Compared to this, HEVC video decoding, which is widely implemented on cell-phones using custom hardware, takes only 16.7 milliseconds per frame and consumes 0.4 nanojoules of energy per pixel. Therefore, 2 - 4 orders of magnitude improvement in execution time and 3 - 6 orders of magnitude improvement in energy is required for most of these applications to support real-time processing with long battery life on mobile platforms.

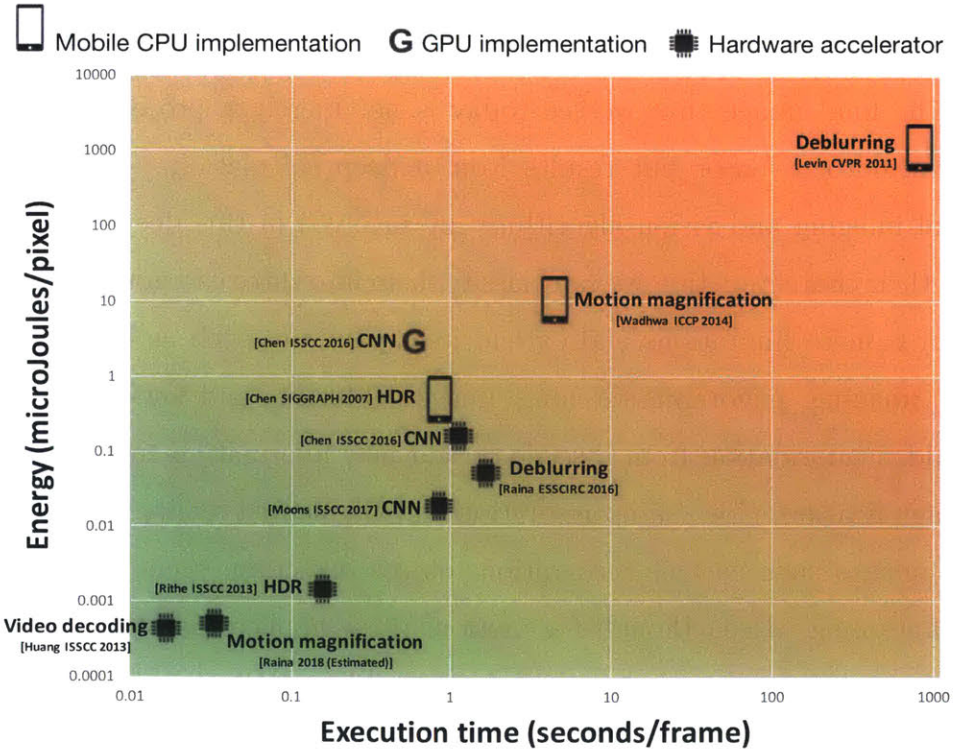


Figure 1-1: Energy efficiency and execution time (for the same frame size) for different modern computational imaging and vision applications when run on different platforms: CPU, GPU and dedicated hardware accelerator. A low value for both energy and execution time is desirable for mobile devices.

1.2 Thesis Contributions

This work obtains this performance improvement and energy reduction by designing energy-efficient hardware accelerators targeted at these applications. It presents three complete computational imaging systems: (1) an energy-scalable accelerator for image deblurring, (2) a low-power processor for real-time motion magnification in videos, and (3) a 3D imaging platform for automated surface area assessment of dermatologic lesions. The first two are accelerator-based systems and they provide 2 - 3 orders of magnitude improvement in runtime and 3 - 4 orders of magnitude improvement in energy compared to existing implementations. We have proposed a number of energy minimization techniques to obtain these improvements, for example, (1) performing algorithmic and architectural optimizations such as computation reuse and reordering to either dynamically reduce energy by exploiting sparsity in updates to data, or statically reduce energy by reusing computation results that are common across several data elements, (2) scaling down the precision of computation in arithmetic pipelines to reduce energy while maintaining just enough visual quality, and (3) compressing and caching data to reduce off-chip memory bandwidth and energy. In addition to employing these techniques, each of these systems is made configurable in terms of a number of parameters to get energy scalability by trading off accuracy with execution time. This is essential in real-life applications where one might still want to run a complex algorithm in a low-battery scenario but might be willing to sacrifice some visual quality. The following subsections give an overview of these three key projects and the following chapters describe them in detail.

1.2.1 Energy-Scalable Accelerator for Blind Image Deblurring

Camera shake is a common cause of blur in cell-phone camera images. To remove this blur, one needs to estimate the trajectory of the camera during the exposure. This trajectory is represented by a convolution kernel, and the deblurring problem reduces to (1) estimating the kernel from the blurred image and (2) performing deconvolution to obtain a sharp image. State of the art kernel estimation algorithms [18] are

computationally intensive and take more than 2 minutes to run for a single image on a desktop CPU, accounting for 99% of the deblurring time. They also consume 225 microjoules of energy per pixel which makes them unsuitable for implementation in software on mobile devices.

This work presents an energy-efficient and high-performance hardware accelerator for kernel estimation which solves for the kernel using an iterative expectation maximization (EM) based algorithm [18]. It starts with an initial random guess for the kernel, and alternates between solving for the image or the kernel given the other using a numerical solver. We propose several runtime reduction and energy minimization techniques in this design; some are used both in the software and hardware realizations and some are used in the hardware realization only. One of the key energy minimization techniques is computation reuse and reordering to either dynamically reduce energy by exploiting sparsity in updates to data, or statically reduce energy by reusing computation results that are common across several data elements. An example of dynamic reuse that we employ in both software and hardware happens while solving for the blur kernel. To get the kernel, one has to minimize a constrained quadratic program of the form $\frac{1}{2}k^TAk - b^Tk$ such that $k \geq 0$ to obtain the kernel k. Matrix-vector multiplications of the form $y = Ax$ dominate the computational time, as A can be very large, up to 841 $\times$ 841 pixels. However, we observe that during each iteration the intermediate vector x that needs to be multiplied with A updates only at a few indices. Therefore, we exploit this sparsity in updates, and convert matrix-vector multiplies to a few vector-vector operations. This reduces the number of solver multiplies, adds and DRAM accesses by 11$\times$ on an average.

The accelerator is implemented in TSMC 40 nm CMOS technology and a complete system demonstration platform is designed for real-time image deblurring that offloads the computationally expensive kernel estimation to the accelerator chip and performs the inexpensive final deconvolution on a CPU. The accelerator achieves a 78$\times$ reduction in runtime for kernel estimation with respect to our optimized software realization (one that employs all the energy minimization techniques that can be performed in software) of [18] on a desktop CPU for a 13 $\times$ 13 kernel, and a 56$\times$

reduction in the complete deblurring time of a FullHD (1920 × 1080) image. Compared to an implementation of the same algorithm on a mobile processor, it achieves a 480× reduction in kernel estimation time. The accelerator consumes only 105 mJ at its nominal operating point compared to 467 J consumed by the desktop CPU, and 2284 J consumed by the mobile CPU. The system is also made configurable in terms of a number of parameters such as the number of iterations and kernel size to get energy scalability by trading off accuracy with execution time. This is useful in the following scenarios: (1) If coarse camera motion is available through inertial sensors, it can be used to initialize the kernel, and the accelerator can be configured to run for a fewer number iterations to reduce energy. (2) In an energy constrained scenario, one can down-sample the input image, use the accelerator to estimate a kernel of a smaller size, and then up-sample the kernel to do deconvolution on the entire image, for example, if the true kernel is 13 × 13, down-sampling by a factor of 2 leads to a 2.15× reduction in energy. This energy-scalable implementation enables efficient integration of image deblurring into battery-operated mobile devices.

1.2.2 Low-Power Processor for Real-Time Motion Magnification in Videos

There are a number of phenomena around us that exhibit small motions that are invisible to the naked eye. Algorithmic work by [30] has shown that computational amplification can be used to reveal such motions. These motion magnification algorithms use video as input, and analyze each pixel for slight variations in phase over time and amplify these variations. This technology can be used for various applications such as non-invasive health monitoring, industrial infrastructure monitoring and structural integrity assessment, which currently rely on more expensive hardware and human intervention.

However, the state of the art motion magnification algorithms are extremely computationally intensive and do not achieve real-time performance on modern CPUs. Reported results [30] show that the fastest algorithm can only achieve a frame rate

of 2.37 frames per second (FPS) on FullHD (1920 × 1080) and 5.33 FPS on HD (1280 × 720) video when run on an Intel Xeon E5-2623 desktop CPU and only 0.23 FPS on FullHD and 0.52 FPS on HD video on an ARM Cortex-A15 mobile CPU. This implies that there is a 6× to 130× performance gap that needs to be bridged to achieve real-time performance on HD/FullHD video. Moreover, high computational complexity leads to high energy consumption which makes the algorithm unsuitable for implementation on battery operated portable devices like wearables, cell phones and tablets. The motion magnification algorithm while running on an ARM Cortex-A15 mobile processor consumes $10.36\mu J$ per pixel, whereas video compression on mobile phones takes on the order of $2nJ$ per pixel with dedicated hardware, therefore, four orders of magnitude reduction in energy per pixel is required to enable energy-efficient implementation of motion magnification on mobile devices.

In this work, we present a low-power processor to accelerate motion magnification that achieves real-time performance on HD (1280×720) video at 30 frames per second, while consuming less than 0.2 nanojoules of processing energy per pixel. The processor accelerates the algorithm proposed in [30], which uses a Riesz pyramid to decompose each frame of the input video and separate the amplitude of the local wavelets from their phase. It then performs temporal filtering of the phases independently at each location, orientation and scale to isolate the frequencies of interest (for example, a band around 1 Hz for breathing rate amplification). This is followed by spatial smoothing to reduce noise in the phase, and finally amplification and reconstruction of the output video by inverting the pyramid.

We propose to use the following techniques in the design of the accelerator to achieve energy-efficiency and real-time performance: (1) Separable filtering in pyramid computation and spatial filtering to reduce the number of multiplies and adds. (2) Zero-skipping to reduce the buffering requirements of pyramid computation. (3) Reducing the precision of computation in several modules to reduce energy and selectively using block floating point representation to maintain accuracy. (4) Caching intermediates in on-chip SRAM based line buffers to reduce external memory bandwidth and save system energy. The complete motion magnification architecture is

demonstrated on an FPGA and simulated area and power measurements in 40 nm CMOS technology are presented in this thesis. These techniques enable efficient integration of motion magnification technology into mobile devices.

1.2.3 3D Imaging Platform for Automated Surface Area Assessment of Dermatologic Lesions

This work develops a tool and associated image processing/enhancement algorithms to enable the rapid, reproducible, and objective determination of the surface area and volume of a cutaneous lesion or lesions in an automated fashion via 3D imaging. This work has been done in collaboration with MIT student, Jiarui Huang, who worked on algorithm design, its python implementation and initial testing as a part of his Master's thesis, and Dr. Victor Huang from Brigham and Women's Hospital: Boston Hospital & Medical Center, who collected the patient data using the system, and provided clinical insights into the results.

For many dermatologic conditions, the initial assessment of disease severity as well as the primary outcome measure of progression/treatment success are dependent on an assessment of body surface area (BSA). This assessment is based on a physician's estimation of area involvement, which has low accuracy and reproducibility. Moreover, these assessments often are performed by non-dermatologists with no training to complete such assessments. This limitation represents a major hurdle for clinical trials, translational research efforts, as well as daily clinical care.

In this work, we have chosen vitiligo as a model condition to demonstrate the utility of such a tool because of low acuity of pathology experienced by the patients allowing for investigative imaging with minimal impact on clinical care, the varied locations of presentation across the body, the relatively high prevalence affecting all skin types and ethnicities equally (1-2% of the general population), and the underserved nature of the disease. Existing approaches to measure the area of vitiligo lesions suffer from many shortcomings - some require manual tracing of the lesion which is very time-consuming because the lesions boundaries are very complex; others only give a

relative measure of area and not an absolute one which makes comparing different lesions for the same person or across different people impossible; and most of these methods ignore the curvature of the lesions and measure the projected area rather than the true surface area of the lesion.

This work presents a new solution to surface area measurement of vitiligo lesions by incorporating a depth camera and image processing algorithms. We show that this system can perform lesion segmentation robustly and measure lesion area accurately over any skin surface. Compared to the currently existing approaches, this system has several advantages. It is easy to use, does not require any precise calibration or professional training. It is contact-free. This eliminates the possibility of contamination and discomfort caused by manual tracing. It can measure the absolute area of any surface.

The anticipated applications are to determine the burden of disease and response to treatment for patient care and clinical trial applications. In particular, vitiligo, eczema, psoriasis, and chronic wounds are immediate areas for application. In addition, determination of BSA of adverse dermatologic side effects would allow for granular, objective grading of adverse events or determination of BSA involvement in burns or wounds.

1.3 Thesis Outline

Chapter 2 presents a detailed architectural description and results of the image deblurring accelerator. Chapter 3 presents the motion magnification processor. Chapter 4 presents the 3D imaging platform for surface area assessment of dermatologic lesions.

Chapter 2

Image Deblurring Accelerator

2.1 Motivation

Camera shake is a common cause of blur in images. The most widely used solution to avoid blur is to perform physical compensation using image stabilization, where either the camera lens or the image sensor are physically moved in order to compensate for the shake. However, this solution is limited to small camera motion. The second solution is to do algorithmic compensation using blind image deblurring. Here, one needs to estimate the camera trajectory which is usually unknown, hence the deblurring is "blind". The trajectory is represented by a convolution kernel, and the deblurring problem reduces to (1) estimating the kernel from the blurred image and (2) performing deconvolution to obtain a sharp image, as shown in Figure 2-1. Kernel estimation algorithms existing in literature [18, 12] are computationally intensive and take several minutes to run in software on a CPU, accounting for 99% of the deblurring time. These are unsuitable for implementation in software on mobile devices because of both performance and energy concerns, and therefore, are our target for hardware acceleration in this work.

Recent research has shown energy and performance benefits of hardware accelerators for several computational photography and computer vision applications such as high dynamic range and low-light imaging [21], obstacle detection [15] and deep learning based object recognition [9]. However, image deblurring has so far been im-

plemented only in software running on CPU/GPU platforms, which do not support real-time processing and have high energy consumption, making these implementations unsuitable for mobile devices.

2.2 Background

The expectation maximization (EM) based kernel estimation algorithm proposed in [18] is highly accurate but computationally expensive (taking more than 2 minutes to run for a single image frame on a CPU), making it an ideal candidate for hardware acceleration. Since camera shake blur is spatially-invariant, a 128×128 patch is chosen from the blurred image for kernel estimation to reduce the size of the problem. The blurred patch B is modeled as $B = K \otimes S + N$ where K is the unknown kernel, S is the unknown sharp image and N is noise. The convolution operation can equivalently be expressed as a matrix multiplication - blurred image $b = T_k s + n = T_s^T k + n$, where T_k and T_s are the Toeplitz matrices for K and S, and b, s, k and n are raster scan flattened versions of their 2D counterparts. In this thesis, both representations are used - the 1D representation is used in context of the two optimization problems that are solved to get the kernel and the sharp image, and the 2D representation is used in context of the processing of the inputs to these optimization problems, which are 2D images.

Given this model, the EM algorithm has two key steps:

1. **E-step or Image Refinement:** In the E-step, the algorithm starts with an

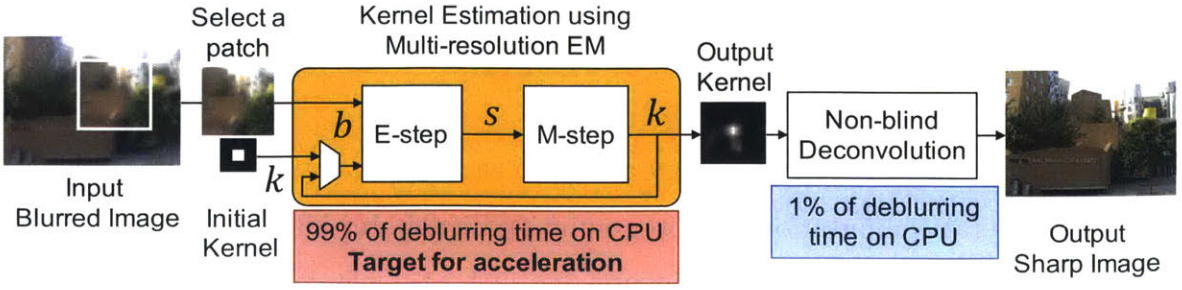


Figure 2-1: Expectation maximization (EM) based blind image deblurring.

initial guess for the kernel, which can be random, or obtained from inertial sensors in a mobile device, and **finds the sharp image** s which minimizes the cost function $||b - T_k s||^2 + R(s)$ where the first term is the convolution error and the second term is a regularization term which ensures that the derivatives of the estimated sharp image are sparse, like in naturally occurring sharp images. The algorithm also **computes a covariance matrix** C around the sharp image estimate, which signifies how confident it is about this sharp image. Since the regularization term makes the cost function non-quadratic, its minimization is done using an **iteratively re-weighted least squares (IRLS) deconvolution engine** and the covariance is computed using a **covariance estimator**.

2. **M-step or Kernel Refinement:** In the M-step, the algorithm uses the refined sharp image s and the covariance C from the E-step, and **finds the kernel** k which minimizes the expected convolution error subject to the constraint that all kernel entries are positive. This simplifies to solving a constrained quadratic program (QP) $\frac{1}{2}k^T \bar{A}_k k + \bar{b}_k^T k$ such that $k \geq 0$ where the matrix $\bar{A}_k$ is given by $\bar{A}_k(i_1, i_2) = \sum_i s(i + i_1)s(i + i_2) + C(i + i_1, i + i_2)$ and the vector $\bar{b}_k$ is given by $\bar{b}_k(i_1) = \sum_i s(i + i_1)b(i)$ ([18] provides a detailed derivation of this simplification). In this work, the computation of the QP coefficients, $\bar{A}_k$ and $\bar{b}_k$, is done using an **image correlator**, and solution of the constrained QP is found using a **gradient projection solver**.

As shown in Figure 2-1, the refined kernel obtained from the M-step is fed back into the E-step and the two steps are iterated multiple times until the desired number of iterations are performed. To avoid local optima, the EM iterations are carried out at successively higher resolutions of the patch size (32×32, 64×64, 128×128), with the kernel at higher resolutions seeded from the result of the lower resolutions. We varied the size of the image patch used for kernel estimation from 256×256 to 32×32 in software, and observed that choosing a patch size smaller than 128×128 compromises the quality of the obtained kernel, and the resulting deblurred image

has several ringing artifacts. The estimated kernel from a single patch is finally used for deconvolution of the full 1920×1080 blurred image. This deconvolution accounts for only 1% of the runtime, and is therefore done on a CPU using Gaussian or sparse prior based approaches described in [19].

In this thesis, we present the first hardware accelerator for kernel estimation, which accounts for 99% of the time in image deblurring applications. It reduces kernel estimation time by 78× (from 2 minutes to about a second) and energy by three orders of magnitude compared to a software implementation of the same algorithm on an Intel Core i5 CPU, making it suitable for integration into mobile devices. The accelerator also provides 10× energy scalability through configurability in the number of iterations and the kernel size (from 7×7 to 29×29), allowing the system to trade off runtime with image quality in energy-constrained scenarios.

We use the following techniques to obtain high throughput and low energy and area of the kernel estimation accelerator:

1. A **multi-resolution IRLS deconvolution engine** with DFT based matrix multiplication for image refinement. (Section 2.3)

2. A **high-throughput image correlator** for computing the coefficient matrices of the kernel QP. (Section 2.4)

3. A **high-speed selective update based gradient projection solver** for solv-

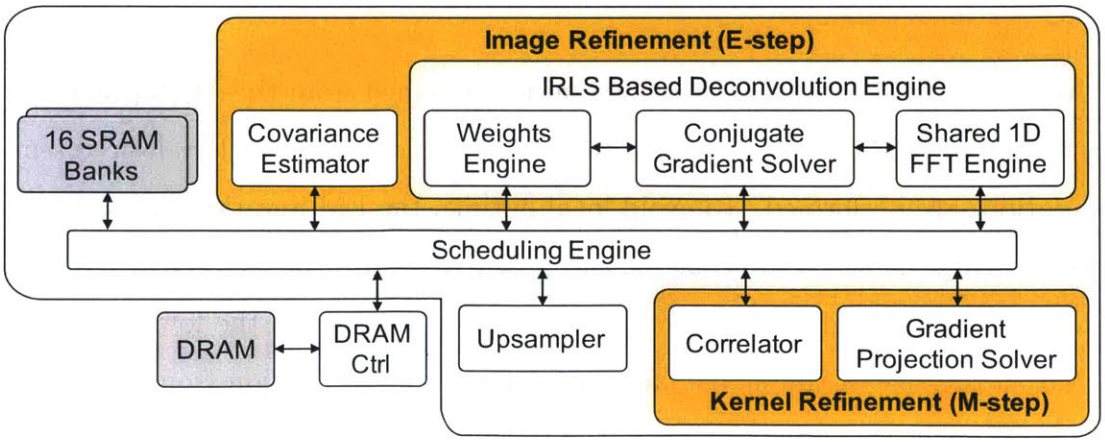


Figure 2-2: Accelerator system architecture.

ing the kernel QP. (Section 2.5)

These modules are controlled by a centralized scheduling engine to run the EM algorithm (Figure 2-2). They use 16 shared 4096×32 bit SRAMs as scratch memory and interface to an external DRAM for intermediate storage. A 32 bit floating point (FP) datapath is used in all modules to avoid inaccuracies in the estimated kernel, which can cause undesirable ringing artifacts in the deconvolved image. Since the image and kernel refinement steps are non-concurrent, data and clock gating are used by the scheduler to save energy.

Additionally, instead of running the EM algorithm on the image (S, B) itself, it is run on the gradients of the image (S_γ, B_γ) to determine the kernel, where $\gamma = 0$ corresponds to the horizontal gradient and $\gamma = 1$ corresponds to vertical gradient, since it is found to give better results in practice [18].

2.3 Multi-Resolution IRLS Deconvolution Engine with DFT Based Matrix Multiplication

Image refinement entails solving a minimization problem to get the sharp image s, but since the cost function to be minimized, $||b_\gamma - T_k s_\gamma||^2 + R(s_\gamma)$, is not quadratic because of the regularization term, we use the iteratively re-weighted least squares (IRLS) approach [19]. This involves repeating the following two steps until convergence. In each iteration i,

1. Approximate the non-linear cost function with a quadratic one using a regularization weights matrix $W_\gamma^{(i-1)}$, and minimize the quadratic by solving the linear system $(T_k^T T_k + W_\gamma^{(i-1)}) s_\gamma^{(i)} = T_k^T b_\gamma$ to obtain $s_\gamma^{(i)}$.

2. Use the solution $s_\gamma^{(i)}$ to find updated regularization weights $W_\gamma^{(i)}$ and repeat.

The linear system in each iteration is solved using a conjugate gradient (CG) solver. Here, the regularization weights penalize the gradients in the smooth regions more than in the edge regions, enforcing a sparse prior on the sharp image gradients. For a detailed derivation, see [18].

2.3.1 IRLS Optimizations

We propose the following optimizations to the IRLS-based deconvolution engine to reduce the number of FP operations and area compared to [19]:

1. **DFT Based Matrix Multiplication:** We observe that the most computationally intensive step in the IRLS deconvolution is the multiplication with $n^2 \times n^2$ matrix $T_k^T T_k$ in each CG iteration, where $n \times n$ is the size of the sharp image. This multiplication is equivalent to convolution with the auto-correlation of kernel k. Since k is large, we implement this convolution in the frequency domain as a multiplication with the squared magnitude of the 2D DFT of k.

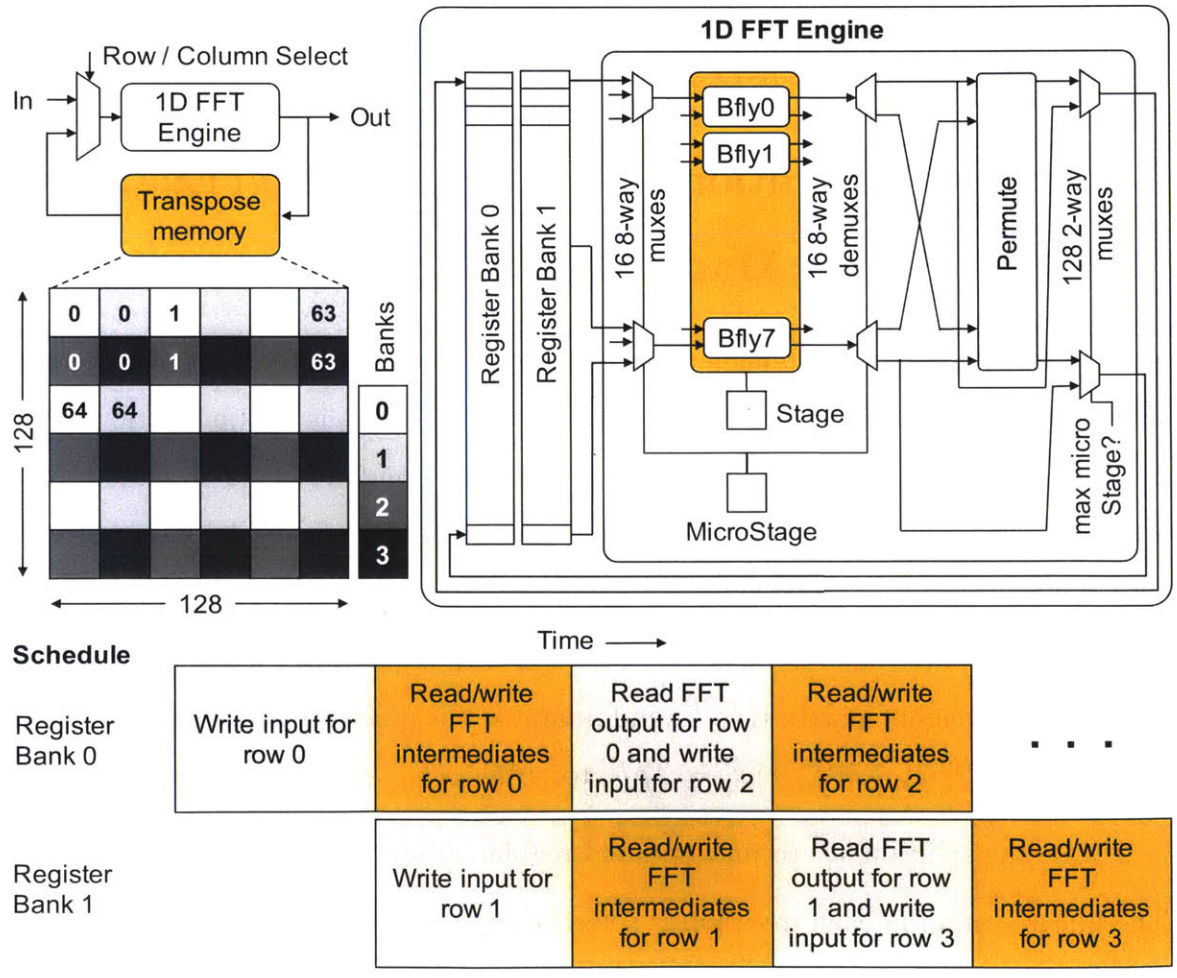


Figure 2-3: Left: 2D DFT with shared 1D FFT and transpose memory. Right: Reconfigurable 1D FFT architecture. Bottom: Schedule for register bank access for I/O and FFT computation.

This reduces the computational complexity of the matrix multiplication and gives an improvement in the number of floating point operations by 8.8× for 128×128, 10.2× for 64×64 and 12.2× for a 32×32 image and a 13×13 kernel compared to a spatial convolution based approach.

2. **Time-Shared Floating Point 1D FFT Architecture:** We perform the 2D DFT by taking 1D DFT of rows followed by columns. Instead of having dedicated 1D FFT engines for computing row and column DFTs, we propose using a shared 1D FFT engine and an SRAM-based transpose memory similar to [13] (Figure 2-3). The 1D FFT engine supports point sizes (N) of 32, 64 and 128 for multi-resolution processing by time-sharing only 8 radix-2 butterflies to perform all log(N) steps of the FFT and each step in N/16 sub-steps (Figure 2-3). This choice of radix and number of butterflies minimizes the area required to support the required point sizes while meeting 2 pixels/cycle throughput. To avoid stalls, the engine uses 2 register banks, each of which can store 128 samples, as a ping-pong buffer as shown in the schedule in Figure 2-3.

2.4 High-Throughput Image Correlator

After completing the E-step, we get the sharp image (S_γ) and covariance image (C_γ) for both the horizontal ($\gamma = 0$) and the vertical ($\gamma = 1$) gradient components and we use these to obtain a better kernel by solving the quadratic program (QP) from section 2.2 in the M-step. Setting up the QP requires computing its coefficient matrix $\bar{A}_k$. For an $n \times n$ sharp image, S_γ, and an $m \times m$ kernel, the $m^2 \times m^2$ matrix, $\bar{A}_k$, is the sum of the two $\bar{A}_{k\gamma}$ matrices corresponding to the two gradient components. These are given by

$$\bar{A}_{k\gamma}(mx_1 + y_1, mx_2 + y_2) = \sum_{x=m-1}^{n-1} \sum_{y=m-1}^{n-1} S_\gamma(x - x_1, y - y_1) * S_\gamma(x - x_2, y - y_2) + \Big(C_\gamma(x - x_1, y - y_1) \text{ when } x_1 = x_2 \text{ and } y_1 = y_2 \Big)$$

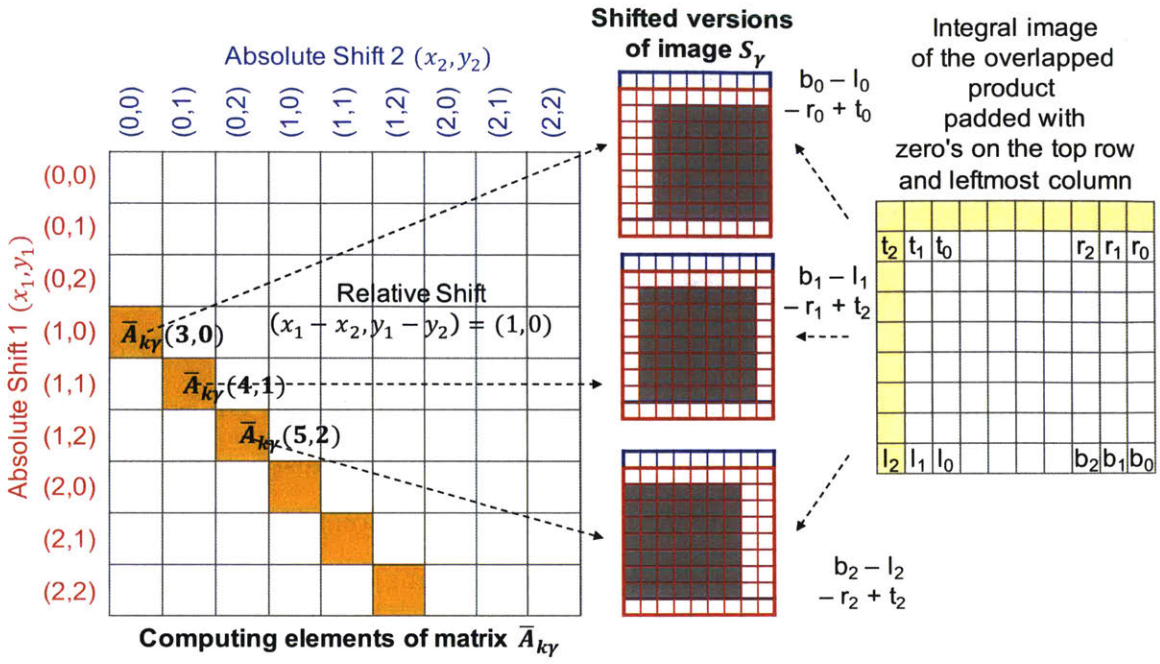


Figure 2-4: Computation of correlation matrix $\bar{A}_{k\gamma}$ with diagonal computation reuse for a toy kernel of size 3×3. Along each diagonal same corresponding elements are multiplied, but are summed over a different area using the integral image of the overlapped product.

where the shifts (x_1, y_1) and (x_2, y_2) vary from to $(0,0)$ to $(m-1, m-1)$. The processing for both the S_γ term and the C_γ term is similar, so let us focus our attention on the S_γ term. The computation time for this term is $O(m^4)$, which is too high to meet the runtime specifications for the accelerator.

2.4.1 Correlator Optimizations

We propose the following optimizations to the correlator to reduce the correlation execution time and buffering requirements compared to the software implementation of the same algorithm proposed in [18]:

1. **Diagonal Computation Reuse:** We observe that along the diagonals of $\bar{A}_{k\gamma}$, the relative shift $(\Delta x, \Delta y) = (x_1 - x_2, y_1 - y_2)$ remains constant between the two shifted images $S_\gamma(x - x_1, y - y_1)$ and $S_\gamma(x - x_2, y - y_2)$ with absolute shifts (x_1, x_2) and (y_1, y_2), as shown in Figure 2-4. Therefore, the same corresponding

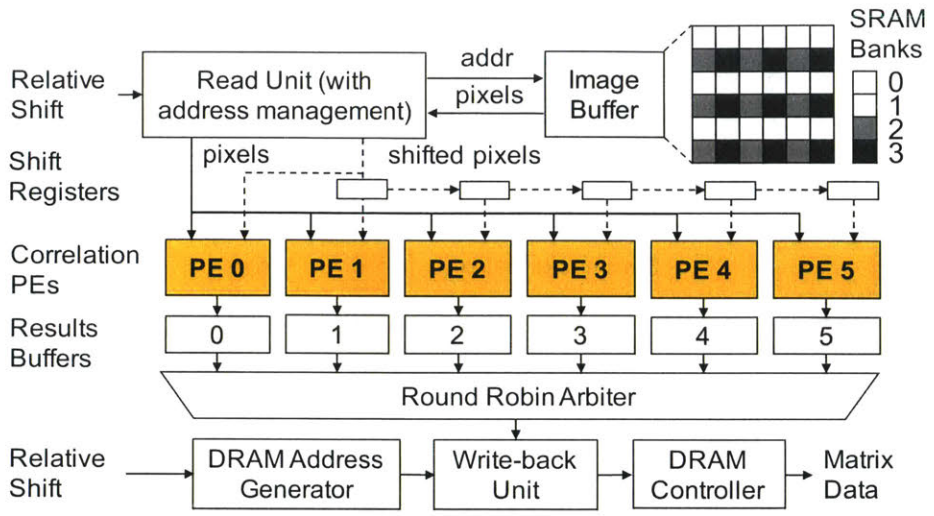


Figure 2-5: Highly parallel correlator architecture with 6 processing elements enabled by image tiling in the image buffer gives 6× improvement in throughput.

elements are multiplied, but the summation of the product is performed over different pixels. We propose doing this multiplication $S_\gamma(x,y)S_\gamma(x-\Delta x, y-\Delta y)$ only once for each relative shift $(\Delta x, \Delta y)$, and then using integral image [28] of the product to compute each entry along the diagonals. For example, for all the entries along the fourth diagonal shown in yellow in Figure 2-4, the relative shift is $(1,0)$. We compute the product $S_\gamma(x,y)S_\gamma(x-1,y-0)$ and its integral image, and then we accumulate the product over different rectangles shaded in gray to get the values for $\bar{A}_{k\gamma}(3,0)$, $\bar{A}_{k\gamma}(4,1)$, $\bar{A}_{k\gamma}(5,2)$ and so on in constant time with 3 operations for each of the elements of the integral image. This technique makes the computation of matrix $\bar{A}_{k\gamma}$, $O(m^2)$ rather than $O(m^4)$, providing a speedup of 42× for computing the correlation for a typical 13×13 kernel.

2. **Six-Parallel Correlator Architecture with Image Tiling:** Diagonal computation reuse alone cannot meet the throughput requirements of the correlator. We further increase throughput by designing a highly parallel correlator architecture, which consists of an array of 6 processing elements (PEs) as shown in Figure 2-5, each of which computes the correlation matrix elements for all absolute shifts which correspond to the same relative shift using diagonal computation reuse. The mapping of correlation computation to the PEs is shown

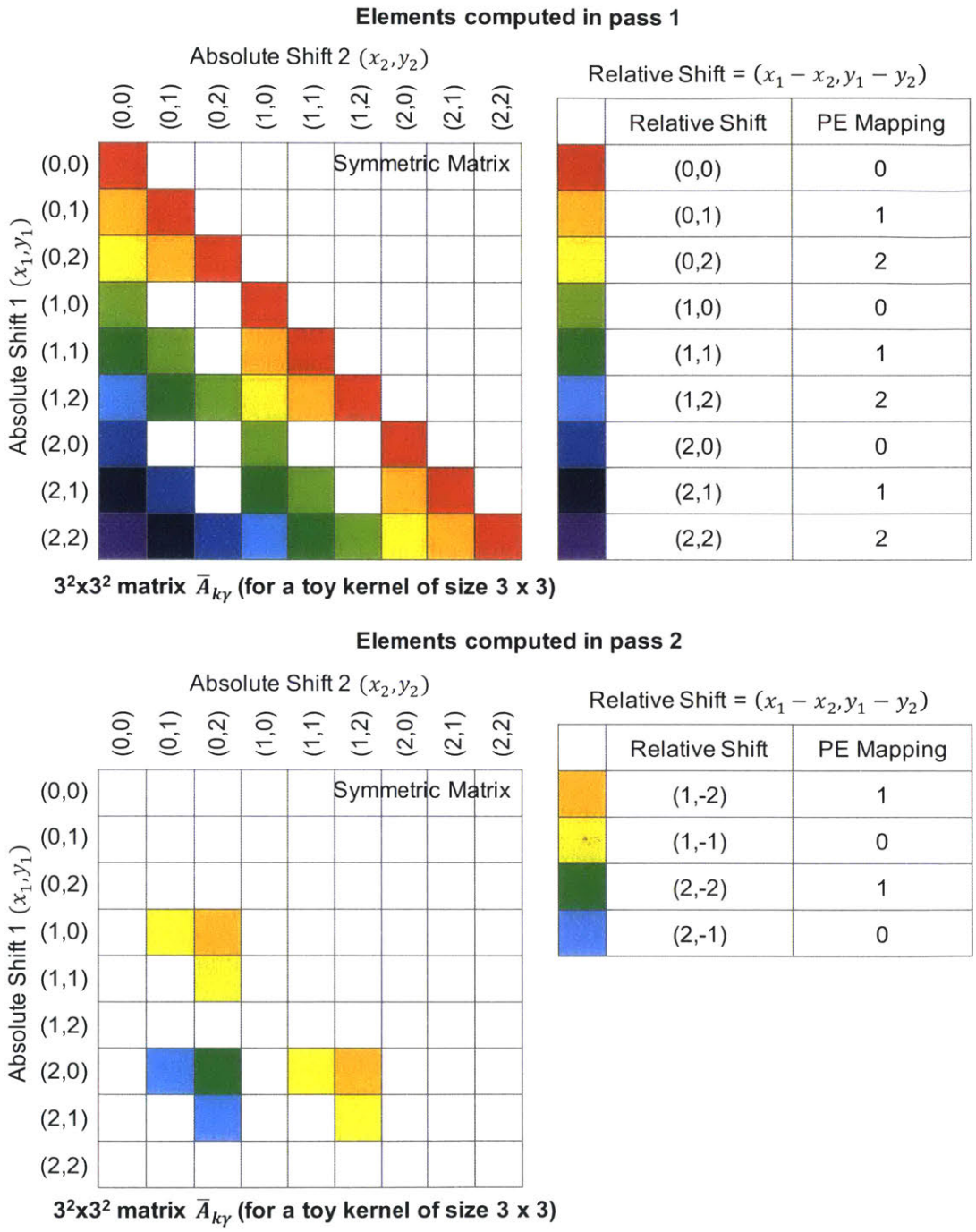


Figure 2-6: The mapping of computation of correlation matrix elements to different PEs based on the relative shifts for a toy kernel of size 3×3, which leads to a $3^2 \times 3^2$ correlation matrix. Only the lower triangular part is computed since the matrix is symmetric.

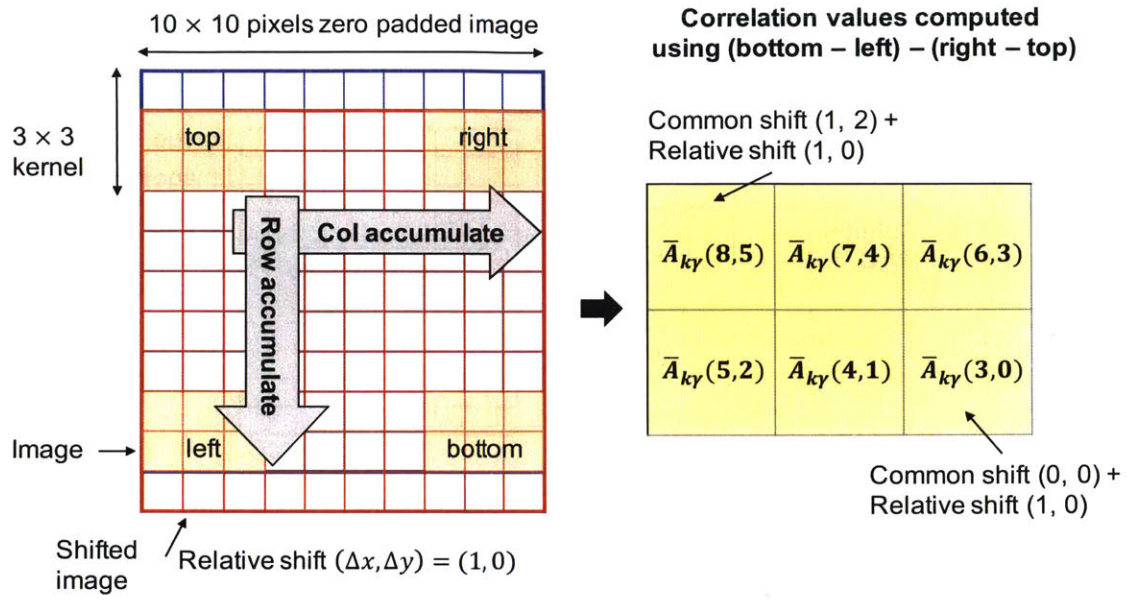


Figure 2-7: A toy example to illustrate the integral image computation followed by matrix element calculation happening inside each correlation processing element (PE).

in Figure 2-6 for a toy kernel of size 3×3. The computation happens in two passes over the image, where in the first pass, the relative shift is positive, and in the second pass the relative shift is negative.

The challenge here is how is to feed these PEs in parallel from a single image buffer. To solve this, we make the PEs work on horizontally consecutive relative shifts as shown in Figure 2-5 and feed them by adding a series of shift registers which delay one stream of image pixels. Then to read the image and the shifted image simultaneously from the image buffer, we tile the image across 4 single port SRAM banks, so that two versions of the image with any relative shift between them can be accessed in parallel. If the pixel location and the shifted location point to the same SRAM bank, the conflict is resolved by pre-fetching the next required pixel from a different SRAM bank in the current access and then toggling the data. This allows the PEs to be fed in parallel and gives a $6\times$ improvement in throughput over the design with only diagonal computation reuse.

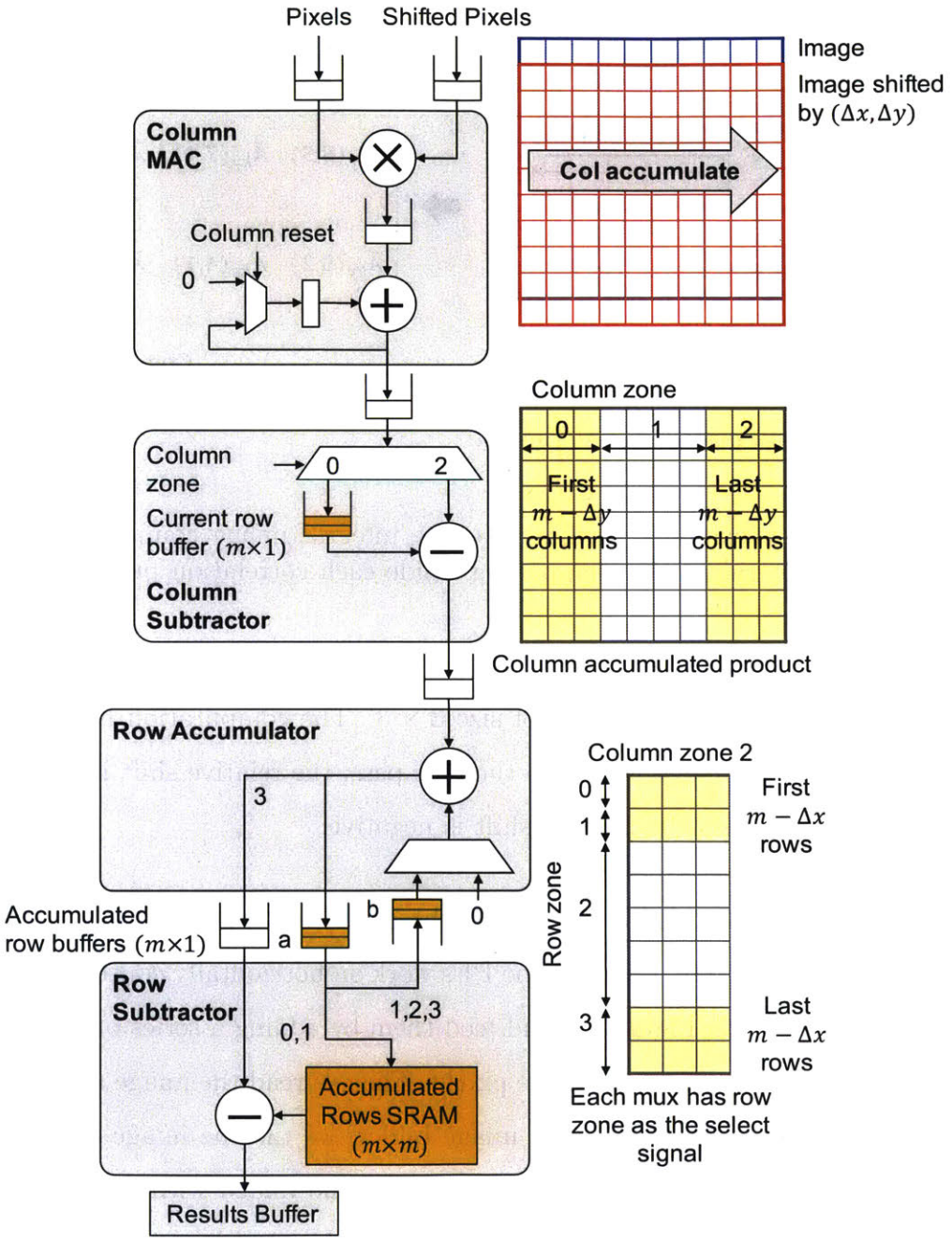


Figure 2-8: Inside the correlation PE, the pixels and shifted pixels are multiplied together and accumulated along each row by the column MAC. The first $m - \Delta y$ accumulated columns are subtracted from the last $m - \Delta y$ accumulated columns and sent to the row accumulator. Accumulation and subtraction is similarly performed across the rows.

3. **Merged Correlation PE Architecture:** Figures 2-7 and 2-8 show the architecture of a correlation PE. Each PE is initialized with the input image size ($n \times n$), the kernel size ($m \times m$) and the 2D relative shift ($\Delta x, \Delta y$) and it calculates the correlation matrix elements for that relative shift as shown in Figure 2-7. One approach for computing correlation by the PEs would be to first multiply the corresponding pixels and compute the integral image by accumulating across columns followed by rows, and then to access four elements at a time from the integral image and use them to compute each matrix element as shown in Figure 2-4. This approach is used several computer vision algorithms that require integral image computation such as real-time object detection [28], robust feature extraction [5] and surface normal calculation [15]. This approach requires at least storing the entire $n \times n$ integral image, which in the worst case is 128×128. However, in our case, we only need to access a subset of elements from the integral image, and the locations of those elements are known a priori, so we can do better. To reduce the buffering requirements, we propose a different architecture that merges the integral image computation step with the subsequent step of calculating the matrix elements. Our approach reduces the buffering requirement to $m \times m + 3 * m \times 1$, where $m = 32$ in the worst case to accommodate the largest kernel, achieving a $14.6\times$ reduction compared to the naive approach. Figure 2-8 highlights the buffers in yellow, and the paragraphs below give the processing details.

Following the initialization, the PE receives one pixel from the image and one from its shifted version every cycle. Inside the PE, the column MAC unit performs multiply accumulate across the columns for each row, and is cleared once a row is complete. At the start of each row, the first $m - \Delta y$ results from the MAC are written to the current row buffer. As the last $m - \Delta y$ results start coming out of the MAC, entries from current row buffer are read and subtracted from the MAC results by the column subtractor as shown in Figure 2-8 and results are sent to the row accumulator.

The column subtractor results are accumulated across rows by the row accumulator and results are written to accumulated row buffer (a). Accumulated row buffer (a) is copied into accumulated rows SRAM when the next row processing starts and into accumulated row buffer (b). Copying into row buffer (b) is necessary since the buffers are implemented using single port SRAMs, and in the row accumulation step we need to be able to read and write at the same time. An alternative implementation would be to ping-pong between the buffers (a) and (b) for read and write, but that would introduce extra multiplexers. After the first $m - \Delta x$ rows, the processing happens as with earlier rows with the difference that the results are not copied into the accumulated rows SRAM. For the last $m - \Delta x$ rows, row subtractor reads the corresponding rows from accumulated rows SRAM, and performs the final row difference and writes the results to the results buffer.

2.5 Selective Update Based Gradient Projection Solver

The gradient projection solver [7] takes the kernel from the previous EM iteration, and the matrix $\bar{A}_k$ and the vector $\bar{b}_k$ computed by the correlator as inputs. It computes the refined kernel by minimizing the constrained quadratic program, $J(k) = \frac{1}{2}k^T\bar{A}_k k + \bar{b}_k^T k$, such that $k \geq 0$. The constraints ensure that the kernel is a blur kernel. The solver refines the kernel by iteratively executing two sequential steps as shown in Figure 2-9:

1. Search along the steepest descent direction, that is the direction $-g$ where $g = \bar{A}_k k + \bar{b}_k$, from the current point k. Whenever a constraint is encountered, bend the search direction so that it stays feasible, and locate the first local minimizer of J along this piece-wise linear path. The minimizer is called the Cauchy point k_c.

2. Explore the face of the feasible box on which the Cauchy point lies by solving a sub-problem, in which the value of k at the indices i at which the constraints

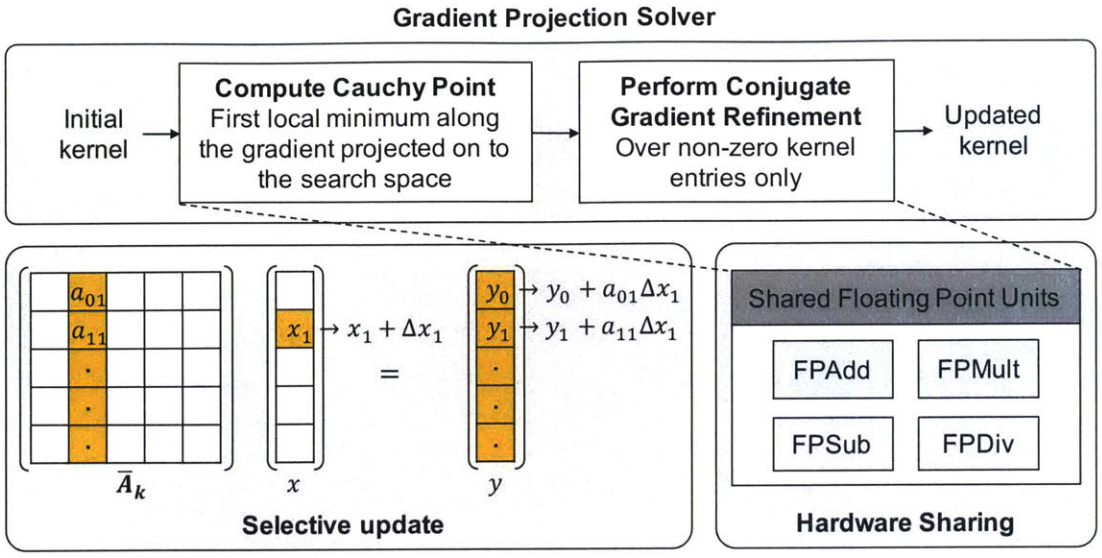


Figure 2-9: Gradient projection solver with selective update and hardware sharing.

are active (which means equal to zero in our case) are held fixed at k_i^c, using a conjugate gradient solver, for faster convergence.

At the end of each iteration, the quadratic cost function is evaluated and compared with its value from the previous iteration; if the difference is small, the kernel is taken as final.

2.5.1 Gradient Projection Solver Optimizations

We perform the following optimizations to the gradient projection solver to ensure fast convergence and reduce energy compared to [7]:

1. **Selective Update:** Matrix-vector multiplications with $\bar{A}_k$ dominate the computational time in the algorithm, as $\bar{A}_k$ can be very large (49×49 to 841×841 depending on the kernel size). However, we observe that during each iteration the vector to be multiplied with $\bar{A}_k$ updates only at a few indices with respect to its value from the previous iteration. Our selective update algorithm (Figure 2-9) converts matrix-vector multiplication to a few element-wise multiplications and additions on two vectors. For example, as shown in Figure 2-9, if the vector x updates only at index 1 with respect to its value in the previous iteration,

that is, it becomes $x_1 + \Delta x_1$ instead of x_1 in the previous iteration, we read only the second column of the $\bar{A}_k$ matrix instead of all columns to compute the new result vector y. This reduces the number of solver multiplies, adds and DRAM accesses by 11× on an average.

2. **Datapath Sharing:** We propose an architecture that shares floating point units between non-concurrent steps (Cauchy point computation and conjugate gradient refinement), resulting in 56% area savings in the solver hardware (Figure 2-9).

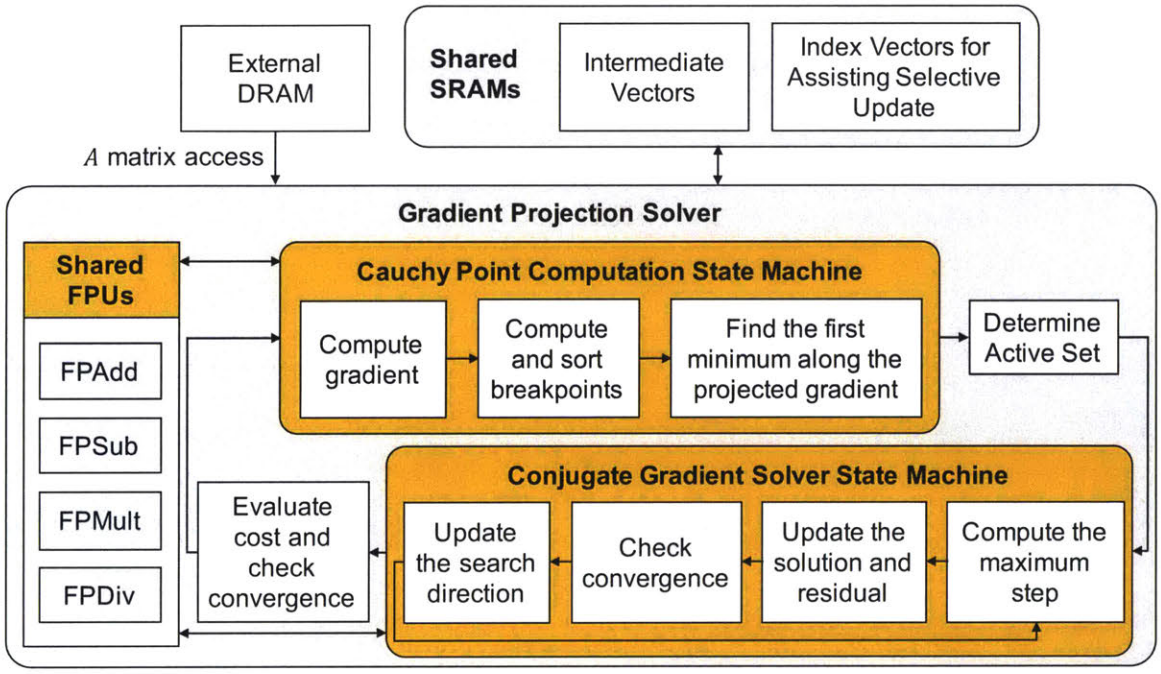


Figure 2-10: Cauchy point computation and conjugate gradient solver state machines.

2.5.2 Gradient Projection Solver Architecture

The gradient projection solver consists of schedulers for Cauchy point computation and conjugate gradient solver as shown in Figure 2-10, which are essentially hard-coded state machines that execute the steps of the algorithm on the shared floating point units. The steps that do not have a data dependency between them are executed in parallel, and the ones that do are executed sequentially. In this way the scheduler

Resources →

Time ↓

Step	Mult 0	Mult 1	Mult 2	AddSub 0	AddSub 1	AddSub 2	AddSub 3	Div 0
Update solution (k), gradient (g) and residual (r) at each index (in subscript)	αp_0	αq_0						
	αp_1	αq_1		$k_0 \leftarrow k_0 + \alpha p_0$	$r_0 \leftarrow r_0 - \alpha q_0$	$g_0 \leftarrow g_0 + \alpha q_0$		
	...	...	$r_{sq0} \leftarrow r_0 r_0$	$k_1 \leftarrow k_1 + \alpha p_1$	$r_1 \leftarrow r_1 - \alpha q_1$	$g_1 \leftarrow g_1 + \alpha q_1$		
			$r_{sq1} \leftarrow r_1 r_1$	...	...	...	$\rho \leftarrow \rho + r_{sq0}$	
			...				$\rho \leftarrow \rho + r_{sq1}$	
							...	
Compute β								$\beta \leftarrow \rho / \rho_{last}$
Update search direction (p)	βp_0							
	βp_1			$p_0 \leftarrow r_0 + \beta p_0$				
	...			$p_1 \leftarrow r_1 + \beta p_1$				
				...				

Figure 2-11: Scheduling of conjugate gradient refinement steps over shared floating point arithmetic units. Yellow blocks denote the pipeline bubbles. Three dots denote that the processing happens over all vector indices.

extracts the maximum amount of parallelism given the I/O bandwidth constraints. Figure 2-11 shows an example of the mapping of computation during three different sequential steps of conjugate gradient refinement onto the shared arithmetic units. The utilization of the arithmetic units depends on the maximum available parallelism.

The following paragraphs provide details on how the optimizations described earlier are used in context of the algorithm.

1. **Cauchy Point Computation:** To initialize the solver, we read the $m^2 \times 1$ vectors k (from the previous EM iteration) and $\bar{b}_k$, where $49 \leq m^2 \leq 841$, from the DRAM and store them in local SRAMs to avoid DRAM access latency and energy penalty in each gradient projection solver iteration. Then, we stream in matrix $\bar{A}_k$ from the DRAM in row-major order, and evaluate the gradient $g = \bar{A}_k k + \bar{b}_k$ and the cost function $k^T(\frac{1}{2}\bar{A}_k k + \bar{b}_k)$. At each index (i), we use gradient g_i to calculate the *breakpoint* t_i, which is the maximum step that one can take along the negative gradient direction starting from the current solution k_i before the solution along that dimension becomes zero (hits a constraint), that is, $t_i = k_i/g_i$. We calculate breakpoints for all indices where the gradient g_i is positive. We also store these indices in a local buffer to later assist in executing the selective update based matrix multiplication algorithm described earlier.

   Once all breakpoints are computed, we sort the unique breakpoints in ascending order using a merge sort based approach using a single floating point comparator. For each distinct pair of breakpoints t^j and t^{j+1} in this sorted list:

   (a) We compute the projected gradient p where, $p_i = -g_i$ when $t_i > t^j$ and 0 otherwise.

   (b) Starting at the current k, the objective function till the next breakpoint can be written as a function of the step size Δt in the direction of the projected gradient p as $J(k + \Delta t p) = f_0 + f_1(\Delta t) + \frac{1}{2} f_2(\Delta t)^2$ where $f_1 = \bar{b}_k^T p + k^T \bar{A}_k p$ and $f_2 = p^T \bar{A}_k p$. We minimize this objective function with respect to the step size Δt, to get the optimal $\Delta t^* = -f_1/f_2$. The most

computationally expensive operation here is matrix multiplication of $\bar{A}_k$ with p for computing f_1 and f_2. However, we observe that the projected gradient p updates only at a few indices at a time, depending on the number of breakpoints t_i that are less than or equal to t^j in each interval, and we use the selective update technique described earlier to reduce the number of computations while calculating $\bar{A}_k p$.

(c) If Δt^* is ≥ 0 and $< (t^{j+1} - t^j)$ and the second derivative $f_2 > 0$, then it is the local minimum and it lies on the current segment. In this case, we update the Cauchy point $k_c = k + \Delta t^* p$ and proceed to conjugate gradient refinement. Otherwise, if $f_1 > 0$, the minimum is at the boundary. In this case, $k_c = k$ and we proceed to conjugate gradient refinement. Otherwise, we update k to $k + (t^{j+1} - t^j)p$ and return to step (1) and examine the next pair of breakpoints.

2. **Active Set Determination:** Once we have the Cauchy point k_c, we compute the updated gradient g for the next step and find the set of non-active indices, that is, indices at which k_c is non-zero. The conjugate gradient refinement is then run on the non-active set only. We store the non-active set indices in a run-length encoded format, so that DRAM requests for the coefficients of the $\bar{A}_k$ matrix can be issued in bursts rather than one at a time, thus reducing the amount of time spent waiting for DRAM reads. In practice, only 50% of the indices are non-active on an average, which makes the matrices and vectors within the conjugate gradient loop 50% smaller, leading to correspondingly fewer DRAM accesses and FP operations.

3. **Conjugate Gradient (CG) Refinement:** It solves the linear system $\bar{A}_{k,na} k_{na} = -g_{na}$, where $\bar{A}_{k,na}$, k_{na} and g_{na} denotes are the respective matrices and vectors at the non-active (na) indices, with a check to restrict the step length in case the solution violates any constraints. The initial value for k_{na} is the Cauchy point k_c at non-active indices ($k_{c,na}$). The refinement step typically runs for 10 - 15 iterations till it converges to the minimum. The conjugate gradient state

machine sequences the following steps on the shared floating point units:

(a) Initialize the residual r_{cg} and the search direction p_{cg} equal to $r_{cg} = p_{cg} = -\bar{A}_{k,na}k_{na} - g_{na}$. Also, initialize the residual norm $\rho = r_{cg}^T r_{cg}$.

(b) Repeat the following steps until convergence or until a new constraint is activated:

i. First, compute the step size $\alpha = \rho / p_{cg}^T q_{cg}$ where $q_{cg} = \bar{A}_{k,na} p_{cg}$. Computing q_{cg} is the most computationally intensive step and it is performed using the non-active sub-matrix multiplication as described earlier. Then, determine the maximum allowable step size in each dimension based on the constraints $\alpha_{max}^i = -k_{na}^i / p_{cg}^i$. Choose the final step size $\alpha^* = min[\alpha, min_i(max(0, \alpha_{max}^i))]$.

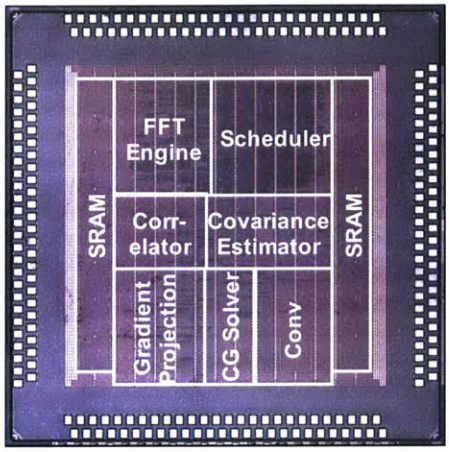
ii. Update the solution $k_{na} = k_{na} + \alpha^* p_{cg}$. In addition, update the gradient as well, incrementally, by reusing the q_{cg} computation, rather than waiting until the end of conjugate gradient refinement and reading the complete $\bar{A}_k$ matrix to compute the updated gradient, $g_{na} = g_{na} + \alpha^* q_{cg}$. If during α computation, a constraint is hit then exit the loop.

iii. Compute the new residual $r_{cg} = r_{cg} - \alpha^* q_{cg}$ and check for convergence by comparing the norm of the new residual $\rho = r_{cg}^T r_{cg}$ with the norm of the last residual ρ_{last}. If this difference is smaller than the tolerance, exit the loop.

iv. Compute a step size for the search direction update $\beta = \rho / \rho_{last}$, update the search direction $p_{cg} = r_{cg} + \beta p_{cg}$, and return to step (a).

4. **Convergence Checking:** After the completion of CG refinement, we have the updated solution k at all indices, but the updated gradient g at only the non-active indices. To compute the updated gradient at the active indices, we again use the selective update based matrix multiplication algorithm, where we read in only a sub-matrix of $\bar{A}_k$ at the active rows and non-active columns and multiply it with the change in the solution at the non-active indices, to get the

change in the gradient at the active indices. Since in the average case only 50% of the indices are non-active, this results in DRAM accesses and FP operations for only 25% of the $\bar{A}_k$ matrix. Once we have the updated gradient, we also compute the new cost without re-reading the $\bar{A}_k$ matrix, using $J = \frac{1}{2}k^T(g + \bar{b}_k)$. We compare this cost with its value from the previous iteration, and if the difference is smaller than some tolerance, the solution k is taken as final, else the two steps are repeated.



Chip Features

Technology	40 nm CMOS
Gate Count	2728 kgates
Core Area	2.1 mm x 2.1 mm
SRAM	272 kBytes
Core VDD	0.9 V
I/O VDD	2.5 V
Frequency	25 - 83 MHz
Core Power	59.5 mW (0.9 V, 83 MHz)
Kernel Size	7 x 7 to 29 x 29 pixels

Figure 2-12: Die photo and chip features.

2.6 Results

The accelerator is fabricated in 40 nm CMOS technology. The chip micrograph and features are shown in Figure 2-12 and the test setup is shown in Figure 2-13. Logic utilization and power breakdown of the chip are shown in Figure 2-14 and Figure 2-15. Figure 2-16 shows the deblurring results for test images of size 1920×1080 and two different kernel sizes. It can be seen that the determined blur kernel successfully deblurs the input blurred images on the left.

2.6.1 Runtime and Energy Reduction

The accelerator achieves runtime reduction by 78× for kernel estimation with respect to our realization of [18] (which incorporates all the software only algorithmic opti-

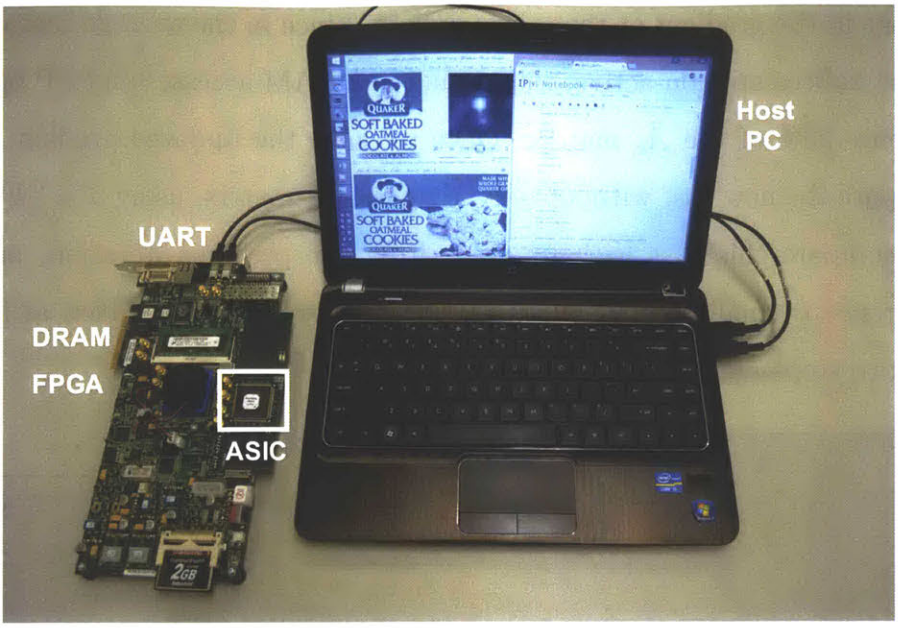


Figure 2-13: Test setup for image deblurring accelerator. The chip is connected to Virtex-6 FPGA on Xilinx ML605 development board. The estimated kernel and the deblurred Full HD image (1920 × 1080) are displayed on the host PC.

mizations that we have proposed) on an Intel i5 CPU for a 13 × 13 kernel, and by 56× for complete deblurring of a 1920 × 1080 image, with final deconvolution performed on the CPU (Table 2.1). Compared to our realization of [18] on a mobile processor (Cortex-A15), it achieves a 480× reduction in kernel estimation time. Also, compared to a related work [12] which implements deblurring on a GPU, this work achieves a substantial reduction in kernel estimation time. Our kernel estimation time is independent of image size, while deconvolution time scales linearly.

The accelerator consumes 105 mJ for the above test case at the nominal operating point (0.9 V, 83 MHz) compared to 467 J consumed by the CPU (measured using Intel PowerTOP), and 2284 J consumed by the mobile CPU (measured using on-board energy monitors). At the minimum energy point at (0.67 V, 38 MHz), the energy consumption is 33% lower than at nominal (Figure 2-17), which can be used for batch processing of blurred images in cell-phones.

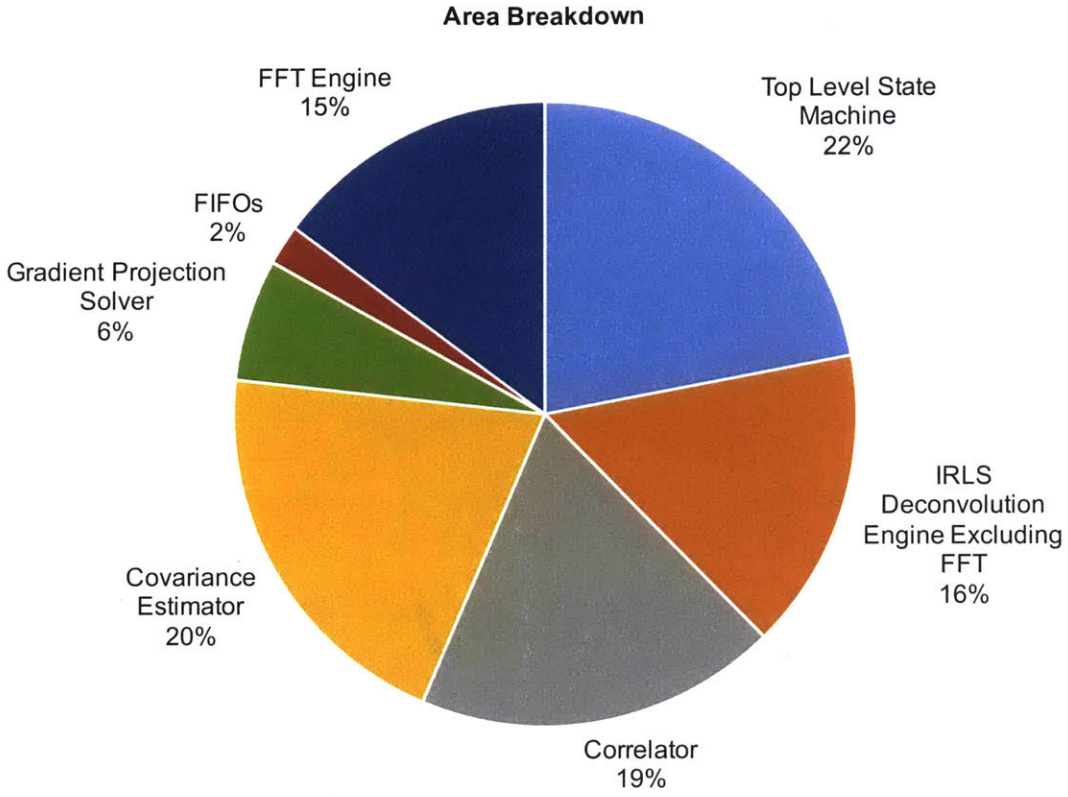


Figure 2-14: Logic utilization for each processing block (total 2728 kgates).

2.6.2 Energy Scalability

The number of EM iterations can be tuned to trade off image quality with runtime, achieving 10× energy scalability from 11 mJ to 105 mJ as shown in Figure 2-18. If coarse camera motion is available through inertial sensors, it can be used to initialize the kernel, leading to fewer iterations and lower energy. Similarly, for deblurring videos, the kernel estimated from one frame can be used as a starting point for the next frame. The processor also allows configurability in kernel size from 7×7 to 29×29 which leads to quadratic variation in energy (Figure 2-19). In an energy constrained scenario, one can down-sample the input image before selecting a 128×128 patch, use the accelerator to estimate a kernel of a smaller size, and then up-sample the kernel to do deconvolution on the entire image, e.g., if the true kernel is 13×13, down-sampling the input image by a factor of 2 in each dimension allows using a 7×7 kernel, leading to a 2.15× reduction in energy as shown in Figure 2-19.

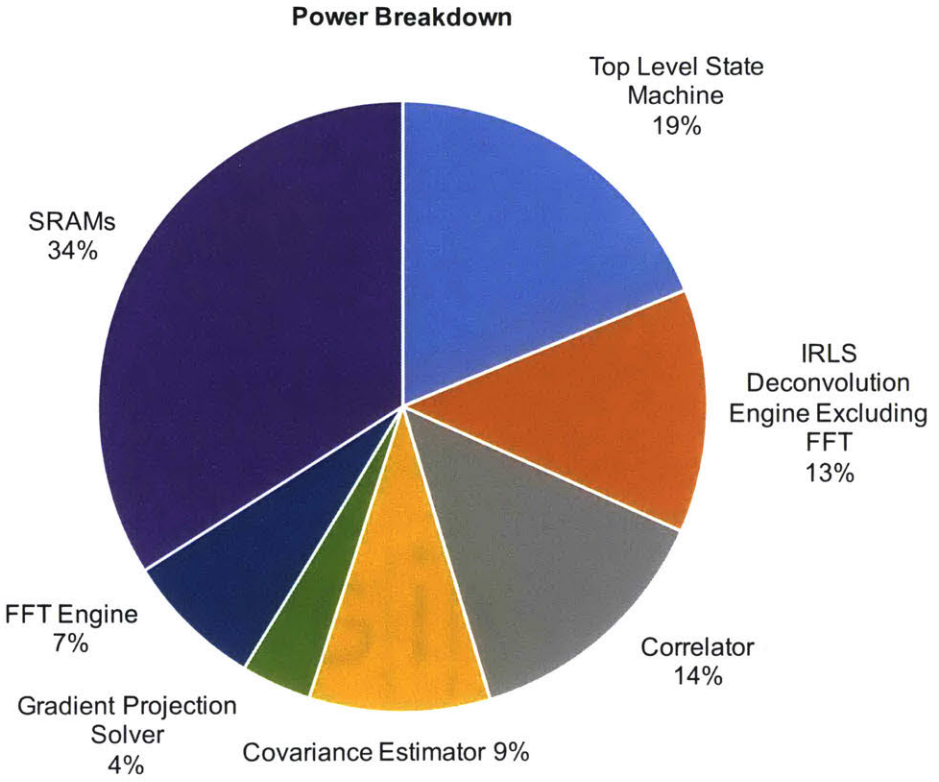


Figure 2-15: Total power breakdown for logic blocks and memory (total 59.5 mW).

2.7 Conclusion

In this chapter of this thesis, we presented the first hardware accelerator for kernel estimation in image deblurring applications. It features

1. A multi-resolution IRLS-based deconvolution engine with DFT based matrix multiplication which achieves at least 8.8× reduction in the number of floating point operations.

2. A highly parallel image correlator with diagonal computation reuse and image tiling which achieves a speedup of two orders of magnitude over the baseline.

3. A selective update based gradient projection solver which achieves 11× increase in speed and 56% reduction in area compared to the baseline.

These techniques result in a 78× reduction in kernel estimation time, and a 56× reduction in the total deblurring time of 1920 × 1080 images with respect to a CPU,

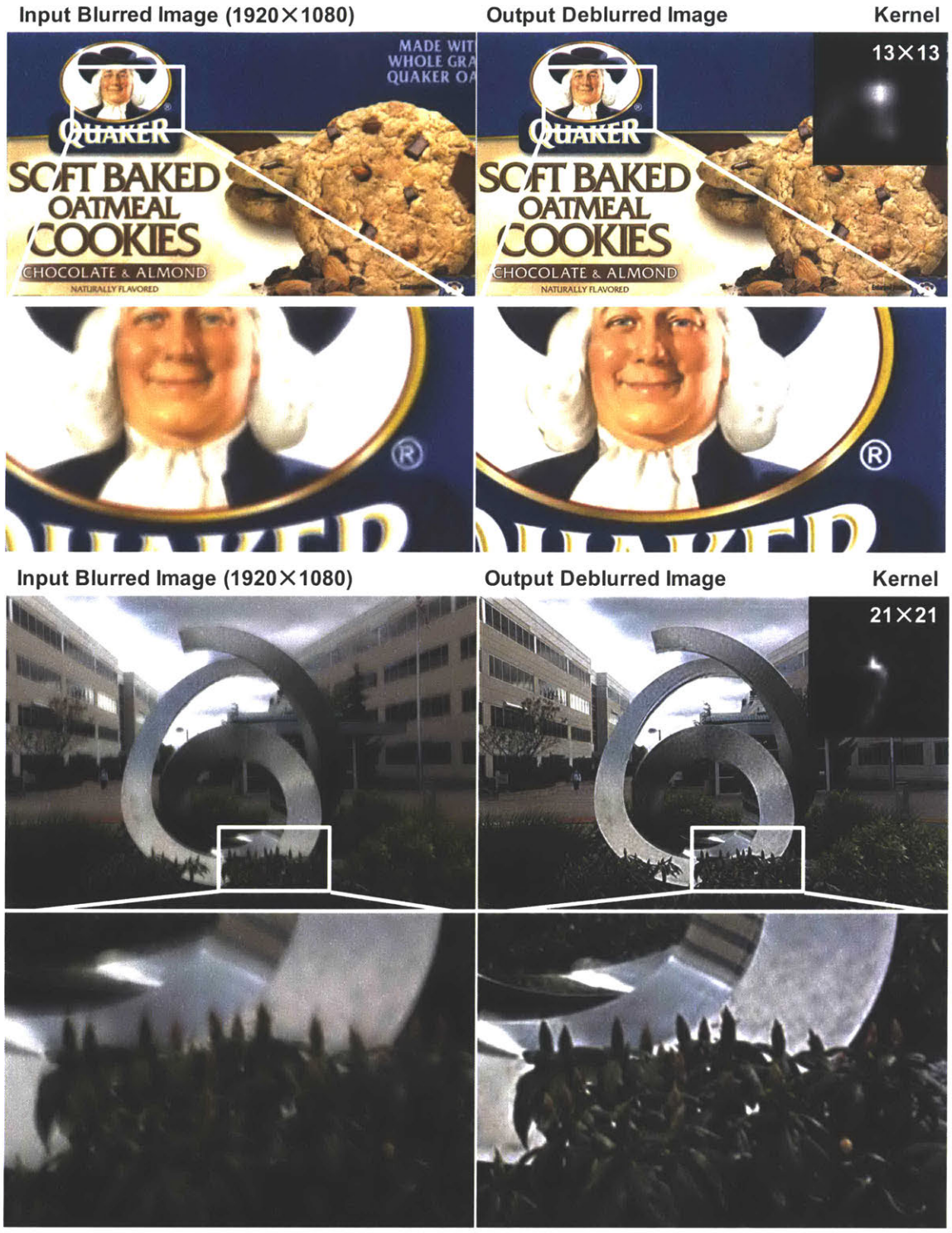


Figure 2-16: Test 1920×1080 blurred image, output kernel of size 13×13 (top) and 21×21 (bottom) and deblurred image.

Table 2.1: Comparison with state of the art algorithms on different platforms

	Size			Time (s)			Energy (J)
Algorithm and Platform	Kernel Size	Patch Size	Image Size	Kernel Estimation	Final Deconvolution	Full Deblurring	Kernel Estimation
This Work (based on [18] with Accelerator + CPU)	13 × 13	128 × 128	1920× 1080	1.70	0.75	2.45	0.105
[18] on Intel Core i5	13 × 13	128 × 128	1920× 1080	134.00	0.75	134.75	467.000
[18] on Samsung Exynos 5422 Cortex-A15	13 × 13	128 × 128	1920× 1080	816.00	-	-	2284.800
[12] on GPU	15 × 15	-	441 × 611	169.70	0.80	170.50	-

and three orders of magnitude reduction in energy. The accelerator supports up to 10× energy scalability through configurability in iterations and kernel size, allowing the system to trade off runtime with image quality in energy-constrained scenarios. This energy-scalable implementation enables efficient integration of image deblurring into mobile devices.

2.8 Future Work

There are several possible directions in which this project can be extended, for example,

- The current algorithm assumes a spatially invariant blur, that is, the same blur kernel is used for deblurring different parts of the image. While this assumption is approximately valid in many common camera shake scenarios, it is not valid in general. One possible direction for future work would be to extend this work to handle spatially variant blur, by determining blur kernels from several patches in a single image, using them to deblur the image locally, and then reconstituting

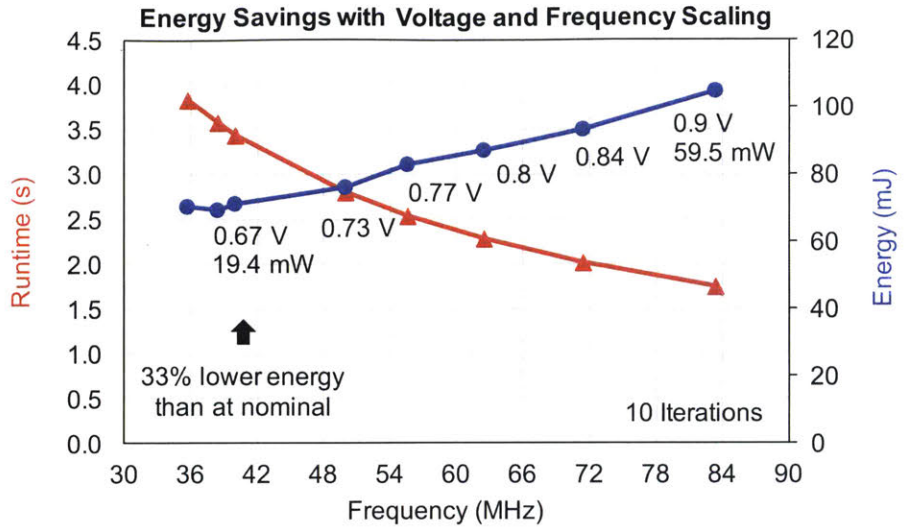


Figure 2-17: By employing voltage and frequency scaling, the system obtains the minimum energy point at (0.67V, 38MHz), where the energy consumption is 33% lower than at nominal, and can be used for batch processing.

the final deblurred image. Since the accelerator determines the kernel from one patch at a time, it should be portable to this multi-patch scenario.

- Another possible extension would be to evaluate the energy scalable nature of this accelerator while deblurring a video. It would be interesting to measure how quickly the algorithm converges if the kernel for one frame is initialized to the final kernel from the previous frame, and measure the resulting energy savings.

- Numerical optimization is required in several computer vision algorithms such as visual odometry and video super-resolution. One extension could involve porting and evaluating the architecture of the numerical optimizers (conjugate gradient and gradient projection) proposed in this work in the context of these new applications.

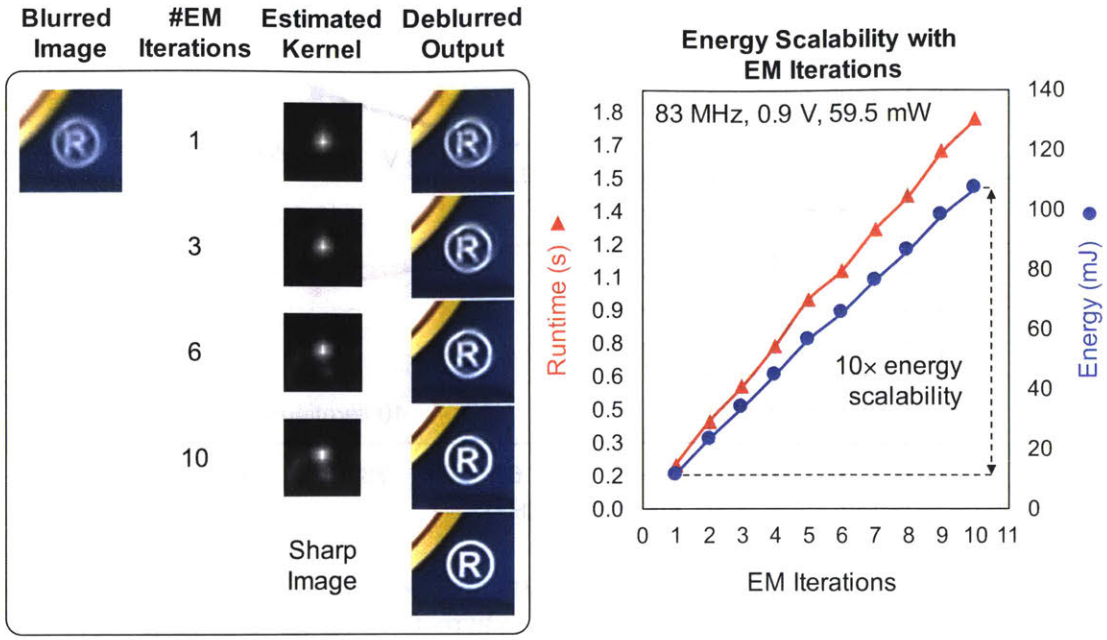


Figure 2-18: Number of EM iterations can be tuned to trade off image quality with runtime giving 10× energy scalability.

2.9 Acknowledgements

We would like to thank Foxconn Technology Group for sponsorship, TSMC University Shuttle Program for chip fabrication, and Professor Frédo Durand and Professor William T. Freeman for valuable feedback. We would like to thank Dr. Mehul Tikekar for contributing to the design of the demonstration platform for this work.

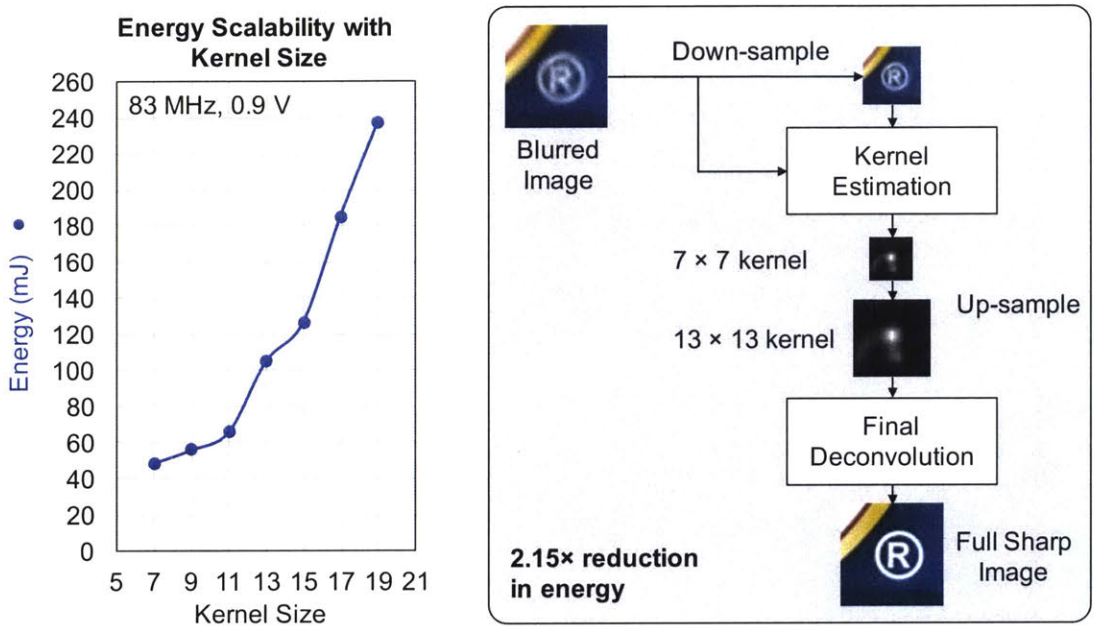


Figure 2-19: Kernel size can also be tuned to achieve energy scalability.

Chapter 3

Motion Magnification Processor

3.1 Motivation

Over the past few centuries, microscopes have revolutionized our world. They reveal to us a tiny world of objects, life and structures that are too small for us to see with our naked eyes. Recent research in computer vision [30, 29, 31] has led to a new class of algorithms that can magnify tiny motions, or in other words, can act as motion microscopes. They do not use optics like a regular microscope to make small objects bigger, but instead use a video camera and image processing to reveal to us the tiniest motions and color changes in objects and people - changes that are impossible for us to see with our naked eyes. These motion magnification algorithms work by analyzing the changes in light at every pixel that are caused when objects move, and by making those changes bigger while carefully separating them from the noise that exists in images. The applications of this technology include, but are not limited to, visualization of blood perfusion [31], contact-less health monitoring by extraction of vital signs from motion magnified video, baby monitoring, analysis of vibrations in mechanical systems and structures [8] and remote sound recovery [10].

However, state of the art motion magnification algorithms are extremely computationally intensive and do not achieve real-time performance on modern CPUs. Reported results [30] show that the fastest algorithm can only achieve a frame rate of 2.37 frames per second (FPS) on FullHD (1920×1080) and 5.33 FPS on HD

(1280 × 720) video when run on an Intel Xeon E5-2623 desktop CPU and only 0.23 FPS on FullHD and 0.52 FPS on HD video on an ARM Cortex-A15 mobile CPU. This implies that there is a 6× to 130× performance gap that needs to be bridged to achieve real-time performance on HD/FullHD video.

Moreover, the high computational complexity leads to high energy consumption which makes the algorithm unsuitable for implementation on battery-operated portable devices like cell phones and tablets. The motion magnification algorithm while running on an ARM Cortex-A15 mobile processor consumes $10.36\mu J$ per pixel, where as video compression on mobile phones takes only $2nJ$ per pixel with dedicated hardware, therefore, four orders of magnitude reduction in energy per pixel is required to enable energy-efficient implementation of motion magnification on mobile devices.

We envision a world where motion microscopes would be as accessible to users as are cameras today, and they would be used in real-time for applications such as non-invasive breathing rate monitoring in battery-operated baby monitors and measuring fluid depth and velocity using handheld devices [32]. In this work, we design a dedicated processor that can be integrated with a cellphone camera to perform real-time motion magnification. This would enable efficient integration of motion magnification technology into portable devices.

The next section starts with a brief description of the phase-based motion magnification algorithm pipeline and then outlines the processor architecture and the major circuit blocks that we implement to realize the motion magnification processor. For each circuit block, we present the area and power simulation results in TSMC 40 nm LP process. The power numbers are for a clock period of 3.3 ns and a supply voltage of 0.99 V.

3.2 Processor Architecture

The motion magnification processor performs phase-based motion magnification based on the algorithm proposed by [30, 29], which manipulates motion in videos by analyzing the signals of local phase over time in different spatial scales and orientations.

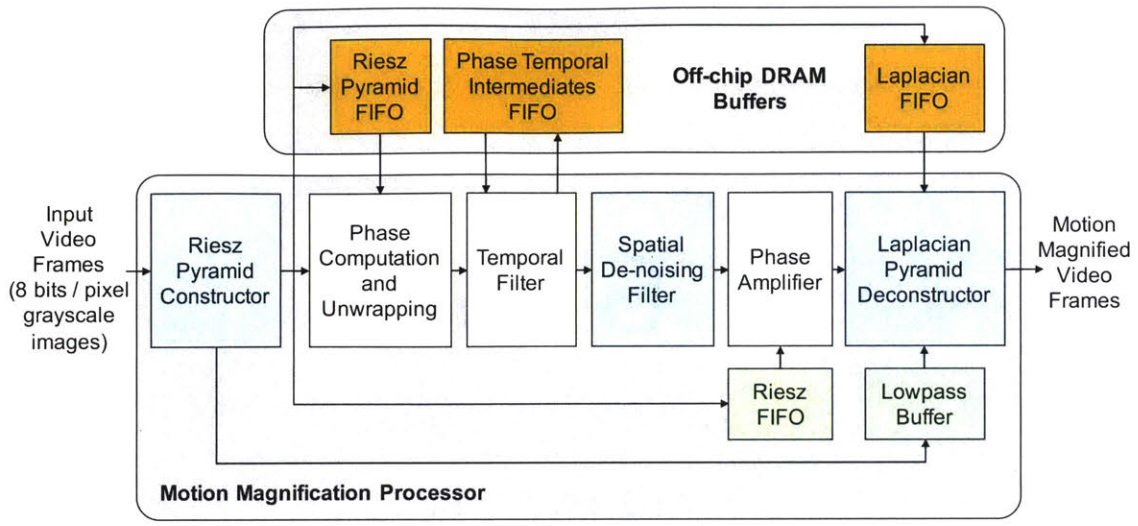


Figure 3-1: Motion magnification pipeline.

Figure 3-1 shows the processing pipeline of the processor. It uses a Riesz pyramid to decompose the video into several local sinusoids/wavelets with different spatial frequencies and separate the amplitude of these wavelets from their phase. It then performs temporal filtering of the phases independently at each location, orientation and scale. Temporal filtering isolates the motions in the frequency bands of interest, for example in 1 - 2 Hz band for breathing rate monitoring. It is followed by spatial smoothing to reduce noise in the phase, which improves the final magnification results. The processor amplifies or attenuates the temporally band-passed phases and reconstructs the video by inverting the pyramid. The following subsections describe the architecture of each stage in the processing pipeline in greater detail.

3.2.1 Color-Space Convertor

The first step in the processing is color-space conversion which is performed off-chip. This takes in each RGB pixel (8 bits per pixel per channel) from the input video frame and converts it into a YIQ pixel using a simple matrix multiplication:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{3.1}$$

Motion magnification is performed only on the Y (luminance) channel by the accelerator. Doing this conversion off-chip reduces the I/O bandwidth used by the pixels from 24 bits to 8 bits for both the input and the output images. This is important because the on-chip implementation of the motion magnification accelerator is I/O limited. The motion magnified luminance is finally combined with the IQ (chrominance) channels of the original video, and passed through inverse color space conversion to get the output RGB video:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.106 & 1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix} \tag{3.2}$$

3.2.2 Riesz Pyramid Constructor

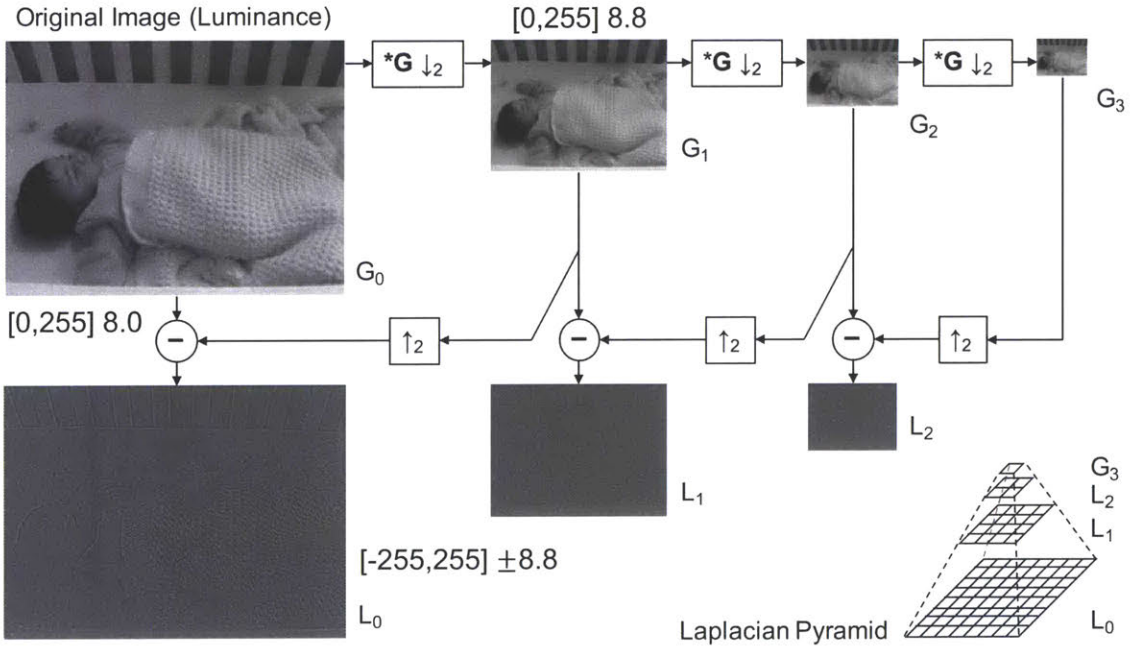


Figure 3-2: Laplacian pyramid computation from luminance frame. Here, G denotes a 5×5 2D Gaussian convolution kernel, and it is used for smoothing before downsampling by a factor of 2 in each dimension.

Image pyramid (Figure 3-2) is the basic structure for multi-resolution image processing used in a large number of computer vision algorithms. The pixels of an image

in the pyramid are recursively processed and up-sampled or down-sampled to create an increasingly finer or coarser image for analysis.

Riesz pyramid constructor takes each input luminance frame and creates a Riesz pyramid from it. Each pixel in the Riesz pyramid is a triplet (L, R1, R2) where L is the Laplacian, and R1 and R2 are the x and y derivatives of the Laplacian. Figure 3-2 shows the algorithm for computing the Laplacian pyramid from the luminance. It takes the input luminance image, blurs it by convolving it with a 5×5 Gaussian kernel and down-samples it by a factor of 2 in each direction. The result is an image which is half the size in each dimension. Then it takes this down-sampled image, up-samples it by 2 by inserting zeros after alternate pixels in both directions and smoothing the result with a 5×5 Gaussian filter, and subtracts it from the original. The results of this subtraction are the high frequency details, the Laplacian, of the image. Then this processing is repeated recursively on the output image to get the lower frequency components. At the end we get a pyramid of images, which is a decomposition of the original image into localized sinusoids/wavelets of different frequencies. Once we have the Laplacian pyramid we can take its x and y derivatives to get the Riesz pyramid.

Figure 3-3 shows the hardware architecture of the Riesz pyramid constructor which consists of 6 pyramid levels operating in parallel. The outputs of the finer levels are the inputs to the coarser levels. Figure 3-4 shows the architecture of each level. The input image to the level is first blurred with an anti-aliasing Gaussian filter, G and then down-sampled by 2 to give the output image for the next level. Additionally, the down-sampled image is up-sampled, which means that we first introduce a zero after each pixel in both dimensions and then smooth it out with an interpolation filter, which is also a Gaussian. This is subtracted from the original image (buffered in the luminance buffer), to get the Laplacian. The Laplacian needs to be buffered in the Laplacian buffer, so that derivative filters D_x and D_y can operate on it to generate

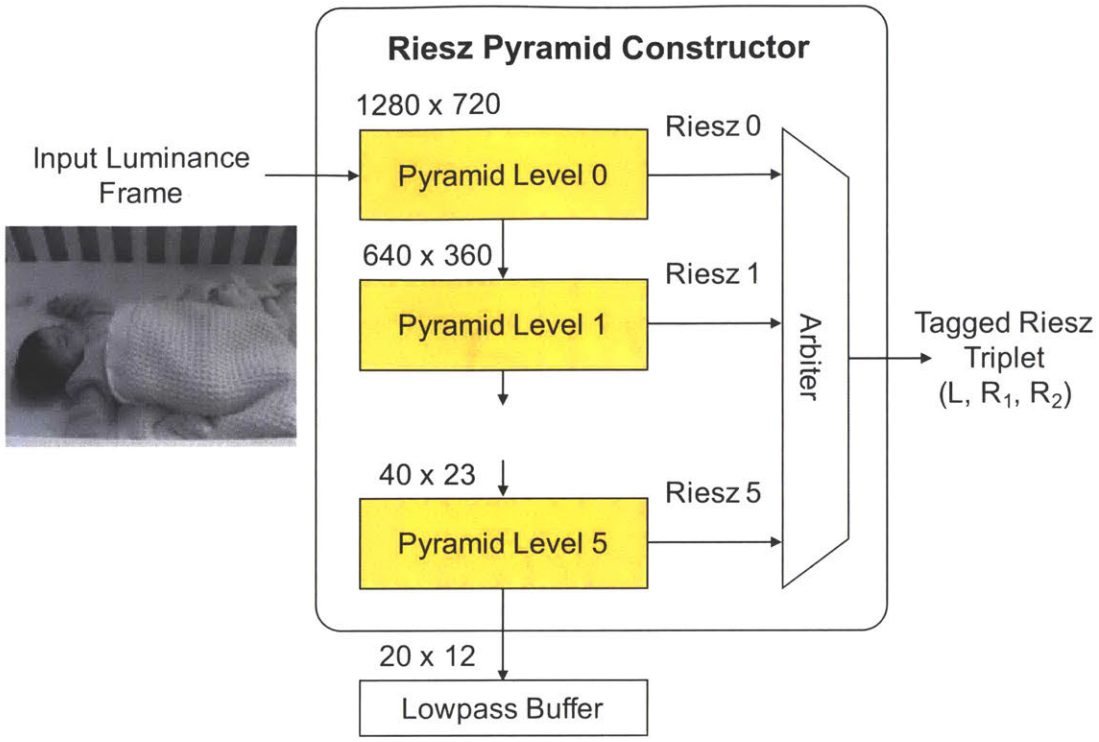


Figure 3-3: Architecture of Riesz pyramid constructor.

the final Riesz triplet.

$$D_x = \begin{bmatrix} -0.5 & 0 & +0.5 \end{bmatrix} \tag{3.3}$$

$$D_y = D_x^T \tag{3.4}$$

Separable filtering with reordering Now we are going to look more carefully at the filtering operations happening at each pyramid level. The easiest way to implement such filtering is by using a 2D filter which, in this case, is a 5×5 Gaussian kernel, G:

$$G = \frac{1}{256} * \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{3.5}$$

In this case, the filter is slid over the image in row or column major order and in

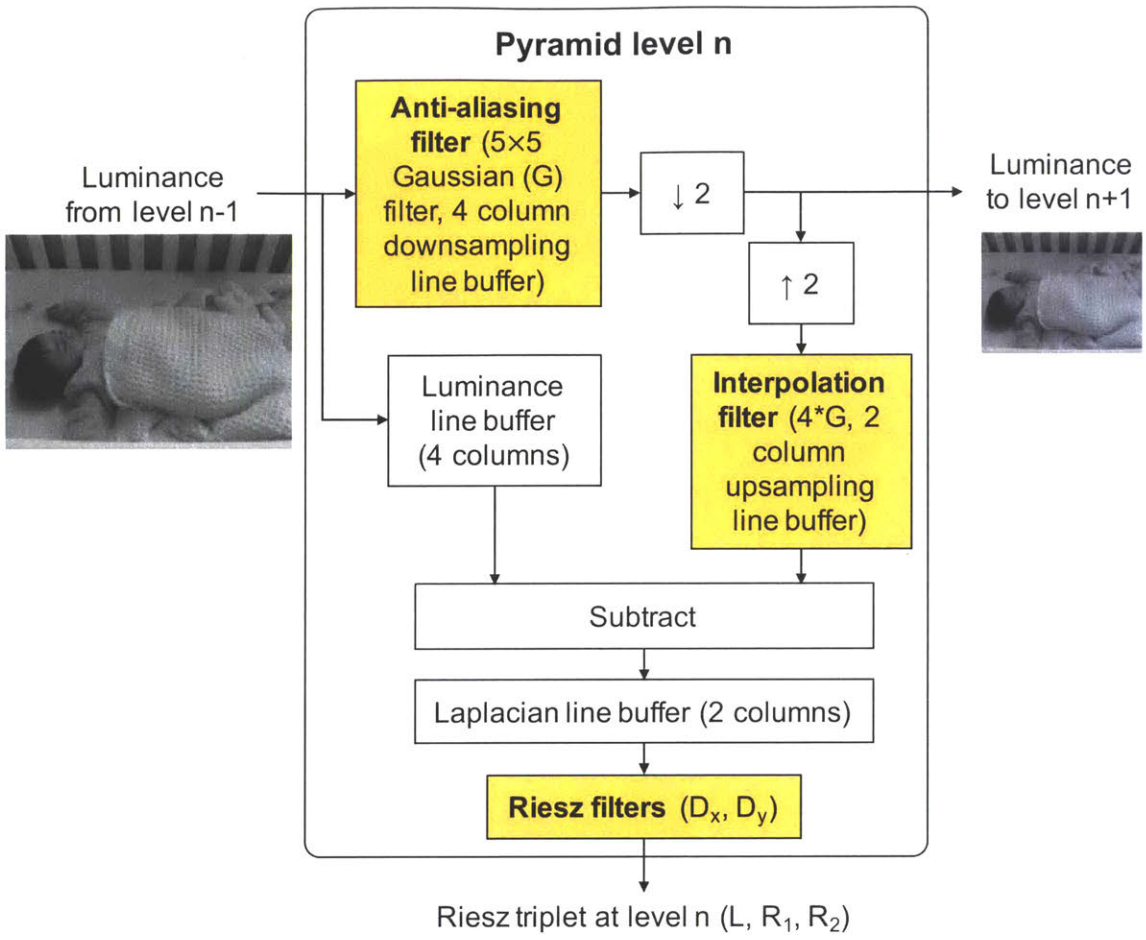


Figure 3-4: Architecture of each level of the Riesz pyramid constructor.

the area in which it overlaps with the image, the corresponding pixels are multiplied with the filter coefficients and added to get the output at the center of the filter. This leads to 21 constant multiplies and 24 adds for computing one output pixel. However, in this case we can do better because of the nature of the filter. A 2D Gaussian filter is separable, which means that you can achieve 2D filtering by applying the 1D filters, G_y and G_x, along columns followed by rows (or vice versa).

$$G_x = \frac{1}{16} * \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} \tag{3.6}$$

$$G_y = G_x^T \tag{3.7}$$

This change leads to 6 constant multiplies with smaller constants and 8 adds per output pixel for both column and row filtering. We can, in fact, do even better by

noticing that the filter is symmetric. For the coefficients that are the same, we can add the corresponding pixels first and then multiply with the coefficient. This leads us to 4 constant multiplies and 8 adds. In total, separable filtering with reordering in Riesz pyramid computation reduces the number of multiplies by 5.25× and number of adds by 3×.

Buffer area reduction using zero-skipping Since we want to read the input pixels only once from the external memory to save energy and time, we buffer them locally. For column filtering, we buffer 4 pixels in registers and when we read the fifth pixel we can compute the filter output at the third pixel. Now the output of column filtering is produced column major, so we need to buffer 4 columns before we can apply the row filter. These columns are buffered in 4 banks of an SRAM-based line buffer, where the banks can be read in parallel. Given this, let us look at the buffering requirements for the entire pyramid level. The anti-aliasing and interpolation filters are both separable 5 point Gaussian filters, so they need buffering of 4 columns. The D_x Riesz filter is a 3-point derivative filter, so it needs buffering of 2 columns. Since there is a latency till the output of the interpolation filter starts to appear, we need to buffer the original input pixels in the luminance buffer and this takes up 4 more columns, making the total buffer requirement 14 columns.

We can reduce the buffering requirements to 11 columns using the following techniques:

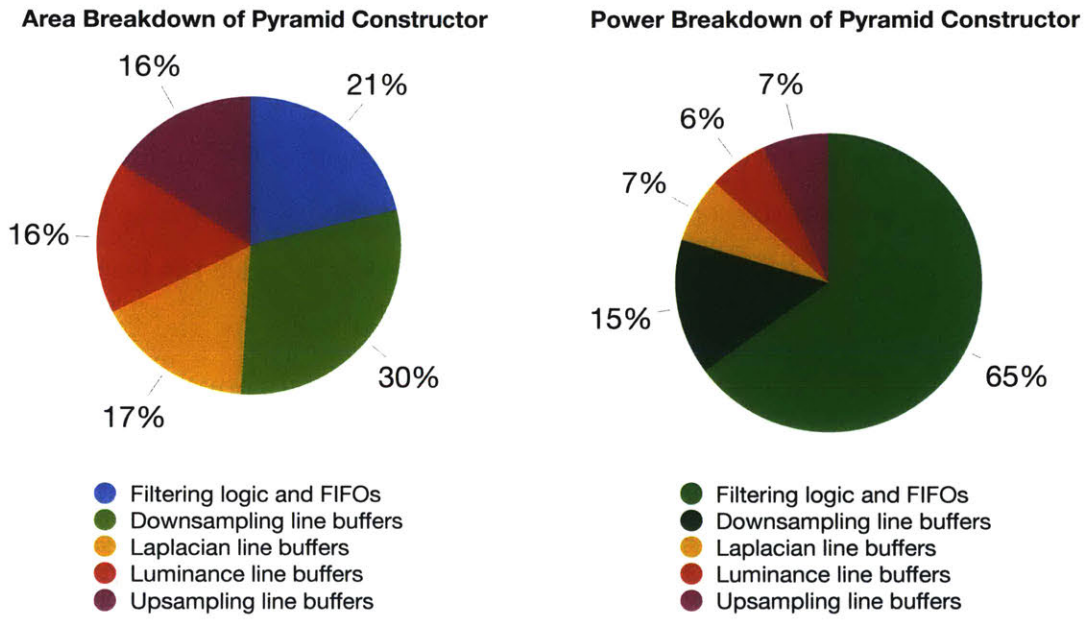
- Column first rather than row first filtering reduces the buffering requirement by 43% over row first filtering.
- Interpolation filter needs to store only half of the samples after column filtering since every alternate column is a column of zeros. By skipping the zero samples, we reduce the total buffering requirement of the Riesz pyramid constructor by another 16%.

Riesz pyramid level pipeline precision Incoming luminance pixels are unsigned 8 bit numbers (which we denote as U8.0, where U means unsigned, and 8.0 means

8 integer bits and 0 decimal bits). The separable 5 point Gaussian filter in the pyramid constructor has coefficients $[1, 4, 6, 4, 1]$, this filtering effectively adds 4 bits after column filtering and 4 more bits after row filtering, a total of 8 decimal bits (U8.8). Laplacian calculation results in addition of a sign bit to these 16 bit numbers, so L becomes 17 bits (S8.8, where S means signed). Riesz calculation results in subtracting two Laplacians, which adds 1 more decimal bit, but we truncate that least significant bit and keep 17 bits for each Riesz component.

For the next level of the pyramid, the input is 16 bits, we could do full precision computation which will add 8 more decimal bits, but instead we truncate the filtered values to 16 bits. To use the same RTL specification for each level, we add 8 zeros to the right of the luminance values, to make them unsigned 16-bit numbers, and then feed them into the first level of the pyramid constructor. We implement a right shift by 4 after each filtering operation. The width of the column buffer in the first pyramid level, however, is kept at 12 bits rather than 16 bits, since the last 4 bits are guaranteed to be zero, and the width of the luminance buffer for the first pyramid level is kept at 8 bits rather than 16 bits, since the last 8 bits are guaranteed to be zero. The buffer sizes are summarized in Figure 3-6. Each buffer is implemented using a two-port register file which allows a read and a write to occur simultaneously.

Area and power breakdown The total area of pyramid constructor is $486854\mu m^2$ and its breakdown is shown in Figure 3-5a. The total power consumed by the phase calculator engine is $9.86mW$ and its breakdown is shown in Figure 3-5b. While the line buffers consume 79% of the area, they consume 35% of the power. 65% of the power is consumed by the filtering logic and the FIFOs.



(a) Area breakdown of pyramid constructor. (b) Power breakdown of pyramid constructor.

Figure 3-5: Area and power breakdown of pyramid constructor.

Type	Banks	Level	Height	Width per bank (bits)	SRAM Kbits
Downsampling line buffer	4	0	720	12	33.75
		1	360	16	23.00
		2	180		12.00
		3	90		6.00
		4	45		3.00
		5	23		1.50
Upsampling line buffer	2	0	720	16	22.50
		1	360		11.50
		2	180		6.00
		3	90		3.00
		4	45		1.50
		5	23		0.75
Luminance line buffer	1	0	4x720 = 2880	8	22.50
		1	4x360 = 1440	16	22.50
		2	4x180 = 720		11.25
		3	4x90 = 360		5.75
		4	4x45 = 180		3.00
		5	4x23 = 92		1.50
Laplacian line buffer	2	0	720	17	23.90
		1	360		12.21
		2	180		6.37
		3	90		3.18
		4	45		1.59
		5	23		0.79
Total buffer size					230.07

Figure 3-6: Line buffer sizes at different levels of the Riesz pyramid constructor.

3.2.3 Phase Calculator

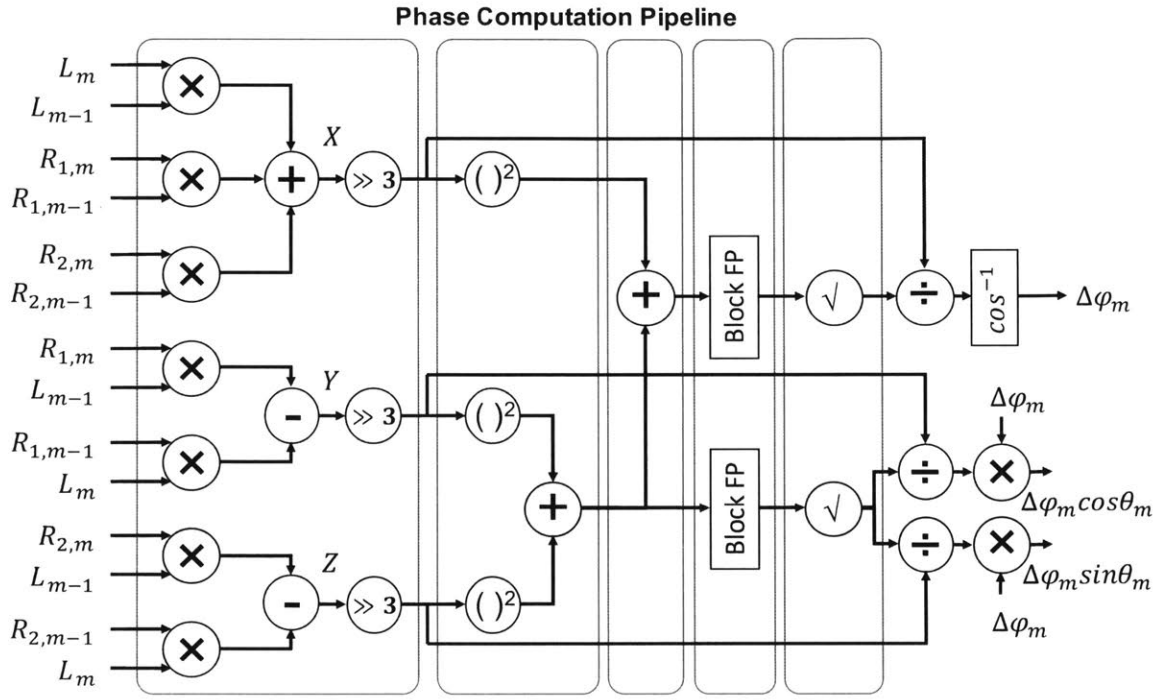


Figure 3-7: Phase calculator architecture.

The phase computation block receives $(L_m, R_{1,m}, R_{2,m})$ from the Riesz pyramid converter at each pixel of the pyramid frame m and the block reads the same pixel from the previous frame (frame $m-1$) from the DRAM as shown in Figure 3-7. These two values can be interpreted as being in rectangular coordinates. The phase calculator converts them into spherical coordinates (A_m, θ_m, ϕ_m), where A_m is the magnitude, θ_m is the orientation and ϕ_m is the phase, and computes the phase difference $\Delta\phi_m$ between the two pixels.

$$L_m = A_m \cos(\phi_m) \tag{3.8}$$

$$R_{1,m} = A_m \sin(\phi_m)\cos(\theta_m) \tag{3.9}$$

$$R_{2,m} = A_m \sin(\phi_m)\sin(\theta_m) \tag{3.10}$$

Given $(L_m, R_{1,m}, R_{2,m})$ we can solve these equations to get the spherical representation (A_m, θ_m, ϕ_m) but the solution is not unique. If (A_m, ϕ_m, θ_m) is a solution, so is

$(A_m, -\phi_m, \theta_m + \pi)$. So we compute $\phi_m cos(\theta_m)$, $\phi_m sin(\theta_m)$ which are invariant to this sign ambiguity. In addition, since we need to spatially filter the phase subsequently, we must unwrap the phase (such that there are no sharp discontinuities every 2π). We do this by first computing the phase difference and then accumulating it to get the unwrapped phase.

$$
\begin{aligned}
X &:= L_m L_{m-1} && + R_{1,m}R_{1,m-1} + R_{2,m}R_{2,m-1} \\
&= A_m A_{m-1}(\cos(\phi_m)\cos(\phi_{m-1}) && + \sin(\phi_m)\cos(\theta_m)\sin(\phi_{m-1})\cos(\theta_{m-1}) \\
& && + \sin(\phi_m)\sin(\theta_m)\sin(\phi_{m-1})\sin(\theta_{m-1})) \\
&= A_m A_{m-1}(\cos(\phi_m)\cos(\phi_{m-1}) && + \sin(\phi_m)\sin(\phi_{m-1})\cos(\theta_m - \theta_{m-1})) \\
&\approx A_m A_{m-1}(\cos(\phi_m)\cos(\phi_{m-1}) && + \sin(\phi_m)\sin(\phi_{m-1})) \\
&= A_m A_{m-1}\cos(\phi_m - \phi_{m-1}) \\
&= A_m A_{m-1}\cos(\Delta\phi_m)
\end{aligned}
$$

$$
\begin{aligned}
Y &:= R_{1,m}L_{m-1} && - R_{1,m-1}L_m \\
&= A_m \sin(\phi_m)\cos(\theta_m) A_{m-1}\cos(\phi_{m-1}) && - A_{m-1}\sin(\phi_{m-1})\cos(\theta_{m-1})A_m\cos(\phi_m) \\
&\approx A_m A_{m-1}\cos\theta_m \sin(\phi_m - \phi_{m-1}) \\
&= A_m A_{m-1}\cos\theta_m \sin(\Delta\phi_m)
\end{aligned}
$$

$$
\begin{aligned}
Z &:= R_{2,m}L_{m-1} && - R_{2,m-1}L_m \\
&= A_m \sin(\phi_m)\sin(\theta_m) A_{m-1}\cos(\phi_{m-1}) && - A_{m-1}\sin(\phi_{m-1})\sin(\theta_{m-1})A_m\cos(\phi_m) \\
&\approx A_m A_{m-1}\sin\theta_m \sin(\phi_m - \phi_{m-1}) \\
&= A_m A_{m-1}\sin\theta_m \sin(\Delta\phi_m)
\end{aligned}
$$

To compute the phase difference, we use the relations specified above. We first compute X, Y and Z defined above by substituting the expressions for $L_m, R_{1,m}, R_{2,m}$ at frames m and $m-1$ in spherical coordinates; and then use these to compute $\Delta\phi_m \cos\theta_m, \Delta\phi_m \sin\theta_m$ as shown in the equations below. Here, we make a simplify-

ing assumption that $\theta_m \approx \theta_{m-1}$.

$$
\begin{aligned}
X^2 &= A_m^2 A_{m-1}^2 cos^2(\Delta\phi_m) \\
Y^2 + Z^2 &= A_m^2 A_{m-1}^2 sin^2(\Delta\phi_m) \\
X^2 + (Y^2 + Z^2) &= A_m^2 A_{m-1}^2 \\
\Delta\phi_m &= \cos^{-1}\left(\frac{X}{\sqrt{X^2+Y^2+Z^2}}\right) \\
\cos\theta_m &= \frac{Y}{\sqrt{Y^2+Z^2}} \\
\sin\theta_m &= \frac{Z}{\sqrt{Y^2+Z^2}}
\end{aligned}
$$

The phase calculator circuit implements this computation, as can be seen in Figure 3-7 using fixed point arithmetic. Next, we shall talk about the precision scaling techniques we have employed in this arithmetic pipeline to reduce energy.

Phase computation engine receives $L_m, R_{1,m}, R_{2,m}$ each of which is a 17 bit value (S8.8). We move the decimal point to the left to normalize them and make them S0.16. The computation of X, Y and Z is done in full precision, with X being S2.32 (35 bits) and Y and Z being S1.32 (34 bits). We then truncate the last 3 bits of the result making X S2.29, and Y and Z S1.29. We square each of these intermediates; X^2 is U4.58 (62 bits), and Y^2 and Z^2 are U2.58 (60 bits). $Y^2 + Z^2$ is U3.58 (61 bits). This is extended to 63 bits by left padding with 2 zeros, so that we can use the same block FP shifter architecture described in the next paragraph. $X^2 + (Y^2 + Z^2)$ is U5.58 (63 bits). Next we take the square root of these two sums $(Y^2 + Z^2)$ and $(X^2 + (Y^2 + Z^2))$.

Using block floating point to reduce square root complexity To take the square root of the two sums generated using the logic described above, we need a square root block with 64 integer bits, and 16 decimal bits for precision. An 80-bit square root module would have an area of $28519\mu m^2$. The two square root modules that we require would account of 43.7% of the area, and consume 37.5% of the power of the phase calculator block. To reduce the complexity of the square root block, we

convert the input of the square root to a 16-bit block floating point (FP) notation (as shown in Figure 3-7) using a Block FP module. This allows us to use a 32 bit input square root module (16 integer bits and 16 decimal bits), which gives a 16-bit output (8 integer bits and 8 decimal bits). This reduces the area of the square root module by 85.3% and the power by 69.3%, and the area of the phase calculator module by 37.3% and power by 26% at the expense of reduced accuracy that does not adversely affect the motion magnified results.

The block FP module scans the unsigned 63 bit number and finds the index j of the most significant 1. If j is even, it adds 1 to it to make it odd; this is needed for the correct computation of the square root. If $j > 15$, then it returns bits at locations $j, j-1, \ldots, j-15$. It also returns the integer $j-15$, which will tell us how much towards the left we need to shift the output of the square root module. If $j < 15$, it returns the 16-bit number shifted left by $15-j$ and also the integer $15-j$ as the amount of right shift needed to get the original number back. Note that since j is odd, the shift in both cases is guaranteed to be even. We send half of this shift to the downstream pipeline, since it will be required to convert the result of the square root into a fixed point number. The square root returns an unsigned 16 bit number (U8.8). We shift this number by half of the shift of the original number to get a 40 bit number.

Pipelined square root The pipelined square root implements a digit-by-digit square root calculation algorithm (similar to long division). Its input is a 16-bit unsigned number with all integer bits, its output is a 16-bit unsigned number with 8 integer bits and 8 decimal bits. Figure 3-8 shows the architecture of one pipeline stage of the square rooter; the pipeline stage index i varies from 0 to 15. The stages are separated by 1-element pipeline FIFOs. One pipeline stage computes 1 digit of the output. The 32 bit operand *op* is initialized to the 16-bit input number, concatenated with 16 zeros on the right. The result *res* is initialized to zero. In each pipeline stage the operand op_i is compared to $res_i + 2^{30-2i}$. If it is greater, the operand for the next stage is obtained by subtracting $res_i + 2^{30-2i}$ from the operand, else the operand

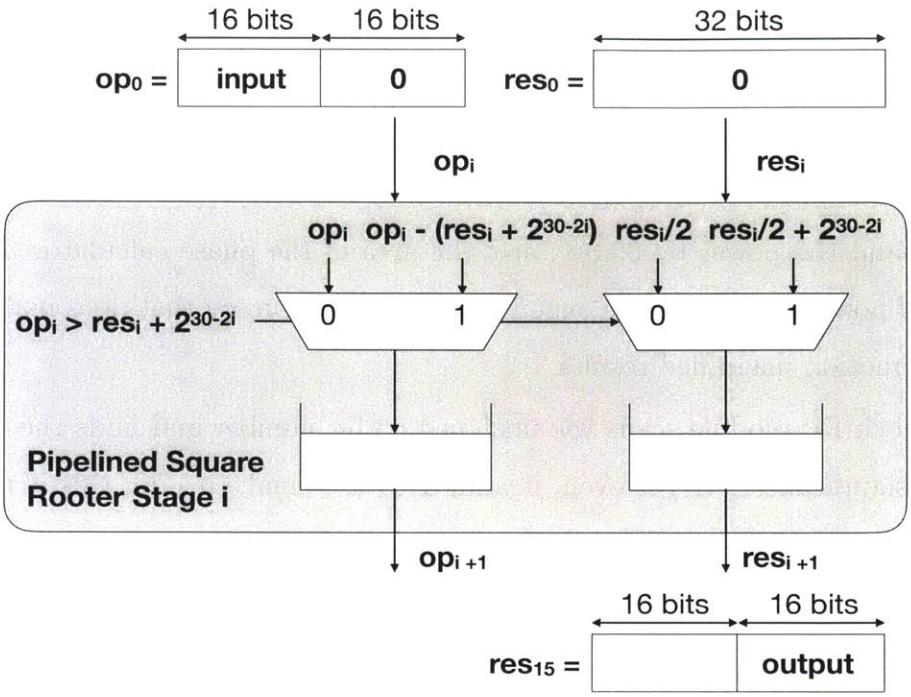


Figure 3-8: Architecture of pipelined square rooter.

is passed on as is. In the first case, the result is updated to $res_i/2 + 2^{30-2i}$ and in the second case to $res_i/2$. Finally, the square root is obtained by taking the least significant 16 bits of the result of the last pipeline stage.

Pipelined divider After the square root, we compute $\cos\Delta\phi_m$, $\cos\theta_m$ and $\sin\theta_m$ using the three dividers. The dividend of the top divider from Figure 3-7 is a truncated version of X, which is a 32 bit signed integer (S2.29) as described earlier. The divisor of the top divider is the output of the top square root which is an unsigned 40 bit number (U3.37). The divider converts the dividend to a sign magnitude representation, and extends it on the left and the right with zeros to make it U3.37 (same as the divisor). The divider has a pipelined long-division type architecture that computes one bit of the quotient in each pipeline stage. It compares the dividend so far with the divisor, if the divisor is smaller, it outputs 1 as the quotient bit and subtracts the divisor from the dividend and shifts the result left by 1, else it outputs zero as the quotient bit and shifts the dividend left by 1. The quotient is computed to 1 integer bit and 12 decimal bits, since cosine lies between -1 and 1. The result

is converted back into two's complement (14 bit signed integer, S1.12) based on the sign of the dividend. The other two divisions are performed similarly yielding signed 14 bit integers for $\cos\theta_m$ and $\sin\theta_m$.

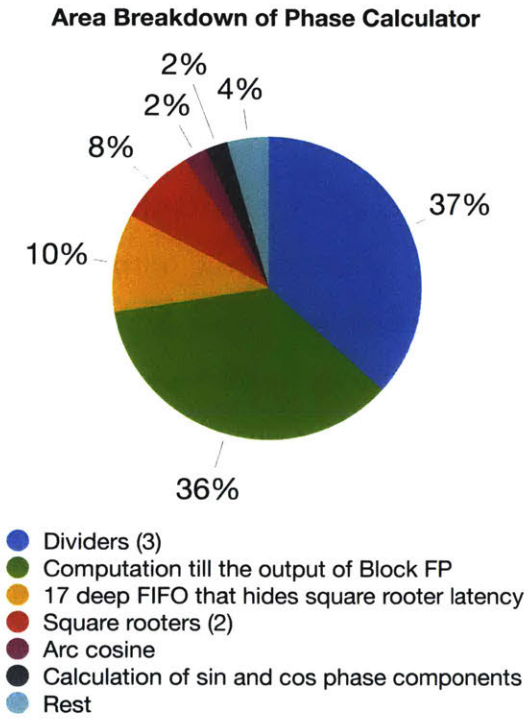
Arc cosine Finally, to get $\Delta\phi_m$ we need to compute the cosine inverse of $\cos\Delta\phi_m$, which is a number between -1 and 1 in S1.12 representation. We use a lookup table based arc cosine architecture. The result is normalized by π. The lookup table has 4097 entries for positive values of the input; for the negative values we compute the result for input x using $\pi - \cos^{-1}(-x)$. The arc cosine output is an 11 bit unsigned integer of the form U1.10.

Obtaining the final phase components Once we have $\Delta\phi_m$ from the arc cosine block, we have to multiply it with $\cos\theta_m$ and $\sin\theta_m$ which are the outputs of the bottom two dividers in Figure 3-7. The top multipliers multiplies $\Delta\phi_m$, which is U1.10 (or a 12 bit signed integer, S1.10 which lies between 0 and 1 inclusive) with $\cos\theta_m$, which is S1.12 (a 14 bit signed integer which lies between -1 and 1 inclusive), to produce a 25 bit signed integer (S2.22) output. We then truncate the multiplier output to a 12 bit signed integer only (S1.10) which is sufficient for precise motion magnification results.

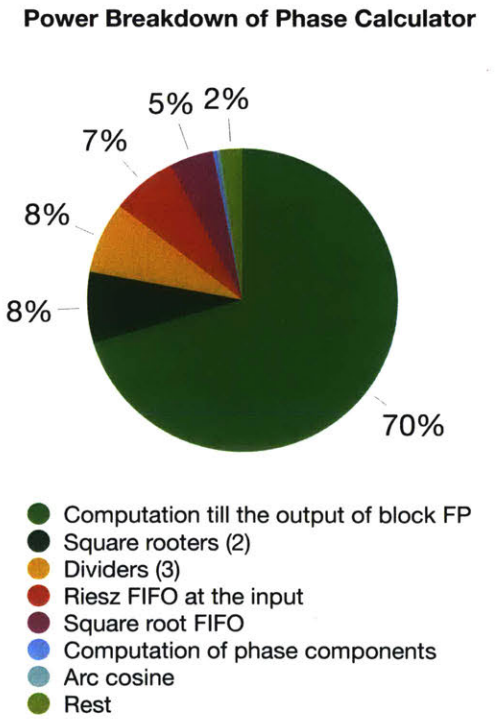
Area and power breakdown The total area of the phase calculator engine is $82996\mu m^2$ and its breakdown is shown in Figure 3-9a. The total power consumed by the phase calculator engine is $3.01mW$ and its breakdown is shown in Figure 3-9b. After the optimization of the square root using block floating point intermediates, most of the power is consumed by the multipliers and adders that generate the input to the square root blocks.

3.2.4 Accumulation of Phase Components

The output of the phase calculator is the phase difference between two successive frames $(\Delta\phi_m\cos\theta_m, \Delta\phi_m\sin\theta_m)$. The absolute phase is obtained by adding this



(a) Area breakdown of phase calculator.



(b) Power breakdown of phase calculator.

Figure 3-9: Area and power breakdown of phase calculator.

phase difference to the absolute phase of the previous frame, which is stored on and read from off-chip memory. The phase difference is guaranteed to be between $-\pi$ and π (i.e. the normalized phase difference is guaranteed to be between -1 and 1), but the absolute phase is unbounded, and its range depends on the number of previous frames it is accumulated over. To have sufficient dynamic range, the absolute phase is computed and stored in a signed 16-bit floating point representation (with 5 exponent bits and 10 mantissa bits). The top level motion magnification module contains two modules that convert each phase difference component from a 12 bit signed integer (S1.10) to a 16-bit float. It also contains two floating point adders that perform the accumulation for each phase component.

3.2.5 Temporal Filter

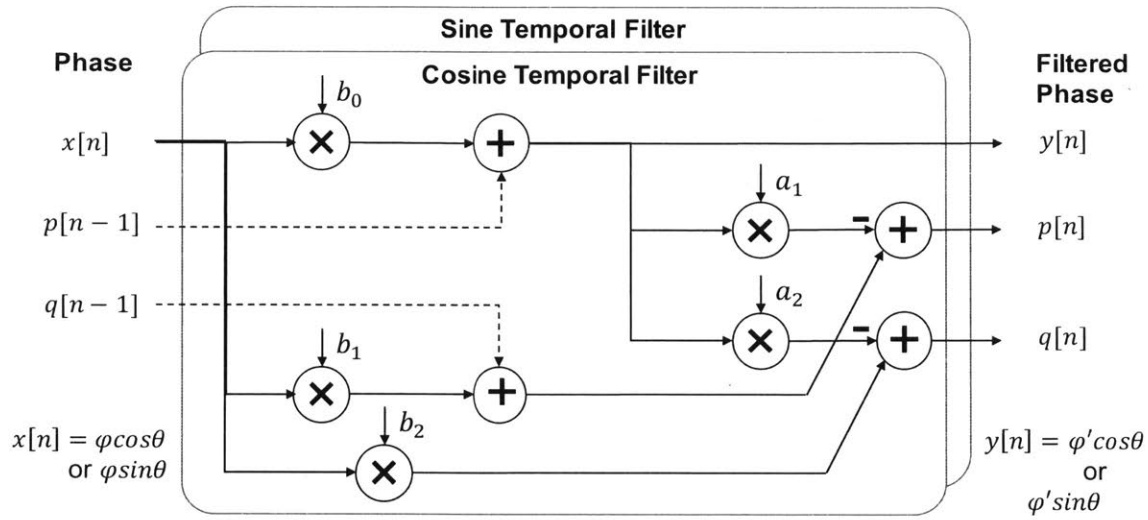


Figure 3-10: Architecture of temporal filter.

The motions of interest from the video are isolated and de-noised with temporal filters, typically band-pass infinite impulse response (IIR) filters. Let $x[n]$ denote the phase (either $\phi cos\theta$ or $\phi sin\theta$) at each pyramid pixel as a function of time (n). This is the input to the temporal filter. Let $y[n]$ denote the filtered phase, the output of the temporal filter. The relationship between $x[n]$ and $y[n]$ can be expressed in terms of their z-transforms $X(z)$ and $Y(z)$ as:

$$\frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} \ldots}{1 + a_1 z^{-1} + a_2 z^{-2} \ldots} \tag{3.11}$$

The values of the coefficients a_i's and b_i's can be calculated from the band-pass filter's center frequency and bandwidth, which depend on the application.

This IIR filter is implemented recursively in the time domain using difference equations (shown below for a second order filter).

$$\begin{aligned}
\frac{Y(z)}{X(z)} &= \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} \ldots}{1 + a_1 z^{-1} + a_2 z^{-2} \ldots} \\
y[n] &= b_0 x[n] + \\
&\quad (b_1 x[n-1] - a_1 y[n-1]) + \\
&\quad (b_2 x[n-2] - a_2 y[n-2]) \\
y[n] &= b_0 x[n] + p[n-1] \\
p[n] &= b_1 x[n] - a_1 y[n] + q[n-1] \\
q[n] &= b_2 x[n] - a_2 y[n]
\end{aligned}$$

Figure 3-10 shows the hardware architecture of the temporal filter that implements these difference equations. The phase (either $\phi cos\theta$ or $\phi sin\theta$) comes from the phase computation engine, and the intermediates are accessed from off-chip memory. The number of coefficients (or the order of the band-pass filter) directly impacts the number of previous frames accessed for filtering a pixel, which determines the number of off-chip memory accesses. In this work, we have fixed the number of coefficients to 5 but kept the values configurable, so that the processor works for a range of applications.

For the final implementation, the phase inputs to this module are 16-bit floating point values with 5 bit exponent and 10 bit mantissa. All computation happens in 16-bit floating point representation. The multipliers and adders in the module are pipelined.

Area and power breakdown The total area of the temporal filter is $13312\mu m^2$. The total power consumed by the temporal filter is $0.902mW$.

3.2.6 Spatial Filter

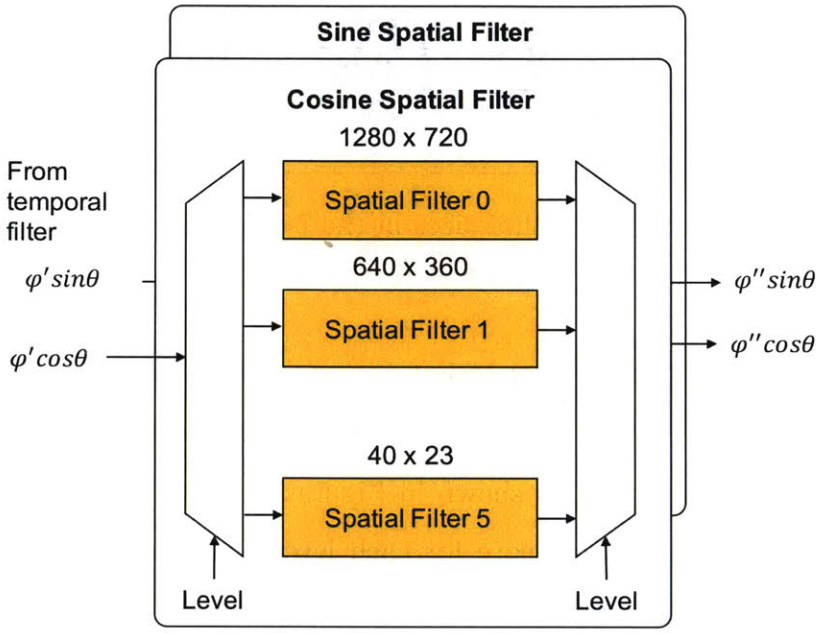


Figure 3-11: Spatial filter architecture.

Temporally filtered phase is de-noised using a spatial filter. The sine and cosine components of the phase are filtered independently, and within each component each pyramid level is filtered independently. Figure 3-11 shows the architecture of the spatial filter. It consists of a 7×7 Gaussian filter, which is implemented using separable horizontal and vertical 7 point filters with the following filter coefficients:

$$\frac{1}{256}\begin{bmatrix}18 & 33 & 49 & 56 & 49 & 33 & 18\end{bmatrix} \tag{3.12}$$

Since the throughput of the temporal filter is one pyramid pixel per cycle, not all spatial filters can be kept busy concurrently, so the filtering core (the multiply accumulate tree) can be shared between the filtering engines (they still, however, need their own separate column buffers). Since the spatial filter is a 7 point separable filter, 6 columns are buffered at each level. The column buffer sizes are shown in Figure

3-12.

Type	Banks	Level	Height	Width per bank (bits)	SRAM Kbits
Spatial line buffer	6	0	720	16	67.50
		1	360		34.50
		2	180		18.00
		3	90		9.00
		4	45		4.50
		5	23		2.25
Total buffer size					135.75

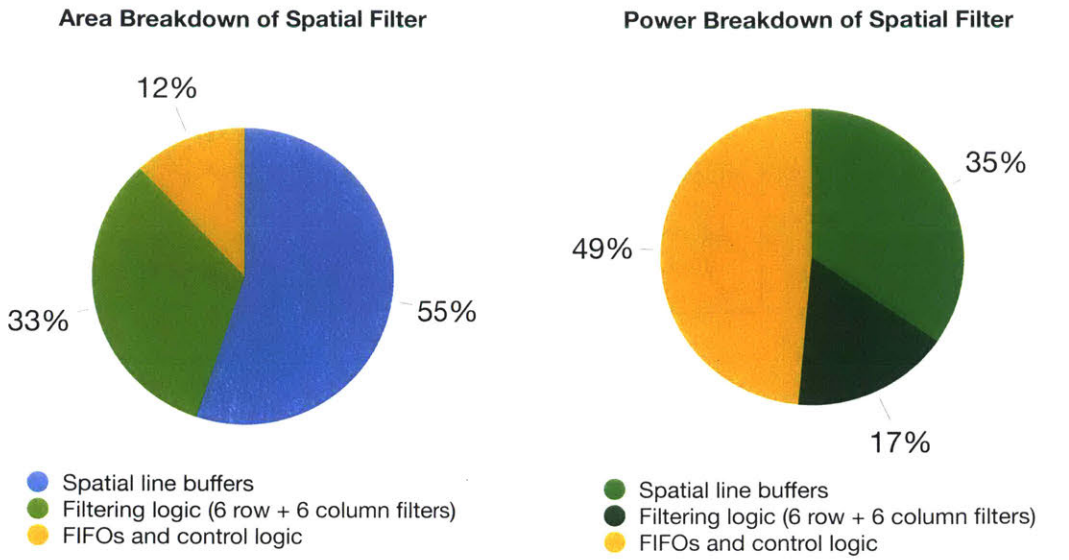
Figure 3-12: Line buffer sizes in the 7 point spatial filter.

Area and power breakdown The total area of spatial filter is $421438\mu m^2$ and its breakdown is shown in Figure 3-13a. The total power consumed by the spatial filter is $5.925mW$ and its breakdown is shown in Figure 3-13b. This breakdown assumes that there is a separate filtering core for each level. Some reduction in area (but not power) might be possible by sharing the filtering core between different levels of the spatial filter, however, it would introduce extra multiplexing/demultiplexing logic overhead. This is not explored in this architecture.

3.2.7 Phase Amplifier

Spatio-temporally filtered phase components ($\phi''cos\theta$, $\phi''sin\theta$) are finally squared and added to obtain ϕ''^2, which is passed through a pipelined 1-select square rooter to obtain the magnitude of the phase ϕ''. This phase is multiplied with an amplification factor α, which is then converted into the motion magnified Laplacian pyramid L_{out} using the following relations. Figure 3-14 shows the architecture of the phase amplifier engine that performs this computation.

$$\begin{aligned} L_{out} &= Acos(\phi)cos(\alpha\phi'') - ((R_1\phi''cos(\theta) + R_2\phi''sin(\theta))/\phi'')sin(\alpha\phi'') \\ &= Acos(\phi)cos(\alpha\phi'') - ((Asin(\phi)cos(\theta)\phi''cos(\theta) + Asin(\phi)sin(\theta)\phi''sin(\theta))/\phi'')sin(\alpha\phi'') \\ &= Acos(\phi)cos(\alpha\phi'') - Asin(\phi)sin(\alpha\phi'') = Acos(\phi + \alpha\phi'') \end{aligned}$$



(a) Area breakdown of spatial filter.

(b) Power breakdown of spatial filter.

Figure 3-13: Area and power breakdown of spatial filter.

CORDIC algorithm

We use CORDIC algorithm to compute sine and cosine of the amplified phase.

The basic operation in the algorithm is a vector rotation as shown in Figure 3-15. It involves rotating a vector (x_i, y_i) by an angle α_i to get the vector (x_{i+1}, y_{i+1}). The rotation can be expressed using the following set of equations:

$$x_{i+1} = r\cos(\alpha_i + \beta_i) \tag{3.13}$$

$$= r\cos\alpha_i\cos\beta_i - r\sin\alpha_i\sin\beta_i \tag{3.14}$$

$$= (\cos\alpha_i)x_i - (\sin\alpha_i)y_i \tag{3.15}$$

$$y_{i+1} = r\sin(\alpha_i + \beta_i) \tag{3.16}$$

$$= r\sin\alpha_i\cos\beta_i + r\cos\alpha_i\sin\beta_i \tag{3.17}$$

$$= (\sin\alpha_i)x_i + (\cos\alpha_i)y_i \tag{3.18}$$

The above equations can be equivalently written in matrix format as shown below. The extra term μ_i denotes the direction of the rotation, it is -1 for clockwise and 1

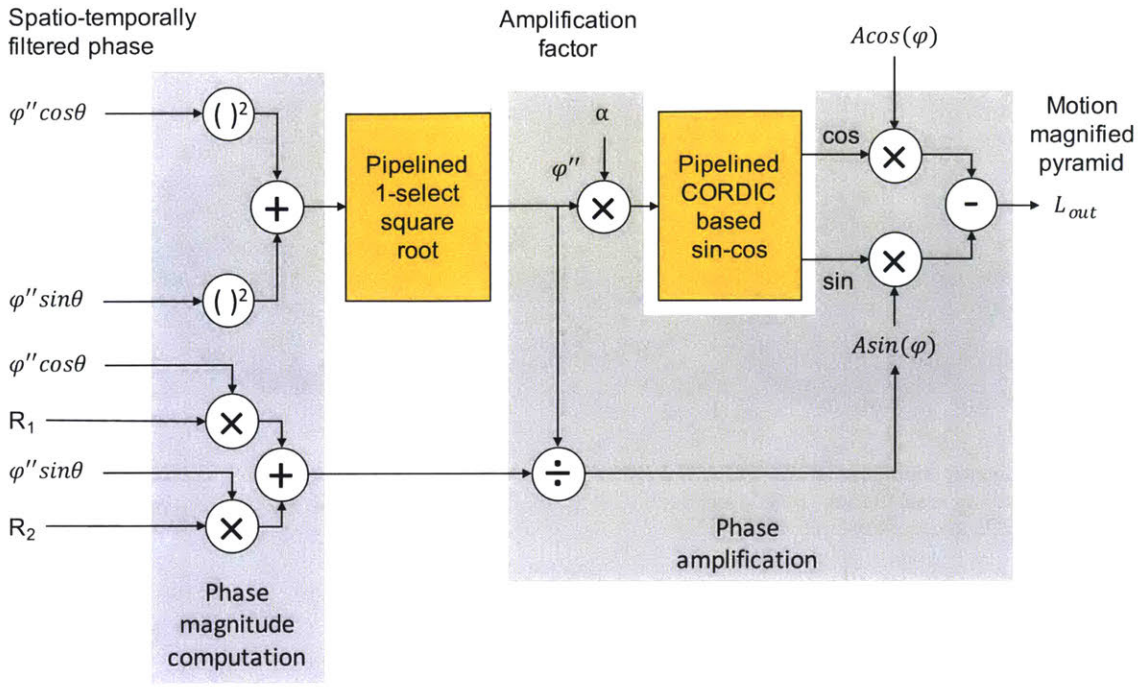


Figure 3-14: Phase amplification architecture.

for anti-clockwise rotation.

$$\begin{bmatrix} x_{i+1} \\ y_{i+1} \end{bmatrix} = \begin{bmatrix} \cos\alpha_i & -\mu_i.\sin\alpha_i \\ \mu_i \sin\alpha_i & \cos\alpha_i \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.19}$$

$$= \cos\alpha_i \begin{bmatrix} 1 & -\mu_i.\tan\alpha_i \\ \mu_i \tan\alpha_i & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{3.20}$$

By starting with $(x_0, y_0) = (1, 0)$, we perform a sequence of rotations until the total rotation angle, $\sum_{i=0}^{n-1} \mu_i\alpha_i$ becomes equal to the input angle z_0, whose sine and cosine we want to compute. In other words, the remaining rotation angle $z_i = z_0 - \sum_{i=0}^{n-1} \mu_i\alpha_i$, becomes equal to zero. When this happens, the final value x_n, will give us the cosine, and the final value y_n will give us the sine of the input angle.

We choose the rotation angle at each step, α_i, and the rotation direction at each

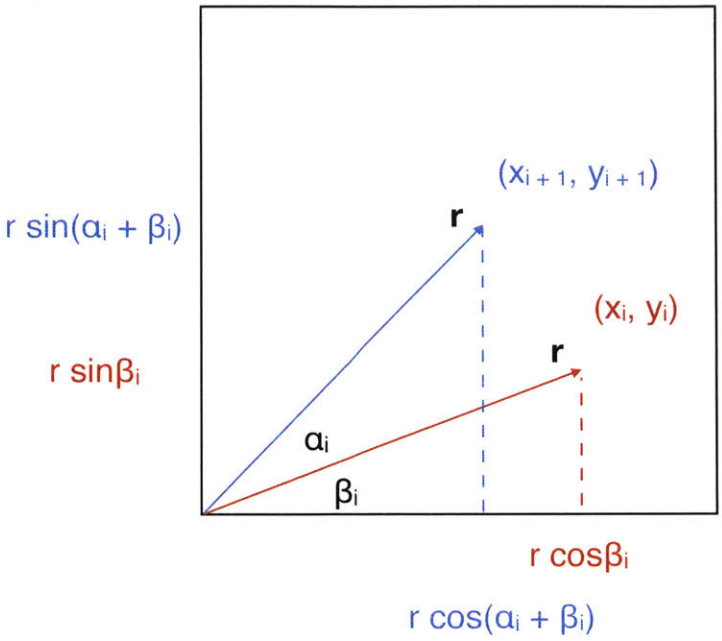


Figure 3-15: Vector rotation.

step, μ_i, using

$$\alpha_i = \tan^{-1}(2^{-i}) \text{ with } i \in \{0, \ldots, n-1\} \tag{3.21}$$

$$\mu_i = \text{sign}(z_i) \tag{3.22}$$

to ensure convergence. Choosing $\tan \alpha_i$ to be a power of 2 ensures that the multiplication with the scaled rotation matrix can be accomplished using only shifts and adds. The rotation angle sequence, α_i, is calculated in advance, quantized according to the chosen quantization scheme, and stored in a look-up table from which it can be retrieved during the execution of the CORDIC algorithm.

Additionally, to avoid multiplication with a scale factor $\cos \alpha_i$ in each iteration i, the computation in each iteration is simplified to the following, and the initial vector $(x_0, y_0) = (1, 0)$ is multiplied with the cumulative scale factor for all iterations $\prod_{i=0}^{\infty} \cos \alpha_i$. This cumulative scale factor can be pre-computed. xs_i and ys_i below

denote the scaled versions of x_i and y_i.

$$\begin{bmatrix} xs_{i+1} \\ ys_{i+1} \end{bmatrix} = \begin{bmatrix} 1 & -\mu_i . \tan\alpha_i \\ \mu_i \tan\alpha_i & 1 \end{bmatrix} \begin{bmatrix} xs_i \\ ys_i \end{bmatrix} \tag{3.23}$$

To summarize the CORDIC algorithm to compute cosine and sine of an angle θ becomes:

$$\mu_i = \text{sign}(z_i) \tag{3.24}$$

$$xs_{i+1} = xs_i - \mu_i(ys_i >> i) \tag{3.25}$$

$$ys_{i+1} = ys_i + \mu_i(x_i >> i) \tag{3.26}$$

$$z_{i+1} = z_i - \mu_i.\alpha_i \tag{3.27}$$

$$\tag{3.28}$$

with $z_0 = \theta$, $xs_0 = \prod_{i=0}^{\infty} \cos\alpha_i$ (pre-computed), $ys_0 = 0$, and $\alpha_i = \tan^{-1}(2^{-i})$ (pre-computed).

Input angle pre- and post-processing

The above algorithm is valid for angles θ such that $-\pi/2 < \theta < \pi/2$. For $\pi/2 < \theta < \pi$ and $-\pi < \theta < -\pi/2$, we use the relations given in Figure 3-16. There is no transformation required when θ lies in quadrant I and IV. For quadrant II, the input to CORDIC is $\theta - \pi/2$,

$$\cos(\theta - \pi/2) = \cos\theta\cos(\pi/2) + \sin\theta\sin(\pi/2) = \sin\theta \tag{3.29}$$

$$\text{so, } \sin\theta = \cos(\theta - \pi/2) \tag{3.30}$$

$$\sin(\theta - \pi/2) = \sin\theta\cos(\pi/2) - \cos\theta\sin(\pi/2) = -\cos\theta \tag{3.31}$$

$$\text{so, } \cos\theta = -\sin(\theta - \pi/2) \tag{3.32}$$

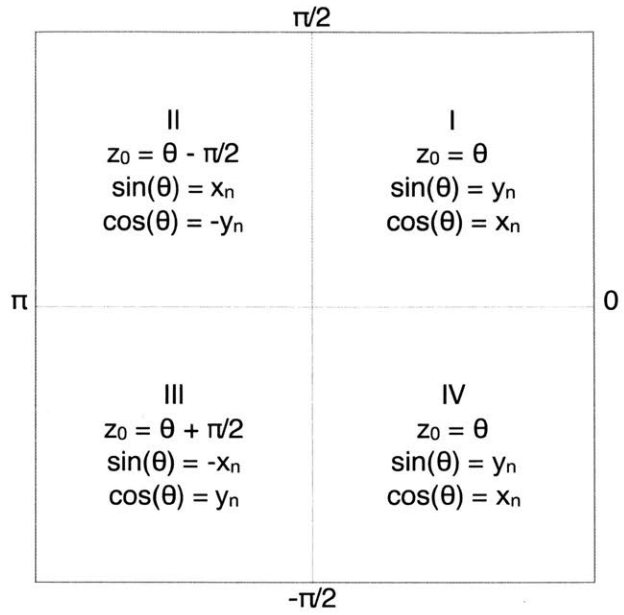


Figure 3-16: Mapping of $-\pi \leq \theta \leq \pi$ to CORDIC unit whose input must lie between $-\pi/2$ and $\pi/2$.

For quadrant III, the input to CORDIC is $\theta + \pi/2$,

$$\cos(\theta + \pi/2) = \cos\theta\cos(\pi/2) - \sin\theta\sin(\pi/2) = -\sin\theta \tag{3.33}$$

$$\text{so, } \sin\theta = -\cos(\theta + \pi/2) \tag{3.34}$$

$$\sin(\theta + \pi/2) = \sin\theta\cos(\pi/2) + \cos\theta\sin(\pi/2) = \cos\theta \tag{3.35}$$

$$\text{so, } \cos\theta = \sin(\theta + \pi/2) \tag{3.36}$$

Pipelined CORDIC Architecture

We implement the CORDIC algorithm described above using a pipelined architecture shown in Figure 3-17. The architecture has 16 identical pipeline stages, each of which corresponds to an iteration of the CORDIC algorithm. xs_i and ys_i are both 18 bit signed integers. The residual angle z_i is a 17 bit signed integer. α_i's are 17 bit signed integers stored in a combinational lookup table. xs_0 is initialized with an 18 bit signed integer 18'h09B74 which is equal to the cumulative scale factor. ys_0 is initialized to

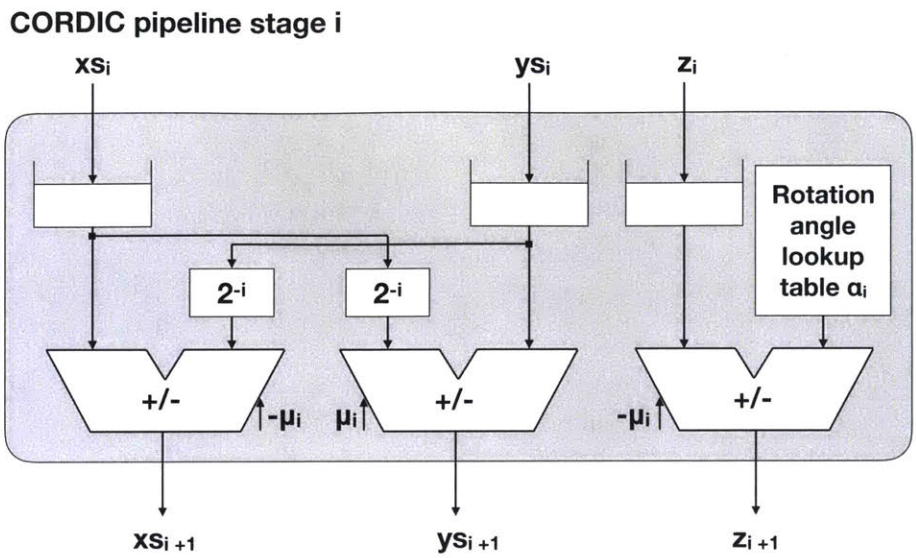


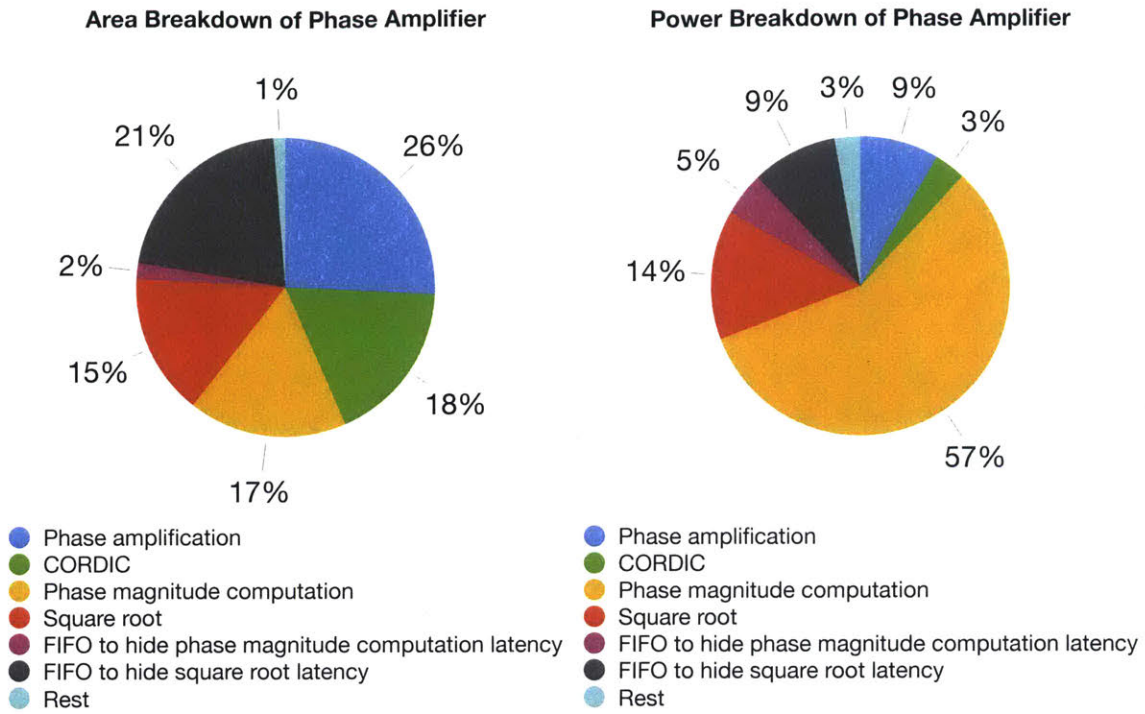
Figure 3-17: Architecture of one pipeline stage of CORDIC based sine and cosine computation. The module has 16 such stages (i varies from 0 to 15).

0.

Area and power breakdown The total area of phase amplifier is $51866\mu m^2$ and its breakdown is shown in Figure 3-18a. The total power consumed by the phase amplifier is $1.318mW$ and its breakdown is shown in Figure 3-18b.

3.2.8 Image Re-constructor

The phase amplified Laplacian pyramid is finally inverted to get the output image using the pyramid inversion algorithm shown in Figure 3-20. This algorithm is implemented by the image re-constructor, which consists of multiple levels similar to the pyramid constructor, only that the data flow is in the reverse direction. The smallest resolution low-pass image is read from the local low-pass buffer. It is up-sampled by a factor of 2 by inserting a zero after each pixel along the rows and the columns, followed by an interpolation using a 5×5 Gaussian filter. This filter is also implemented as separable 1D column and row filters. The Laplacian (high frequency component) is added to this up-sampled image to get the output image at that level, which is the low-pass input to the next higher level. This processes is repeated until we get the



(a) Area breakdown of phase amplifier.

(b) Power breakdown of phase amplifier.

Figure 3-18: Area and power breakdown of phase amplifier.

final image. The architecture of each level is shown in Figure 3-21, which shows the separable interpolation filter and the adder for summing the low-pass and high-pass components to generate the output image. The column buffer sizes are shown in Figure 3-19.

Area and power breakdown The total area of the image reconstructor is $234907\mu m^2$. 33% of this area is taken up by the upsampling line buffers and the rest by the logic. Almost all of the logic area comes from two large FIFOs that hide the DRAM latency which is incurred while accessing the Laplacian values for each level from the DRAM. The total power consumed by the image reconstructor is $4.63mW$.

3.3 FPGA Demonstration

The complete system described so far was tested on a Xilinx VC707 FPGA board. The motion magnification pipeline was verified to be working correctly on the test

Type	Banks	Level	Height	Width per bank (bits)	SRAM Kbits
Upsampling line buffer	2	0	720	16	22.50
		1	360		11.50
		2	180		6.00
		3	90		3.00
		4	45		1.50
		5	23		0.75
Total buffer size					45.25

Figure 3-19: Line buffer sizes in the 5 point up-sampling filter inside image reconstructor.

videos. It operates at a throughput of 1 pyramid pixel per cycle, which implies that running at a frequency of 37 MHz is sufficient to process HD (1280×720) at 30 frames per second. The system achieves this real-time performance on the FPGA. It will be fabricated in 40 nm CMOS technology for which area and power numbers are presented in each subsection. The estimated core power for the chip is approximately 9 mW at the real-time frequency; it can be further reduced to about 3 mW with voltage scaling.

3.4 Conclusion

To conclude, state of the art motion magnification algorithms are extremely computationally intensive and do not achieve real-time performance on modern CPUs. In this work, we present a low-power processor to accelerate motion magnification that achieves real-time performance on HD (1280×720) video at 30 frames per second, while consuming less than 0.2 nanojoules of processing energy per pixel. The processor accelerates the algorithm proposed in [30], which uses a Riesz pyramid to decompose each frame of the input video and separate the amplitude of the local wavelets from their phase. It then performs temporal filtering of the phases independently at each location, orientation and scale to isolate the frequencies of interest (for example, a band around 1 Hz for breathing rate amplification). This is followed by spatial smoothing to reduce noise in the phase, and finally amplification and recon-

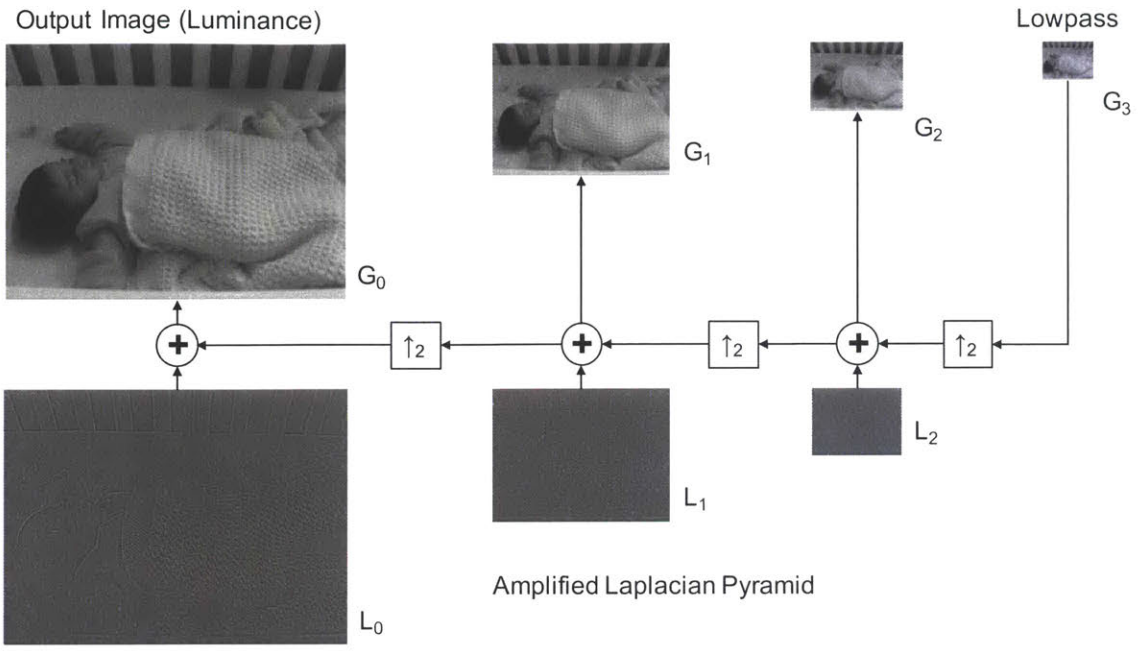


Figure 3-20: Pyramid inversion algorithm.

struction of the output video by inverting the pyramid.

We propose to use the following techniques in the design of the accelerator to achieve energy-efficiency and real-time performance: (1) Separable filtering in pyramid computation and spatial filtering to reduce the number of multiplies and adds. (2) Zero-skipping to reduce the buffering requirements of pyramid computation. (3) Reducing the precision of computation in several modules to reduce energy and selectively using block floating point representation to maintain accuracy. (4) Caching intermediates in on-chip SRAM based line buffers to reduce external memory bandwidth and save system energy. The complete motion magnification architecture is demonstrated on an FPGA and simulated area and power measurements in 40 nm CMOS technology are presented in this thesis. These techniques enable efficient integration of motion magnification technology into mobile devices.

3.5 Future Work

There are several immediate directions for future work. We are working on fabricating the motion magnification processor. Whereas the number of chip I/Os is not

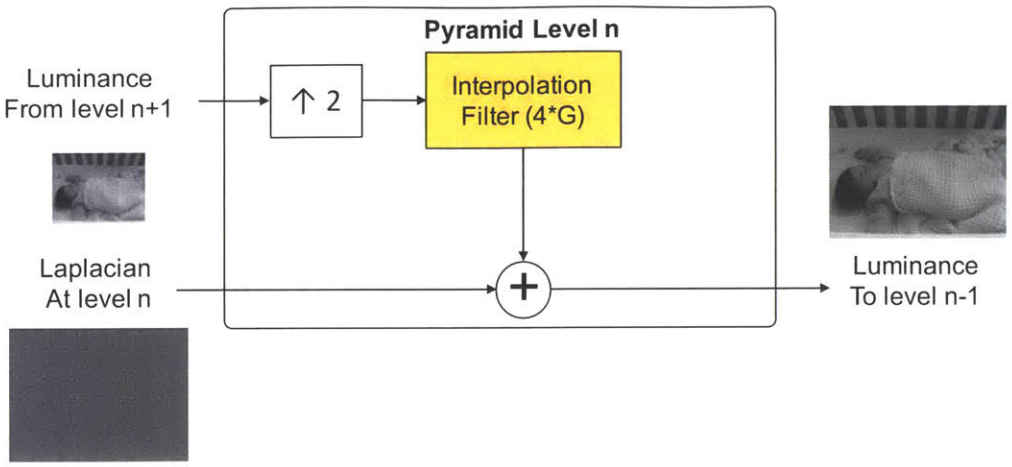


Figure 3-21: Image re-constructor architecture.

a concern in the FPGA implementation, it presents a major limitation for the on-chip implementation. The algorithm generates several intermediates at the various pipeline stages - the Riesz pyramid, the phase components, the temporal filtering intermediates - which are produced by filtering/accumulating temporally over successive frames. The on-chip SRAMs are not big enough to hold a complete frame worth of intermediates locally, so these intermediates are read from and written to the DRAM. Since each intermediate is required in each cycle (by different pipeline stages), this leads to a large I/O and DRAM bandwidth requirement. To solve these problems, we clock the DRAM and the I/O at a higher frequency than the chip itself.

We can trade off DRAM and I/O bandwidth and energy with processor area and energy. For example, computation of phase needs Riesz values from two successive frames. One architecture (that is employed in our current FPGA implementation), has one Riesz pyramid constructor that operates sequentially on the frames. The Riesz values of the current frame are written to the DRAM once they are computed and are accessed again by the phase calculator when it needs to compute the phase difference between the next frame and this frame. Another possible architecture is one where there are two Riesz pyramid constructors, one operating on the current frame and one operating on the previous frame in parallel. In this case, the Riesz values (which take 51 bits per pyramid pixel) are not written and read from the DRAM, only the image pixel values (which are 6 bits per pyramid pixel) are read twice from

the DRAM. Therefore, this architecture reduces both DRAM and I/O bandwidth and energy at the expense of having two instantiations of the Riesz pyramid constructor. We plan to explore such trade-offs in the on-chip implementation.

Another technique that could be used for energy minimization is active pixels detection. We define active pixels as the pixels which are undergoing the motion. Active pixels are typically less than 50% of total pixels since large portions of the scene remains static from frame to frame. These active pixels can be detected from the temporal filter output, and we can perform motion magnification for the subsequent frames on the active pixels only to reduce the amount of computation, memory bandwidth and energy. Finally, we can perform dynamic voltage and frequency scaling together with active pixels detection to reduce energy while maintaining the throughput.

3.6 Acknowledgements

We would like to thank Intel Corporation for sponsorship and Professor Frédo Durand, Professor William T. Freeman, Dr. Mondira Pant and Dr. Dongsuk Jeon for valuable feedback. We would also like to thank MIT undergraduate student, Natalie Mionis for contributing to the design of the test platform for this work.

Chapter 4

A 3D Imaging Platform for Automated Surface Area Assessment of Dermatologic Lesions

4.1 Motivation

For many dermatologic conditions, the initial assessment of disease severity as well as the primary outcome measure of progression or treatment success are dependent on an assessment of body surface area (BSA) involved in the disease. This assessment is based on a physician's visual estimation of area involvement, which has low accuracy and reproducibility. Moreover, these assessments often are performed by non-dermatologists with no training to complete such assessments. This limitation represents a major hurdle for clinical trials, translational research efforts, as well as daily clinical care. For example, estimation of BSA helps determine who is a candidate for systemic immuno-suppression for psoriasis and atopic dermatitis among other diseases. It also is a key factor in determining the grade of cutaneous adverse reactions to chemo- and immuno-therapies, and determines whether a patient can stay on a cancer therapy. This chapter addresses the limitations of existing methods to measure body surface area and presents a tool and associated image processing/enhancement

algorithms to enable rapid, reproducible, and objective determination of surface area of a cutaneous lesion(s) in an automated fashion via 3D imaging.

This work has been done in collaboration with MIT student, Jiarui Huang, who worked on the python implementation of the algorithm and its initial testing as a part of his Master's thesis, and Dr. Victor Huang from Brigham and Women's Hospital: Boston Hospital & Medical Center, who collected patient data using our system, and provided clinical insights into the results. My contributions were the design of the platform architecture, identification of the algorithms to be used for each processing step and performing extensive validation, measurements and analysis on clinical data. We completed and submitted the Harvard Affiliated Application for Institutional Review Board (IRB) Review of Multi-Site Research. Our application has been successfully reviewed and accepted (Federal Wide Assurance: FWA00004881 and FWA00000484).

4.1.1 Vitiligo

In this work, we have chosen vitiligo as a model condition to demonstrate the utility of such a tool for the following reasons:

- Low acuity of pathology experienced by the patients allowing for investigative imaging with minimal impact on clinical care.
- Varied locations of presentation across the body.
- Relatively high prevalence affecting all skin types and ethnicities equally (1-2% of the general population).
- Underserved nature of the disease.

The clinical need for a reliable, reproducible, inexpensive, and practical assessment of surface area and volume of lesions is present, however, across an array of inflammatory skin diseases (for example psoriasis, atopic dermatitis, drug reactions, etc.), wounds (for example graft-versus-host disease, chronic wounds, etc.), and pigmented lesions

(for example atypical moles, melasma, etc.), and this work is extensible to these disease types.

Vitiligo affects 1-2% of the general population, and there are an estimated 3 to 6 million individuals in the United States suffering from it. Recent work has demonstrated the significant psychological, social, professional, and interpersonal repercussions for those afflicted [4, 23]. Clinical and translational efforts to address the burden of vitiligo, however, have lagged behind the recognition of the impact of this disease. The lack of a reliable, objective, and reproducible outcome measure that can be assessed quickly has hampered clinical and translational investigation of treatments for vitiligo. Vitiligo Area Scoring Index (VASI) and Vitiligo European Task Force (VETF) measures, commonly used in clinical trial contexts, which are based on a physician's visual assessment of the percentage of body surface area affected by vitiligo, were demonstrated to have a smallest detectable change of 7.1% and 10.4% respectively, which is quite large [16].

4.1.2 Existing Approaches for Objective Measurement of Lesions

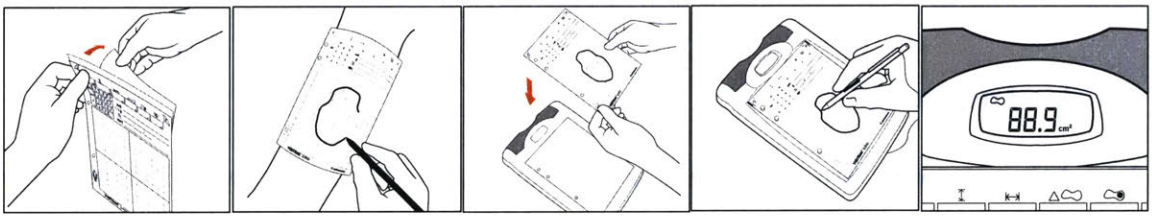


Figure 4-1: Surface area measurement with planimetry using Visitrak. This approach requires contact with the lesion and manual tracing. Image from [3].

The gold standard of lesion area assessment involves planimetry, where a lesion is manually traced onto a transparent grid sheet, which is then retraced onto the Visitrak pad [2] that automatically calculates the area of the trace as shown in Figure 4-1. The advantage of this approach is that the area obtained is an absolute area, however, manual tracing is very time-consuming because of the complex shapes of the lesions and may cause discomfort to the patient. It is also limited to small lesions, and not

suited to evaluate involvement across an entire body.

Attempts at image analysis based on 2D color images of the lesion suffer from the same necessity of manual lesion segmentation. ImageJ is an open-source, Java-based image processing program developed at and distributed by the National Institutes of Health [1, 24]. To use ImageJ to measure a lesion, the lesion needs to be photographed first with a ruler in the photographic frame to allow ImageJ calibration. The lesion outline needs to be manually traced and then the ImageJ software can calculate the lesion area. Compared to Visitrak, this method is non-lesion-contact but it is cumbersome because the lesion outline needs to be manually traced. Also, this method only measures the projected area rather than the curved surface area.

An image analysis program [26] has been developed which performs automatic lesion segmentation from 2D color images of the lesion, but only provides the relative change in lesion area for a sequence of images, and no absolute measurement, which makes comparing different lesions for the same person or across different people impossible. It also fails at segmenting large lesions and gives inaccurate measurements over curved surfaces.

This work addresses all of these shortcomings; it is user-friendly, non-lesion-contact and gives an absolute measurement of lesion area over any curved surface.

4.1.3 Impact

A tool that can measure the surface area in an automated fashion would be useful to pharmaceutical companies designing trials, researchers, and general dermatologists. Patients potentially can use this to monitor their own disease progression.

The most immediate customers would be those involved in clinical trials for drugs treating dermatologic lesions or with dermatologic side effects. Using the vitiligo space as an example, most clinical trials currently require large cohorts followed over 6 months to see changes because of the limitations of outcome assessments used. A more accurate measure that is more sensitive to change would allow more rapid assessments of fewer patients. Our own pilot data suggest that reliable changes can be seen as early as 1-3 months with standard treatments for vitiligo. The other

space where such assessments may have utility is in the assessment of dermatologic side effects. In particular, in the area of targeted immunotherapy for cancer, 70-80% of drugs involve cutaneous side effects. These side-effects are often dose limiting, and the Common Terminology Criteria for Adverse Events uses body surface area to determine Grade 3 toxicity requiring dose reduction or trial termination from Grade 2 toxicity. These assessments are often completed by non-dermatologists with limited experience in making such assessments. This leads to unnecessary dose reduction or medication cessation both in research trials and regular clinical care.

General dermatologists may find such assessments useful, particularly as data using these objective measures becomes increasingly used to determine suitability of therapies for particular diseases. Already, guidelines have been developed for the use of first and second generation biologic medications for the treatment of psoriasis based on body surface area involvement. The application of this technology to the monitoring of pigmented lesions would incorporate a factor not currently being used to evaluate atypical moles, namely lesion volume in addition to area and regularity. Finally, the evaluation of chronic wounds and response to treatment is intimately related to surface area and volume of wounds that is currently too cumbersome to determine in the regular clinical flow.

Finally, patients themselves may be interested in applying such a technology to monitor their own progression and response to treatment over time.

4.2 System Workflow

The 3D imaging system developed in this work has high accuracy, and requires minimal manual labor and resources to implement. It is envisioned to be used in an outpatient clinical setting or at home by patients for the evaluation of disease severity and progression. This system will allow objective and reproducible assessments of disease burden with a high sensitivity to change over time.

The system workflow starting from image capture to area calculation consists of the following steps (Figure 4-2):

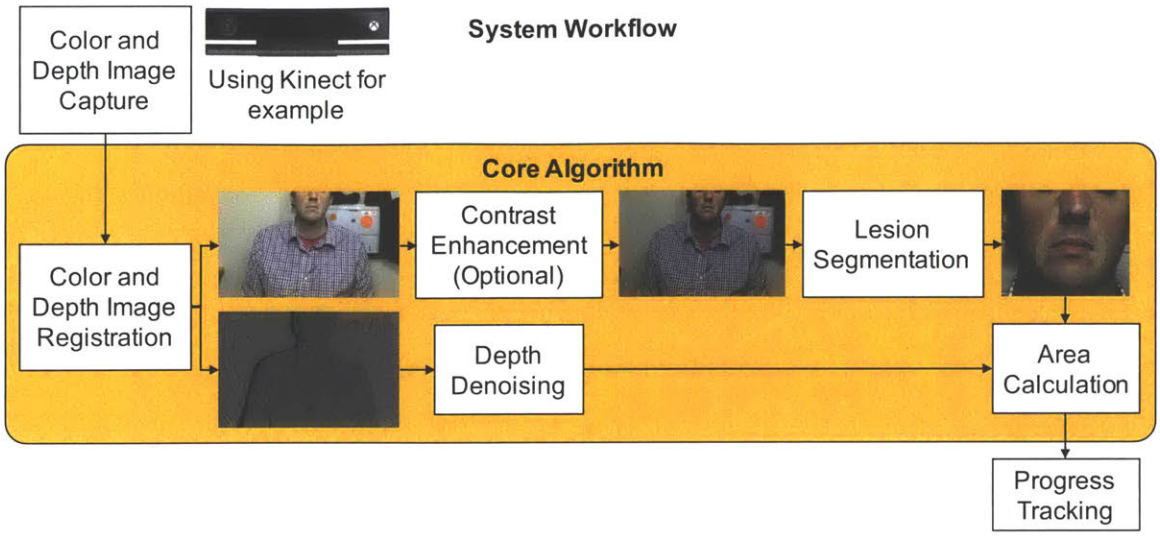


Figure 4-2: System workflow starting from image capture to area calculation.

1. A 3D camera (Kinect V2 in our case) captures color and depth images and stores them in files. The camera can capture the images using available ambient light sources. Supplementary lighting, for example an ultraviolet Wood's lamp or a flash, may also be used to capture more detailed images. These raw data files are labeled with the patient's number and date.

2. The system registers the depth image with the color image since the two images are captured using two cameras at different locations, with different fields of view and resolutions.

3. It pre-processes the color image using contrast enhancement with algorithms such as gamma correction, histogram equalization and contrast limited adaptive histogram equalization (CLAHE) to get a more distinguishable contour for lesion segmentation.

4. It performs lesion segmentation using watershed [6] algorithm.

5. It calculates the surface area of the segmented lesion using:

$$\text{Lesion area} \simeq \sum_{(i,j)\epsilon Lesion} D(i,j)^2 \sqrt{D_x(i,j)^2 + D_y(i,j)^2 + 1}\, \theta_x\, \theta_y \tag{4.1}$$

$$D_x(i,j) = \frac{D(i+1,j) - D(i-1,j)}{2D(i,j)\theta_x} \tag{4.2}$$

$$D_y(i,j) = \frac{D(i,j+1) - D(i,j-1)}{2D(i,j)\theta_y} \tag{4.3}$$

Here (i, j) are pixel coordinates. $D(i,j)$ is the depth of pixel (i,j). The camera has a field of view $F_x \times F_y$ degrees with a resolution of $R_x \times R_y$ pixels and each image pixel corresponds to a field of view of $\theta_x \times \theta_y$ degrees where $\theta_x = F_x/R_x$ and $\theta_y = F_y/R_y$.

The following sections describe each of these steps in detail and the final section presents the results.

4.3 3D Image Acquisition and Processing

4.3.1 3D Image Acquisition

This system uses Kinect for 3D image acquisition; it captures two images - a color image and a depth image. We tested our system with Kinect versions 1 and 2. Kinect version 1 (V1) has a color image resolution of 640 × 480 pixels and a depth image resolution of 320 × 240 pixels and uses stereopsis for depth measurement. Kinect version 2 (V2) has a color image resolution of 1920 × 1080 pixels and a depth image resolution of 512 × 424 pixels and uses time-of-flight for depth measurement. Because of the different techniques used to measure depth, the effective depth measurement ranges are also different. Kinect V1 has an effective measurement range from 0.4 m to 4 m while Kinect V2 has a range from 0.5 m to 4.5 m. Even though we use Kinect for image acquisition, our platform not limited to it; it can support any camera that provides a color and a depth image, and the mapping between the two.

For light-skinned patients, the vitiligo lesions are not completely discernible in am-

bient light. This system uses a supplementary ultraviolet Wood's lamp to illuminate the lesion area, which makes the lesions more discernible. The effect of supplementary lighting is illustrated in the results section.

4.3.2 Color and Depth Image Registration and Depth Image Completion and De-noising

In order to calculate the surface area, we need to register the depth image with the color image because the two images are captured by different cameras in Kinect that are offset with respect to each other and have different resolutions and fields of view. Simply scaling them and stacking one on top of the other does not give desirable results. The difference in resolutions of the two images is shown in Figure 4-3. We use a function called coordinate mapper from Microsoft's SDK to map each color image pixel to a depth pixel. The results are shown in Figure 4-4.

Figure 4-3: Images captured by Kinect V2: Left: Color image (1920×1080), Right: Depth image (512×424). Dimensions of the two images are different. Black area indicates missing depth values.

Notice that the depth value can be missing at several pixels. If an object is outside Kinect's depth measurement range or the depth measurement is too noisy, then the corresponding depth pixel will be missing (black). We fill in pixels that contain no depth values with the average depth of the 4 nearest valid pixels towards the top, the bottom, the left and the right of the missing pixel to improve the accuracy of area calculation. To improve the depth estimates for the pixels with missing depth

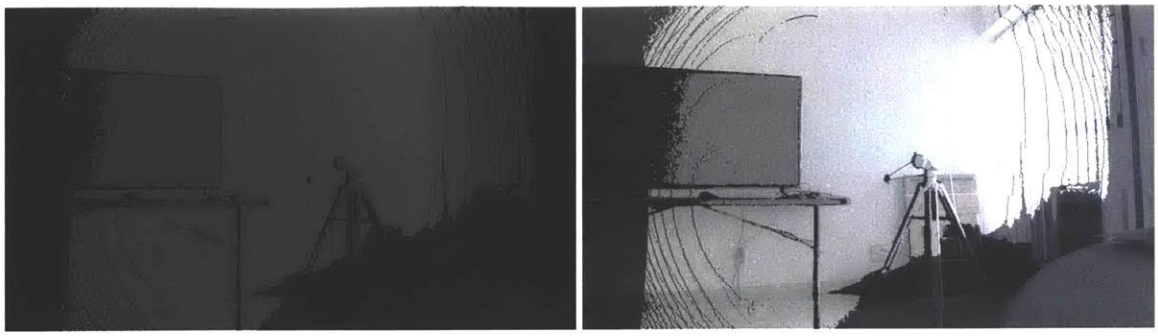


Figure 4-4: Coordinate mapper maps the color image onto the depth image: Left: Mapped depth image, Right: Overlaid color and depth images.

values even further more sophisticated algorithms such as [25, 33, 17, 27] could be used in future work which use information from the color image to fill holes in the depth image.

The other reason for missing depth information is the difference in the fields of view of the color and the depth cameras. The color camera has a field of view of 84.1×53.8 degrees while the depth camera has a field of view of 70.6×60 degrees. Since the color camera has a larger field of view, some objects that are only captured by the color camera cannot find their corresponding depth pixels in the depth image, which causes two black strips at the left and right ends of the mapped image. To avoid error due to this, it is necessary that the lesion not be near the edge of the image when the image is captured.

4.3.3 Color Image Enhancement

Skin de-pigmentation is visually different on various types of skin. Vitiligo lesions on lighter skin are less distinguishable than those on darker skin. The system, therefore, incorporates a contrast enhancement tool to aid lesion segmentation as shown in Figure 4-5. The tool implements the following existing contrast enhancement algorithms:

Gamma Correction Gamma correction is a simple nonlinear transformation. For an input pixel $I(x, y)$, the output pixel after gamma correction $O(x, y)$ is given by:

$$O(x, y) = [\alpha * I(x, y) + \beta]^{\frac{1}{\gamma}} \tag{4.4}$$

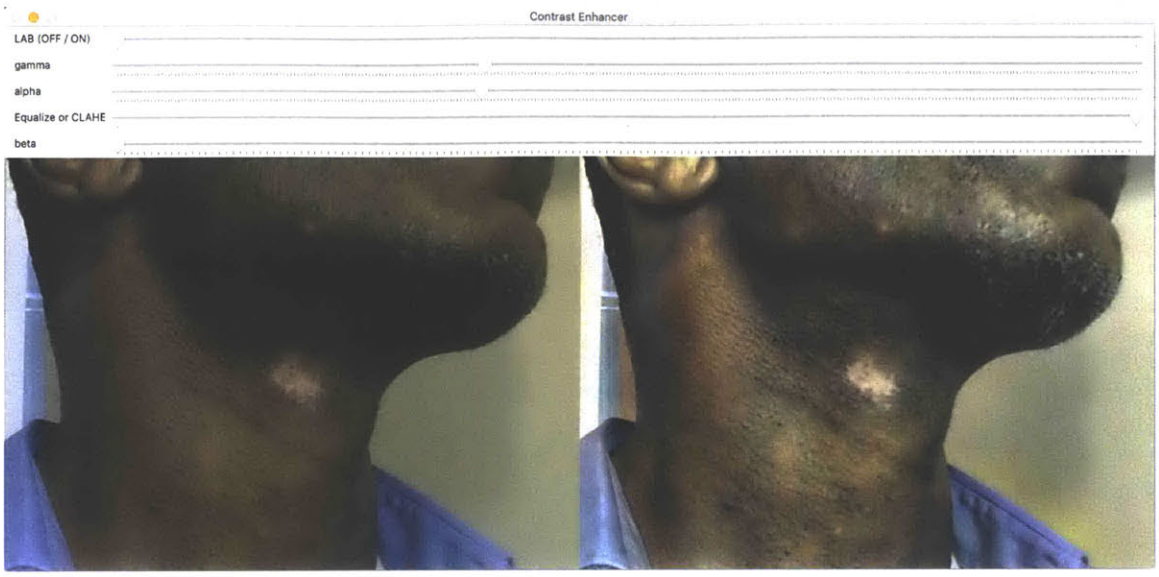


Figure 4-5: A screen-shot of the contrast enhancement tool.

where α operates as an intensity amplifier, β as a range shift, and γ as the gamma coefficient. The difference in intensity between two pixels will be accentuated if $\gamma < 1$ and lessened if $\gamma > 1$. The original image is returned when $\gamma = 1$. This simple transformation in intensity involves only three parameters and therefore it is computationally efficient yet powerful. In this system, gamma correction is applied to all three color channels independently.

Histogram Equalization A histogram is a graphical representation of the pixel intensities in the image. The left side of the graph represents the information in the darker areas in the image and the right side represents the brighter area. The height of each bar indicates the frequency of pixels in a specific intensity range, typically from 0 to 255. Consider a grayscale input image X with gray levels ranging from 0 to $L-1$. From a probabilistic model point of view, each bar in the histogram of this image represents the probability density of pixels with gray level i. Define the cumulative distribution function $F_X(i)$ of pixels with intensity level i as

$$F_X(i) = \sum_{j=0}^{i} \frac{n_j}{n} \tag{4.5}$$

where n_j is the number of pixels with gray level j and n the total number of pixels in the image. Histogram equalization [11] converts the input image to the output image Y such that $F_Y(i)$ is linear, or the mass density function $f_Y(i) = (F_Y(i))'$ is constant. This spreads out the most frequent intensity values in the image and therefore increases the global contrast of the image.

CLAHE Histogram equalization considers the global contrast of the image. However, in many cases, it is sub-optimal. An improved technique known as adaptive histogram equalization [20] is used to provide better contrast processing. Adaptive histogram equalization divides images into tiles with a much smaller size than the dimension of the image. Each tile is first histogram equalized. Then bilinear interpolation is applied to soften tile borders. Adaptive histogram equalization produces excellent results in enhancing the signal component of an image, but it also amplifies the noise in the image. To address this problem, contrast limitation is introduced. In the contrast limited adaptive histogram equalization, or CLAHE [34], a contrast limit threshold β is specified. If any histogram bin is beyond β, the corresponding pixels will be clipped and distributed evenly into other bins before histogram equalization.

Lab Color Space Lab color space is designed to approximate human vision. It describes perceivable colors using three parameters: L for lightness, a for green-red color opponents and b for blue-yellow color opponents. By converting the original image from RGB color space to Lab color space, this system can adjust the lightness layer to reveal hidden details that are not apparent, without losing the color features. After the adjustment, the Lab image is converted back into an RGB image for further processing. This system provides the user the option to perform color adjustment in Lab color space to increase segmentation accuracy.

Integrated Tool All the methods described above are incorporated in the contrast enhancer to reveal unapparent details. It provides a variety of options to the user, including whether to operate in Lab color space, perform gamma correction and different modes of histogram equalization. The user interface along with the sample

processing image is shown in Figure 4-5.

4.3.4 Lesion Segmentation from Color Image

We perform segmentation using the watershed algorithm [6], and compare our results with [26]. A detailed description of the watershed algorithm is beyond the scope of this thesis, please refer to [6] for an explanation. We focus our discussion on how we use the watershed algorithm. Our approach is semi-autonomous: the user is presented with the image in a graphical interface shown in Figure 4-6. The user marks, very coarsely, some background pixels with a red marker and some lesion pixels with a green marker. When the user presses 'spacebar' the segmented output is shown. The interface also shows the background area and the lesion area in red and green respectively overlaid on the original image. Pressing 's' saves the segmentation results. If the user detects a mistake in the segmentation, he/she can go back to the image window and annotate some more pixels with red and green markers, and press 'spacebar' again to see the updated results. Once satisfied with the segmentation, the user can press 's' to save the segmented result. Absence of this interactiveness was a major limitation of [26] since the dermatologist could not correct any mistakes made by the algorithm and that led to inaccurate results in certain scenarios.

4.3.5 3D Surface Area Calculation

After lesion segmentation, the final step is to calculate the lesion area. A naive method for calculating area is to count the number of pixels in the lesion. This method has the following drawbacks: first, it only gives a relative measure of the area (with respect to the size of the image), not an absolute one. Moreover, the size of the lesion in the image changes with distance. The larger the distance the smaller size it appears in the image. This method, therefore, requires the image to be taken at the same distance and with the same camera perspective each time in order to track progress of the lesion. This is difficult to implement reliably in a clinical setting. To get an absolute measure of the area, this method also requires that a ruler be incorporated in the

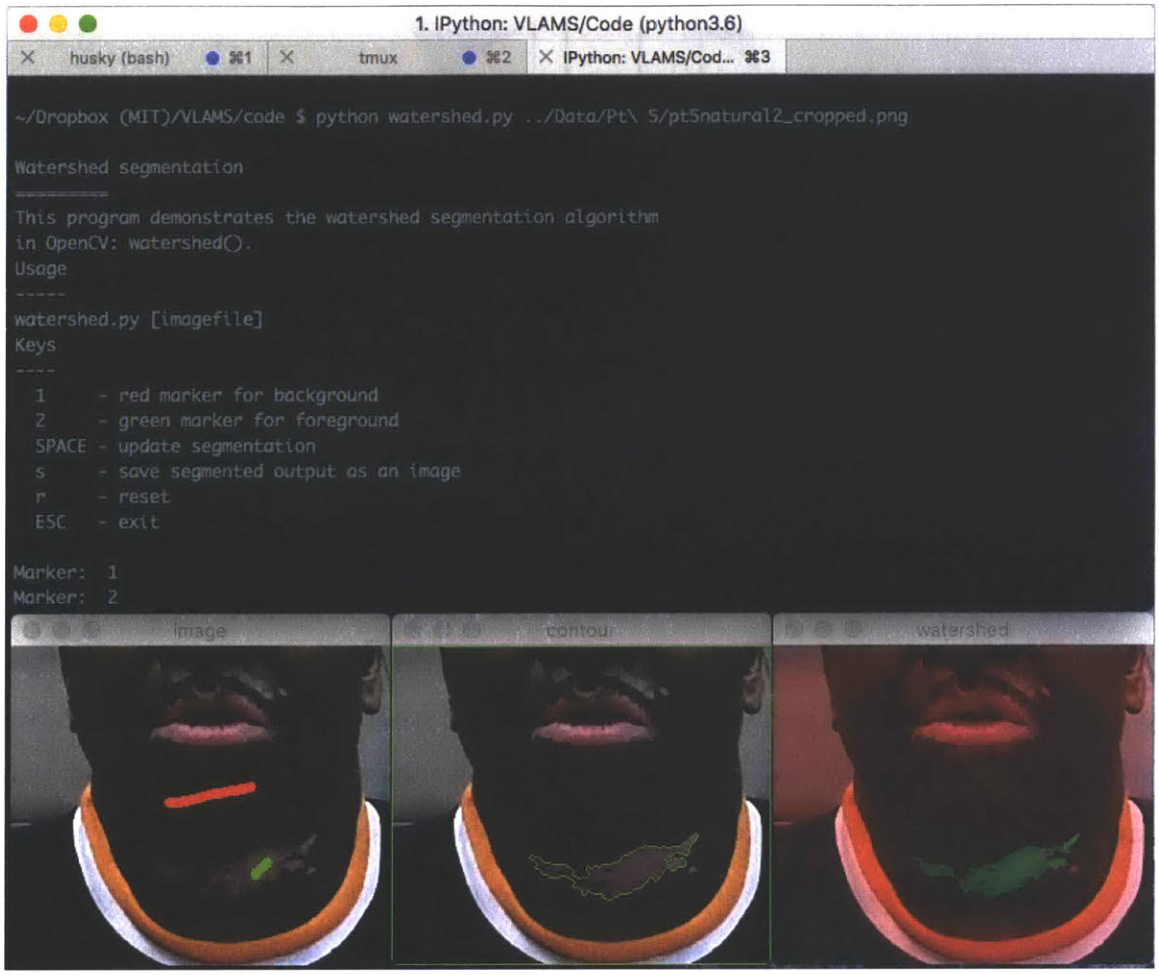


Figure 4-6: User interface for watershed segmentation. Left: The user marks very coarsely some pixels in the background with a red marker and some pixels in the lesion with a green marker. Middle: When the user presses 'spacebar' the segmented output is shown. Right: The interface also shows the background area and the lesion area in red and green respectively overlaid on the original image. Pressing 's' saves the segmentation results.

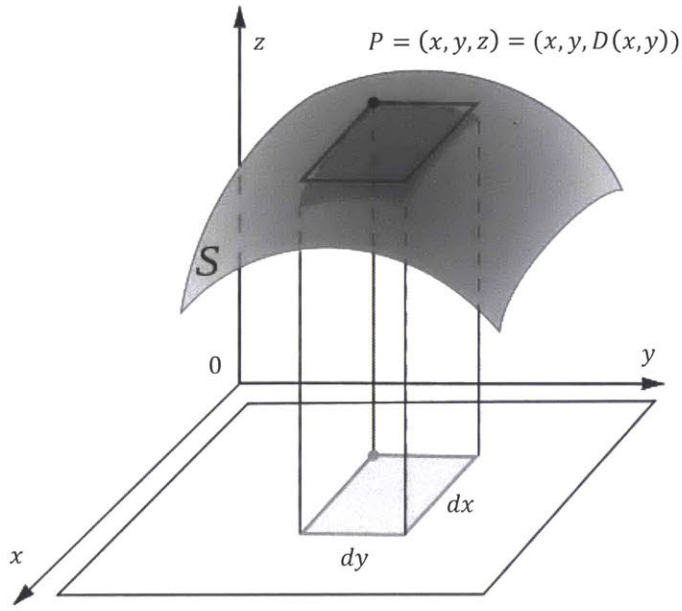


Figure 4-7: If the area of lesion S is directly calculated from the color image without using the depth information, the area projected on the $x-y$ plane is obtained instead of the actual surface area.

image at the distance of the lesion, so that one can get the correspondence between pixels and real distance. Secondly, this method only calculates the projected surface area, not the true curved surface area. In our system, using the depth information, we are able to calculate the absolute area in squared millimeters instead of the relative area. We now describe how we calculate the area of the segmented lesion.

Let S be the lesion to be measured as shown in Figure 4-7. Assume that the sensor plane is the $x-y$ plane. Any point $P = (x, y, z) = (x, y, D(x, y))$ corresponds to a point in the color image with coordinates (x, y) and a depth value $D(x, y)$. The surface area of S is given by:

$$\iint_S \sqrt{\left(\frac{\partial D}{\partial x}\right)^2 + \left(\frac{\partial D}{\partial y}\right)^2 + 1}\, dx\, dy \tag{4.6}$$

If the depth gradient is ignored, that is, $\frac{\partial D}{\partial x} = \frac{\partial D}{\partial y} = 0$, then the above formula gives the projected surface area $\iint_S dx\, dy$.

The surface integral from Equation (4.6) takes a continuous depth function, but the data from the hardware is discrete. Consider that the camera has a field of view

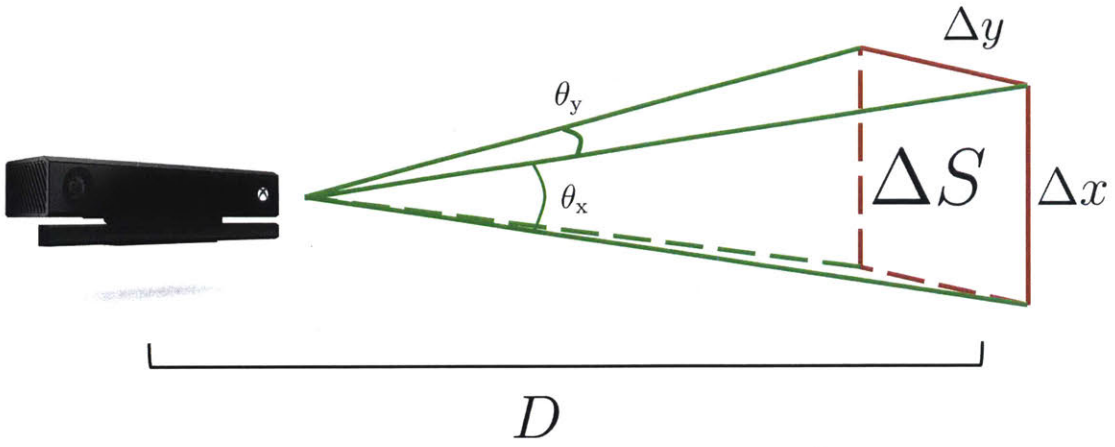


Figure 4-8: One single pixel captures a projected area ΔS at distance D.

$F_x \times F_y$ degrees with a resolution of $R_x \times R_y$ pixels. Assume that the camera is ideal without distortion. Then, each pixel corresponds to a field of view of $\theta_x \times \theta_y$ degrees where $\theta_x = F_x/R_x$ and $\theta_y = F_y/R_y$. Consider that a projected surface ΔS with width Δx and height Δy at distance D is captured in one single pixel. Assume that both θ_x and θ_y are small, then by trigonometry, $\Delta x = D\theta_x$ and $\Delta y = D\theta y$. Thus, we have $\Delta S = \Delta x \Delta y = D^2\theta_x\theta_y$, as shown in Figure 4-8.

Since the surface is not necessarily flat, we also need to incorporate the gradient of depth in order to give a better approximation. The gradient of depth at pixel (i, j) can be approximated using the central difference theorem. Consider three horizontally aligned pixels $(i-1, j)$, (i, j) and $(i+1, j)$ that correspond to physical locations $x-t$, x and $x+t$. These locations have distances $L(x-t)$, $L(x)$ and $L(x+t)$ respectively, where $L(x) = D(i, j)$, as shown in Figure 4-9. Given that a single pixel has a horizontal field angle of θ_x, we can see that $t = D\theta_x$. Then, at location x or pixel (i, j), the horizontal gradient approximation is given by:

$$\frac{\partial L(x)}{\partial x} \approx \frac{L(x+t) - L(x-t)}{2t} = \frac{D(i+1, j) - D(i-1, j)}{2D(i, j)\theta_x} \equiv D_x(i, j) \tag{4.7}$$

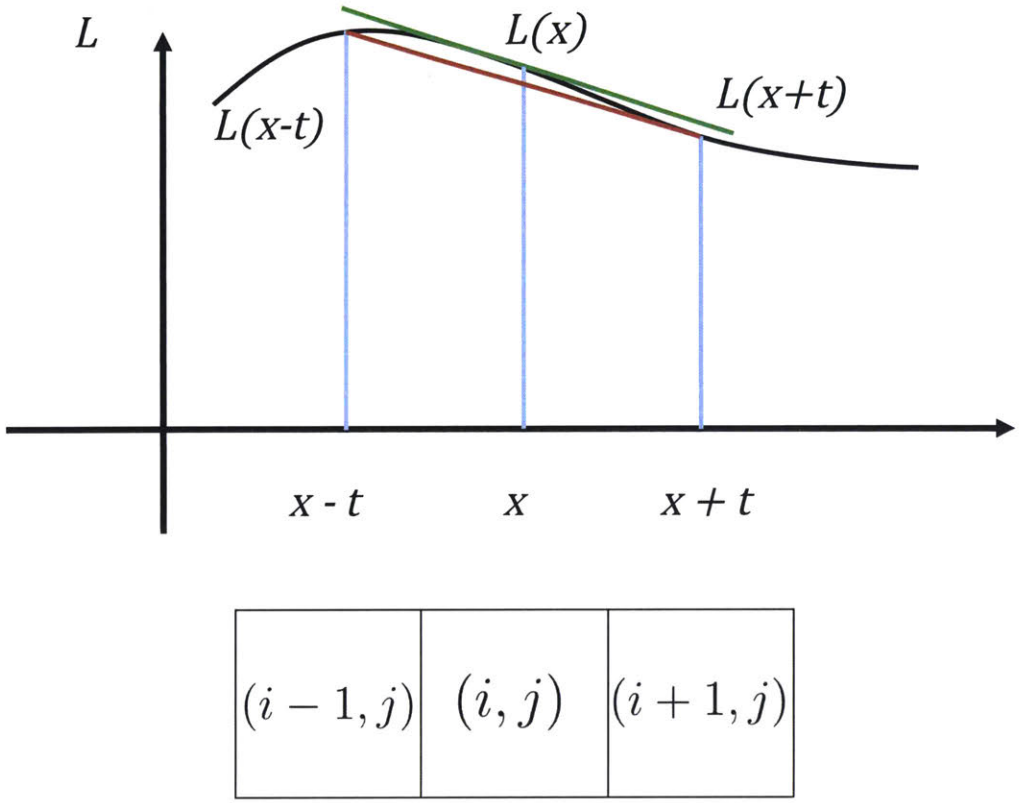


Figure 4-9: Example of using central difference to approximate gradient. Three pixels $(i-1,j)$, (i,j) and $(i+1,j)$ correspond to three physical location $x-t$, x and $x+t$. The gradient (green) at point $(x, L(x))$ is approximated by the slope (red) of the line connecting point $(x-t, L(x-t))$ and point $(x+t, L(x+t))$.

Using the same argument, the vertical gradient approximation is given by:

$$\frac{\partial L(y)}{\partial y} \approx \frac{L(y+t) - L(y-t)}{2t} = \frac{D(i,j+1) - D(i,j-1)}{2D(i,j)\theta_y} \equiv D_y(i,j) \tag{4.8}$$

The surface area is the sum of the area of each pixel in the lesion and can be written as:

$$A \simeq \sum_{p \epsilon Lesion} D(p)^2 \sqrt{D_x(p)^2 + D_y(p)^2 + 1}\, \theta_x\, \theta_y \tag{4.9}$$

Once the lesion area is obtained, it can be stored in a clinical database for progress tracking.

4.4 Results

4.4.1 Patient Demographics

To validate our imaging approach, after institutional review board approval, 9 patients with a diagnosis of vitiligo were recruited for the study, with different skin types and having varied presentation and size of vitiligo lesions. All the data was collected on-site at Brigham and Women's Hospital/Faulkner by Dr. Victor Huang.

4.4.2 Distance Calibration

Figure 4-10: Area measurement of a box with a known surface area at 914 mm (approximately 3 ft). The image taken with a Kinect V2. A ruler on the floor measures the true distance of the box. Left: The color image. Middle: The color image overlaid with the mapped depth image. Right: Segmentation result.

In order to test the accuracy of area calculation of this system, we measure a box with known surface area at different distances. This box has a surface area of $524.80mm^2$. The experimental set-up is shown in Figure 4-10. The test results are shown in Table 4.1. From the results, we see that Kinect version 1 has its optimal measurement at around 0.6 m while Kinect version 2 has its optimal measurement at around 0.9 m. We use Kinect version 2 in our experiments since it gives a more accurate measurement, but the accuracy of depth measurement goes down with an increase in depth. This is because Kinect V2 uses time-of-flight to measure depth, and the light returned from farther away pixels is weaker and leads to more noise in the depth estimate. From this experiment we conclude that to obtain the most accurate surface area measurements using Kinect V2, we should image the lesions at about 0.9 m distance from the camera.

Table 4.1: Area measurement at different distances

Distance (mm)	609	762	914	1067	1219	1371
Area from Kinect V1 (mm^2)	492.11	487.16	491.70	464.86	463.56	461.53
Area from Kinect V2 (mm^2)	437.64	504.94	507.88	490.30	494.41	479

4.4.3 Surface Area Measurements and Comparison

We use our approach to measure the lesion surface area for 9 patients in two different lighting conditions: natural lighting and UV Wood's lamp lighting. These results are presented in Figure 4-11. For comparison, we also manually trace the lesion boundary on a transparent sheet with a marker at the same time. This contour is then retraced on the planimetry device and area measurement is recorded for each lesion. We also take an image of the transparent sheet with the manually traced lesion and a ruler, use our segmentation algorithm to segment out the contour and measure the number of pixels occupied by the lesion. We then convert the area in number of pixels to an absolute area by measuring how many pixels span 15 cm on the ruler. Note that the planimetry measurements are not perfectly accurate and error may result from inaccuracy of manual segmentation, inaccuracy of the planimetry machine, and the thickness of the markers used to trace the contour of the lesion etc. However, since planimetry is the most widely used method in a clinical setting, we use those results as comparison benchmarks. The figures following the results chart show the segmentation results obtained using our system for the 9 patients, the captions have a discussion of the results. For privacy reasons, we have cropped the images to hide the identities of the patients. The figures also show the hand traced lesion boundaries side by side for comparison.

From the chart in Figure 4-11, it can be seen that out of the 14 measurements, 10 show close correspondence between the area calculated by our system in at least one lighting condition and at least one benchmark measurement (planimetry or pixel counting with calibration). It should be noted that even the benchmark measurements may have many sources of error as described earlier. First failure case is for patient 5. In this case, the image of the lesion is taken at a very oblique angle, and

we suspect that the large error arises due to the limited depth measurement accuracy of the Kinect device. The second one is for patient 3. In this case, the measurements with our system were taken without ultraviolet lighting whereas the benchmark measurements were taken with ultraviolet lighting. Since patient 3 is light skinned, the full extent of the lesion is only discernible in the presence of ultraviolet lighting, which explains the lower area measurement obtained from our system. The last two failure cases are for patient 8. In this case, the patient is very light skinned and the lesion boundaries are barely discernible even with ultraviolet lighting. In all the other cases, our system performs well.

4.5 Conclusion

This work presents a new solution to surface area measurement of vitiligo lesions by incorporating a depth camera and image processing algorithms. We show that this system can perform lesion segmentation robustly and measure lesion area accurately over any skin surface. Compared to the currently existing approaches, this system has several advantages. It is easy to use, does not require any precise calibration or professional training. It is contact-free. This eliminates the possibility of contamination and discomfort caused by manual tracing. It can measure the absolute area of any surface.

The anticipated applications are to determine the burden of disease and response to treatment for patient care and clinical trial applications. In particular, vitiligo, eczema, psoriasis, and chronic wounds are immediate areas for application. In addition, determination of BSA of adverse dermatologic side effects would allow for granular, objective grading of adverse events or determination of BSA involvement in burns or wounds.

4.6 Future Work

There are several research directions to extend this work in the future, such as:

- **Progression measurements.** One immediate goal is to obtain area measurements from images of the same lesion taken over multiple visits. The percentage difference between them will give us a measure of treatment efficacy. Using this we can quantify the sensitivity (the minimum observable change) of our approach.

- **Depth de-noising using color image.** To improve the depth estimates for pixels with missing depth values even further, more sophisticated algorithms such as [25, 33, 17, 27] could be used which use information from the color image to fill holes in the depth image.

- **Extension to other dermatological conditions.** Our intermediate to long term goal is to adapt these approaches to be appropriate for the evaluation of inflammatory skin diseases (e.g. psoriasis, atopic dermatitis, drug reactions, etc.), wounds (e.g. graft-versus-host disease, chronic wounds, etc.) and pigmented lesions (e.g. atypical moles, melasma, etc.). This would entail making modifications to the algorithm based on pathology (e.g. de-pigmented lesions of vitiligo versus pigmented lesions such as melanocytic nevi versus erythematous lesions of drug reactions versus volumetric lesion determination in wounds). Our final objective is to optimize the imaging hardware for clinical use and workflows.

4.7 Acknowledgements

This work has been done in collaboration with MIT student, Jiarui Huang, who worked on the python implementation of the algorithm and its initial testing as a part of his Master's thesis, and Dr. Victor Huang from Brigham and Women's Hospital: Boston Hospital & Medical Center, who collected the patient data using our system, and provided clinical insights into the results.

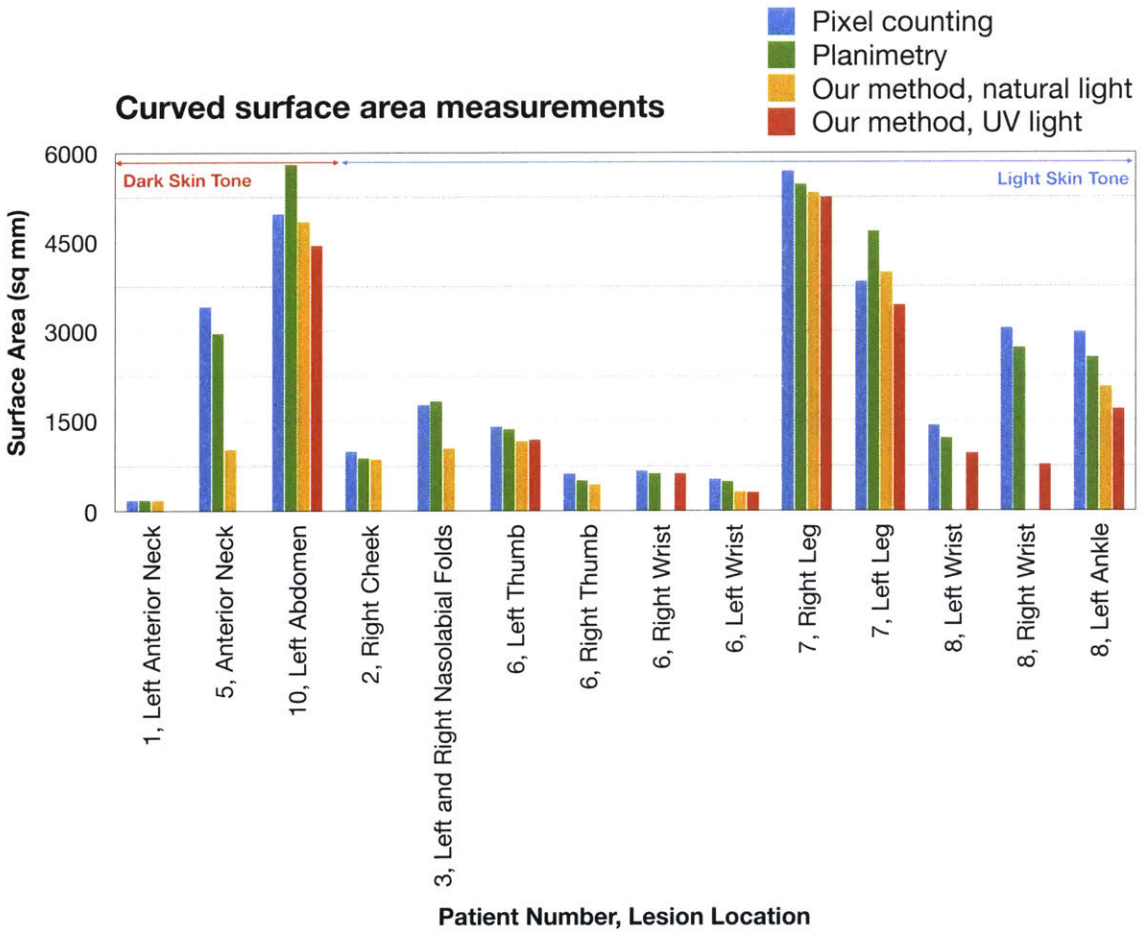


Figure 4-11: Surface area measurements for 9 patients using images taken with different lighting conditions and comparison to planimetry measurements taken with the planimetry device and obtained by pixel counting.

Figure 4-12: Results for patient 1, lesion on the left anterior neck. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary.

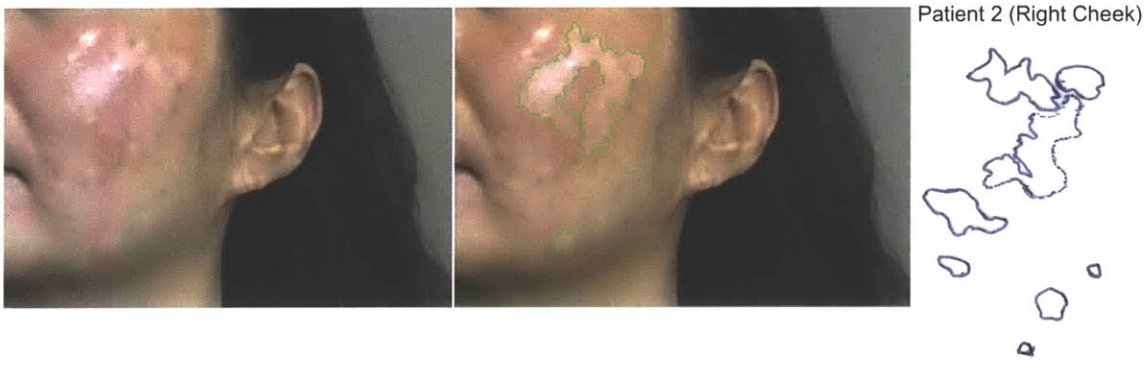


Figure 4-13: Results for patient 2, lesion on the right cheek. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary.

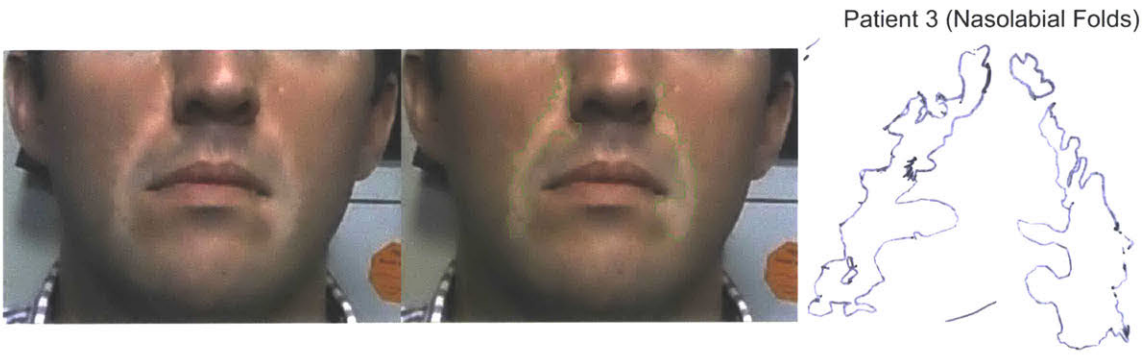


Figure 4-14: Results for patient 3, lesions on the left and right nasolabial folds. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. In this case the measurements with our system were taken without ultraviolet lighting whereas the benchmark measurements were taken with ultraviolet lighting. Since the patient is light skinned, the full extent of the lesion is only discernible in the presence of ultraviolet lighting, which explains the lower area measurement obtained from our system.

Figure 4-15: Results for patient 5, lesion on the anterior neck. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. In this case, the image of the lesion is taken at a very oblique angle, and we suspect that the large error arises due to the limited depth measurement accuracy of the Kinect device.

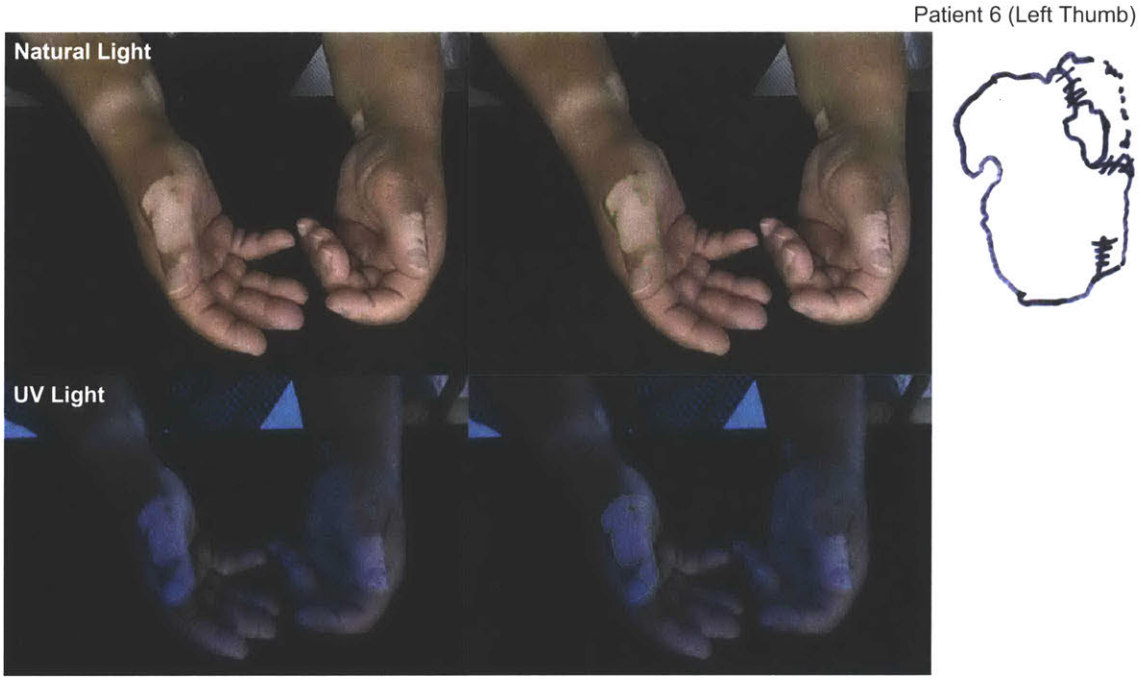


Figure 4-16: Results for patient 6, lesion on the left thumb. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. Top row shows the results with natural light, bottom row shows the results with UV light.

Figure 4-17: Results for patient 6, lesion on the right thumb. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary.

Figure 4-18: Results for patient 6, lesion on the right wrist. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary.

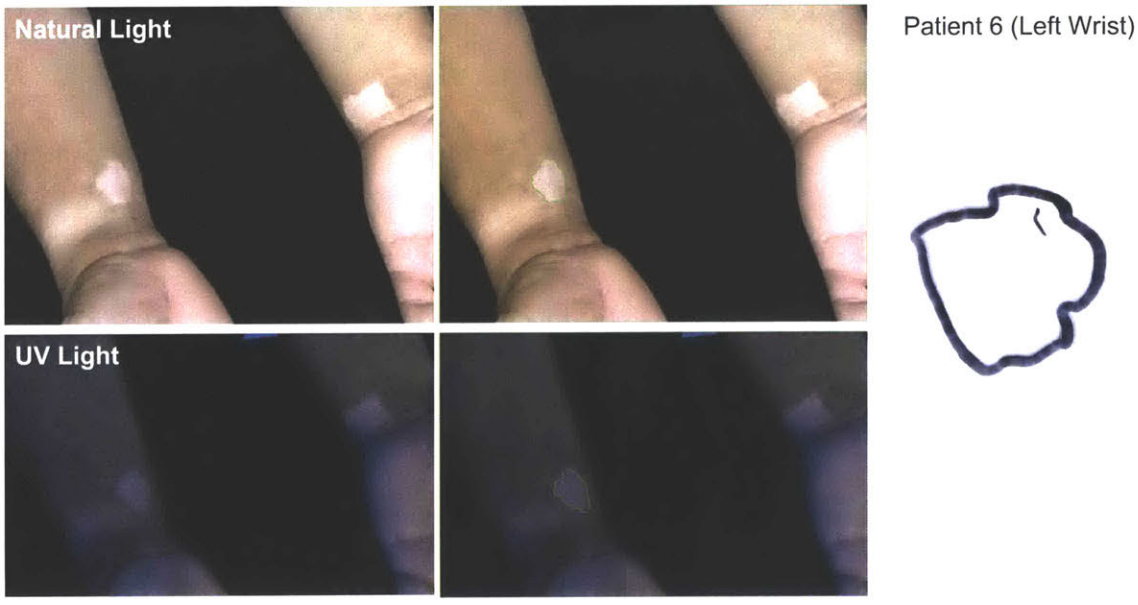


Figure 4-19: Results for patient 6, lesion on the left wrist. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. Top row shows the results with natural light, bottom row shows the results with UV light.

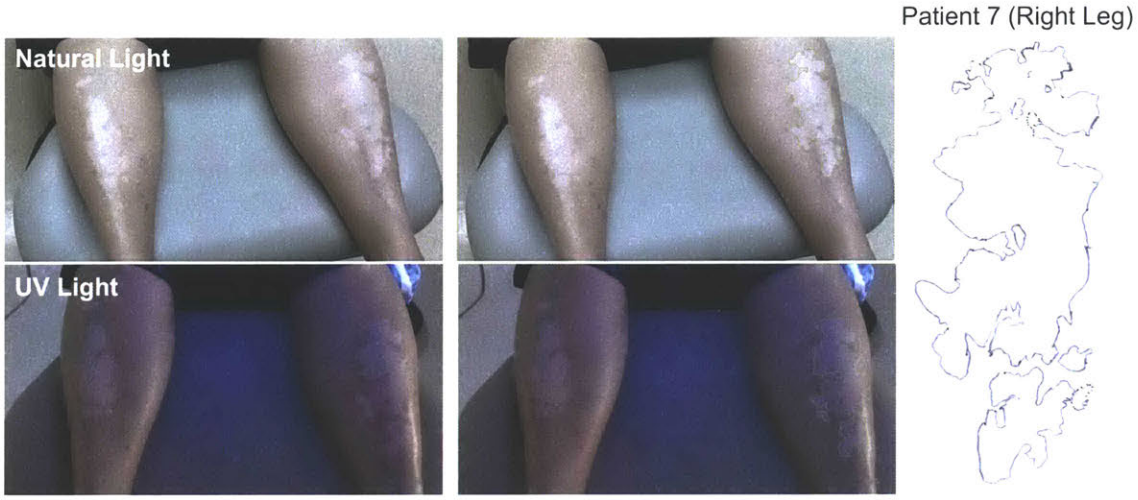


Figure 4-20: Results for patient 7, lesion on the right leg. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. Top row shows the results with natural light, bottom row shows the results with UV light.

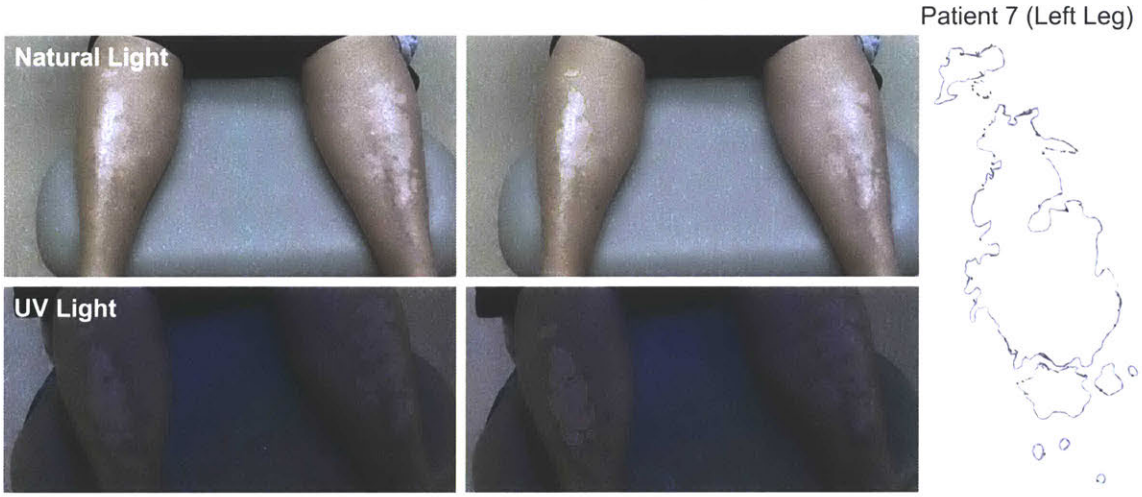


Figure 4-21: Results for patient 7, lesion on the left leg. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. Top row shows the results with natural light, bottom row shows the results with UV light.

Figure 4-22: Results for patient 8, lesion on the left wrist. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. The patient is very light skinned and the lesion boundaries are barely discernible even with ultraviolet lighting.

Figure 4-23: Results for patient 8, lesion on the right wrist. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. The patient is very light skinned and the lesion boundaries are barely discernible even with ultraviolet lighting.

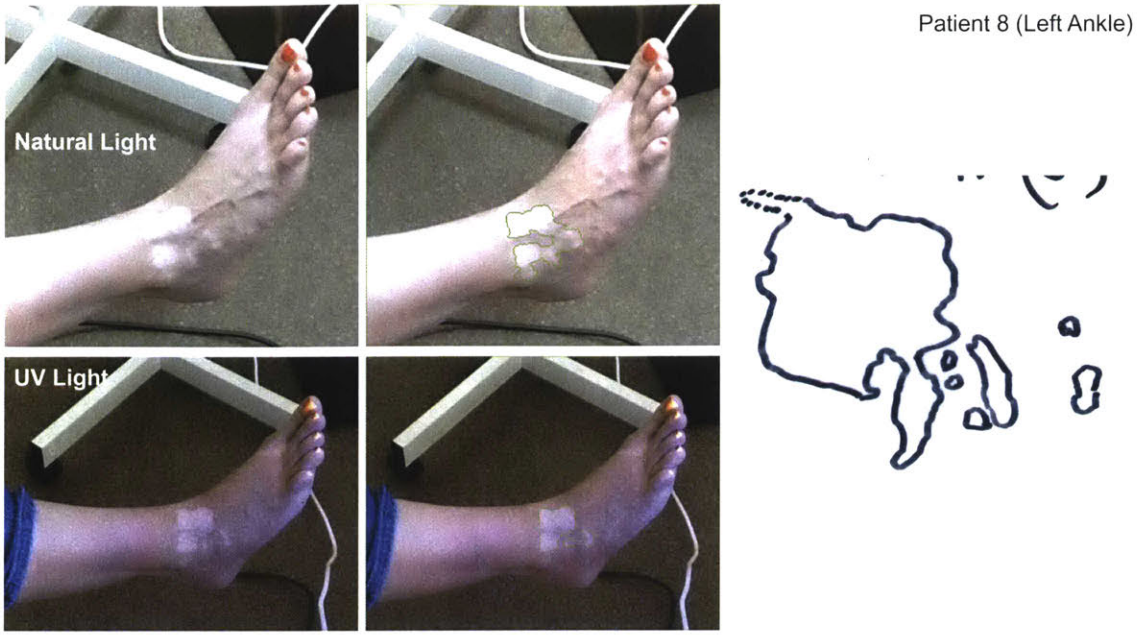


Figure 4-24: Results for patient 8, lesion on the left ankle. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. Top row shows the results with natural light, bottom row shows the results with UV light.

Figure 4-25: Results for patient 9, lesion on the left cheek. Left: Original image. Middle: Our segmentation result. Right: Manually segmented lesion boundary. Top row shows the results with natural light, bottom row shows the results with UV light.

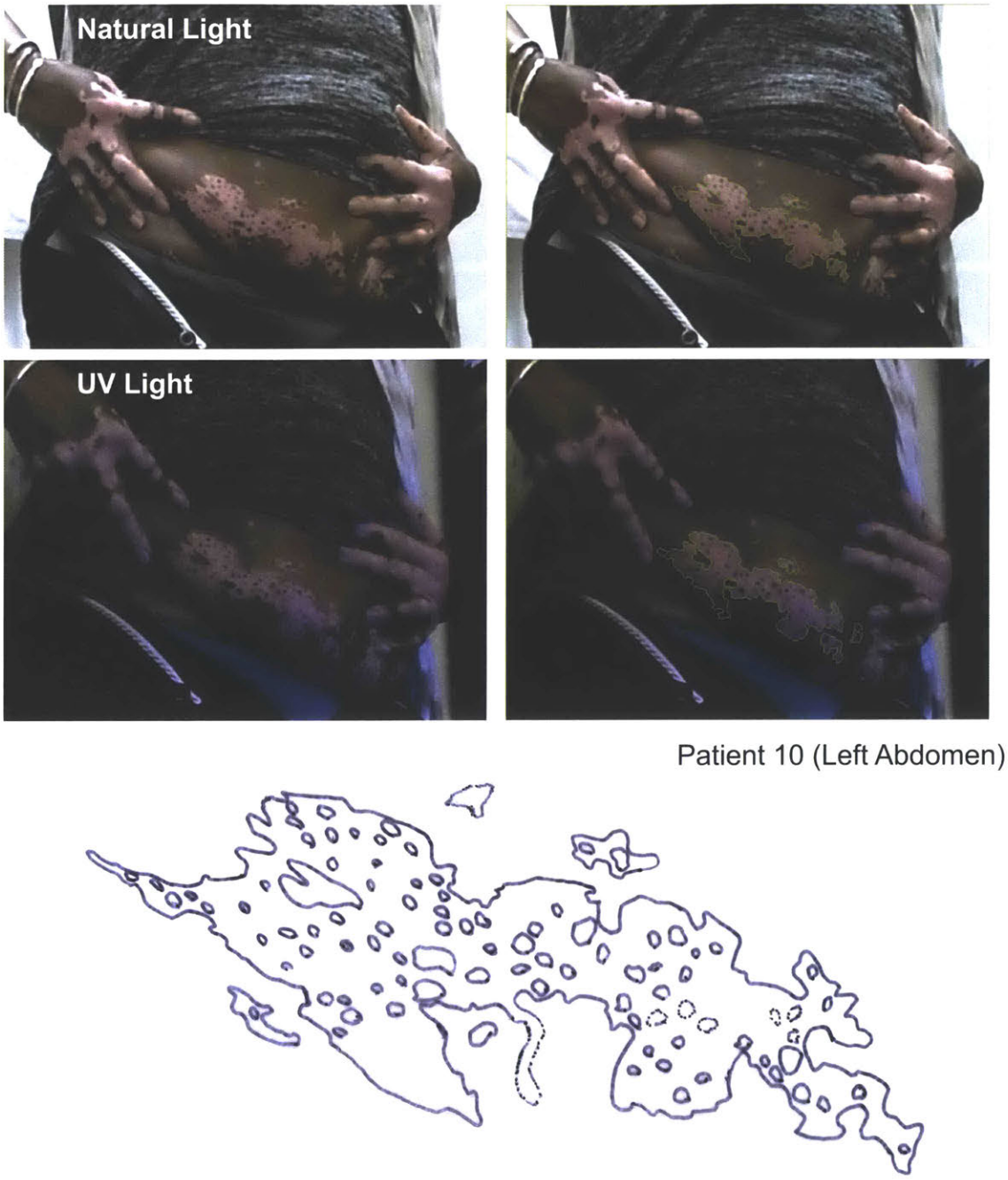


Figure 4-26: Results for patient 10, lesion on the left abdomen.
Top row - Left: Original image. Right: Our segmentation result. Natural lighting.
Middle row - Left: Original image. Right: Our segmentation result. UV lighting.
Bottom row - Manually segmented lesion boundary.

Chapter 5

Conclusion

To summarize, this thesis argues that energy-efficient hardware accelerators are required to enable complex computational imaging, vision and machine learning applications on mobile devices. We demonstrate that hardware acceleration can lead to two to three orders of magnitude improvement in performance and three to four orders of magnitude improvement in energy for applications such as image deblurring and motion magnification using several energy minimization techniques.

We present three complete computational imaging systems: (1) an energy-scalable accelerator for image deblurring, (2) a low-power processor for real-time motion magnification in videos, and (3) a 3D imaging platform for automated surface area assessment of dermatologic lesions. In future work this approach can be extended to a variety of new applications such as autonomous visual understanding, augmented reality, and portable medical imaging systems, to create efficient implementations of the algorithms involved for energy-constrained mobile devices.

The first part of this thesis presents the first hardware accelerator for kernel estimation in image deblurring applications. It features a multi-resolution IRLS-based deconvolution engine with DFT based matrix multiplication which achieves at least 8.8× reduction in the number of floating point operations, a highly parallel image correlator with diagonal computation reuse and image tiling which achieves a speedup of two orders of magnitude over the baseline, and a selective update based gradient projection solver which achieves 11× increase in speed and 56% reduction in area

compared to the baseline.

The accelerator achieves a 78× reduction in kernel estimation time, and a 56× reduction in the total deblurring time of 1920 × 1080 images with respect to a CPU, and three orders of magnitude reduction in energy. The accelerator supports up to 10× energy scalability through configurability in iterations and kernel size, allowing the system to trade off runtime with image quality in energy-constrained scenarios. This energy-scalable implementation enables efficient integration of image deblurring into mobile devices.

There are several possible directions in which this project can be extended, for example, the current algorithm assumes a spatially invariant blur, that is, the same blur kernel is used for deblurring different parts of the image. While this assumption is approximately valid in many common camera shake scenarios, it is not valid in general. One possible direction for future work would be to extend this work to handle spatially variant blur, by determining blur kernels from several patches in a single image, using them to deblur the image locally, and then reconstituting the final deblurred image. Since the accelerator determines the kernel from one patch at a time, it should be portable to this multi-patch scenario. Another possible extension would be to evaluate the energy scalable nature of this accelerator while deblurring a video. It would be interesting to measure how quickly the algorithm converges if the kernel for one frame is initialized to the final kernel from the previous frame, and measure the resulting energy savings. Numerical optimization is required in several computer vision algorithms such as visual odometry and video super-resolution. One extension could involve porting and evaluating the architecture of the numerical optimizers (conjugate gradient and gradient projection) proposed in this work in the context of these new applications.

In the second part of this thesis, we present a low-power processor to accelerate motion magnification that achieves real-time performance on HD (1280 × 720) video at 30 frames per second, while consuming less than 0.2 nanojoules of processing energy per pixel. We propose to use the following techniques in the design of the accelerator to achieve energy-efficiency and real-time performance: (1) Separable

filtering in pyramid computation and spatial filtering to reduce the number of multiplies and adds. (2) Zero-skipping to reduce the buffering requirements of pyramid computation. (3) Reducing the precision of computation in several modules to reduce energy and selectively using block floating point representation to maintain accuracy. (4) Caching intermediates in on-chip SRAM based line buffers to reduce external memory bandwidth and save system energy. The complete motion magnification architecture is demonstrated on an FPGA and simulated area and power measurements in 40 nm CMOS technology are presented in this thesis. These techniques enable efficient integration of motion magnification technology into mobile devices.

There are several directions in which this work can be extended. We are working on fabricating the motion magnification processor. Whereas the number of chip I/Os is not a concern in the FPGA implementation, it presents a major limitation for the on-chip implementation. The on-chip SRAMs are not big enough to hold a complete frame worth of intermediates locally, so these intermediates are read from and written to the DRAM. Since each intermediate is required in each cycle, this leads to a large I/O and DRAM bandwidth requirement. To solve these problems, we clock the DRAM and the I/O at a higher frequency than the chip itself. We can also trade off DRAM and I/O bandwidth and energy with processor area and energy. Another technique that could be used for energy minimization is active pixels detection. We define active pixels as the pixels which are undergoing the motion. Active pixels are typically less than 50% of total pixels since large portions of the scene remains static from frame to frame. These active pixels can be detected from the temporal filter output, and we can perform motion magnification for the subsequent frames on the active pixels only to reduce the amount of computation, memory bandwidth and energy. Finally, we can perform dynamic voltage and frequency scaling together with active pixels detection to reduce energy while maintaining the throughput.

The third part of this thesis presents a new solution to surface area measurement of vitiligo lesions by incorporating a depth camera and image processing algorithms. We show that this system can perform lesion segmentation robustly and measure lesion area accurately over any skin surface. Compared to the currently existing

approaches, this system has several advantages. It is easy to use, does not require any precise calibration or professional training. It is contact-free. This eliminates the possibility of contamination and discomfort caused by manual tracing. It can measure the absolute area of any surface.

We use our approach to measure the lesion surface area for 9 patients in two different lighting conditions: natural lighting and UV Wood's lamp lighting. We compare the results obtained from our system with those obtained by manual tracing/segmentation of the lesion followed by either planimetry or pixel counting to compute area. Out of 14 measurements, 10 show close correspondence between the area calculated by our system in at least one lighting condition and at least one benchmark measurement (planimetry or pixel counting with calibration). We analyze the failure cases to inform future image collection, for example, we conclude that very oblique angles and absence of supplementary lighting while imaging patients with a light skin-tone can lead to a large error in the measured area.

There are several research directions to extend this work in the future. One immediate goal is to obtain area measurements from images of the same lesion taken over multiple visits to assess disease progression. The percentage difference between them will give us a measure of treatment efficacy. Using this we can quantify the sensitivity (the minimum observable change) of our approach. To improve the depth estimates for pixels with missing depth values even further, more sophisticated algorithms could be used which use information from the color image to fill holes in the depth image. Our intermediate to long term goal is to adapt these approaches to be appropriate for the evaluation of inflammatory skin diseases (e.g. psoriasis, atopic dermatitis, drug reactions, etc.), wounds (e.g. graft-versus-host disease, chronic wounds, etc.) and pigmented lesions (e.g. atypical moles, melasma, etc.). This would entail making modifications to the algorithm based on pathology (e.g. de-pigmented lesions of vitiligo versus pigmented lesions such as melanocytic nevi versus erythematous lesions of drug reactions versus volumetric lesion determination in wounds). Our final objective is to optimize the imaging hardware for clinical use and workflows.

Bibliography

[1] Imagej. Available at `http://imagej.nih.gov`.

[2] Visitrak. `http://www.smith-nephew.com/new-zealand/healthcare/products/product-types/visitrak/`.

[3] Visitrak digital. `http://www.smith-nephew.com/documents/anz/visitrakdigitaluserguide\%5b1\%5d.pdf`.

[4] Abdulrahman A. A. Amer and Xing-Hua Gao. Quality of life in patients with vitiligo: an analysis of the dermatology life quality index outcome over the past two decades. *International Journal of Dermatology*, 55(6):608–614, 2016.

[5] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008.

[6] S. Beucher and Centre De Morphologie Mathmatique. The watershed transformation applied to image segmentation. In *Scanning Microscopy International*, pages 299–314, 1991.

[7] Paul H Calamai and Jorge J Moré. Projected gradient methods for linearly constrained problems. *Mathematical programming*, 39(1):93–116, 1987.

[8] Justin G Chen, Neal Wadhwa, Young-Jin Cha, Frédo Durand, William T Freeman, and Oral Buyukozturk. Structural modal identification through high speed camera video: Motion magnification. In *Topics in Modal Analysis I, Volume 7*, pages 191–197. Springer, 2014.

[9] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, Jan 2017.

[10] Abe Davis, Michael Rubinstein, Neal Wadhwa, Gautham Mysore, Frédo Durand, and William T. Freeman. The visual microphone: Passive recovery of sound from video. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 33(4):79:1–79:10, 2014.

[11] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[12] M. Hirsch, C. J. Schuler, S. Harmeling, and B. Schölkopf. Fast removal of non-uniform camera shake. In *2011 International Conference on Computer Vision*, pages 463–470, Nov 2011.

[13] Chao-Tsung Huang, Mehul Tikekar, Chiraag Juvekar, Vivienne Sze, and Anantha Chandrakasan. A 249mpixel/s hevc video-decoder chip for quad full hd applications. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 162–163. IEEE, 2013.

[14] InfoTrends. 2016 U.S. consumer digital camera end user study, May 2016.

[15] D. Jeon, N. Ickes, P. Raina, H. C. Wang, D. Rus, and A. P. Chandrakasan. A 0.6V 8mW 3D vision processor for a navigation device for the visually impaired. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 416–417, Jan 2016.

[16] L. Komen, V. da Graça, A. Wolkerstorfer, M.A. de Rie, C.B. Terwee, and J.P.W. van der Veen. Vitiligo area scoring index and vitiligo european task force assessment: reliable and responsive instruments to measure the degree of depigmentation in vitiligo. *British Journal of Dermatology*, 172(2):437–443, 2015.

[17] Anh Le, Seung-Won Jung, and Chee Won. Directional joint bilateral filter for depth images. *Sensors*, 14(7):11362–11378, Jun 2014.

[18] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman. Efficient marginal likelihood optimization in blind deconvolution. In *2011 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2657–2664, June 2011.

[19] Anat Levin, Rob Fergus, Frédo Durand, and William T Freeman. Deconvolution using natural image priors. 2007.

[20] Stephen M. Pizer, E. Philip Amburn, John D. Austin, Robert Cromartie, Ari Geselowitz, Trey Greer, Bart ter Haar Romeny, John B. Zimmerman, and Karel Zuiderveld. Adaptive histogram equalization and its variations. *Computer Vision, Graphics, and Image Processing*, 39(3):355–368, 9 1987.

[21] R. Rithe, P. Raina, N. Ickes, S. V. Tenneti, and A. P. Chandrakasan. Reconfigurable processor for energy-efficient computational photography. *IEEE Journal of Solid-State Circuits (JSSC)*, 48(11):2908–2919, Nov 2013.

[22] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. "grabcut": Interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph.*, 23(3):309–314, August 2004.

[23] Camille Salzes, Sophie Abadie, Julien Seneschal, Maxine Whitton, Jean-Marie Meurant, Thomas Jouary, Fabienne Ballanger, Franck Boralevi, Alain Taieb, Charles Taieb, and Khaled Ezzedine. The vitiligo impact patient scale (vips): Development and validation of a vitiligo burden assessment tool. *Journal of Investigative Dermatology*, 136(1):52–58, 2015.

[24] Caroline A Schneider, Wayne S Rasband, and Kevin W Eliceiri. Nih image to imagej: 25 years of image analysis. *Nature methods*, 9(7):671–675, 07 2012.

[25] Ju Shen and Sen-Ching S. Cheung. Layer depth denoising and completion for structured-light rgb-d cameras. In *Proceedings of the 2013 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR '13, pages 1187–1194, Washington, DC, USA, 2013. IEEE Computer Society.

[26] Vaneeta M. Sheth, Rahul Rithe, Amit G. Pandya, and Anantha Chandrakasan. A pilot study to determine vitiligo target size using a computer-based image analysis program. *Journal of the American Academy of Dermatology*, 73(2):342–345, 2015.

[27] Wanbin Song, Anh Vu Le, Seokmin Yun, Seung-Won Jung, and Chee Sun Won. Depth completion for kinect v2 sensor. *Multimedia Tools Appl.*, 76(3):4357–4380, February 2017.

[28] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–511. IEEE, 2001.

[29] Neal Wadhwa, Michael Rubinstein, Frédo Durand, and William T. Freeman. Phase-based video motion processing. *ACM Trans. Graph. (Proceedings SIGGRAPH 2013)*, 32(4), 2013.

[30] Neal Wadhwa, Michael Rubinstein, Frédo Durand, and William T. Freeman. Riesz pyramids for fast phase-based video magnification. In *2014 IEEE International Conference on Computational Photography (ICCP)*, pages 1–10, May 2014.

[31] Hao-Yu Wu, Michael Rubinstein, Eugene Shih, John Guttag, Frédo Durand, and William T. Freeman. Eulerian video magnification for revealing subtle changes in the world. *ACM Trans. Graph. (Proceedings SIGGRAPH 2012)*, 31(4), 2012.

[32] Tianfan Xue, Michael Rubinstein, Neal Wadhwa, Anat Levin, Frédo Durand, and William T. Freeman. Refraction wiggles for measuring fluid depth and velocity from video. *Proc. of European Conference on Computer Vision (ECCV)*, 2014.

[33] J. Yang, X. Ye, K. Li, C. Hou, and Y. Wang. Color-guided depth recovery from rgb-d data using an adaptive autoregressive model. *IEEE Transactions on Image Processing*, 23(8):3443–3458, Aug 2014.

[34] Karel Zuiderveld. Graphics gems iv. chapter Contrast Limited Adaptive Histogram Equalization, pages 474–485. Academic Press Professional, Inc., San Diego, CA, USA, 1994.