Architectures for Computational Photography

by

Priyanka Raina

B. Tech., Indian Institute of Technology Delhi (2011)

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

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2013

Certified by .....
Anantha P. Chandrakasan
Joseph F. and Nancy P. Keithley Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Theses

Architectures for Computational Photography

by

Priyanka Raina

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

Computational photography refers to a wide range of image capture and processing techniques that extend the capabilities of digital photography and allow users to take photographs that could not have been taken by a traditional camera. Since its inception less than a decade ago, the field today encompasses a wide range of techniques including high dynamic range (HDR) imaging, low light enhancement, panorama stitching, image deblurring and light field photography.

These techniques have so far been software based, which leads to high energy consumption and typically no support for real-time processing. This work focuses on hardware architectures for two algorithms - (a) bilateral filtering which is commonly used in computational photography applications such as HDR imaging, low light enhancement and glare reduction and (b) image deblurring.

In the first part of this work, digital circuits for three components of a multi-application bilateral filtering processor are implemented - the grid interpolation block, the HDR image creation and contrast adjustment blocks, and the shadow correction block. An on-chip implementation of the complete processor, designed with other team members, performs HDR imaging, low light enhancement and glare reduction. The 40 nm CMOS test chip operates from 98 MHz at 0.9 V to 25 MHz at 0.9 V and processes 13 megapixels/s while consuming 17.8 mW at 98 MHz and 0.9 V, achieving significant energy reduction compared to previous CPU/GPU implementations.

In the second part of this work, a complete system architecture for blind image deblurring is proposed. Digital circuits for the component modules are implemented using Bluespec SystemVerilog and verified to be bit accurate with a reference software implementation. Techniques to reduce power and area cost are investigated and synthesis results in 40nm CMOS technology are presented.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Joseph F. and Nancy P. Keithley Professor of Electrical Engineering

Acknowledgments

First, I would like to thank Prof. Anantha Chandrakasan, for being a great advisor and a source of inspiration to me and for guiding and supporting me through the course of my research.

I would like to thank Rahul Rithe and Nathan Ickes, my colleagues on the bilateral filtering project. Working with both of you on this project right from conceptualization and design to testing has been a great learning experience for me. I would also like to thank Rahul Rithe for going through this thesis with me and providing some of the figures.

I would like to thank all the members of *ananthagroup* for making the lab such an exciting place to work to at. I would like to thank Mehul Tikekar, for being a great teacher and a friend, for helping me out whenever I was stuck, for teaching me Bluespec, for teaching me python and for going through this thesis. I would like to thank Michael Price and Rahul Rithe for helping me out with the gradient projection solver, and Chiraag Juvekar for answering thousands of my quick questions.

I would like to thank Foxconn Technology Group for funding and supporting the two projects. In particular, I would like to thank Yihui Qiu for the valuable feedback and suggestions. I would like to thank TSMC University Shuttle Program for the chip fabrication.

I would like to thank Abhinav Uppal for being a great friend, for being supportive as well as for being critical, for teaching me how to code and for teaching me how to use vim. I would like to thank Anasuya Mandal for being a wonderful roommate, for all the good times we have spent together and for all the good food.

Lastly, I would like to thank my parents for their love, support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The world of photography has seen a drastic transformation in the last 23 years with the advent of digital cameras. They have made photography accessible to all, with images ready for viewing the moment they are captured. In February, 2012, Facebook announced that they were getting 3000 pictures uploaded to their servers every second - that is 250 million pictures every day. The field of computational photography takes this to the next level. It refers to a wide range of image capture and processing techniques that enhance or extend the capabilities of digital photography and allow users to take photographs that could not have been taken by a traditional digital camera. Since its inception less than a decade ago, the field today encompasses a wide range of techniques such as high dynamic range (HDR) imaging [1], low-light enhancement [2, 3], panorama stitching [4], image deblurring [5] and light field photography [6], which allow users to not just capture a scene flawlessly, but also reveal details that could otherwise not be seen. However, most of these techniques have high computational complexity which necessitates fast hardware implementations to enable real-time applications. In addition, energy-efficient operation is a critical concern when running these applications on battery-powered hand-held devices such as phones, cameras and tablets.

Computational photography applications have so far been software based, which leads to high energy consumption and typically no support for real-time processing. This work identifies the challenges in hardware implementation of these techniques

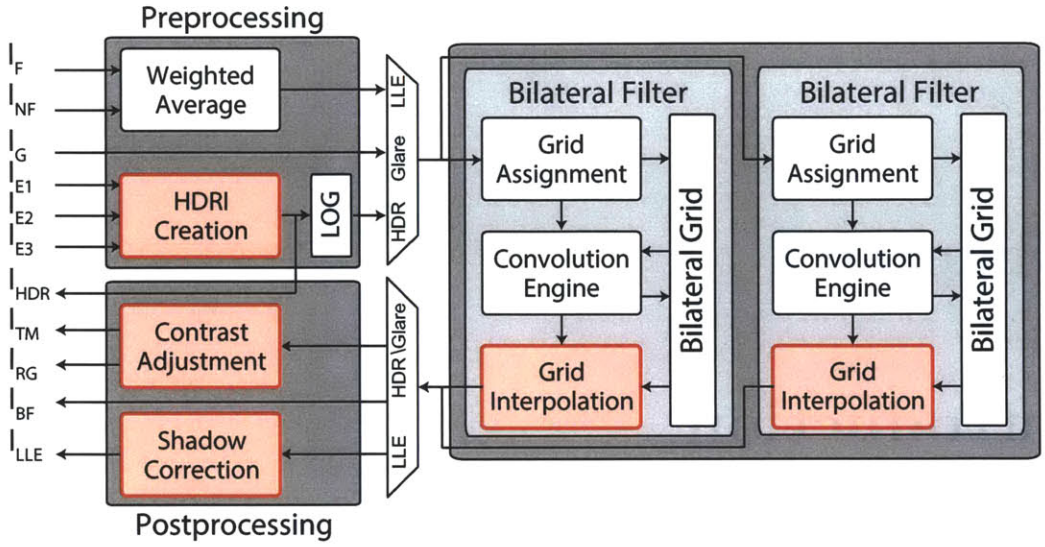


Figure 1-1: System block diagram for reconfigurable bilateral filtering processor. The blocks in red highlight the key contributions of this work. (Courtesy R.Rithe)

and investigates ways to reduce their power and area cost. This involves algorithmic optimizations to reduce computational complexity and memory bandwidth, parallelized architecture design to enable high throughput while operating at low frequencies and circuit optimizations for low voltage operation. This work focuses on hardware architectures for two algorithms - (a) bilateral filtering, which is commonly used in applications such as HDR imaging, low light enhancement and glare reduction and (b) image deblurring. The next section gives a brief overview of the two algorithms and highlights the key contributions of this work. The following two chapters describe the proposed architectures for bilateral filtering and image deblurring in more detail and the last chapter summarizes the results.

1.1 Contributions of this work

Bilateral Filtering

A bilateral filter is an edge-preserving smoothing filter, which is a commonly used filter in computational photography applications. In the first part of this work, digital circuits for the following three components of a multi-application bilateral filtering

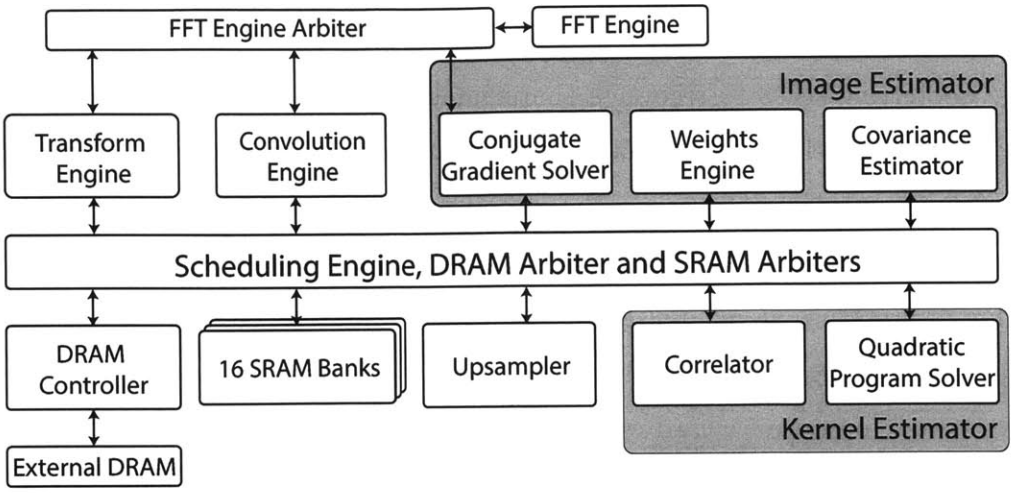


Figure 1-2: System block diagram for image deblurring processor.

processor are designed and implemented - the grid interpolation block, the HDR image creation and contrast adjustment blocks, and the shadow correction block.

These components are put together along with others as shown in Figure 1-1 into a reconfigurable multi-application processor with two bilateral filtering engines at its core. The processor, designed with other team members, can be configured to perform HDR imaging, low light enhancement and glare reduction. The filtering engine can also be accessed from off-chip and used with other applications. The input images are preprocessed for the specific functions and fed into the filter engines which operate in parallel and decompose an image into into a low frequency base layer and a high frequency detail layer. The filtered images are post processed to generate outputs for the specific functions.

The processor is implemented in 40 nm CMOS technology and achieves 15× reduction in run-time compared to a CPU implementation, while consuming 1.4 mJ/megapixel energy, a significant reduction compared to CPU or GPU implementations. This energy scalable implementation can be efficiently integrated into portable multimedia devices for real time computational photography.

Image Deblurring

Image deblurring seeks to recover a sharp image from its blurred version, given no knowledge about the camera motion when the image was captured. Blur can be caused due to a variety of reasons; the second part of this work focuses on recovering images that are blurred due to camera shake during exposure. Deblurring algorithms existing in literature are computationally intensive and take on the order of minutes to run for HD images, when implemented in software.

This work proposes a hardware architecture based on the deblurring algorithm presented in [5]. Figure 1-2 shows the system architecture. The major challenges are the high computation cost and memory requirements for the image and kernel estimation blocks. These are addressed in this work by designing an architecture which minimizes off chip memory accesses by effective caching using on-chip SRAMs and maximizes data reuse through parallel processing. The system is designed using Bluespec SystemVerilog (BSV) as the hardware description language and verified to be bit accurate with the reference software implementation. Synthesis results in TSMC 40nm CMOS technology are presented.

The next two chapters describe the algorithms and the proposed architectures in detail.

Chapter 2

Bilateral Filtering

A bilateral filter is a non-linear filter which smoothes an image while preserving edges in the image [7]. For an image I, at position p, it is defined by:

$$I_{BFp} = \frac{1}{W_p} \sum_{q \epsilon N(p)} G_{\sigma_s}(\| p - q \|) G_{\sigma_r}(|I_p - I_q|) I_q \tag{2.1}$$

$$W_p = \sum_{q \epsilon N(p)} G_{\sigma_s}(\| p - q \|) G_{\sigma_r}(|I_p - I_q|) \tag{2.2}$$

The output value at each pixel in the image is a weighted average of the values in a neighborhood, where the weight is the product of a Gaussian on the spatial distance (G_{σ_s}) and a Gaussian on the pixel value/range difference (G_{σ_r}). In linear Gaussian filtering, on the other hand, the weights are determined solely by the spatial term. The inclusion of the range term makes bilateral filter effective in respecting strong edges, because pixels across an edge are very different in the range dimension, even though they are close in the spatial dimension, and get low weights. Figure 2-1 compares the effectiveness of a bilateral filter and a linear Gaussian filter in reducing noise in an image while preserving scene details.

A direct implementation of a bilateral filter, however, is computationally expensive since the kernel (weights array) used for filtering has to be updated at every pixel according to the values of neighboring pixels, and it can take on the order of several minutes to process HD images [8]. Faster approaches for bilateral filtering have been

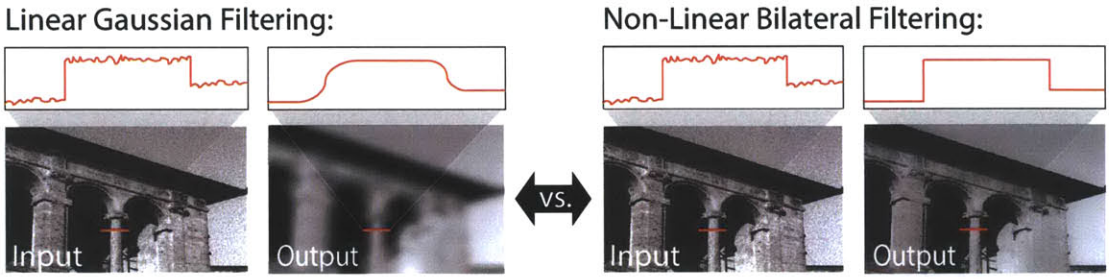


Figure 2-1: Comparison of Gaussian filtering and bilateral filtering. Bilateral filtering effectively reduces noise while preserving scene details. (Courtesy R.Rithe)

proposed that reduce processing time by using a piece-wise linear approximation in the intensity domain and appropriate sub-sampling [1]. [9] uses a higher dimensional space and formulates the bilateral filter as a convolution followed by simple nonlinearities. A fast approach to bilateral based on a box spatial kernel, which can be iterated to yield smooth spatial fall-off is proposed in [10]. However, real-time processing of HD images requires further speed-up.

2.1 Bilateral Grid

A software-based bilateral grid data structure proposed in [8] enables fast bilateral filtering. In this approach, a 2-D input image is partitioned into blocks (of size $\sigma_s \times \sigma_s$) as shown in Figure 2-2 and a histogram of pixel intensity values (with $256/\sigma_r$ bins) is generated for each block. These block level histograms, when put together for all the blocks in the image, create a 3-D representation of the 2-D image, called a bilateral grid, where each grid cell stores the number of pixels in the corresponding histogram bin of the block and their summed intensity. A bilateral grid has two key advantages:

- **Aggressive down-sampling** The size of the blocks used while creating the grid and the number of intensity bins determine the amount by which the image is down-sampled. Aggressive down-sampling reduces the number of computations required for processing as well as the amount of memory required for storing the grid.

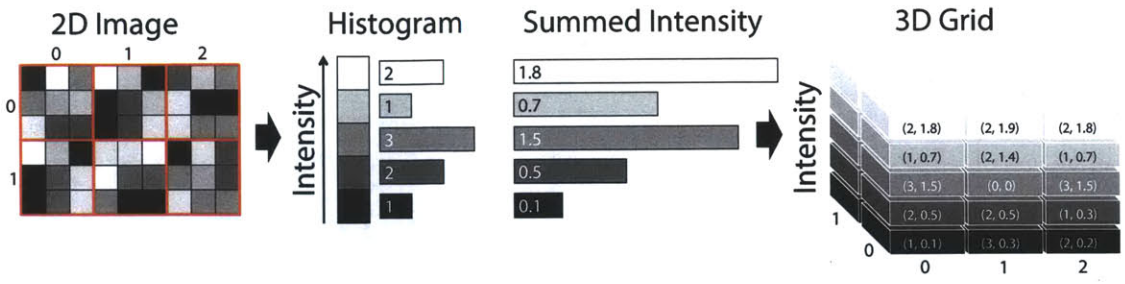


Figure 2-2: Construction of a 3-D bilateral grid from a 2-D image. (Courtesy R.Rithe)

- **Built-in edge awareness** Two pixels that are spatially adjacent but have very different intensities end up far apart in the grid in the intensity dimension. When linear filtering is performed on the grid using a 3-D Gaussian kernel, only nearby intensity levels influence the result whereas levels that are farther away do not contribute to the result. Therefore, linear Gaussian filtering on the grid followed by slicing to generate a 2-D image is equivalent to performing bilateral filtering on the 2-D image.

2.2 Bilateral Filter Engine

Bilateral filter engine using a bilateral grid is implemented as shown in Figure 2-3. It consists of three components - grid assignment engine, grid filtering engine and grid interpolation engine. Grid assignment and filtering engines are briefly described next for completeness. The contribution of this work is the design of grid interpolation engine described later in this section.

2.2.1 Grid Assignment

The input image is scanned pixel by pixel in a block-wise manner and fed into 16, 8 or 4 grid assignment engines operating in parallel (depending on the number of intensity bins being used). The number of pixels in a block is scalable from 16×16 pixels to 128×128. Each grid assignment engine compares the intensity of the input pixel with the boundaries of the intensity bin assigned to it and if the pixel intensity lies in the

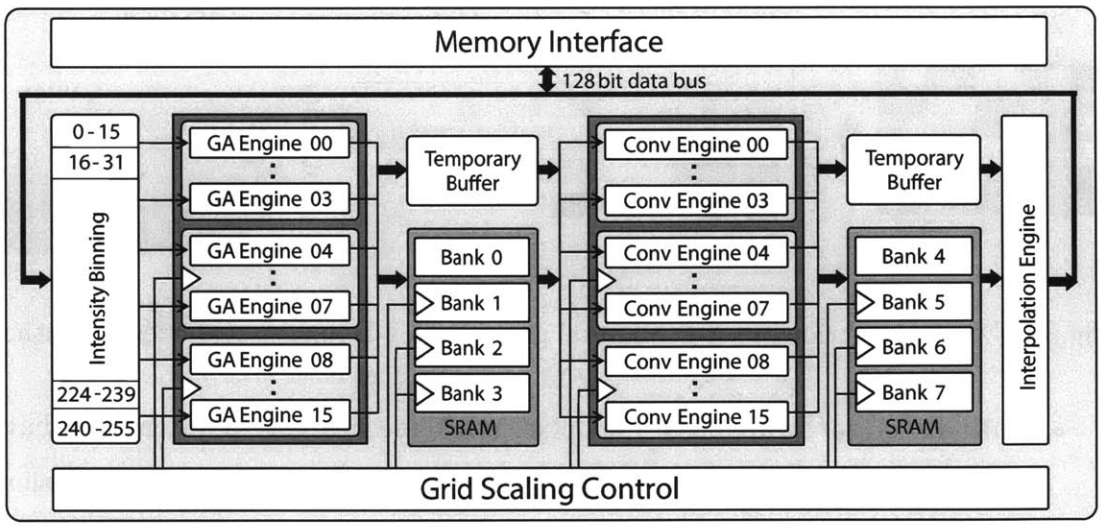


Figure 2-3: Architecture of the bilateral filtering engine. Grid scalability is achieved by gating processing engines and SRAM banks. (Courtesy R.Rithe)

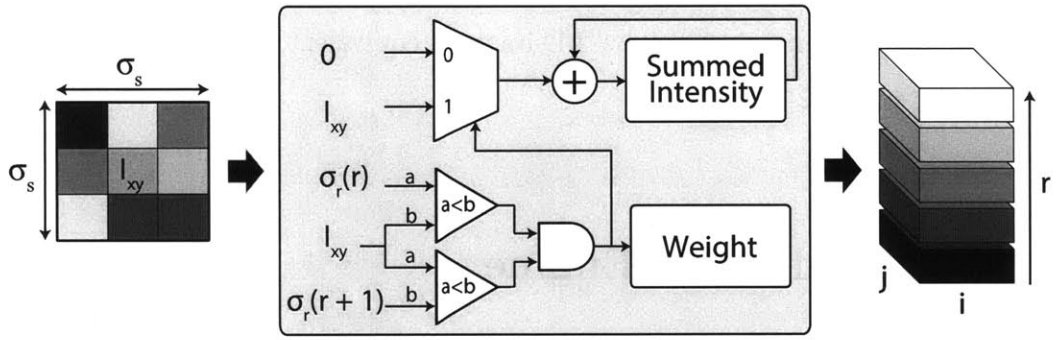


Figure 2-4: Architecture of grid assignment engine. (Courtesy R.Rithe)

range, it is accumulated into the intensity bin and a weight counter is incremented. Figure 2-4 shows the architecture of grid assignment engine. Both summed intensity and weight are stored for each bin in on-chip memory.

2.2.2 Grid Filtering

Convolution (Conv) engine, shown in Figure 2-5, convolves the grid intensities and weights with a $3 \times 3 \times 3$ Gaussian kernel and returns the normalized intensity. The convolution is performed by multiplying the 27 coefficients of the filter kernel with the 27 grid cells and adding them using a 3-stage adder tree. The intensity and weight

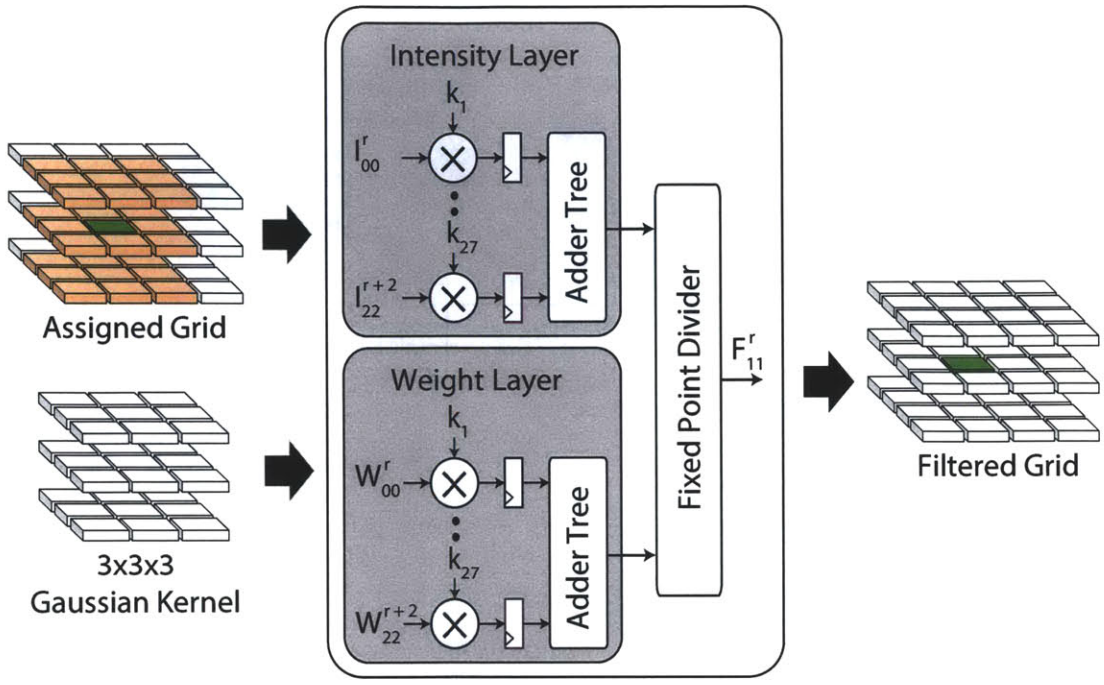


Figure 2-5: Architecture of convolution engine for grid filtering. (Courtesy R.Rithe)

are convolved in parallel and the convolved intensity is normalized with the convolved weight by using a fixed point divider to make sure that there is no intensity scaling during filtering. The hardware has 16 convolution engines that can operate in parallel to filter a grid with 16 intensity levels, but 4 or 8 of them can be activated if fewer intensity levels are used.

2.2.3 Grid Interpolation

Interpolation engine constructs the output 2-D image from the filtered grid. To obtain the output intensity value at pixel (x, y), its intensity value in the input image I_{xy} is read from the DRAM. So, the number of interpolation engines that can be activated in parallel is limited by the number of pixels that can be read from the memory per cycle. The output intensity value is obtained by trilinear interpolation of $2 \times 2 \times 2$ neighborhood of filtered grid values. Trilinear interpolation is implemented as three successive pipelined linear interpolations as shown in Figure 2-6.

Figure 2-7 shows the architecture of the linear interpolator, which basically con-

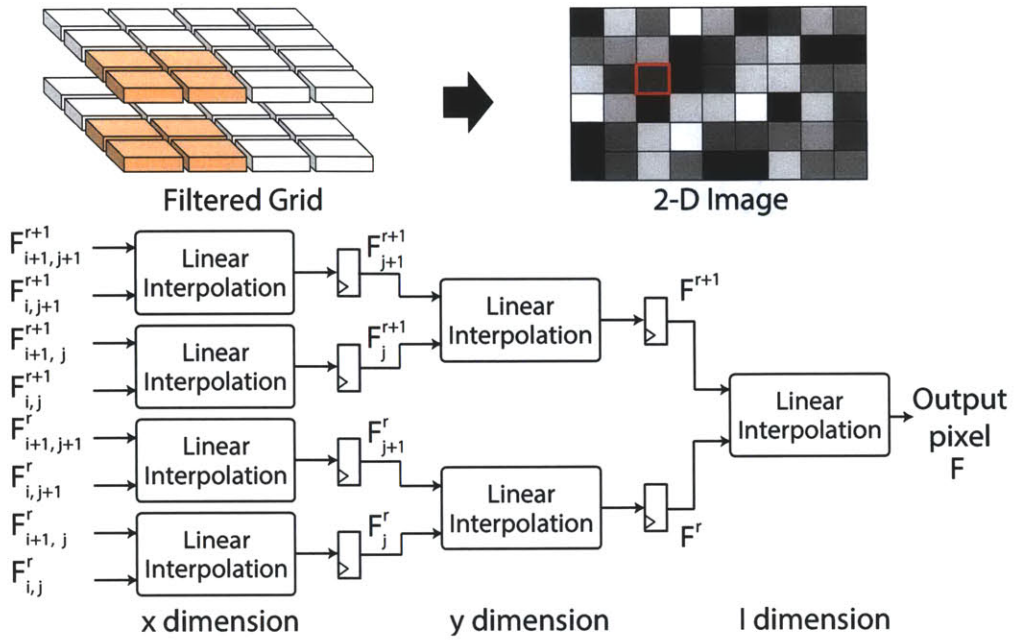


Figure 2-6: Architecture of the interpolation engine. Tri-linear interpolation is implemented as three pipelined stages of linear interpolations.

sists of performing a weighted average of the two inputs, where weights are determined by the distance to the output pixel along the dimension in which the interpolation is being performed. The division by σ_s at the end reduces to a shift because σ_s values used in the system are powers of 2. The output value F is calculated from filtered

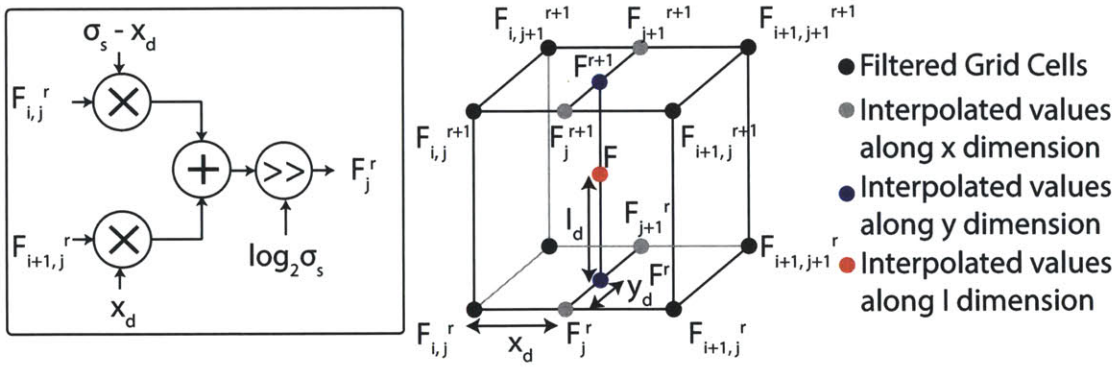


Figure 2-7: Architecture of the linear interpolator.

grid values $F^r_{i,j}$ using four parallel linear interpolations along the x dimension:

$$F^r_j = (F^r_{i,j} * (\sigma_s - x_d) + F^r_{i+1,j} * x_d)/\sigma_s \quad (2.3)$$

$$F^r_{j+1} = (F^r_{i,j+1} * (\sigma_s - x_d) + F^r_{i+1,j+1} * x_d)/\sigma_s \quad (2.4)$$

$$F^{r+1}_j = (F^{r+1}_{i,j} * (\sigma_s - x_d) + F^{r+1}_{i+1,j} * x_d)/\sigma_s \quad (2.5)$$

$$F^{r+1}_{j+1} = (F^{r+1}_{i,j+1} * (\sigma_s - x_d) + F^{r+1}_{i+1,j+1} * x_d)/\sigma_s \quad (2.6)$$

followed of two parallel interpolations along the y dimension:

$$F^r = (F^r_j * (\sigma_s - y_d) + F^r_{j+1} * y_d)/\sigma_s \quad (2.7)$$

$$F^{r+1} = (F^{r+1}_j * (\sigma_s - y_d) + F^{r+1}_{j+1} * y_d)/\sigma_s \quad (2.8)$$

followed by a final interpolation along the r dimension:

$$F = (F^r * (\sigma_r - I_d) + F^{r+1} * I_d)/\sigma_r \quad (2.9)$$

where $i = \lfloor x/\sigma_s \rfloor$, $j = \lfloor y/\sigma_s \rfloor$ and $r = \lfloor I_{xy}/\sigma_r \rfloor$, and $x_d = x - \sigma_s * i$, $y_d = y - \sigma_s * j$, and $I_d = I_{xy} - \sigma_r * r$. The interpolated output is fed into the downstream application-specific processing blocks.

2.2.4 Memory Management

The bilateral filtering engine does not store the complete bilateral grid on chip. Since the kernel size is $3 \times 3 \times 3$ and since the processing happens in row major order, only 2 complete grid rows and 4 grid blocks of the next row are required to be stored locally. As soon as the grid assignment engines assign $3 \times 3 \times 3$ blocks, the convolution engines can start filtering the grid. Once the grid cells have been filtered they are replaced by newly assigned cells. The interpolation engine also processes in row major order and therefore, requires one complete row and 3 blocks of the next row of the filtered grid to be stored on chip. The interpolation engine starts as soon as 2×2 filtered grid blocks become available. This scheduling scheme shown in

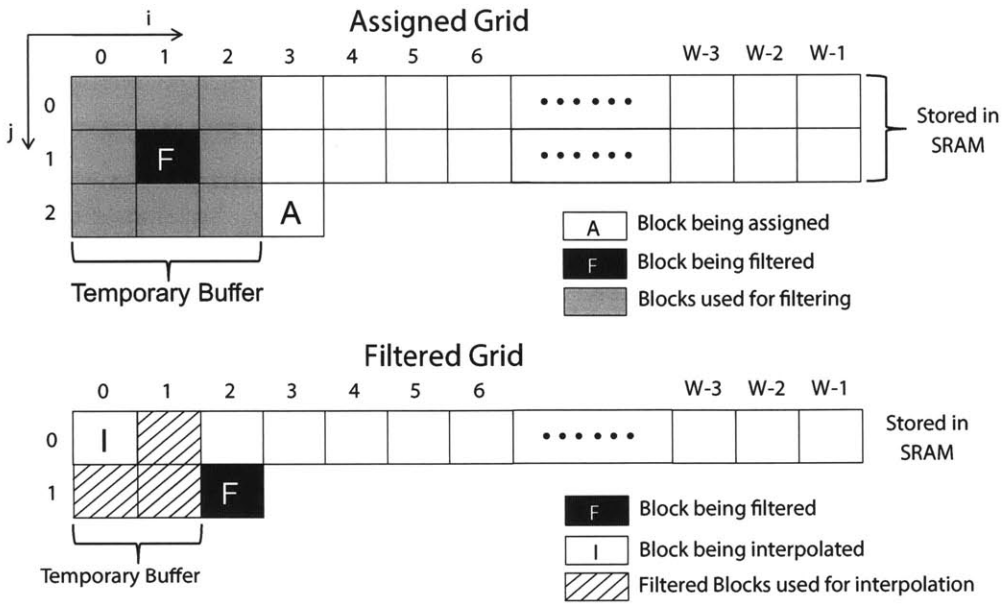


Figure 2-8: Memory management by task scheduling. (Courtesy R.Rithe)

Figure 2-8 allows processing without storing the entire grid and reduces the memory requirement to 21.5 kB from more than 65 MB (for a software implementation) for processing a 10 megapixel image and allows processing grids of arbitrary height using the same amount of on-chip memory. The test chip has two bilateral filter engines, each processing 4 pixels/cycle.

2.3 Applications

The processor performs high dynamic range (HDR) imaging, low light enhancement and glare reduction using the two bilateral filter engines. The contribution of this work is the design of the application specific modules required for performing these algorithms. HDRI creation, contrast adjustment and shadow correction modules are implemented to enable these applications. The following subsections describe the architecture of each of these modules in detail and how they interact with the bilateral filtering engines to produce the desired output.

2.3.1 High Dynamic Range Imaging

High dynamic range (HDR) imaging is a technique for capturing a greater dynamic range between the brightest and darkest regions of an image than a traditional digital camera. It is done by capturing multiple images of the same scene with varying exposure levels, such that the low exposure images capture the bright regions of the scene well without loss of detail and the high exposure images capture the dark regions of the scene. These differently exposed images are then combined together into a high dynamic range image, which more faithfully represents the brightness values in the scene. Displaying HDR images on low dynamic range (LDR) devices, such as a computer monitor and photographic prints, requires dynamic range compression without loss of detail. This is achieved by performing tone mapping using a bilateral filter which reduces the dynamic range or contrast of the entire image, while retaining local contrast.

HDRI Creation

The first step in HDR imaging is to create a composite HDR image from multiple differently exposed images which represents the true scene radiance value at each pixel of the image. To recover the true scene radiance value at each pixel from its recorded intensity values and the exposure time, the algorithm presented in [11] is used, which is briefly described below:

The exposure E is defined as the product of sensor irradiance R (which is the amount of light hitting the camera sensor and is proportional to the scene radiance) and the exposure time Δt. After the digitization process, we obtain a number I (intensity) which is a non-linear function of the initial exposure E. Let us call this function f. The non-linearity of this function becomes particularly significant at the saturation point, because any point in the scene with a radiance value above a certain level is mapped to the same maximum intensity value in the image. Let us assume that we have a number of different images with known exposure times Δt_j. The pixel

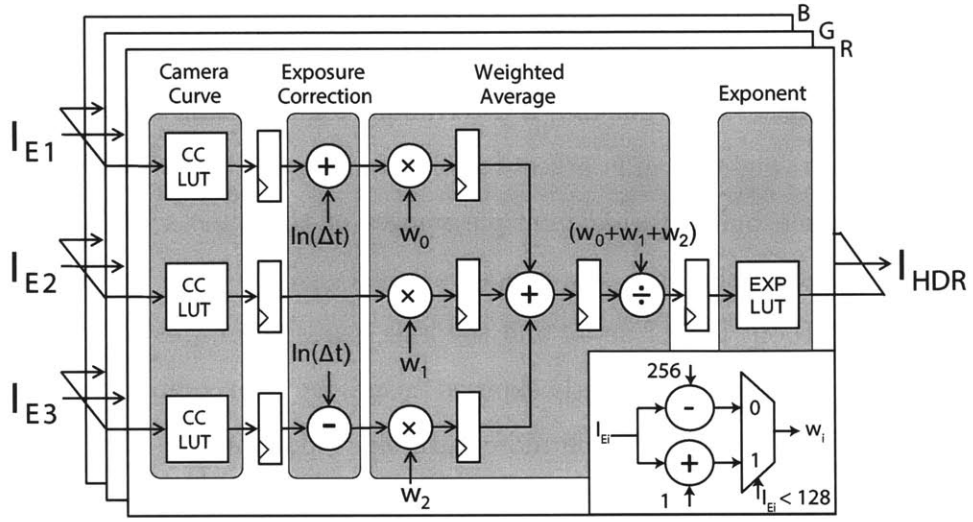


Figure 2-9: HDRI creation module.

intensity values are given by

$$I_{ij} = f(R_i \Delta t_j) \tag{2.10}$$

where i is the spatial index and j indexes over exposure times Δt_j. We then have the log of the irradiance values given by:

$$ln(R_i) = g(I_{ij}) - ln(\Delta t_j) \tag{2.11}$$

where lnf^{-1} is denoted by g. The mapping g is called the camera curve, and can be obtained by the procedure described in [11]. Once g is known, the true scene radiance values can be recovered from image pixel intensity values using the relationship described above.

The HDRI creation block, shown in Figure 2-9 takes values of a pixel from three different exposures (I_{E1}, I_{E2}, I_{E3}) and generates an output pixel which represents the true scene radiance value at that location. Since we are working with a finite range of discrete pixel values (8 bits per color), the camera curves are stored as combinational look-up tables to enable fast access. The camera curves are looked up to get the true (log) exposure values followed by exposure time correction to obtain (log) radiance.

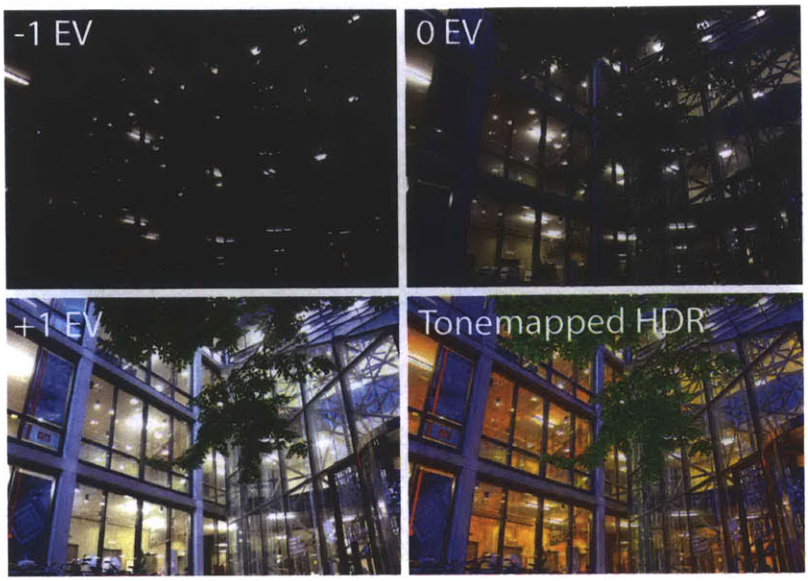


Figure 2-10: Input low-dynamic range images: -1 EV (under exposed image), 0 EV (normally exposed image) and 1 EV (over exposed image). Output image: tone mapped HDR image. (Courtesy R.Rithe)

The three resulting (log) radiance values obtained from the three images represent the radiance values of the same location in the scene. A weighted average of these three values is taken to obtain the final (log) radiance value. The weighting function, shown in Figure 2-9 gives a higher weight to the exposures in which pixel value is closer to the middle of the response function (thus avoiding the high contributions from images where the pixel value is saturated). In the end an exponentiation is performed to get the final radiance value (16 bits per pixel per color).

Tone Mapping

To perform tone mapping, the 16 bits per pixel per color HDR image is split into intensity and color channels. A low frequency base layer and a high frequency detail layer are created by bilateral filtering the HDR image in the log domain. The dynamic range of the base layer is compressed by a scaling factor in the log domain. The detail layer is untouched to preserve details and the colors are scaled linearly to 8 bits per pixel per color. Merging the compressed base layer, the detail layer and the color channels results in a tone mapped HDR image (I_{TM}). In HDR mode both bilateral

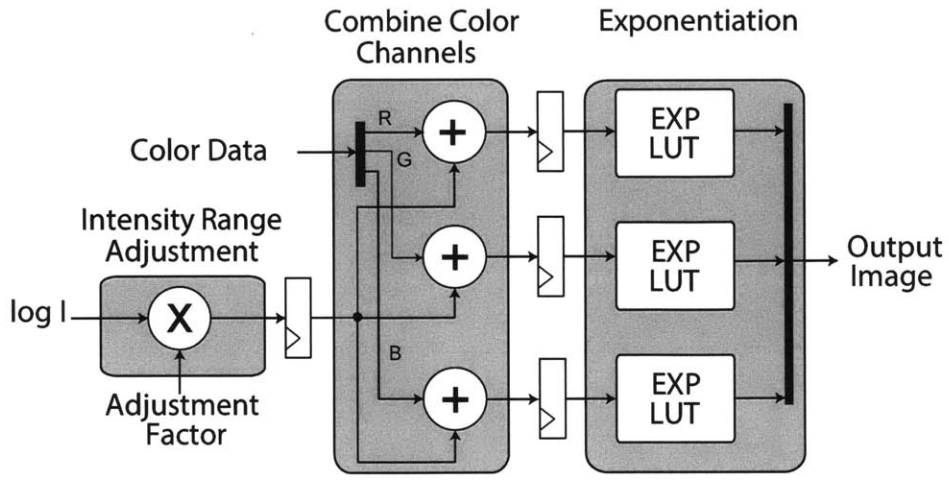


Figure 2-11: Contrast adjustment module. Contrast is increased or decreased depending on the adjustment factor.

grids are configured to perform filtering in an interleaved manner, where each grid processes alternate pixel blocks in parallel. Figure 2-10 shows a set of input low dynamic range exposures and the tone mapped HDR output image.

2.3.2 Glare Reduction

Glare reduction is similar to performing single image HDR tone mapping. The processing flow is shown in Figure 2-13(a). The input image is split into intensity and color channels. A low frequency base layer and a high frequency detail layer are obtained the bilateral filtering the intensity. The contrast layer of the base layer is enhanced using the contrast adjustment module shown in Figure 2-11 which is also used in HDR tone mapping. The contrast can be increased or decreased depending on the adjustment factor.

Figure 2-12 shows an input image with glare and the glare reduced output image. Glare reduction recovers details that are white-washed in the original image and enhances the image colors and contrast.

Figure 2-12: Input images: (a) image with glare. Output image: (b) image with reduced glare. (Courtesy R.Rithe)

2.3.3 Low Light Enhancement

Low light enhancement (LLE) is performed by merging two images captured in quick succession, one taken without flash (I_{NF}) and one with flash (I_F), as shown in Figure 2-13(b). The bilateral grid is used to decompose both images into base and detail layers. In this mode, one grid is configured to perform bilateral filtering on the non-flash image and the other to perform cross-bilateral filtering, given by Equation 2.12, on the flash image using the non-flash image. The location of the grid cell is determined by the non-flash image and the intensity value is determined by the flash image.

$$I_{CBFp} = \frac{1}{W_p} \sum_{q \epsilon N(p)} G_{\sigma_s}(\| p - q \|) G_{\sigma_r}(|I_{Fp} - I_{Fq}|) I_{NFq} \tag{2.12}$$

$$W_p = \sum_{q \epsilon N(p)} G_{\sigma_s}(\| p - q \|) G_{\sigma_r}(|I_{Fp} - I_{Fq}|) \tag{2.13}$$

The scene ambience is captured in the base layer of the non-flash image and details are captured in the detail layer of the flash image.

The image taken with flash contains shadows that are not present in the non-flash image. A novel shadow correction module is implemented which merges the details from the flash image with base layer of the cross-bilateral filtered non-flash image and corrects for the flash shadows to avoid artifacts in the output image. A mask

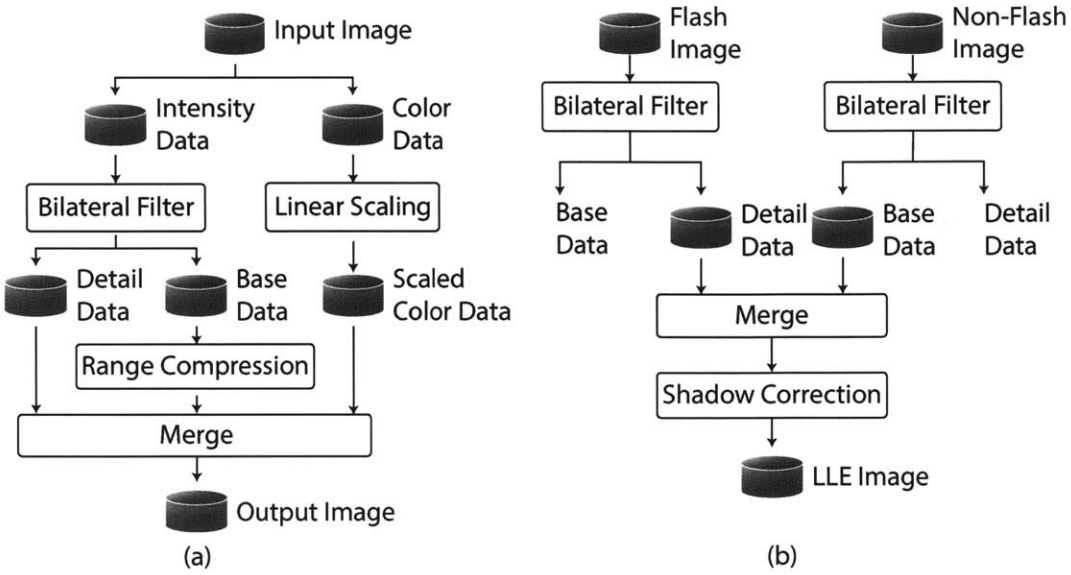


Figure 2-13: Processing flow for (a) glare reduction and (b) low light enhancement by merging flash and non-flash images. (Courtesy R.Rithe)

representing regions with high details in the filtered non-flash image is created, as shown in Figure 2-14. Gradients are computed at each pixel for blocks of 4×4 pixels. If the gradient at a pixel is higher than the average gradient for that block, the pixel is labeled as an edge pixel. This results in a binary mask that highlights all the strong edges in the scene but no false edges due to the flash shadows. The details from the flash image are added to the filtered non-flash image only in the regions represented by the mask. A linear filter is used to smooth the mask to ensure that that the resulting image does not have discontinuities. This implementation of the

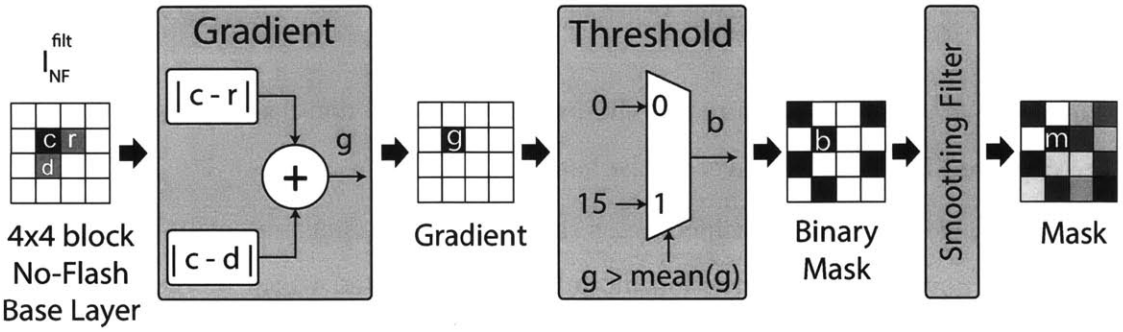


Figure 2-14: Mask creation for shadow correction.

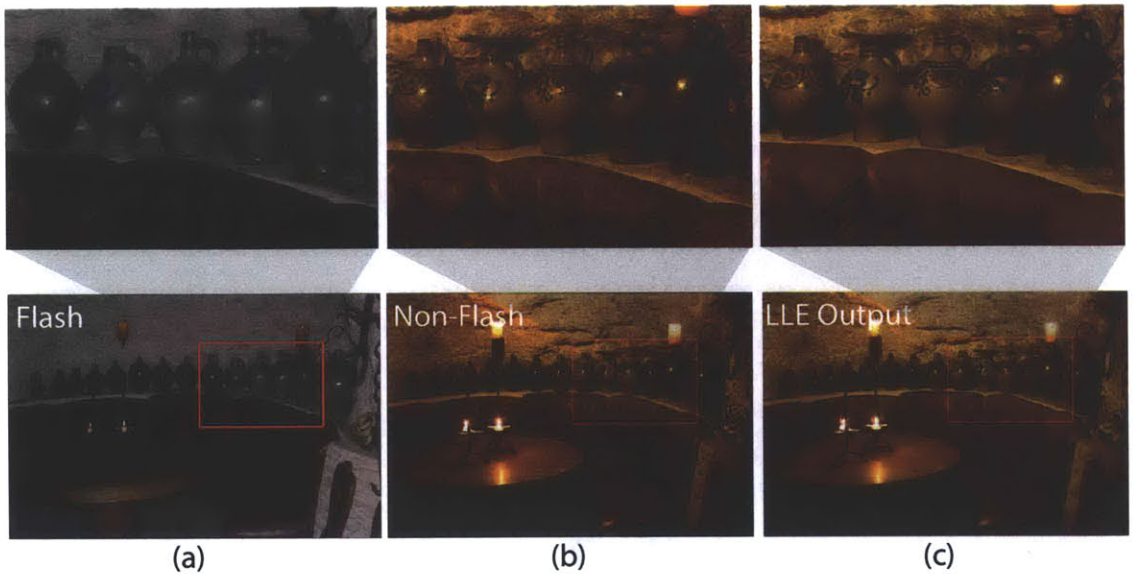


Figure 2-15: Input images: (a) image with flash, (b) image without flash. Output image: (c) low-light enhanced image. (Courtesy R.Rithe)

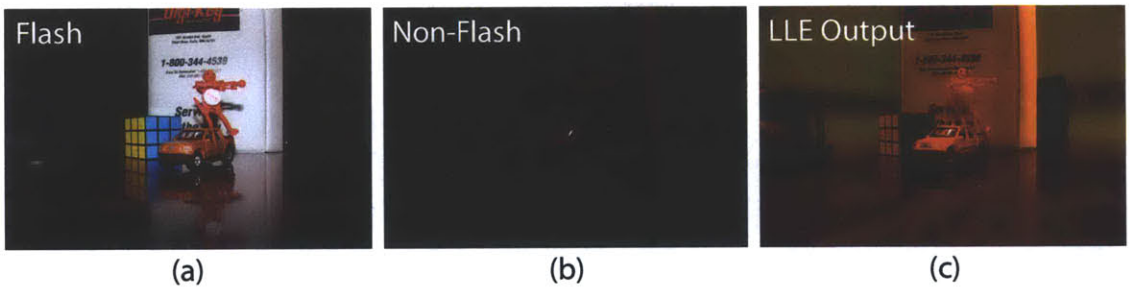
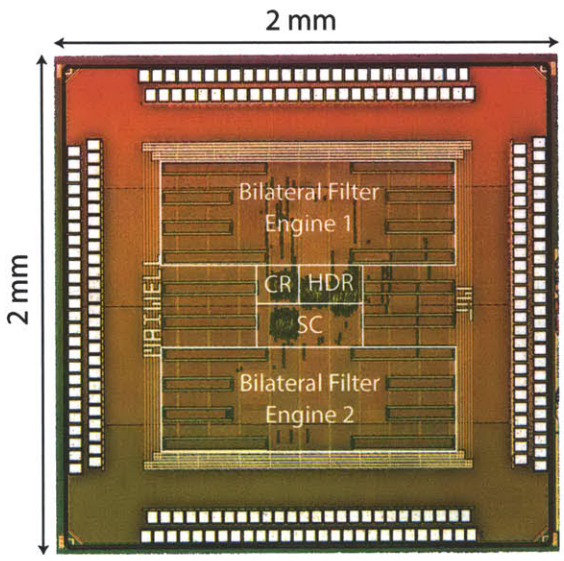


Figure 2-16: Input images: (a) image with flash, (b) image without flash. Output image: (c) low-light enhanced image. (Courtesy R.Rithe)

shadow correction module handles shadows effectively to produce enhanced images without artifacts.

Figure 2-15 shows a set of input flash and non-flash images and the low-light enhanced output image. The enhanced output effectively reduces noise while preserving details. Another set of images is shown in Figure 2-16. The flash image has shadows that are not present in the non-flash image. The bilateral filtered non-flash image reduces the noise but lacks details. The enhanced output, created by adding the details from the flash image, effectively reduces noise while preserving details and corrects for flash shadows without creating artifacts.



Chip Features

Technology	40 nm CMOS
Core Area	1.1 mm × 1.1 mm
Transistor Count	1.94 million
SRAM	21.5 kB
Core Supply Voltage	0.5 V to 0.9 V
I/O Supply Voltage	1.8 V to 2.5 V
Frequency	25 – 98 MHz
Core Power	17.8 mW (0.9 V)

Figure 2-17: Die photo of test-chip with its features. Highlighted boxes indicate SRAMs. HDR, CR and SC refer to HDRI creation, contrast reduction and shadow correction modules respectively. (Courtesy R.Rithe)

2.4 Results

The test chip, shown in Figure 2-17, is implemented in 40 nm CMOS technology and verified to be operational from 25 MHz at 0.5 V to 98 MHz at 0.9 V with SRAMs operating at 0.9 V. This chip is designed to function as an accelerator core as part of a larger microprocessor system, utilizing the systems existing DRAM resources.

For standalone testing of this chip a 32 bit wide 266 MHz DDR2 memory controller was implemented using a Xilinx XC5VLX50 FPGA shown in Figure 2-19.

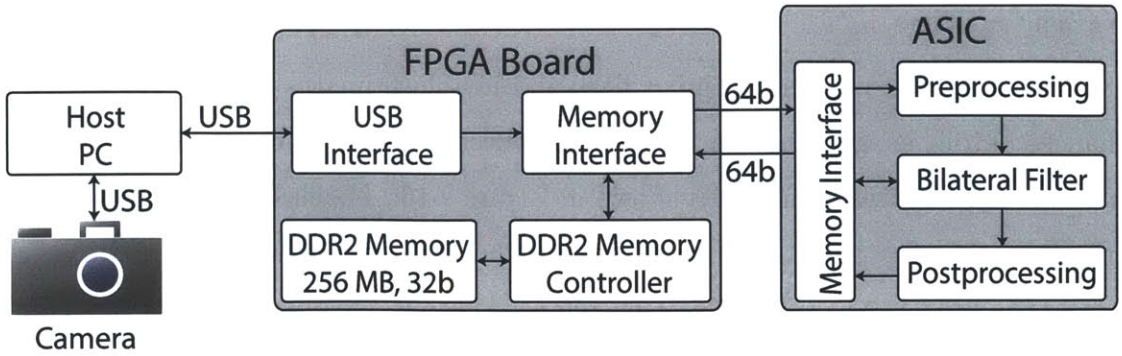


Figure 2-18: Block diagram of demo setup for the processor. (Courtesy R.Rithe)

The energy consumption and frequency of the test-chip is shown in Figure 2-20(b) for a range of V_{DD}. The processor is able to operate from 25 MHz at 0.5 V with 2.3 mW power consumption to 98 MHz at 0.9 V with 17.8 mW power consumption. The run-time scales linearly with the image size, as shown in Figure 2-20(a), with 13 megapixel/s throughput. Table 2.1 shows a comparison of the processor performance with other CPU/GPU implementations. The processor achieves significant energy reduction compared to other software implementations.

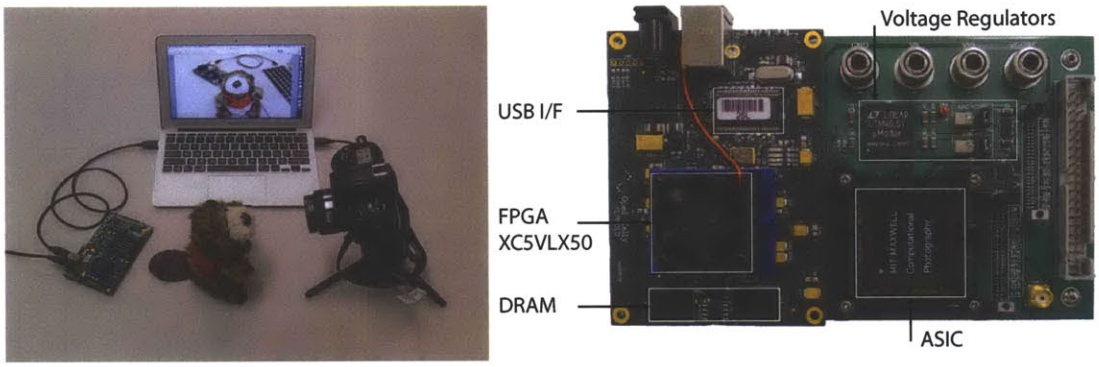


Figure 2-19: Demo board and setup integrated with camera and display. (Courtesy N.Ickes)

Processor	Runtime
NVIDIA G80	209 ms
NVIDIA NV40	674 ms
Intel Core i5 Dual Core (2.5 GHz)	12240 ms
This work (98 MHz)	771 ms

Table 2.1: Run-time comparison with CPU/GPU implementations.

The processor is integrated, as shown in Figure 2-18, with a camera and a display through a host PC using the USB interface. A software application, running on the host PC, is developed for processor configuration, image capture, processing and result display. The system provides a portable platform for live computational photography.

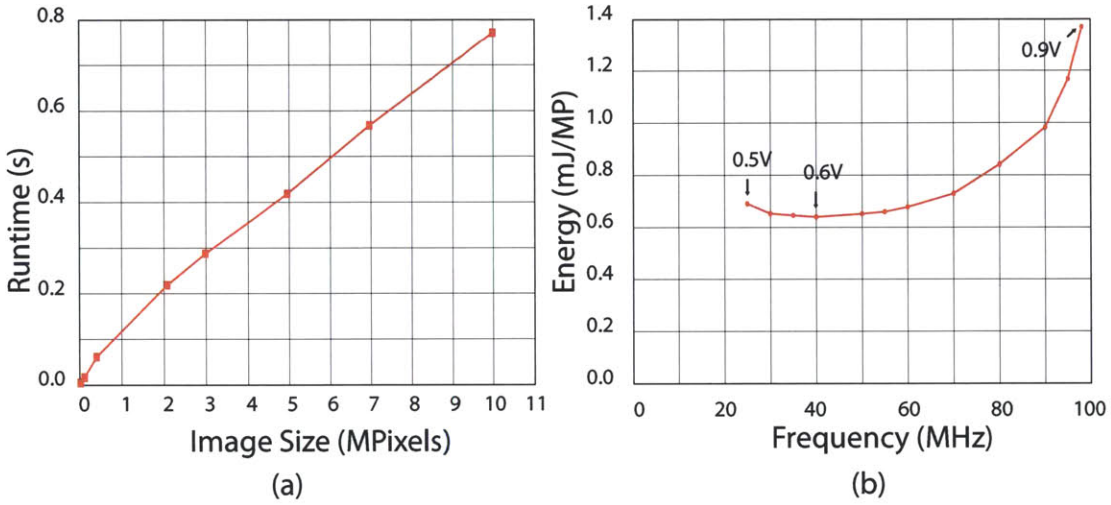


Figure 2-20: (a) Processing run-time for different image sizes. (b) Frequency of operation and energy consumption for varying V_{DD}. (Courtesy R.Rithe)

Chapter 3

Image Deblurring

When we use a camera to take a picture, we want the recorded image to be a faithful representation of the scene. However, more often than not, the recorded image is blurred and thus unusable. Blur can be caused due to a variety of reasons such as camera shake, object motion, defocus and lens defects and directly affects image quality. This work focuses on recovering a sharp image from a blurred one when blur is caused due to camera shake and there is no prior information about the camera motion during exposure.

3.1 MAP_k Blind Deconvolution

For blur caused due to camera shake, an observed blurred image y can be modeled as a convolution of an unknown sharp image x with an unknown blur kernel k (hence blind), corrupted by measurement noise n:

$$y = k \otimes x + n \tag{3.1}$$

This problem is severely ill-posed and there is an infinite set of pairs (x, k) that can explain an observed blurred image y. For example, an undesirable solution is the no-blur explanation where k is the delta kernel and $x = y$. So, in order to obtain the true sharp image, additional assumptions are required. A common approach is to

put the problem into a probabilistic framework and utilize prior knowledge about the statistics of natural images and the blur kernel to solve for the latent sharp image. To summarize, we have the following three sources of information:

1. The reconstruction constraint ($y = k \otimes x + n$). This is expressed as the likelihood of observing a blurred image y, given some estimates for the latent sharp image x and the blur kernel k and the noise variance η^2:

$$p(y|x,k) = \prod_i N(y(i)|k \otimes x(i), \eta^2) \tag{3.2}$$

2. A prior on the sharp image. A common natural prior is to assume that the image derivatives are sparse [12, 13, 14, 15]. The sparse prior can be expressed as a mixture of J Gaussians (MOG):

$$p(x) = \prod_{i,\gamma} \sum_{j=1}^{J} \pi_j N(f_{i,\gamma}(x)|0, \sigma_j^2) \tag{3.3}$$

3. A sparse or a uniform prior on the kernel $p(k)$ which enforces all kernel entries to be non-negative and to sum to one.

The common approach [14, 15, 16] is to search for the $MAP_{x,k}$ solution which maximizes the posterior probability of the estimates for the kernel k and the sharp image x given the observed blurred image y:

$$(\hat{x}, \hat{k}) = \arg\max p(x,k|y) = \arg\max p(y|x,k)p(x)p(k) \tag{3.4}$$

However, [17] shows that this approach does not provide the expected solution and favors the no-blur explanation. Instead, since the kernel size is much smaller than the image size, MAP estimation of the kernel alone marginalizing over all latent images gives a much more accurate estimate of the kernel:

$$\hat{k} = \arg\max p(k|y) = \arg\max p(y|k) = \arg\max \int p(x,y|k)dx \tag{3.5}$$

where we consider a uniform prior on k. However, calculating the above integral over latent images is hard. [5] proposes an algorithm which approximates the solution using an Expectation-Maximization (EM) framework, which is used as the baseline algorithm in this work.

3.2 EM Optimization

The EM algorithm takes as inputs a blurred image and a guess for the blur kernel. It alternates between two steps. In the E-step, it solves a non-blind deconvolution problem to estimate the mean sharp image and the covariance around it given the current estimate for the blur kernel. In the M-step, the kernel estimate is refined given the mean and covariance sharp image estimates from the E-step and the process is iterated.

Since convolution is a linear operator, the optimization can be done in the image derivative space rather than in the image space. In practice, the derivative space approach gives better results as shown in [5] and is therefore adopted in this work. In the following sections, we assume that k is an $m \times m$ kernel, and $M = m^2$ is the number of unknowns in k. y_γ and x_γ denote the blurred and sharp images derivatives where $\gamma = 0$ refers to the horizontal derivative and $\gamma = 1$ refers to the vertical derivative. x_γ is an $n \times n$ image and $N = n^2$ is the number of unknowns in x_γ.

3.2.1 E-step

For a sparse prior, the mean image and the covariance around it cannot be computed in closed form. The mean latent image μ_γ is estimated using iterative re-weighted least squares given the kernel estimate and the blurred image, where in each iteration an $N \times N$ linear system is solved to get μ_γ:

$$A_x \mu_\gamma = b_x \tag{3.6}$$

$$A_x = 1/\eta^2 T_k^T T_k + W_\gamma \tag{3.7}$$

$$b_x = 1/\eta^2 T_k^T y_\gamma \tag{3.8}$$

The solution to this linear system minimizes the convolution error plus a weighted regularization term on the image derivatives. The weights are selected to provide a quadratic upper bound on the MOG negative log likelihood based on previous μ_γ solution:

$$w_{\gamma,i,j_0} = \frac{\pi_{j_0}}{\sigma_{j_0}} e^{-\frac{E[\|x_{\gamma,i}\|^2]}{2\sigma_{j_0}^2}} / \sum_j \frac{\pi_j}{\sigma_j} e^{-\frac{E[\|x_{\gamma,i}\|^2]}{2\sigma_j^2}} \tag{3.9}$$

Here i indexes over image pixels and the expectation is computed using:

$$E[\| x_{\gamma,i} \|^2] = \mu_{\gamma,i}^2 + c_{\gamma,i} \tag{3.10}$$

W_γ is a diagonal matrix with:

$$W_\gamma(i,i) = \sum_j \frac{w_{\gamma,i,j}}{\sigma_j^2} \tag{3.11}$$

The $N \times N$ covariance matrix C_γ around the mean image is approximated with a diagonal matrix and is given by:

$$C_\gamma(i,i) = \frac{1}{A_x(i,i)} \tag{3.12}$$

Only the diagonal elements of this matrix are stored as an $n \times n$ covariance image c_γ. The E-step is run independently on the two derivative components, and for each component it is iterated three times before proceeding to the M-step.

3.2.2 M-step

In the M-step, given the mean image and covariance around it obtained from the E-step, a quadratic programing problem is solved to get the kernel:

$$\bar{A}_{k,\gamma}(i_1, i_2) = \sum_i \mu_\gamma(i + i_1)\mu_\gamma(i + i_2) + C_\gamma(i + i_1, i + i_2) \quad (3.13)$$

$$\bar{b}_{k,\gamma}(i_1) = \sum_i \mu_\gamma(i + i_1)y_\gamma(i) \quad (3.14)$$

$$\bar{A}_k = \sum_\gamma A_{k,\gamma} \quad (3.15)$$

$$\bar{b}_k = \sum_\gamma b_{k,\gamma} \quad (3.16)$$

$$\hat{k} = \arg\min \frac{1}{2}k^T\bar{A}_k k + \bar{b}_k^T k \text{ s.t. } k \geq 0 \quad (3.17)$$

Here i sums over image pixels and i_1 and i_2 are kernel indices. These are 2-D indices but the expression uses the 1-D vectorized version of the image and the kernel.

The algorithm assumes that the noise variance η^2 is known, and is taken as an input. To speed convergence of EM algorithm, the initial noise variance is assumed to be high and it is gradually reduced during optimization by dividing by a factor of 1.15 till the desired noise variance value of reached.

3.3 System Architecture

Figure 3-1 shows the top level flow for the deblurring algorithm. The input blurred image is preprocessed to remove gamma correction since deblurring is performed in the linear domain. If the blur kernel is expected to be large, the blurred image is down-sampled. This is followed by selecting a window of pixels from the blurred image from which the blur kernel is estimated using blind deconvolution. Using a window of pixels rather than the whole image reduces the size of the problem, and hence the time taken by the algorithm, while giving a fairly accurate representation of the kernel if the blur is spatially uniform. A maximum window size of 128×128 pixels is used, which is found to work well in practice. A smaller window size results

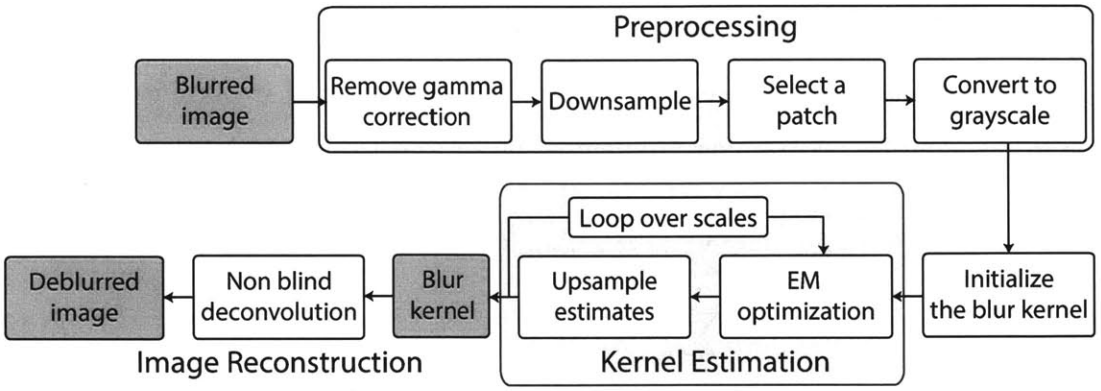


Figure 3-1: Top level flow for deblurring algorithm.

in an inaccurate representation of the kernel which produces ringing artifacts after final deconvolution.

To estimate the kernel, a multi-resolution approach is adopted. The blurred image is down-sampled and passed through the EM optimizer to obtain a coarse estimate of the kernel. The coarse kernel estimate is then up-sampled and used as the initial guess for the finer resolutions. In the current implementation, the coarser resolutions are created by down-sampling by a factor of 2 at each scale. The final full resolution kernel is up-sampled (if the blurred image was down-sampled initially) and non blind deconvolution is performed on the full image for each color channel to get the deblurred image.

Figure 3-2 shows a block diagram of the system architecture. It consists of independent modules which execute different parts of deblurring algorithm, controlled by a centralized scheduling engine.

3.3.1 Memory

The processor has 4 SRAMs each having 4 banks that can be accessed in parallel, which are used as scratch memory by the modules. The access to the SRAMs is arbitrated through centralized SRAM arbiters, one for each bank. Each bank is 32 bits wide and contains 4096 entries. The processor is connected to an external DRAM through a DRAM controller. The access to the DRAM is arbitrated through a DRAM

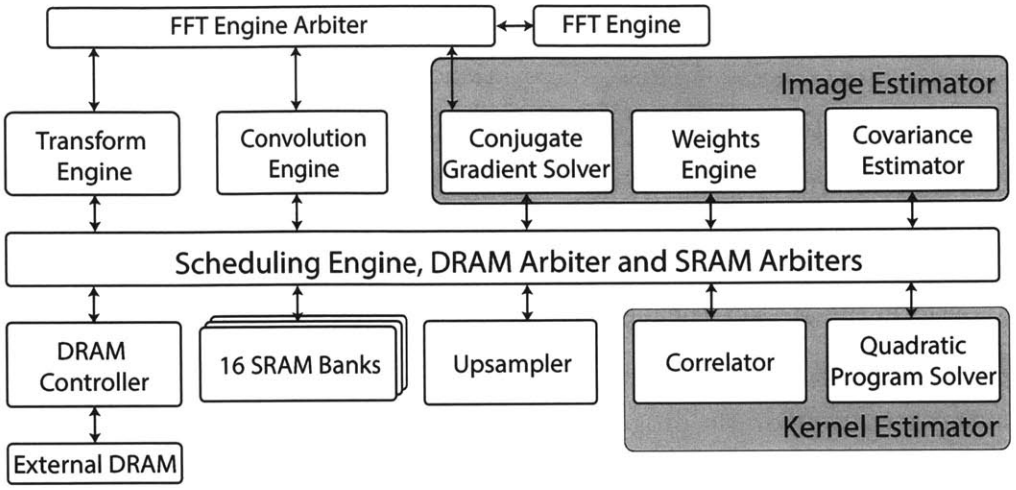


Figure 3-2: System block diagram for image deblurring processor.

arbiter. All data communication between the modules happens through the scratch memory and the DRAM.

3.3.2 Scheduling Engine

The scheduling engine schedules the modules to execute different parts of the deblurring algorithm based on the data dependencies between them. At the start of each resolution, the blurred image is read in from the DRAM and down-sampled to the scale of operation. The scheduler starts the transform engine, which computes the horizontal and vertical gradients of the blurred image and their 2-D discrete Fourier transform and writes them back to the DRAM. At any given resolution, for each EM iteration, the convolution engine is enabled, which computes the kernel transform and uses it to convolve the gradient images with the kernel. The result of the convolution engine is used downstream in the E-step of the EM iteration.

To perform the E-step, a conjugate gradient solver is used to solve the system given by Equation 3.6, given the current kernel estimate, noise variance and weights. The solution to this system is the mean gradient image μ_γ. The covariance estimator then estimates the covariance image c_γ given the current kernel and the weights. Covariance and mean image estimation can be potentially done in parallel as they

do not have data dependencies but limited memory bandwidth and limited on-chip memory size do not allow it. Instead, covariance estimation is done in parallel with the weight computation for the next iteration. The E-step is performed for both horizontal and vertical components of the gradient, and for each component it is iterated three times before performing the M-step.

To perform the M-step, the mean image and the covariance image generated by the E-step are used by the correlator to generate the coefficients matrix A_k and vector b_k for the kernel quadratic program. A gradient projection solver is then used to solve the quadratic program subject to the constraint that all kernel entries are non-negative. The solution is a refined estimate of the blur kernel which is fed back into the E-step for the next iteration of the EM algorithm. The number of EM iterations is configurable and can be set before the processing starts. Once EM iterations for the current resolution complete, the kernel estimate at the end is up-sampled and used as the initial guess for the next finer resolution.

3.3.3 Configurability

The architecture allows several parameters to be configured at runtime by the user. The kernel size can be varied from 7×7 pixels to 31×31 pixels. For larger kernel sizes, the image can be down-sampled for kernel estimation and the resulting kernel estimates can be scaled up to get the full resolution blur estimate. The number of EM iterations and the number of iterations and convergence tolerance for conjugate gradient and gradient projection solvers can be configured to achieve energy scalability at runtime.

Setting parameters aggressively results in a very accurate kernel estimate, which takes longer to compute and consumes more energy. In an energy constrained scenario, less aggressive parameter settings result in a reasonably accurate kernel while taking less time to compute and consuming lower energy.

3.3.4 Precision Requirements

The algorithm requires all the arithmetic to be done with high precision, to get an accurate estimate of the kernel. An inaccurate kernel estimate results in undesirable ringing artifacts in the deconvolved image which makes the image unusable. A 32 bit fixed point implementation of the algorithm was developed in software but the resulting kernel was far from accurate due to very large dynamic range of the intermediates.

For example, to set up the kernel quadratic program, the coefficients matrix is computed using an auto-correlation of the estimated sharp image. For an image of size 128×128 with b bit pixels, the auto-correlation result requires $2 * b + 14$ bits to be represented accurately. This coefficient matrix is then multiplied with b bit vectors of size m^2 where $m \times m$ is the size of the kernel while solving the kernel quadratic program, resulting is an output which requires $3 * b + 14 + 10$ bits for accurate representation. If b is 32 this intermediate needs 120 bits.

Also, since the algorithm is iterative, the magnitude of the errors keeps growing with successive iterations. Moreover, a static scaling schedule is not feasible because the dynamic range of the intermediates is highly dependent on the input data. Therefore, the complete datapath is implemented for 32 bit single precision floating point numbers and all arithmetic units (including the FFT butterfly engines) are implemented using Synopsys Designware floating point modules to handle the required dynamic range.

The following sections detail the architecture of the component modules.

3.4 Fast Fourier Transform

Discrete Fourier Transform (DFT) computation is required in the transform engine, the convolution engine and the conjugate gradient solver, and it is implemented using a shared FFT engine. The FFT engine computes the DFT using Cooley-Tukey FFT algorithm. It supports run-time configurable point sizes of 128, 64 and 32. For an N-point FFT, the input and output data are vectors of N complex samples represented as

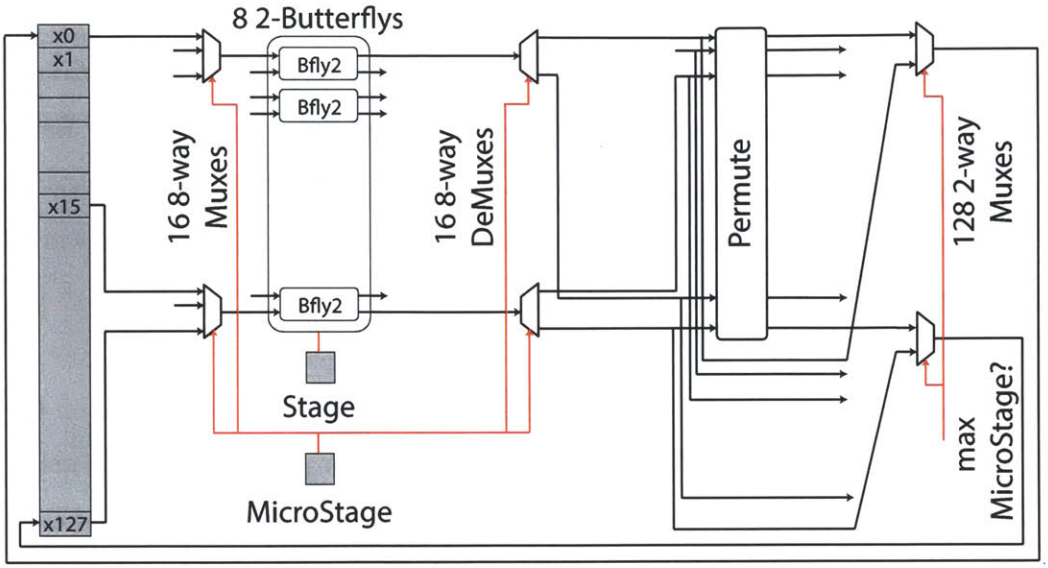


Figure 3-3: Architecture of the FFT engine.

dual 32-bit floating-point numbers in natural order. Figure 3-3 shows the architecture of the FFT engine. It has the following key components:

3.4.1 Register Banks

The FFT engine is fully pipelined and provides streaming I/O for continuous data processing. This is enabled by 2 register banks each of which can store up to 128 single-precision complex samples. The interface to the FFT engine consists of 2 sample wide input and output FIFOs. To enable continuous data processing, the engine simultaneously performs transform calculations on the current frame of data (stored in one of the two register banks), and loads the input data for the next frame of data and unloads the results of the previous frame of data (using the other register bank). At the end of each frame, the register bank being used for processing and the register bank being used for I/O are toggled. The client module can continuously stream in data into the input FIFO at a rate of two samples per cycle and after the calculation latency (of N cycles for an N-point FFT) unload the results from the output FIFO. Since the higher level modules accessing the FFT engine use it to

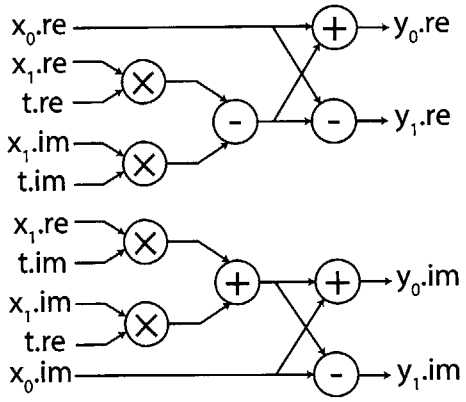


Figure 3-4: Architecture of the radix-2 butterfly module.

perform 2-D transforms of $N \times N$ arrays, the latency of the FFT engine to compute a single N-point transform is amortized over N transforms.

3.4.2 Radix-2 Butterfly

The core of the engine has 8 radix-2 butterfly modules operating in parallel. This number (of butterfly modules) has been selected to minimize the amount of hardware required to support a throughput of 2 samples per cycle. This is the maximum achievable throughput given the system memory bandwidth of 64-bits read and write per cycle. Figure 3-4 shows the block diagram of a single butterfly module. The floating point arithmetic units in the butterfly module have been implemented using Synopsys Designware floating point library components. Each butterfly module is divided into 2 pipeline stages (not shown in Figure 3-4) to meet timing specifications.

3.4.3 Schedule

An N-point FFT is computed in $log_2 N$ stages, and each stage is further sub-divided into $N/16$ micro-stages. Each micro-stage takes 2 cycles and consists of operating 8 butterfly modules in parallel on 16 consecutive input/intermediate samples, however, since the butterfly modules are pipelined, the data for the next micro-stage can be fed into the butterfly modules when they are operating on the data from the current micro-stage. So, an N-point FFT takes $log_2 N * (N/16 + 1)$ cycles (i.e.

63, 30 and 15 cycles for 128, 64 and 32 point FFTs). The twiddle factors fed into the butterfly modules are stored in a look-up table implemented as combinational logic. A controller FSM generates the control signals to MUX/DeMUX the correct input/intermediate samples and twiddle factors to the butterfly modules and the register banks depending upon the stage and micro-stage registers. A permute engine permutes the intermediates at the end of each stage before they get written to the register bank.

3.4.4 Inverse FFT

The FFT engine can also be used to take inverse transform since inverse transform can be expressed simply in terms of forward transform. To take the inverse transform, the FFT client must flip the real and imaginary parts of the input vector and the output vector, and scale the output by a factor of $1/N$.

3.5 2-D Transform Engine

At the start of every resolution, the 2-D transform engine reads in the (down-sampled) blurred image y from the DRAM and computes the horizontal (y_0) and vertical (y_1) gradient images and their 2-D DFTs which are used downstream by the convolution engine. 2-D DFT of the gradient image is computed by performing 1-D FFT along all rows of the gradient image to get an intermediate row transformed matrix, followed by performing 1-D FFT along all columns of the intermediate matrix to get the 2-D DFT of the gradient image. Both row and column transform is performed using a shared 1-D FFT engine as shown in Figure 3-5.

3.5.1 Transpose Memory

Two 64 kB transpose memories are required for the largest transform size of 128×128 to store the real and imaginary parts of the row transform intermediate. This is prohibitively large to store in registers. Therefore, two shared SRAMs, each having 4

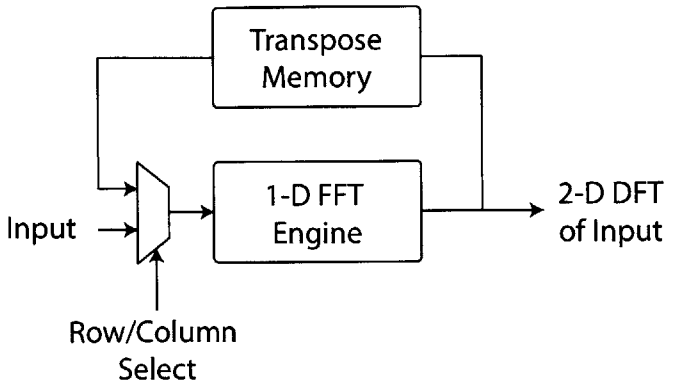


Figure 3-5: Architecture of the 2-D transform engine.

single-port banks of 4096 32-bit wide entries are used for this purpose. The pixels are mapped to locations in the 4 SRAM banks as shown in Figure 3-6(a). By ensuring that 2 adjacent pixels in any row or column sit in different SRAM banks, it is possible to write along rows and read along columns by supplying different addresses to the 4 banks. It is possible to achieve the transpose using only two SRAM banks having two times the number of entries for a throughput of 2 pixels per cycle using the mapping shown in Figure 3-6(b). However, this approach is not adopted because 4 bank SRAMs are required in downstream modules for processing consecutive rows or columns concurrently.

This scheme of mapping a matrix to SRAM banks is used in downstream modules as well. Using this mapping, the elements of a matrix having even row index and even column index, even row index and odd column index, odd row index and even column index and odd row index and odd column index are mapped to different SRAM banks. This allows access of two consecutive elements in along a row or a column in parallel. This also allows concurrent access of consecutive rows or columns from the SRAM. For ease of representation, the following notation is used to describe which SRAM banks store which elements of a matrix: [Bank# for even-row even-column elements, Bank# for even-row odd-column elements, Bank# for odd-row even-column elements, Bank# for odd-row odd-column elements].

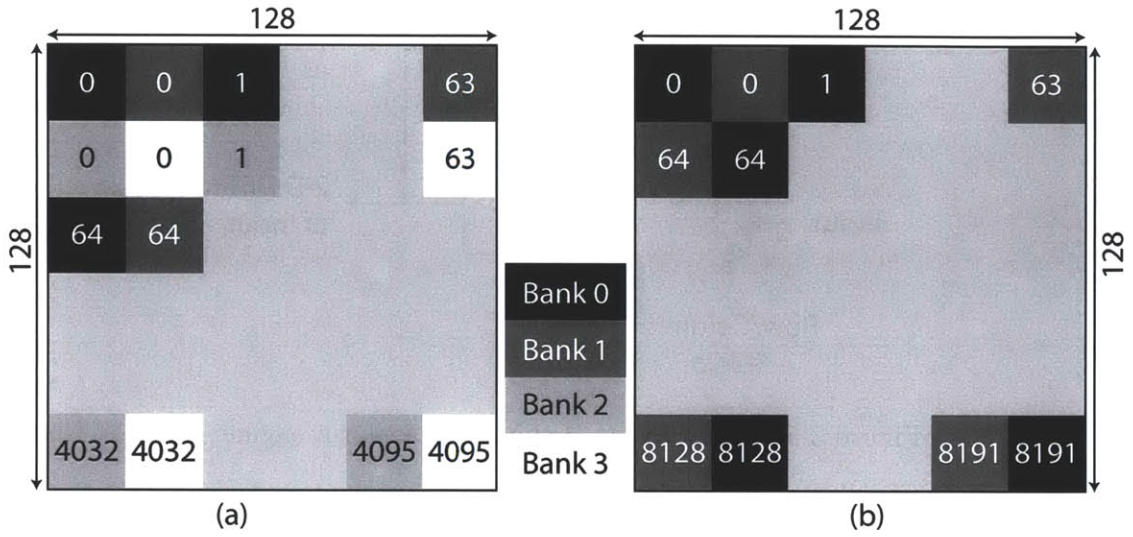


Figure 3-6: Mapping 128 × 128 matrix to (a) 4 SRAM banks or (b) 2 SRAM banks for transpose operation. Color of the pixel indicates the SRAM bank and the number denotes the bank address.

3.5.2 Schedule

The 2-D transform engine performs the following steps sequentially. Figure 3-7 shows the sequencing of steps, the resources used in each step and the state of the shared SRAMs before and after each step. The 2-D transform engine has has 4 floating point subtractors for computing 2 horizontal and vertical gradient pixels in parallel every cycle. It uses the shared FFT engine for computing the 2-D DFT of the gradient images. The sequencing is a result of either data dependencies between successive steps or limited DRAM bandwidth.

1. Two pixels from the down-sampled image (y) are read from the DRAM every cycle in row-major order. Two horizontal gradient (y_0) pixels are computed and are fed into the FFT engine for row transform (RT) and written to the DRAM. The results of row transform appear after a number of cycles equal to the latency of the FFT engine. These are written into SRAMs 2 (real part) and 3 (imaginary part) in banks [0 1 2 3]. The latency penalty applies only in the beginning since the FFT engine is fully pipelined.

   In parallel, the current image pixels (y) are written to SRAM 1 banks [0 1 2 3]

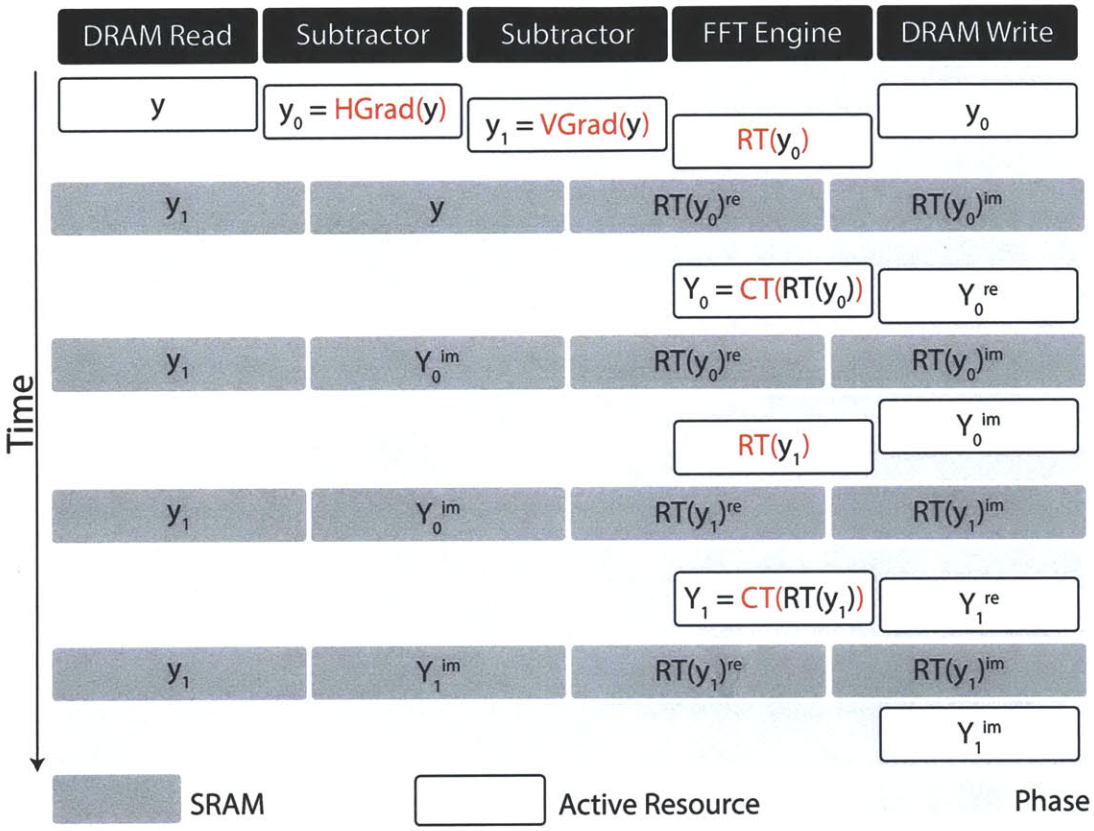


Figure 3-7: Schedule for the 2-D transform engine shows the blocks executing in parallel during each step/phase and state of the 4 SRAMs before the after each phase.

and previous row of image pixels is read from SRAM 1 banks [2 3 0 1]. These are used to compute two vertical gradient (y_1) pixels which are written into SRAM 0 banks [0 1 2 3]. The vertical gradient computation can start only after an entire row of image pixels has been read in.

2. Once row transform of y_0 completes, the column transform (CT) of the row transform intermediate is computed (Y_0) and the real part of the result is written to DRAM and imaginary part to SRAM 1 banks [0 1 2 3].

3. Next, the row transform (RT) of y_1 is computed and the real part of the result is written into SRAM 2 and the imaginary part of the results is written into SRAM 3 banks [0 1 2 3]. In parallel, the imaginary part of column transform of y_0, Y_0^{im}, is written to DRAM.

4. The column transform of row transform intermediate of y_1 is computed (Y_1) and the real part is written to DRAM and the imaginary part to SRAM 1 banks [0 1 2 3].

5. The imaginary part of column transform of y_1, Y_1^{im}, is written from SRAM 1 to DRAM.

The processing is limited by the memory bandwidth - write bandwidth in this case, and using two FFT engines to process the horizontal and vertical gradients in parallel does not help reduce the overall processing time, as the module would still have to wait for all the results to be written to the DRAM.

3.6 Convolution Engine

The convolution engine runs once at the start of every EM iteration and first computes the 2-D DFT of the kernel, and then uses it to convolve the blurred gradient images y_γ with the kernel. The 2-D DFTs of the gradient images, which are computed by the transform engine, are read in from the DRAM. The convolution engine uses the shared FFT engine for DFT computation. It consists of a set of FSMs which execute the following steps, shown in Figure 3-8 and Figure 3-9, sequentially. It should be noted that the time axis on the figures is not uniformly spaced.

1. The kernel (k) is read form the DRAM and written to SRAM 1 banks [0 1 2 3].

2. Once the kernel (k) is loaded, it is read row major from SRAM 1 banks [0 1 2 3] and fed into the FFT engine for computing row transform (RT). The transform results from the FFT engine are written back to SRAMs 0 and 1 banks [2 3 0 1]. In parallel, if it is the first iteration for the current resolution, y_1 is read from SRAM 0 banks [0 1 2 3] and written to the DRAM. Also in parallel, 2-D DFT of y_0, Y_0, is read from DRAM and its real part is written into SRAM 2 banks [0 1 2 3] and the imaginary part to SRAM 3 banks [0 1 2 3].

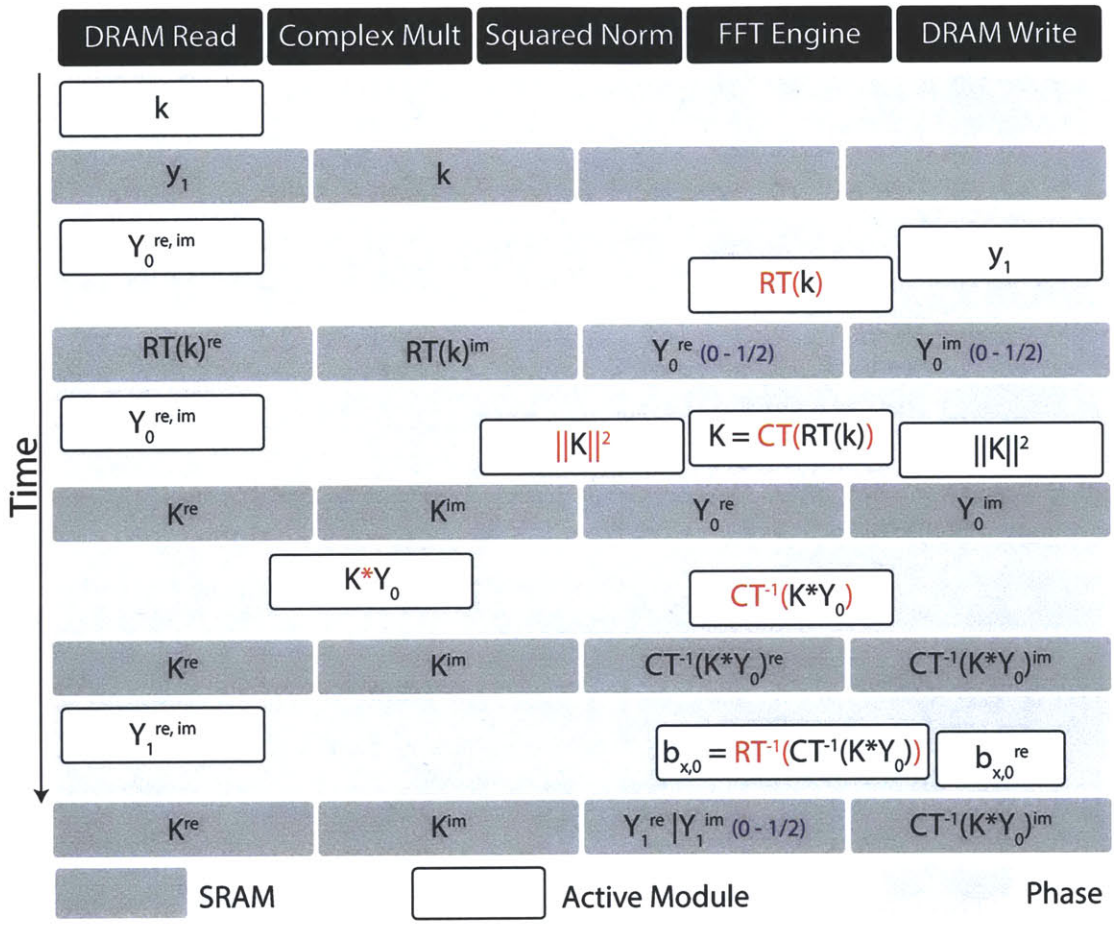


Figure 3-8: Schedule for convolution engine shows the blocks executing in parallel during each phase and state of the 4 SRAMs before the after each phase. The numbers in blue indicate the fraction of the matrix in the SRAM.

3. After the row transform completes, the row transform intermediates for k are read column major from SRAMs 0 and 1 banks [2 3 0 1] and fed into the FFT engine for column transform. The column transform (K) is written back into SRAMs 0 and 1 banks [3 2 1 0] and squared magnitude of the kernel DFT, $||K||^2$, is computed and written to the DRAM. If there are no stalls in reading DRAM data, 2-D DFT of y_0, Y_0, will be loaded completely into SRAMs 2 and 3 by the time the column transform of the kernel completes.

4. To perform convolution, the kernel transform (K) is read from SRAMs 0 and 1 banks [3 2 1 0] and y_0 transform (Y_0) is read from SRAMs 2 and 3 banks [0 1 2

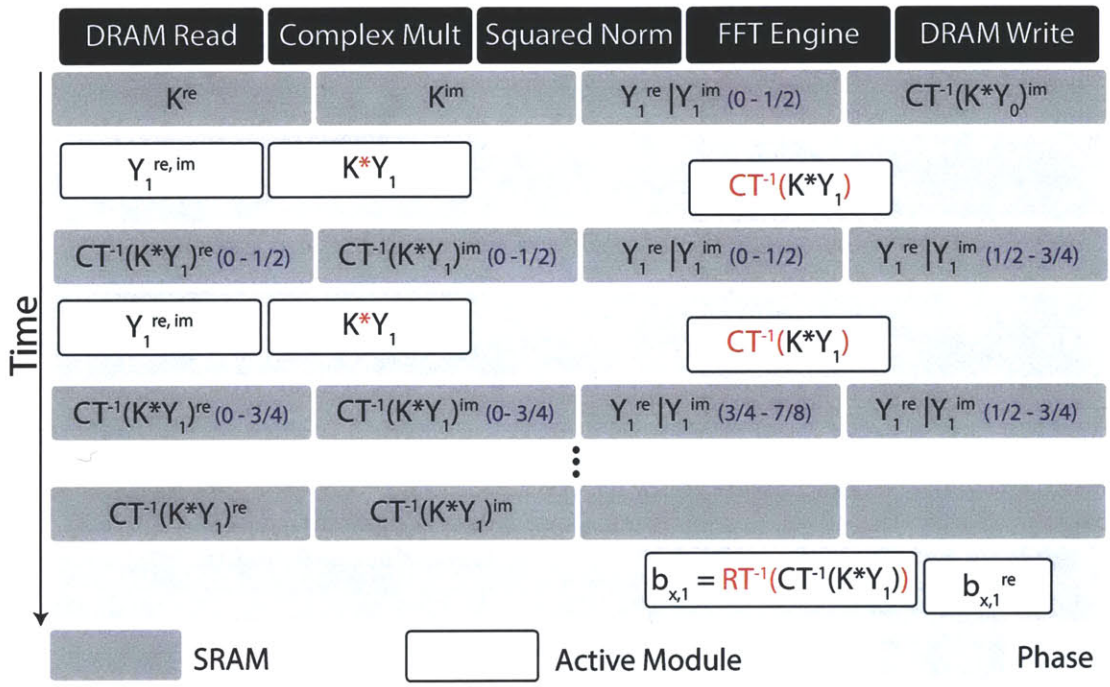


Figure 3-9: Schedule for convolution engine shows the blocks executing in parallel during each phase and state of the 4 SRAMs before the after each phase. The numbers in blue indicate the fraction of the matrix in the SRAM.

3]. These are multiplied using complex multipliers and fed into FFT engine to compute column inverse transform (CT^{-1}) and the results are stored back into SRAMs 2 and 3 banks [1 0 3 2].

5. Column inverse transform intermediate is read in row major order from SRAMs [1 0 3 2] and fed in to the FFT engine to compute row inverse transform (RT^{-1}). The real part of the result is the convolved matrix $b^{re}_{x,0}$ which is written to the DRAM. The imaginary part is discarded. In parallel, 2-D DFT of y_1, Y_1, is read from the DRAM and written to SRAM 2. The real and imaginary parts of Y_1 are stored interleaved at a column level (a column of real part followed by a column of imaginary part) in the DRAM. The real part is written to banks 0 and 2 and the imaginary part is written to banks 1 and 3, to allow parallel access. However, for an $n \times n$ image, it takes n^2 cycles to read Y_1 from the memory, whereas it takes $n^2/2$ cycles to compute the row inverse transform. So, by the time the FFT engine finishes computing the row inverse transform

and only half of the columns of Y_1 are loaded.

6. This partially loaded Y_1 matrix is read in column major order from SRAM 2 (the real part is read from banks 0 and 2 and the imaginary part is read from banks 1 and 3) and multiplied with the kernel transform K read from SRAMs 0 and 1 banks [3 2 1 0]. The multiplication result is fed in to the FFT engine for column inverse transform. The results of the FFT engine are written to SRAMs 0 and 1 banks [2 3 0 1]. In parallel, by the time 1/2 of Y_1 matrix is processed, the next 1/4 columns are loaded into SRAM 3 (the real part is written to banks 0 and 2 and the imaginary part is written to banks 1 and 3) since SRAM 2 is being accessed for inverse transform computation.

7. The roles of SRAMs 2 and 3 are then flipped and SRAM 3 is used for FFT processing while 2 is loaded with the next 1/8 columns and so on. This allows computation of column inverse transform without stalling for the matrix Y_1 to be read completely from the memory.

8. Once column inverse transform is complete, it is read in row major order from SRAMs 0 and 1 banks [2 3 0 1] and fed into the FFT engine to compute the row inverse transform. The real part of the result $b^{re}_{x,1}$ is written to the DRAM and the imaginary part is discarded.

3.7 Conjugate Gradient Solver

The first part of E-step is to solve for the mean sharp image given the kernel from the previous EM iteration. The sharp image can be obtained by solving the following linear system for μ. For simplifying the representation, the γ subscript is dropped in the rest of the section.

$$A_x \mu = b_x \tag{3.18}$$

$$A_x = T_k^T T_k + \eta^2 W \tag{3.19}$$

$$b_x = T_k^T y \tag{3.20}$$

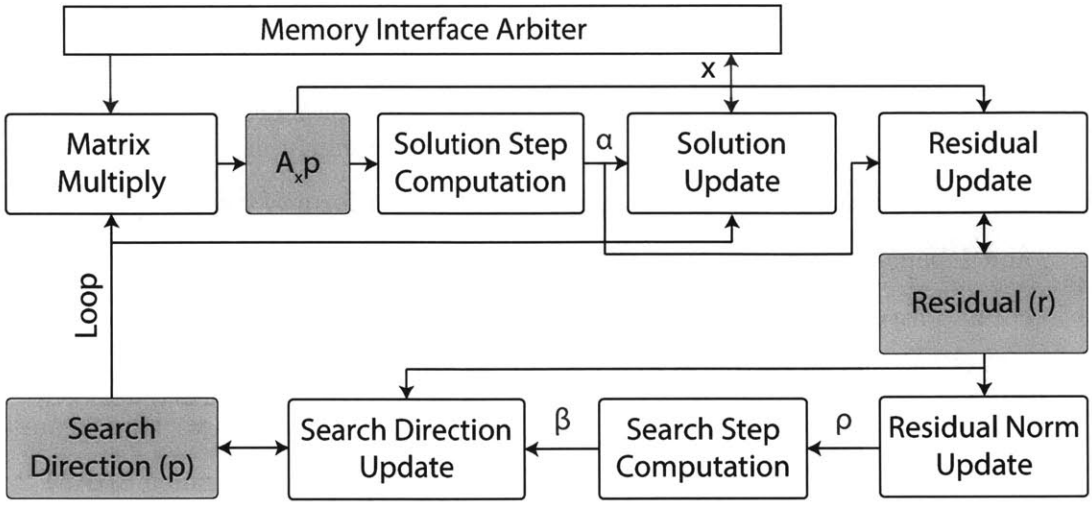


Figure 3-10: Flow diagram for conjugate gradient solver.

3.7.1 Algorithm

The linear system of equations in μ is solved numerically using a conjugate gradient (CG) solver. Starting with an initial guess for the solution, the CG solver iteratively improves the solution by taking steps in search directions that are conjugate to all previous search directions, thus ensuring fast convergence. The negative gradient direction at the initial guess is used as the starting search direction. Figure 3-10 shows the flow diagram for the CG solver. The major steps in the algorithm are outlined below.

- **Initialize:** Initialize μ to blurred image y. Initialize the residual r and the initial search direction p to the negative of the gradient at the initial guess $(b_x - A_x\mu)$. Compute the squared magnitude of the residual $\rho = r^T r$.

- **Iterate till convergence:**

  1. Multiply matrix A_x with the search direction p to get $A_x p$.
  2. Compute the step size for solution update $\alpha = \rho / p^T A_x p$.
  3. Update the solution $\mu = \mu + \alpha p$, the residual $r = r - \alpha A_x p$, and the squared magnitude of the residual $\rho = r^T r$.

4. Compute the step size for search direction update $\beta = \rho/\rho_{last}$

5. Update the search direction $p = r + \beta p$ and loop.

3.7.2 Optimizations

It can be seen that, in this algorithm, multiplication with the matrix A_x has the highest computational complexity, and involves multiplication with an $N \times N$ matrix for a flattened $n \times n$ image μ, where $N = n^2$. However, the structure of the matrix A_x makes it amenable to the following optimizations:

- It can be observed that the first component involves multiplication with the kernel Toeplitz matrix T_k which is equivalent to a convolution with the kernel. Convolution can be implemented efficiently in the frequency domain using FFT which makes the matrix multiply operation $O(NlogN)$ instead of $O(N^2)$.

- The second component involves multiplication with W which is a diagonal matrix. So, multiplication with $N \times N$ W is the same as element by element multiplication with a $n \times n$ weights matrix w comprising the diagonal elements of W.

Therefore, to apply matrix A_x on a flattened $n \times n$ image x, the following steps are involved:

1. First, the 2-D DFT of x is computed by performing row transform followed by column transform using the shared FFT engine.

2. The squared magnitude of 2-D DFT of kernel $||K||^2$ (computed by the convolution engine) is read from the memory, and multiplied with 2-D DFT of x. 2-D inverse DFT of the result is computed by performing column inverse transform followed by row inverse transform using the FFT engine.

3. The weights matrix is read in from the memory and multiplied element by element with x and the result is added to the inverse row transform obtained in 2 to get the matrix multiplication result.

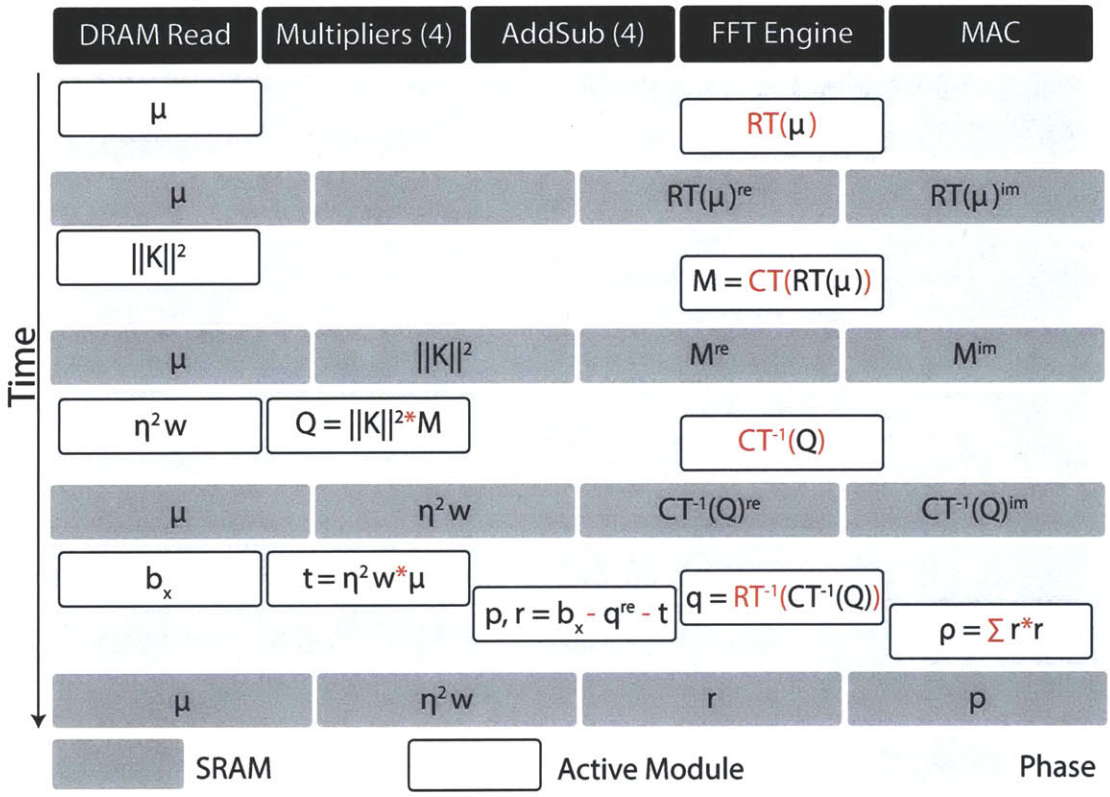


Figure 3-11: Schedule for initialization phase of CG solver.

3.7.3 Architecture

We now describe the architecture of the CG solver in detail. The CG solver consists of an FSM for each step of the conjugate gradient algorithm outlined earlier. The FSMs execute sequentially and the FSM/step boundaries are determined by the data dependencies in the algorithm, for example, the solution and residual update in step 3 cannot happen before the step length is computed in step 2. Each step has a throughput of 2 pixels/cycle.

The CG solver uses shared SRAMs as scratch memory to store intermediate variables. This minimizes memory accesses and enables maximum parallelism. The intermediate matrices are mapped to SRAM banks using the mapping shown in Figure 3-6. The solver runs in two phases: initialization and iteration.

Initialization

The initialization phase initializes $\mu = y$ and $r = p = b_x - A_x\mu$ in SRAMs as shown in Figure 3-11. The matrix multiplication is performed using the method described previously.

1. μ is initialized by reading y from the DRAM and writing it in to SRAM 0 banks [0 1 2 3]. In parallel, it is also fed into the FFT engine for computing row transform. After a delay equal to the latency of the FFT engine, the row transform intermediate results are written in parallel to SRAMs. The real part is stored in SRAM 2 banks [0 1 2 3] and the imaginary part is stored in SRAM 3 banks [0 1 2 3].

2. Squared magnitude of 2-D DFT of the kernel is read from DRAM and written into SRAM 1 banks [1 0 3 2]. In parallel, row transform intermediate computed in 1 is read column major from SRAM 2 banks [0 1 2 3] and SRAM 3 banks [0 1 2 3] and fed into the FFT engine for column transform. As FFT results start coming out, they are written to the same SRAMs but with a bank ordering of [1 0 3 2] to maximize concurrency. So, when a column from banks 0 and 2 is being read and fed into the FFT engine, the column transform coming out of the FFT engine is being written to banks 1 and 3 and vice versa.

3. Once the column transform is complete, it is read column major from SRAMs 2 and 3 banks [1 0 2 3] and multiplied with $||K||^2$ read from SRAM 1 banks [1 0 2 3]. The result (Q) is fed into the FFT engine for inverse column transform. The column inverse transform intermediates are written back into SRAMs 2 and 3 banks [0 1 2 3]. In parallel, if it is not the first iteration, the weights matrix $\eta^2 w$ is read from the DRAM and written to SRAM 1 banks [0 1 2 3].

4. The column inverse transform intermediate of Q is read from SRAMs 2 and 3 banks [0 1 2 3] in row major order and fed into FFT engine for row inverse transform. The real part of the result coming out of the FFT engine $q = T_k^T T_k \mu$. Weights $\eta^2 w$ and initial μ are read from SRAMs 1 and 0 and b_x is read from

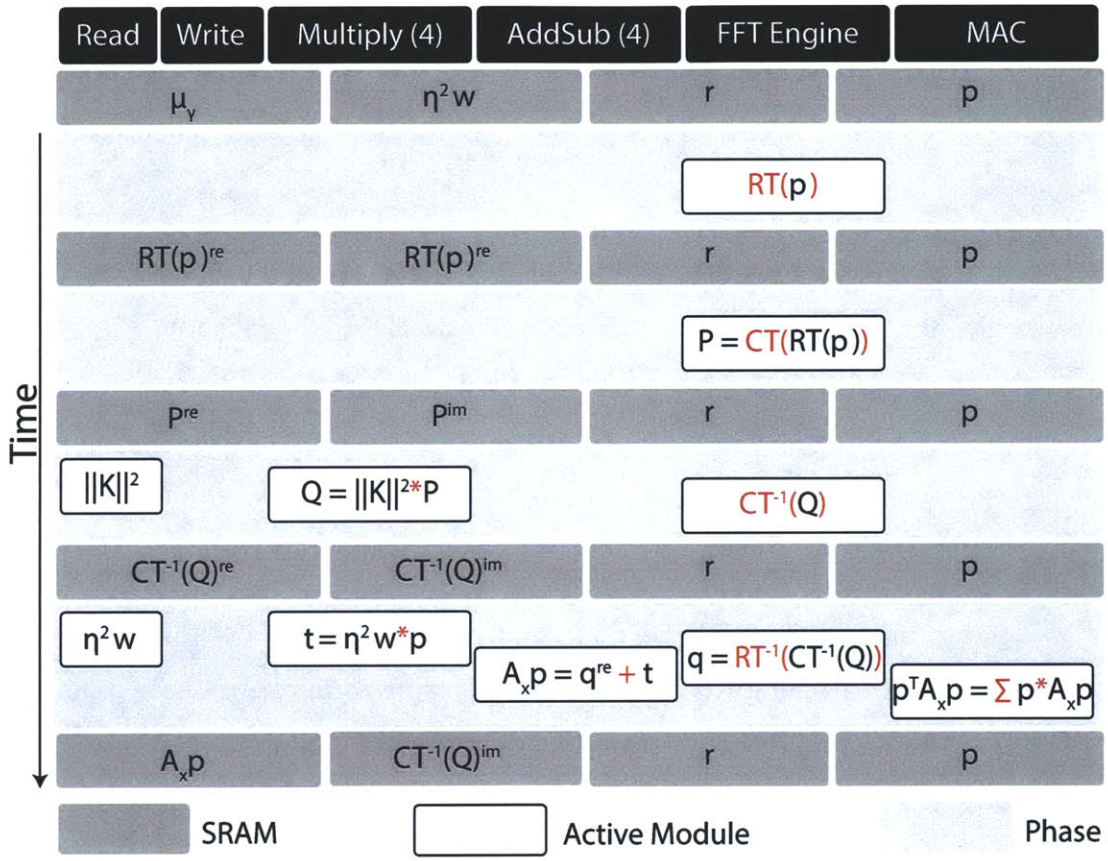


Figure 3-12: Schedule for iteration phase of CG solver.

the DRAM. The residual (r) and the search direction (p) are initialized with $b_x - q - \eta^2 w\mu$ in SRAMs 2 and 3 banks [2 3 0 1]. This computation is pipelined, such that only one floating point operation happens in a cycle. Also, as residual gets computed it is squared and accumulated into ρ.

Iteration

The iteration phase, shown in Figure 3-12 and Figure 3-13 starts with a search direction and computes a step length along it. It then updates the solution and the residual and computes the search step length from the updated residual. This is then used to update the search direction, and the process is iterated over.

1. The search direction (p) is read from SRAM 3 banks [2 3 0 1] in row major order

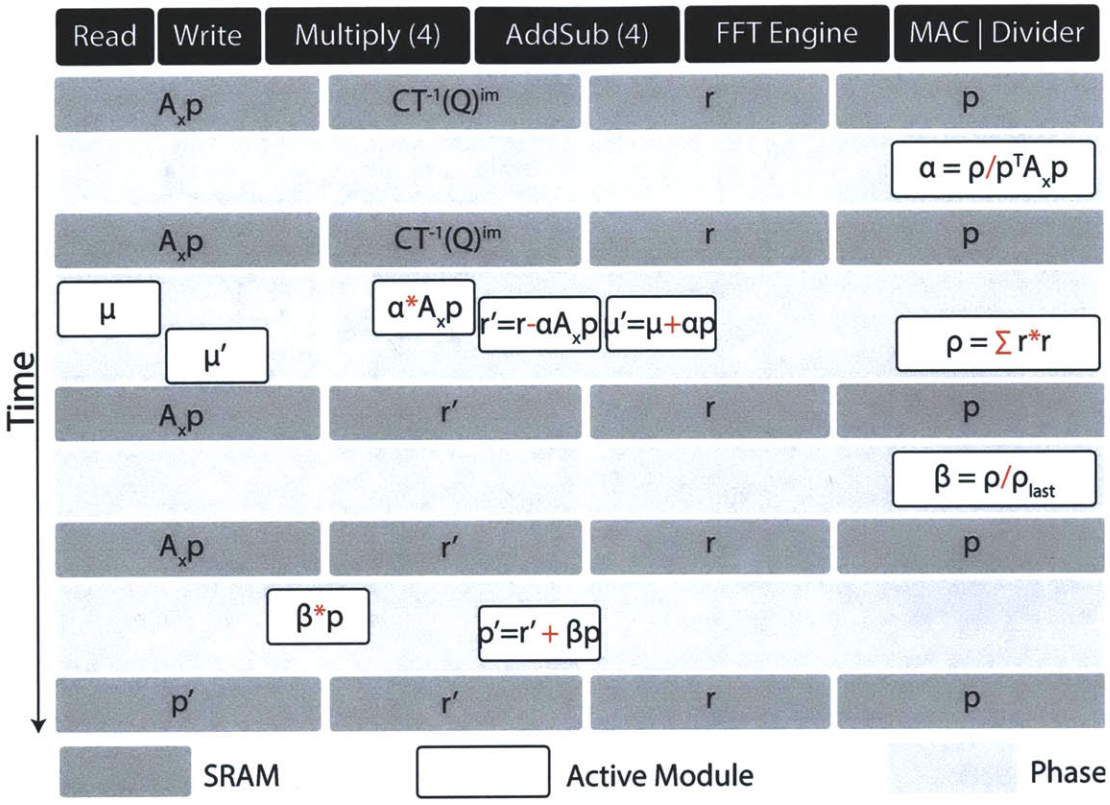


Figure 3-13: Schedule for the iteration phase of CG solver (continued).

and fed into FFT engine for row transform. The row transform intermediates are written to SRAMs 0 and 1 banks [0 1 2 3].

2. The row transform intermediates are read from SRAMs 0 and 1 banks [0 1 2 3] in column major order and fed into FFT engine for column transform. The column transform results (P) are written back into SRAMs 0 and 1 banks [1 0 3 2].

3. The column transform of p, P, is read from SRAMs 0 and 1 banks [1 0 3 2] and multiplied with $||K||^2$ which is read form the DRAM. The result is fed into the FFT engine in column major order for column inverse transform which is written back into SRAMs 0 and 1 banks [0 1 2 3].

4. The column inverse transform is read in row major order form SRAMs 0 and 1 banks [0 1 2 3] and fed into FFT engine for row inverse transform. In parallel,

weights $\eta^2 w$ are read from the DRAM and multiplied with p read from SRAM 3 banks [2 3 0 1]. The result is added to the real part of the results from the FFT engine q^{re} to give $A_x p$ which is written to SRAM 0 banks [2 3 0 1]. $A_x p$ elements are also multiplied with p and accumulated into $p^T A_x p$.

5. Step length α is computed using ρ and $p^T A_x p$ computed in the last step ($\alpha = \rho / p^T A_x p$).

6. Once the step length is computed, μ is read from the DRAM and $A_x p$, r and p are read from SRAMs. Updates to the solution ($\mu = \mu + \alpha p$) and the residual ($r = r - \alpha A_x p$) are computed in parallel. Updated μ is written back to the DRAM. Updated r is written to SRAM 1 banks [2 3 0 1] and its elements are squared and accumulated as well to get ρ.

7. Updated ρ and its value from the previous iteration are used to compute the factor β used in search direction update.

8. Search direction p is read from SRAM 3 banks [2 3 0 1] and updated residual r is read from SRAM 1 banks [2 3 0 1] and updated search direction is computed using $p = r + \beta p$ and written to SRAM 0 banks [2 3 0 1].

9. The FSM loops after the search direction is updated. However, during the iteration, the SRAMs in which the updated search direction and residuals are written are different from the ones from which the original values are read (because SRAMs are single port). In order to avoid copying these intermediates back to the original SRAMs, in the odd numbered iterations, p is accessed from SRAM 0, its transform is written to and read from SRAMs 2 and 3, $A_x p$ is written to SRAM 3, updated r is written to SRAM 2 and updated p is written to SRAM 3, returning the intermediates back to their original locations.

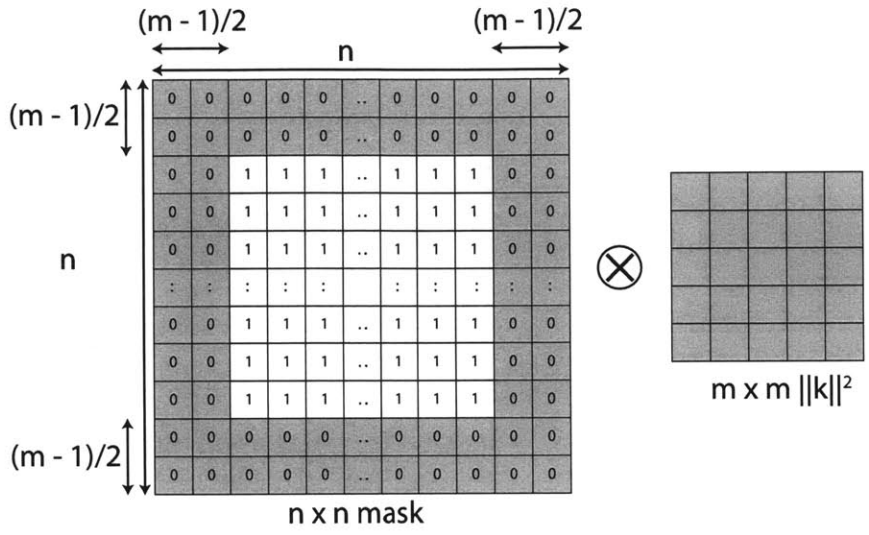


Figure 3-14: Computation of da_1 matrix is expressed as convolution of $n \times n$ mask with the squared kernel.

3.8 Covariance Estimator

The second part of the E-step involves estimating the covariance around the mean sharp image. For a mean image with dimensions $n \times n$ (or $n^2 \times 1$ when flattened), the covariance matrix is an $n^2 \times n^2$ matrix. The covariance estimator module computes this matrix using a diagonal approximation by inverting the diagonal elements of the weighted deconvolution system A_x. This makes the time taken to compute the covariance matrix linear rather than quadratic in the number of pixels in the mean image. Only the diagonal elements of the covariance matrix are stored as an $n \times n$ covariance image.

The matrix A_x has two components: a kernel-dependent component $T_k^T T_k$ and a weights-dependent component W. For simplifying the representation, the subscript γ has been dropped.

$$A_x = 1/\eta^2 T_k^T T_k + W \tag{3.21}$$

Since the matrix A_x is not explicitly computed while solving for the mean image, the diagonal entries of $T_k^T T_k$, denoted by da_1, and the diagonal entries of $\eta^2 W$, denoted

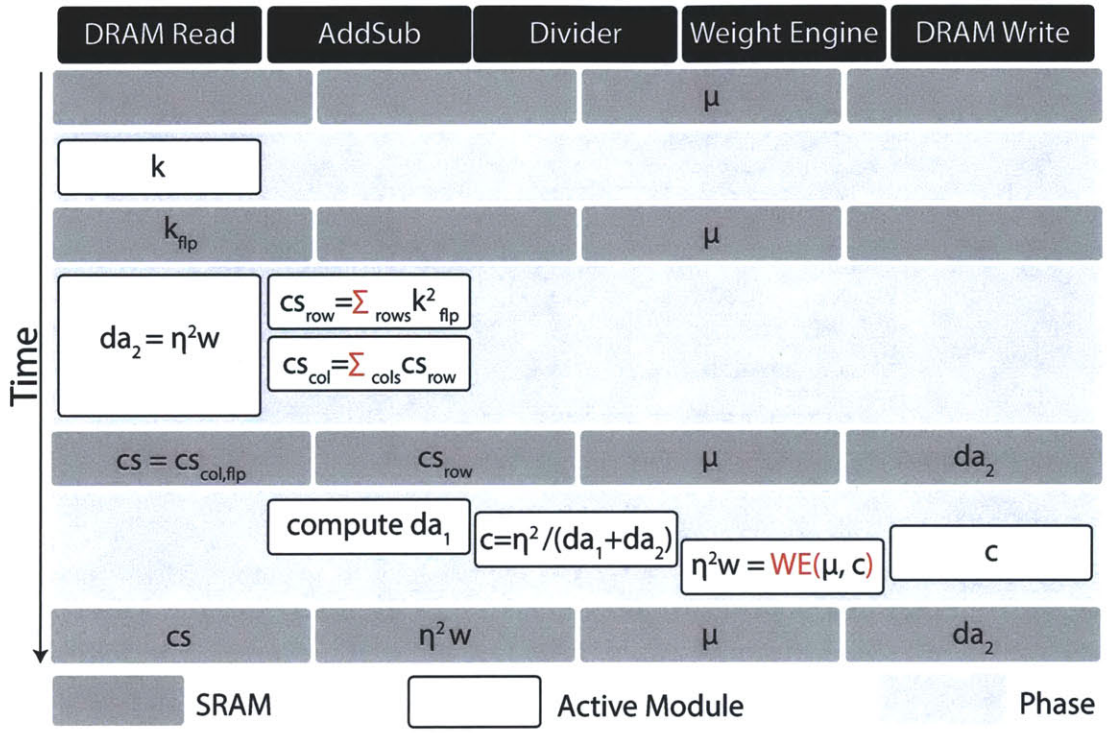


Figure 3-15: Schedule for covariance estimator.

by da_2, are computed and the covariance image entries are set using $\eta^2/(da_1 + da_2)$.

3.8.1 Optimizations

The diagonal entries of $T_k^T T_k$ or da_1 can be obtained by convolving the mask shown in Figure 3-14 with the kernel squared element by element. This requires $O(n^2 * m^2)$ operations. However, it can be seen from Figure 3-14 that the structure of the mask allows the convolution to be computed simply from the integral image of the squared kernel. The integral image (cs) can be computed in $O(m^2)$ operations, followed by a look-up into cs for each element of da_1, which results in only $O(n^2)$ operations.

3.8.2 Architecture

Figure 3-15 shows the scheduling of different arithmetic units to compute the covariance image. First, the kernel (k) is read from the DRAM and written flipped (k_{flp}) along both axes into SRAM 0 banks [0 1 2 3]. The kernel is read in the normal

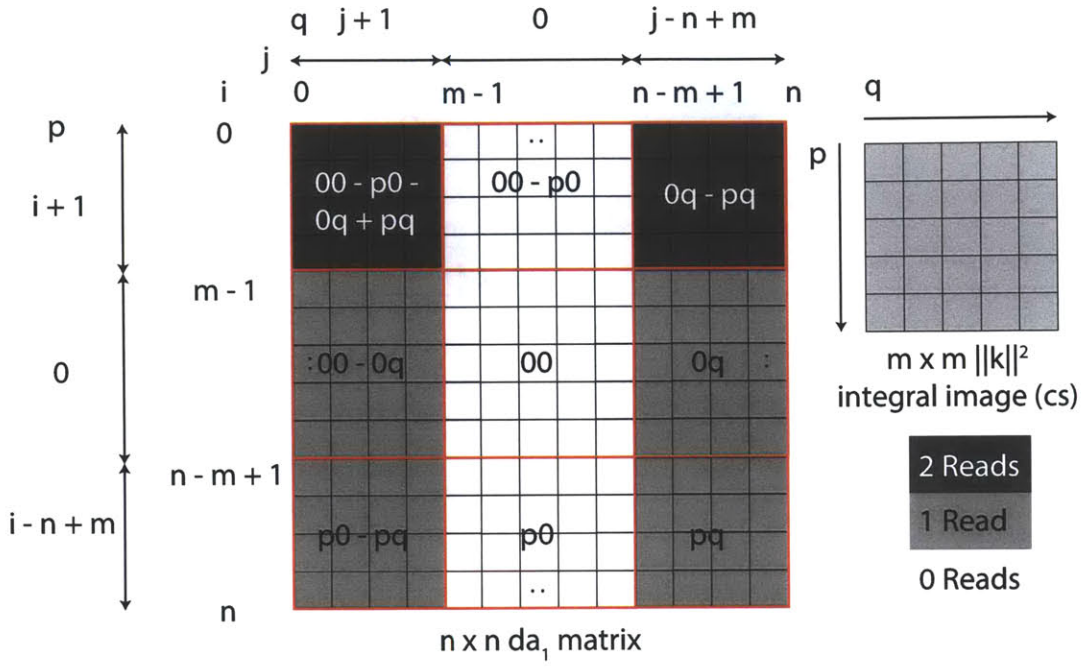


Figure 3-16: Relationship between da_1 matrix entries and integral image cs entries. Indices i and j index over rows and columns of da_1 and p and q denote the corresponding row and column index into cs matrix.

row-major order, and flipping is achieved by manipulating the SRAM bank addresses while writing it into SRAMs.

Computing da_1

Once the kernel is read from the DRAM the cumulative sum of squared kernel entries is computed along rows and the intermediate (cs_{row}) is stored into SRAM 1 banks [0 1 2 3], which serves as a transpose memory. Two rows are processed in parallel because they lie in different SRAM banks. Once the row cumulative sum is complete, the intermediate is read from SRAM 1 and cumulative sum along columns (cs_{col}) is computed and flipped to get the integral image (cs) which is stored in SRAM 0. Two columns are processed in parallel because they lie in different SRAM banks. Flipping is achieved by manipulating SRAM write addresses. The final sum is stored in a register. In parallel, $\eta^2 w$ or da_2 matrix is read from DRAM and written to SRAM 3 banks [0 1 2 3].

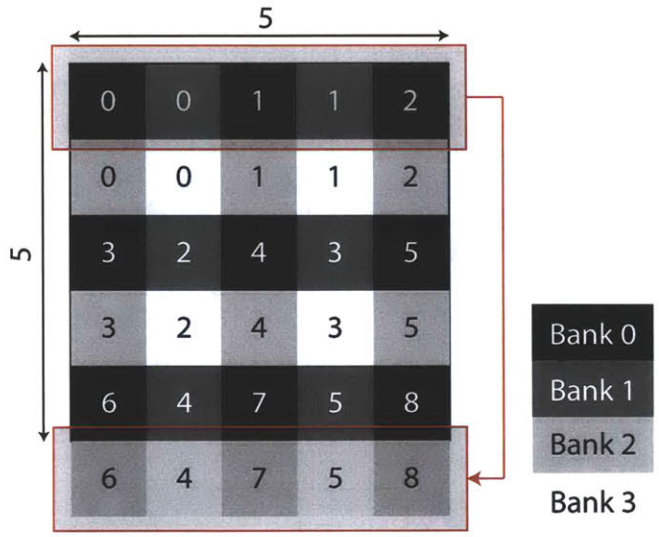


Figure 3-17: The first row of the integral image is copied and written at the end of the matrix to allow conflict free SRAM access (shown here for a 5×5 kernel).

Next, the entries of da_1 matrix are computed using the integral image cs entries using the piece-wise linear relationship shown in Figure 3-16, where (i, j) are the row and column indices for da_1 matrix and (p, q) and row and column indices for the cs matrix. The blocks in the figure represent the da_1 entries where the same relationship holds. Since the covariance matrix entries have to be written back to the DRAM at a rate of 2 pixels/cycle, we require 2 da_1 entries to be computed in parallel every cycle. However, it can be seen that computation of one da_1 entry can require up to 4 cs entries which cannot be accessed in parallel from the single port SRAM which stores the cs matrix. So a naive implementation would not meet the throughput requirement.

Optimizations The following optimizations allow us to meet the 2 pixels/cycle throughput:

- A large part of da_1 is simply the cs value at 00, this value is stored in a register to an avoid SRAM access.
- The da_1 matrix is computed in row-major order, and for a particular da_1 row index i, the row index p into cs matrix remains constant. So, $p0$ is read and

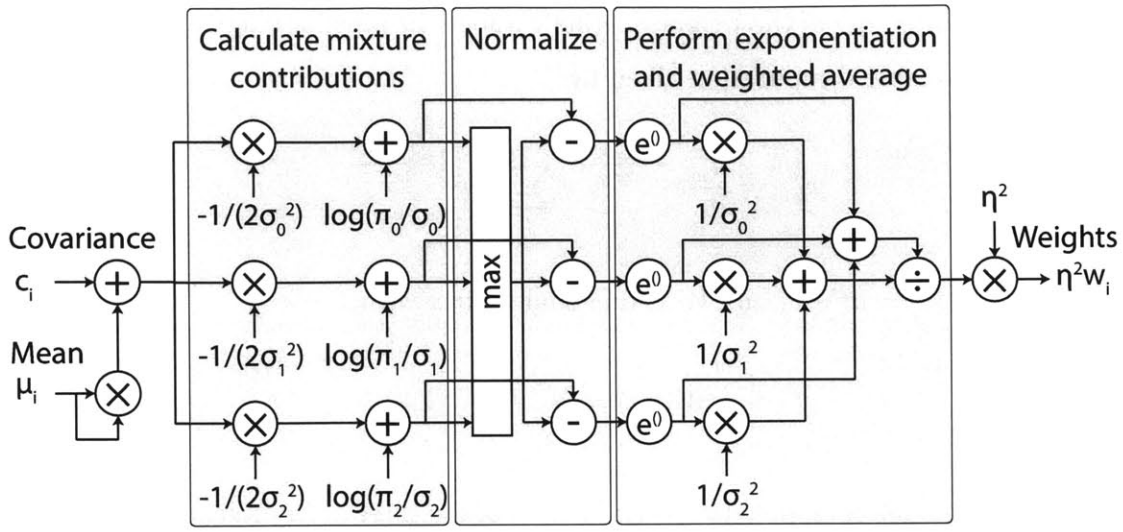


Figure 3-18: Architecture of the weights engine (shown without the pipeline registers) for a throughput of 1 pixel per cycle.

registered at the beginning of each row avoiding 1 read per column.

- The previous two optimizations remove all SRAM access conflicts except when both $0q$ and pq entries are required. These will conflict when p is even, because $0q$ and pq would then be stored in the same SRAM. In order to resolve this conflict, the first row of cs matrix is copied and written at the end of the matrix as shown in Figure 3-17 and instead of accessing $0q$, kq is accessed when p is even. This does not lead to any conflicts since k is odd. This allows a throughput of 1 pixel per cycle.

- A throughput of two pixels per cycles can be achieved simply by computing da_1 entries for two columns in parallel for any row. This can be done easily because adjacent cs columns fall in different banks because of the SRAM mapping policy.

Computing covariance and weights

In parallel, da_2 is read from SRAM 3 and both da_1 and da_2 are used to compute covariance image in a pipelined manner which is written to the DRAM. The resulting covariance image is used along with the mean image to refine the weights for the next iteration. This is performed using a fully pipelined weights engine. Figure 3-18

shows the architecture of the weights engine without the pipeline stage registers which implements weights computation given by:

$$w_{i,j_0} = \frac{\pi_{j_0}}{\sigma_{j_0}} e^{-\frac{E[\|x_i\|^2]}{2\sigma_{j_0}^2}} / \sum_j \frac{\pi_j}{\sigma_j} e^{-\frac{E[\|x_i\|^2]}{2\sigma_j^2}} \tag{3.22}$$

where $E[\| x_i \|^2] = \mu_i^2 + c_i$ and W is diagonal matrix with:

$$W(i,i) = \sum_j \frac{w_{i,j}}{\sigma_j^2} \tag{3.23}$$

The weights engine computes the contributions of the three prior mixture components in parallel. The exponentiation is carried out using a combinational look-up table which stores output values for a sub-sampled version of the input, and only the most significant 16 bits (excluding the sign bit) are used for the look-up. It should be noted that normalization of the three mixture component contributions but subtracting the maximum contribution is performed before taking the exponential. Therefore, the inputs to the exponentiation lookup table are always non-positive. This further restricts the number of entries stored in the look-up table since exponential quickly decays to zero as the input becomes large negative.

Only the diagonal entries of W are computed and written to SRAM 1 as an $n \times n$ weights matrix w. A throughput of 2 pixels/cycle is achieved by operating two weights engine in parallel. The E-step is iterated three times before proceeding on to the M-step. In the last E-step iteration, the weight computation for the next iteration is not performed and the covariance image is written to SRAM 1 instead of the weights matrix.

3.9 Correlator

After completion of the E-step, we have the mean image (μ_γ) and the covariance (C_γ) around it for the both horizontal and vertical gradient components ($\gamma = 0, 1$). These are used in the M-step to refine the kernel estimate by solving the following quadratic

programming problem for the kernel:

$$\bar{A}_{k,\gamma}(i_1, i_2) = \sum_i \mu_\gamma(i + i_1)\mu_\gamma(i + i_2) + C_\gamma(i + i_1, i + i_2) \quad (3.24)$$

$$\bar{b}_{k,\gamma}(i_1) = \sum_i \mu_\gamma(i + i_1)y_\gamma(i) \quad (3.25)$$

$$\bar{A}_k = \sum_\gamma A_{k,\gamma} \quad (3.26)$$

$$\bar{b}_k = \sum_\gamma b_{k,\gamma} \quad (3.27)$$

$$\hat{k} = \arg\min \frac{1}{2}k^T \bar{A}_k k + \bar{b}_k^T k \text{ s.t. } k \geq 0 \quad (3.28)$$

Here i sums over image pixels and i_1 and i_2 are kernel indices. These are 2-D indices but the expression uses the 1-D vectorized version of the image and the kernel. The first part of the M-step is executed by the correlator module which computes the coefficients matrix $\bar{A}_k$ and the vector $\bar{b}_k$ from the mean and covariance images and sets up the quadratic program (QP). Once the QP is set up, the gradient projection solver solves for the kernel subject to non-negativity constraints.

3.9.1 Optimizations

For an $m \times m$ kernel, $\bar{A}_{k,\gamma}$ is an $M \times M$ matrix, where $M = m^2$, representing the covariance of all $m \times m$ windows in μ_γ and $\bar{b}_{k,\gamma}$ is an $M \times 1$ vector representing the correlation with y_γ. The correlator module first computes the auto-correlation $\sum_i \mu_\gamma(i + i_1)\mu_\gamma(i + i_2)$, which is required for computing $\bar{A}_{k,\gamma}$.

The naive way to compute autocorrelation would be to multiply and accumulate two shifted versions of μ_γ, $\mu_\gamma(i + i_1, j + j_1)$ and $\mu_\gamma(i + i_2, j + j_2)$ for all possible shift values with i_1, j_1, i_2 and j_2 varying from 0 to m. For computing each element of the $M \times M$ matrix, this approach requires $O(n \times n)$ operations where $n \times n$ is the size of the mean image.

Let us consider two cases, one where both (i_1, j_1) and (i_2, j_2) are $(0, 0)$ and the other where both shifts are $(1, 1)$. In both cases, the relative shift between the two images is $(0, 0)$, and the same corresponding elements will get multiplied. So, for all

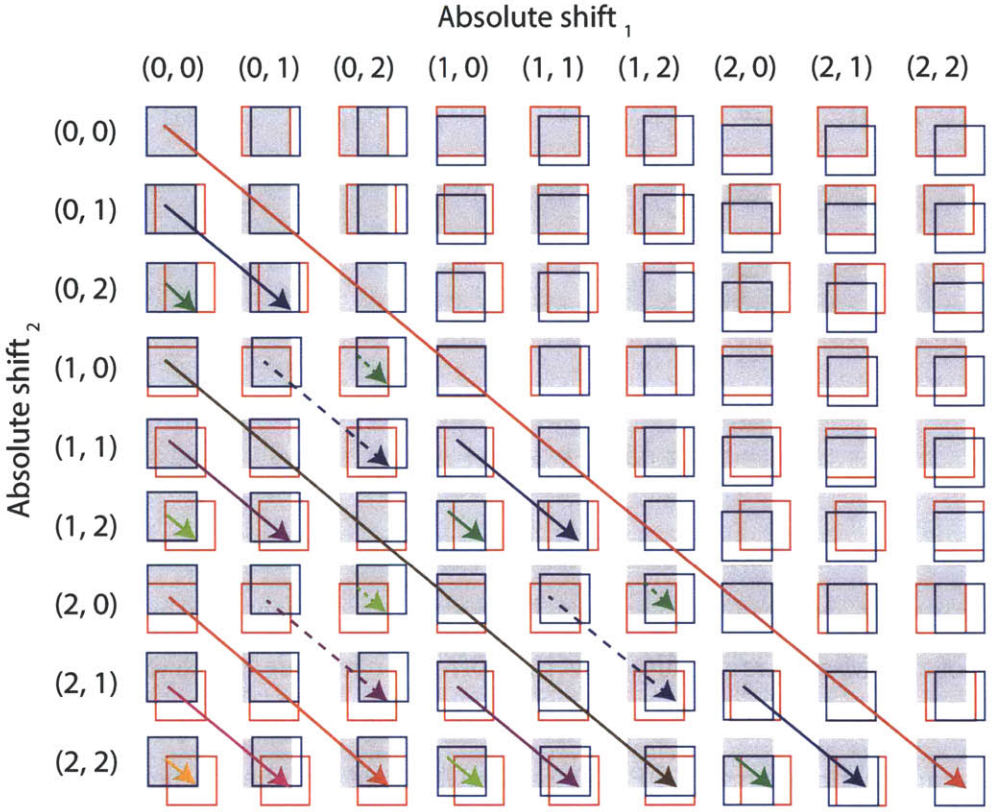


Figure 3-19: Auto-correlation matrix for a 3×3 kernel. The axes represent the absolute shifts of the image with respect to the reference.

absolute shift pairs which have the same relative shift between them, the multiply accumulate result can be computed using the result for the case when the absolute shift pair is same as the relative shift pair. This makes the correlation operation $O(M * N)$ compared to $O(M^2 * N)$ for the naive case. This is achieved by computing the integral image of multiplication of the shifted version of the image with an unshifted version and then using the integral image to obtain the results for all absolute shifts for a given relative shift.

Figure 3-19 shows the $3^2 \times 3^2$ auto-correlation matrix for a 3×3 kernel. The gray boxes represent the location of the unshifted base image. The red and the blue boxes represent the shifted images shifted by the absolute shifts on the vertical and horizontal axes with respect to the base image. The value at each location of the matrix is calculated by multiplying the shifted red and blue boxes element by

element and accumulating them in the region which is the intersection of the red, blue and gray boxes. It can be observed that, the elements along the diagonal solid lines of any color can be computed using the integral image of top left box along the diagonal line. A second pass is required to compute the elements along the dashed lines. The matrix is symmetric, therefore only the lower triangular part is computed and written to two locations into the DRAM (except for the diagonal elements which are written once).

3.9.2 Architecture

An architecture based on this optimized algorithm has been developed, but it has not been completely written and tested yet, and is therefore not described in detail.

3.10 Gradient Projection Solver

Once the quadratic program given by Equation 3.24 is set up by the correlator, the second part of the M-step involves solving it to estimate the blur kernel. Since the constraints in the problem are simple bounds, the most straightforward approach to solving this problem is the gradient projection method which minimizes:

$$J(x) = \frac{1}{2}x^T Ax + b^T x \text{ s.t. } x \geq 0 \tag{3.29}$$

For the kernel estimation problem, A is the $M \times M$ matrix $\bar{A}_k$ and b is the $M \times 1$ vector $\bar{b}_k$ which are generated by the correlator and x is $M \times 1$ flattened version of the kernel k. The matrix A is accessed directly from the DRAM. The vectors x and b are accessed from the DRAM in the first iteration, and are cached in SRAMs for accessing in subsequent iterations. The gradient projection (GP) solver also uses the 4 shared SRAMs with 4 banks each as temporary work space, however unlike other modules which store one matrix spliced across 4 banks in each SRAM, the GP solver uses each bank independently to store a different vector variable.

3.10.1 Algorithm

Gradient projection method alternates between two different types of solution refinement in its outer iteration until convergence:

1. **Find the Cauchy point** x_c, which is the first local minimum along the gradient projected onto the search space. This approach allows activation of multiple constraints per outer iteration, significantly reducing the number of iterations needed when the solution is highly constrained.

2. **Use the conjugate gradient method** to find a solution x_{cg} improving on the Cauchy point ($J(x_{cg}) \leq J(x_c)$), without violating any constraints. This accelerates convergence compared to simply repeating the Cauchy point computation. The conjugate gradient method finds a solution more quickly than gradient descent by generating a sequence of search directions that are mutually conjugate (orthogonal with respect to A), avoiding repeated updates along any one direction.

3. **Check convergence** by evaluating the cost $J = \frac{1}{2}x^T Ax - b^T x$ and seeing how much it has changed since the last iteration. If the difference is larger than the tolerance (e.g. 10^{-12}), return to step 1 and repeat. Otherwise, set $x = x_{cg}$, the most recent result from conjugate gradient refinement, and exit.

Each outer iteration of the gradient projection algorithm finds a Cauchy point and then refines this solution using the conjugate gradient method.

Cauchy point computation

The Cauchy point computation step can be elaborated as follows:

1. Compute the gradient: $g = \delta J/\delta x = Ax + b$

2. Let $x(t) = x - gt$ (the line through the current solution along the gradient direction). Compute the set of break-points t where $x(t)$ crosses any constraint boundary. With our constant bound at zero, there is one breakpoint per dimension: $t^i = x^i/g^i$ where the superscript i is a vector index.

3. Remove the negative breakpoints (since we are concerned only with the negative gradient direction) and sort the remaining breakpoints in ascending order. For each distinct pair of breakpoints t^i and t^{i+1} in this sorted list:

   (a) Compute $x_j = max(x(t^i), 0)$, which is $x(t^i)$ projected onto the search space.

   (b) Compute p_j element-wise as

$$p_j^i = \begin{cases} -g^i & \text{when } x_j^i > 0 \\ 0 & \text{when } x_j^i = 0 \end{cases}$$

   which is the gradient projected onto any constraints that are active at x_j.

   (c) Compute the 1-D quadratic coefficients of the objective function along the line segment between x_j and the next breakpoint (the constant coefficient $f_0 = J(x_j)$ is not needed):

$$J_j(t - t^i) = f_0 + f_1(t - t^i) + f_2(t - t^i)^2 \quad (3.30)$$

$$f_1 = b^T p_j + x_j^T A p_j \quad (3.31)$$

$$f_2 = p_j^T A p_j \quad (3.32)$$

   (d) Compute the parameter of the local minimum: $t^* = t^i - f_1/f_2$. If $t^* < t^{i+1}$, then the local minimum lies on the current segment; set $x_c = x_j + (t^* - t^i)p_j$ and proceed to conjugate gradient refinement. Otherwise, return to step (a) and examine the next pair of breakpoints.

Conjugate gradient refinement

The conjugate gradient refinement step then runs for one or more iterations. Usually only one iteration is needed during the early stages of optimization; in the final steps, once the active set is stable, typically 10 - 30 are required to zero in on the global minimum. The steps involved are the same as performed by the CG solver described previously, however, the matrix multiplication in this case is a direct one and is not

implemented through convolution. Also, there is extra condition checking to restrict the step length in case the resultant solution violates any constraints.

1. Initialize the solution $x_{cg} = x_c$ and the residual and the search direction $r_{cg} = p_{cg} = -Ax_{cg} - b$. Also, initialize the residual norm $\rho = r_{cg}^T r_{cg}$.

2. Repeat the following steps until convergence or until a new constraint is activated:

   (a) Compute a step size: $\alpha = \rho / p_{cg}^T A p_{cg}$

   (b) Determine the maximum allowable step size in each dimension based on the constraints: $\alpha_{max}^i = -x_{cg}^i / p_{cg}^i$

   (c) Select a step size: $\alpha^* = min[\alpha, min_i max(0, \alpha_{max}^i)]$

   (d) Update the solution: $x_{cg} = x_{cg} + \alpha^* p_{cg}$

   (e) Compute a new residual: $r_{cg} = r_{cg} - \alpha^* A p_{cg}$

   (f) 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 (e.g. 10^{12}), exit the loop.

   (g) Compute a step size for the search direction update: $\beta = \rho / \rho_{last}$

   (h) Update the search direction: $p_{cg} = r_{cg} + \beta p_{cg}$.

   (i) Return to step (a).

Optimizations

The gradient projection algorithm employs primarily numerical operations including matrix/vector products, dot products, comparisons, and sorting. Considering that the problem sizes will be on the order of 1,000 variables, matrix/vector multiplications are the most time-consuming part. A reference implementation of the algorithm in C is developed and modified to reduce matrix/vector operations in the following ways:

- The conjugate gradient refinement ignores the dimensions that have active constraints; the solution values on these dimensions are held at zero until the next

outer iteration. If the solution is sparse (as it usually is) and all but 100 of the constraints are active, then matrix and vector operations within the conjugate gradient loop deal with a 100 x 100 matrix instead of 1,000 x 1,000.

- The matrix-vector product Ap_j is updated incrementally during Cauchy point computation. This is possible since the values of p_j start out as $-g$ and change irreversibly to zero as more of the constraints are activated. Hence the complete product is only evaluated once per outer iteration.

- The gradient g is updated incrementally throughout both major steps, based on the incremental changes that are being made to x. This allows initialization of the conjugate gradient residual as $r_{cg} = -g$ instead of evaluating $r_{cg} = -Ax_{cg} - b$.

- At the end of each conjugate gradient refinement, the cost is recalculated using the updated gradient:

$$J = \frac{1}{2}x^T Ax + x^T b \tag{3.33}$$

$$= \frac{1}{2}(x^T(Ax + b) + x^T b) \tag{3.34}$$

$$= \frac{1}{2}(x^T g + x^T b) \tag{3.35}$$

  This lets us avoid explicitly computing the Ax product once per outer iteration.

These optimizations provide a constant-factor improvement in performance. Consider solving a particular problem that takes N_o outer iterations, with an average of N_b breakpoints checked while finding the Cauchy point and N_{cg} conjugate gradient iterations per outer iteration. The direct implementation of the gradient projection method required $N_o(2 + N_b + N_{cg})$ matrix/vector multiplications. This implementation requires $1 + N_o(2 + N_{cg})$, counting a partial product to update the gradient along the constrained dimensions.

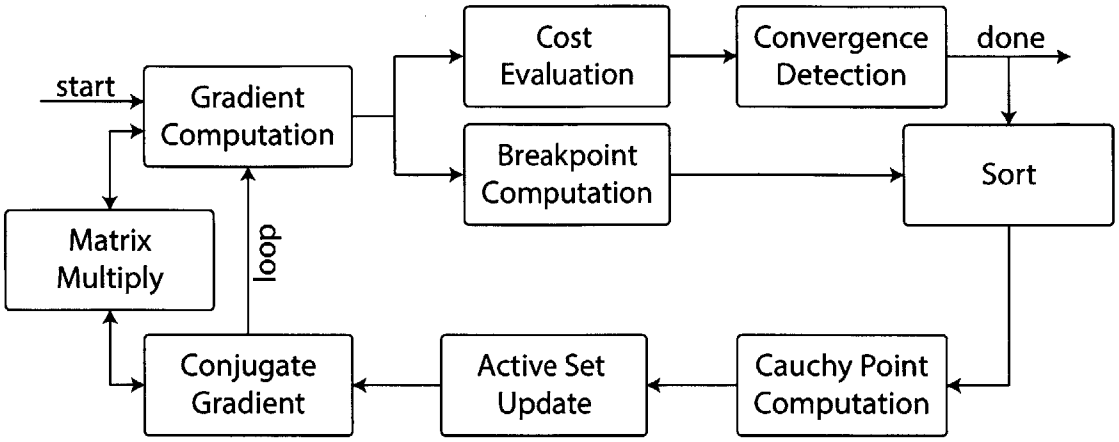


Figure 3-20: Block diagram for gradient projection solver.

3.10.2 Architecture

Figure 3-20 shows a block diagram for the gradient projection solver. The architecture has been implemented in Bluespec SystemVerilog by labmates, Michael Price and Rahul Rithe. The algorithm requires iterative execution of Cauchy point computation and conjugate gradient. The results of Cauchy point computation are used in conjugate gradient and the results of conjugate gradient are used in the next iteration of Cauchy point computation. To optimize the computation resources, the non-concurrency of these modules is exploited. This allows sharing of the matrix multiplier, sort and floating point arithmetic units among these two modules. The Cauchy point computation and conjugate gradient modules implement the control to access these resources and the SRAMs. The module is also connected to the memory interface arbiter which allows it to access the coefficients matrix A from the external DRAM.

Matrix Multiplier

The algorithm performs matrix multiplication at several steps in both Cauchy point computation as well as conjugate gradient. In order to maximize hardware sharing and simplify the control logic, the matrix multiplier is implemented as a separate unit that can be accessed through an arbiter by both modules.

The multiplication is performed in column major order, that is for performing $Ax = b$ a complete column of A is read in from the DRAM and multiplied with one element of x and a temporary vector is generated which has the contribution of the element of x to all the entries in the final product. When the second column of A is read in and multiplied with the second element of x, the resultant is added to the previous product and so on. This allows us to incrementally update of the product when only a few elements of the vector x change.

The matrix multiplier can also handle partial matrix multiply when not all rows and columns of the matrix A are active. It should be noted that the matrix A is symmetric, so column major access of the coefficients is done by reading A along rows. This minimizes DRAM access latency since A is stored in row major order in the DRAM. The available DRAM bandwidth allows two coefficients to be accessed in parallel, so the matrix multiplier is also designed to support the same amount of parallelism.

Sort

Determination of the Cauchy point requires sorting an array of breakpoints so that the number of active constraints increases monotonically during the search. The sort is implemented using a merge sort approach that requires a single floating-point comparator. A simplified block diagram of this module is shown in Figure 3-21

This merge sort needs two SRAMs. Given a list of N input values starting in one of those SRAMs, the merge sort will copy data back and forth between the SRAMs a total of $log_2 N$ iterations. During iteration i, the $N/2^i$ batches of 2^i sorted values will be merged into $N/2^{i+1}$ batches of 2^{i+1} sorted values. At the beginning, the list is not sorted; after the first iteration, batches of 2 values in the list will be sorted, then batches of 4, and so on until the entire list is sorted.

Within each output batch, the sequencer reads the first value from the input batch and stores it in the reference value register. Each subsequent value coming from the input SRAM will be compared to this register. The smaller of the two values is saved in the output SRAM. The larger of the two values is stored in the reference value

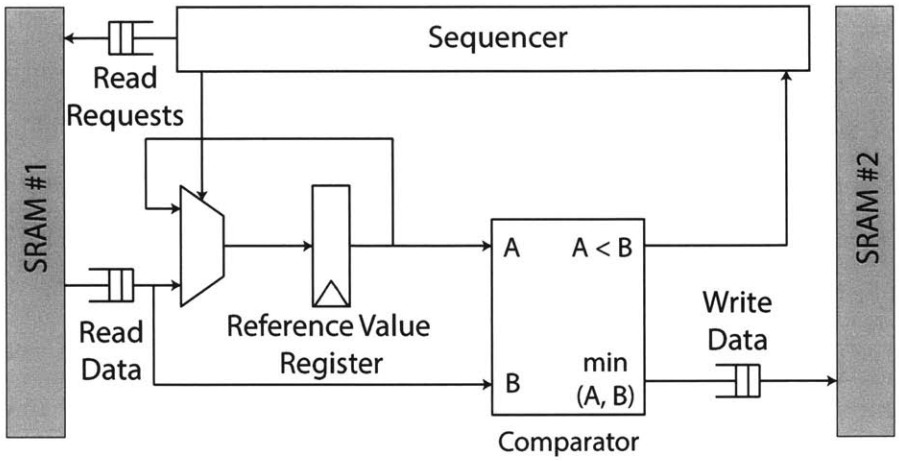


Figure 3-21: Architecture of sort module. (Courtesy M.Price)

register. This ensures that the values written to the output SRAM are in ascending order. At the end of the batch, the reference value (which must be the largest in the input batch) is saved as the last entry in the output batch. The sequencer determines which element to read from the input SRAM based on internal counters and the comparison result.

After each iteration, the interfaces for SRAMs #1 and #2 are swapped. If the number of iterations is odd the data is copied back to the original SRAM once it is fully sorted.

Floating point unit (FPU)

The floating point adder/subtractors, multipliers (except those needed for matrix multiplication) and a divider are implemented in the FPU. All the floating point operators can be accessed in parallel by both Cauchy point computation and the conjugate gradient modules through an arbiter.

The control logic within each of the two major modules - the Cauchy point computation unit and the conjugate gradient solver is a state machine that sequences the external operators as well as any internal operators. Each state corresponds to

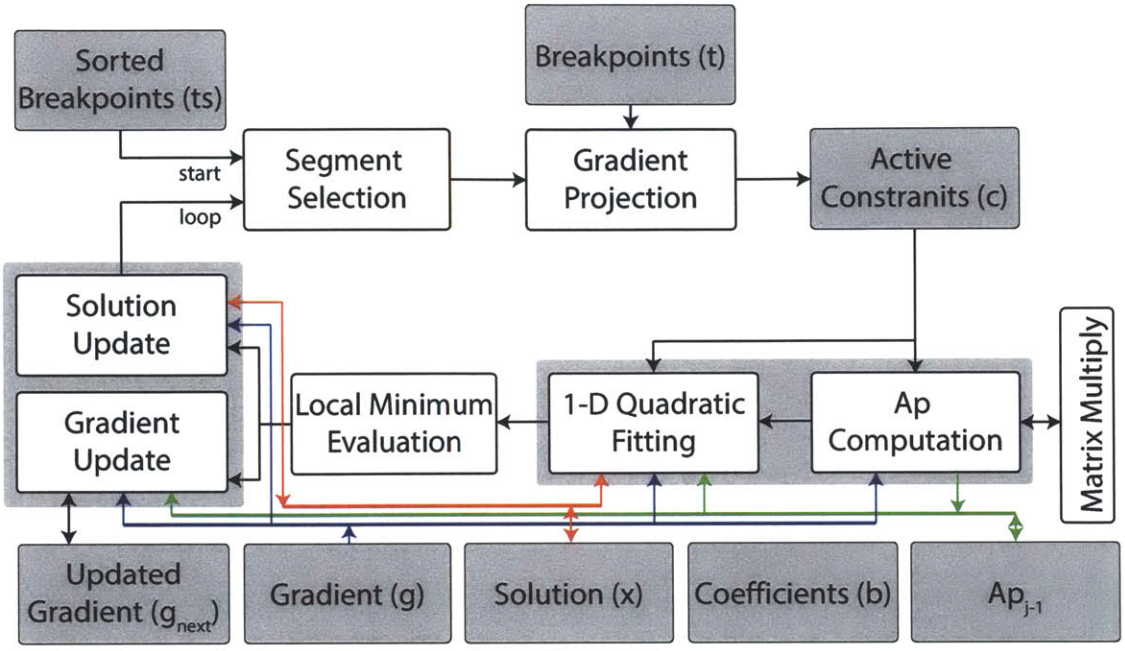


Figure 3-22: Architecture of Cauchy point computation module.

a discrete step of the algorithm, in which as many operations as possible are evaluated concurrently. The steps are partitioned such that the outputs of each step are required for the inputs of the next step (otherwise they could be run at the same time).

The schedule of computation steps, including their input data sources and result data destinations, are shown in the following tables. Each vector variable is mapped to an SRAM; some SRAMs are reused for multiple variables when it is safe. Some variables are moved between SRAMs in order to avoid reading and writing the same SRAM during any given operation.

Before first iteration

The cost function and gradient need to be initialized using the full matrix/vector product before the Cauchy point module can be started.

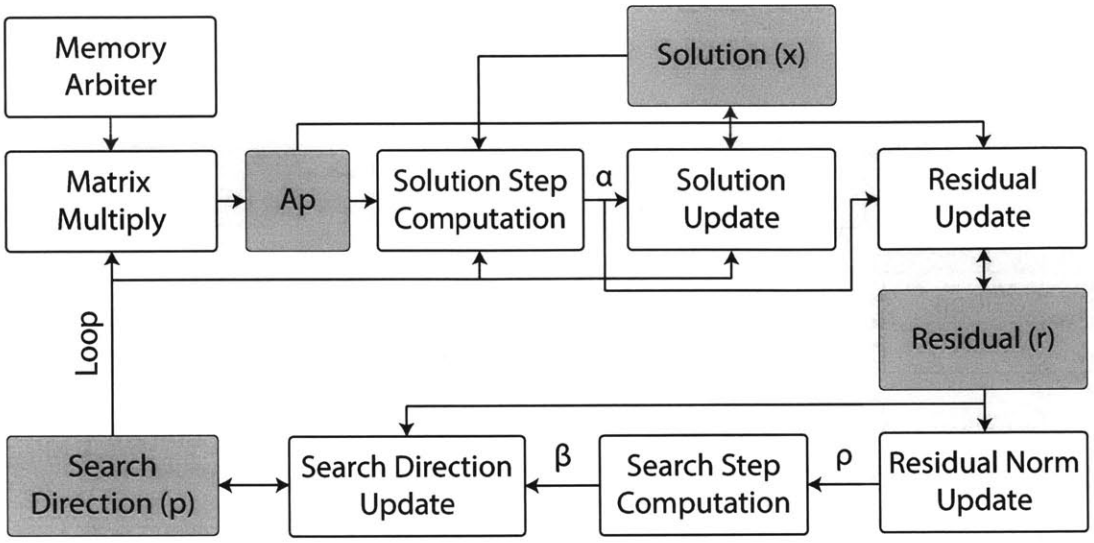


Figure 3-23: Architecture of conjugate gradient module.

	Inputs		Outputs	
Step	Source	Name	Destination	Name
Matrix/vector multiplication	DRAM	$\mathbf{A}$	SRAM 3	$\mathbf{Ax}$
	SRAM 0	$\mathbf{x}$		
Cost and gradient evaluation	SRAM 0	$\mathbf{x}$	SRAM 7	$\mathbf{g}$
	SRAM 1	$\mathbf{b}$	Register	J
	SRAM 3	$\mathbf{Ax}$		

Cauchy point computation

Figure 3-22 illustrates the implementation of Cauchy point computation. The following is the schedule for the Cauchy point module. The steps bracketed by double horizontal lines make up a loop that is executed for each sorted breakpoint.

	Inputs		Outputs	
Step	Source	Name	Destination	Name
Detect convergence	Register	J	Terminate outer iteration	
	Register	J_{last}	Register	J_{last}

Compute and clamp breakpoints	SRAM 0 SRAM 7	$\mathbf{x}$ $\mathbf{g}$	SRAM 2	$\mathbf{t}$
Sort breakpoints	SRAM 2	$\mathbf{t}$	SRAM 8 SRAM 5	$\mathbf{t}_{\text{sorted}}$ $\mathbf{t}_{\text{scratch}}$
Initialize **Ap**	DRAM SRAM 7	$\mathbf{A}$ $\mathbf{g}$	SRAM 4 SRAM 11 Register	$\mathbf{Ap}$ $\mathbf{g}_{\text{next}}$ p_changed
Determine start and end of segment	SRAM 8	$\mathbf{t}_{\text{sorted}}$	Register Register Skip to next segment	t^i t^{i+1}
Determine location and gradient	Register SRAM 0 SRAM 2 SRAM 11 SRAM 7	t^i $\mathbf{x}$ $\mathbf{t}$ $\mathbf{g}_{\text{next}}$ $\mathbf{g}$	SRAM 5 SRAM 6 Register SRAM 9	$\mathbf{x_j}$ $\mathbf{p_j}$ p_changed $\mathbf{g}_{\text{next}}$
Compute a partial update of **Ap** product	Register DRAM SRAM 7 SRAM 4	p_changed $\mathbf{A}$ $\mathbf{g}$ $\mathbf{Ap}$	SRAM 3 SRAM 10	$\mathbf{\Delta Ap}$ $\mathbf{Ap}$
Update **Ap** product	SRAM 3 SRAM 10	$\mathbf{\Delta Ap}$ $\mathbf{Ap}$	SRAM 4	$\mathbf{Ap}$
Fit objective function	SRAM 5 SRAM 1 SRAM 4 SRAM 6	$\mathbf{x_j}$ $\mathbf{b}$ $\mathbf{Ap}$ $\mathbf{p_j}$	Register Register	f_1 f_2
Find local minimum	Register Register	f_1 f_2	Register Cauchy point found	t^*

Evaluate Cauchy point	Register	t^*	SRAM 2	$\mathbf{x_c}$
	SRAM 4	$\mathbf{Ap}$	SRAM 11	$\mathbf{g}_{\text{next}}$
	SRAM 5	$\mathbf{x_j}$	Register	`inds_inactive`
	SRAM 9	$\mathbf{g}_{\text{next}}$	Register	`inds_inactive`
	SRAM 6	$\mathbf{p_j}$		
Update gradient	Register	t^i	SRAM 11	$\mathbf{g}_{\text{next}}$
	Register	t^{i+1}		
	SRAM 9	$\mathbf{g}_{\text{next}}$		
	SRAM 4	$\mathbf{Ap}$		

Conjugate gradient refinement

Figure 3-23 illustrates how conjugate gradient refinement is implemented. The following is the schedule for the CG module. The steps bracketed by double horizontal lines make up a loop that is executed until the solution has converged or activated a new constraint.

	Inputs		Outputs	
Step	Source	Name	Destination	Name
Initialize variables	Register	`inds_inactive`	SRAM 3	$\mathbf{p_{cg}}$
	SRAM 11	$\mathbf{g}_{\text{next}}$	SRAM 4	$\mathbf{\Delta g}_{\text{inactive}}$
	SRAM 2	$\mathbf{x_c}$	SRAM 7	$\mathbf{r_{cg}}$
			SRAM 9	$\mathbf{g}_{\text{next}}$
			SRAM 8	$\mathbf{x}_{\mathbf{c,last}}$
Weight search direction	Register	`inds_inactive`	SRAM 0	$\mathbf{Ap_{cg}}$
	DRAM	$\mathbf{A}$	SRAM 5	$\mathbf{p_{cg}}$
	SRAM 3	$\mathbf{p_{cg}}$		

Accumulate dot products	SRAM 0	$\mathbf{Ap_{cg}}$	Register	ρ
	SRAM 5	$\mathbf{p_{cg}}$	Register	ρ_{last}
	SRAM 7	$\mathbf{r_{cg}}$		
Evaluate target step size	Register	ρ	Register	α^*
	Register	ρ_{last}		
Determine maximum step size	Register	α^*	Register	$\alpha_{\max}$
	SRAM 2	$\mathbf{x_{cg}}$	Register	`constraint_hit`
	SRAM 5	$\mathbf{p_{cg}}$		
Update solution and residual	Register	$\alpha_{\max}$	SRAM 10	$\mathbf{x_{cg}}$
	SRAM 0	$\mathbf{Ap_{cg}}$	SRAM 11	$\mathbf{\Delta g}_{\text{inactive}}$
	SRAM 2	$\mathbf{x_{cg}}$	SRAM 6	$\mathbf{r}_{\mathbf{cg},\text{last}}$
	SRAM 4	$\mathbf{\Delta g}_{\text{inactive}}$	SRAM 12	$\mathbf{r_{cg}}$
	SRAM 5	$\mathbf{p_{cg}}$	Register	$\lvert r_{cg}\rvert$
	SRAM 7	$\mathbf{r_{cg}}$		
Check convergence	Register	$\lvert r_{cg}\rvert$	Whether to terminate	
	Register	`constraint_hit`		
Accumulate dot products	SRAM 6	$\mathbf{r}_{\mathbf{cg},\text{last}}$	Register	ρ
	SRAM 12	$\mathbf{r_{cg}}$	Register	ρ_{last}
			SRAM 7	$\mathbf{r_{cg}}$
Evaluate update size	Register	ρ	Register	β
	Register	ρ_{last}		
Update search direction	Register	β	SRAM 3	$\mathbf{p_{cg}}$
	SRAM 5	$\mathbf{p_{cg}}$	SRAM 2	$\mathbf{x_{cg}}$
	SRAM 7	$\mathbf{r_{cg}}$	SRAM 4	$\mathbf{\Delta g}_{\text{inactive}}$
	SRAM 10	$\mathbf{x_{cg}}$		
	SRAM 11	$\mathbf{\Delta g}_{\text{inactive}}$		

Before next iteration

After conjugate gradient refinement, the gradient and cost estimates (this time using partial matrix/vector products) are updated before starting the next outer iteration.

	Inputs		Outputs	
Step	Source	Name	Destination	Name
Copy results	SRAM 10	$\mathbf{x_{cg}}$	SRAM 2	$\mathbf{x_{cg}}$
	SRAM 11	$\mathbf{\Delta g}_{\text{inactive}}$	SRAM 4	$\mathbf{\Delta g}_{\text{inactive}}$
Calculate gradient update	Register	`inds_inactive`	SRAM 3	$\mathbf{\Delta g}_{\text{active}}$
	DRAM	$\mathbf{A}$		
	SRAM 2	$\mathbf{x_{cg}}$		
	SRAM 8	$\mathbf{x}_{\text{c,last}}$		
Update gradient, cost and solution	Register	`inds_inactive`	SRAM 0	$\mathbf{x}$
	SRAM 3	$\mathbf{\Delta g}_{\text{active}}$	SRAM 7	$\mathbf{g}$
	SRAM 1	$\mathbf{b}$	Register	J
	SRAM 2	$\mathbf{x_{cg}}$		
	SRAM 4	$\mathbf{\Delta g}_{\text{inactive}}$		
	SRAM 9	$\mathbf{g}_{\text{next}}$		

3.11 Results

The complete system is implemented using Bluespec SystemVerilog as the hardware description language, and verified to be bit accurate with the reference software implementation. Figure 3-24 shows an input blurred image and the output deblurred image and the estimated kernel.

The component modules are synthesized in TSMC 40 nm CMOS technology for a clock frequency of 200MHz and V_{DD} of 0.9V. The following tables show the synthesis

Figure 3-24: Results for image deblurring. The image on the left is the blurred input image. The image on the right is the deblurred image along with the estimated kernel.

area breakdown of different modules.

3.11.1 2-D Transform Engine

Table 3.4 shows the area breakdown of the 2-D transform engine. The SRAM interface FIFOs dominate the area.

Module	Logic Area (kgates)
Total Area	42.45
FIFOs	27.04
Arithmetic Units	7.61
Subtractor (4)	1.90
Scheduling and state	7.80

Table 3.4: Area breakdown for 2-D transform engine.

3.11.2 Convolution Engine

Table 3.5 shows the area breakdown of the convolution engine. A significant portion of the area is taken by the floating point arithmetic units as well as by the state machine for scheduling the conjugate gradient steps. The floating point area comes as a result of 2 complex multipliers each of which use 4 parallel multipliers followed by one adder and one subtractor, and two squared magnitude computation unit each of which use 2 multipliers and one adder. About 20% of the area is taken up by

scheduling logic and state registers because of the complicated control flow in the convolution engine, however, meeting timing is not difficult.

Module	Logic Area (kgates)
Total Area	115.06
FIFOs	26.48
Arithmetic Units	80.24
AddSub (6)	1.87
Multiplier (12)	4.62
Scheduling and state	21.93

Table 3.5: Area breakdown for convolution engine.

3.11.3 Conjugate Gradient Solver

Table 3.6 shows the area breakdown of the conjugate gradient solver. The FIFO area is similar to the transform engine, because the two modules have similar interfaces to SRAMs, DRAM and FFT engine. A significant portion of the area is taken by the floating point arithmetic units as well as by the state machine for scheduling the conjugate gradient steps.

It should be noted that the current implementation of CG solver has the minimum number of floating point arithmetic units required to sustain the throughput and avoid stalling because of conflicts on the arithmetic units. Therefore, most of the arithmetic units are shared, which contributes to significant muxing logic that adds to the critical path. It is apparent that floating point modules with higher drive strength are being used to meet timing specifications (for example, the area of AddSub unit can be compared to Subtractor from transform engine). It would be interesting to observe how the area would change if the amount of sharing is reduced. On one hand, reducing sharing would reduce the area by reducing the muxing logic and reducing the size of individual arithmetic units (as they would see a larger effective clock period). On the other hand, it would add to the number of floating point units in the module.

Module	Logic Area (kgates)
Total Area	142.74
FIFOs	27.04
Arithmetic Units	80.24
AddSub (4)	3.27
Multiplier (6)	4.83
Divider	29.89
Sum3	8.28
Scheduling and state	35.19

Table 3.6: Area breakdown for conjugate gradient solver.

Chapter 4

Conclusion

In the first part of this work, circuit implementations for three components of a multi-application bilateral filtering processor have been described - the grid interpolation block, the HDR image creation and contrast adjustment blocks, and the shadow correction block. These modules have been integrated into a reconfigurable multi-application bilateral filtering processor which has been implemented in 40 nm CMOS technology. The processor achieves 15× reduction in run-time compared to a CPU implementation, while consuming 1.4 mJ/megapixel energy, a significant reduction compared to CPU or GPU implementations. This energy scalable implementation can be efficiently integrated into portable multimedia devices for real time computational photography.

In the second part of this work, complete system architecture for hardware image deblurring has been proposed. The system has been designed using Bluespec SystemVerilog (BSV) as the hardware description language and verified to be bit accurate with the reference software implementation. Synthesis results in 40nm CMOS technology have been presented.

4.1 Key Challenges

The following points describe the key challenges in the design and how they were addressed:

- **Complex data dependencies**: The deblurring algorithm is iterative in nature at multiple levels. The estimation is run at different resolutions, for each resolution, EM optimizer is run for several iterations, and inside each EM iteration, the image and kernel are estimated iteratively using gradient solvers. This leads to complex data dependencies and severely limits the amount of parallelism at a macro level. The challenge here is to identify ways to extract parallelism at a micro level while respecting data dependencies and resource constraints.

- **Working set size** denotes the number of pixels on which the output at a given pixel depends. The worst case working set for image deblurring is the entire window of 128×128 pixels from which the kernel is estimated. This translates to a large on-chip memory requirement to minimize DRAM accesses and allow parallel operations. The challenge then is to identify ways to use the on-chip scratch memory effectively and maximize data reuse.

These challenges were addressed by

1. Reducing the computational complexity of the algorithms by identifying steps where computation can be either reused or avoided entirely. The optimizations section in the description of each component module highlights this. These optimizations, in most cases, benefit both software and hardware implementations.

2. Devising a processing pipeline that takes into account the data dependencies in the algorithm and the available resources (including arithmetic units, scratch memory and DRAM bandwidth). This step identifies which parts of the computation can be parallelized, which parts have to be sequential, and which sequential steps can be pipelined. This also involves identifying the best way to map data to on and off chip memory to minimize latency and achieve the desired throughput.

3. Reducing area cost of the implementation by identifying which modules offer the maximum gain in terms of area when shared without introducing too much control overhead. This trade-off has been explored in the current design by

sharing modules such as the FFT engine and the SRAMs at the macro level and the floating point arithmetic modules at the micro level.

The image deblurring system is still work in progress. Immediate future work includes design of the correlator engine and integration of all the component modules, followed by FPGA evaluation and implementation in 40 nm CMOS technology.

4.2 Future Work

Future work on bilateral filtering could explore several directions:

- **New applications** An interesting direction to explore would be use the bilateral filtering processor to accelerate other applications that require edge aware filtering, for example video denoising [18], abstraction and stylization [19] and optical flow estimation and occlusion detection [20].

- **Increasing configurability** The current implementation uses a fixed size Gaussian kernel to perform the filtering. Extending the hardware to support different kernel sizes and types of kernels would allow support for a wider range of applications.

- **Higher dimensional grids** The concept of bilateral grid can be extended to higher dimensions. The current implementation supports 3 dimensions and can be used for filtering only gray-scale images, but having say a 5-D grid would allow filtering color images. Future work could explore hardware implementation of higher dimensional grids or even grids with configurable number of dimensions.

Future work on image deblurring could explore several directions:

- **Power Gating** Since not all modules in the deblurring processor are active at any given time, this architecture presents an opportunity to save power by turning off idle processing blocks. Moreover, when a block is active or inactive can be determined based on configuration parameters for most of the modules

and is not data dependent, which can be taken advantage of for power gating. This direction can be explored in future work.

- **Spatially variant blur** This work addresses camera shake blur which can be modeled as a convolution with a spatially invariant blur kernel. In a lot of real life situations, the blur kernel is spatially variant. This can potentially be addressed by using the proposed deblurring system to infer different kernels from multiple windows in the image and then use them to deconvolve parts of the image. Addressing motion blur and defocus blur can also be explored in future work.

Bibliography

[1] F. Durand and J. Dorsey, "Fast bilateral filtering for the display of high-dynamic-range images," *Vine*, vol. 21, no. 3, pp. 257–266, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?doid=566654.566574

[2] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama, "Digital photography with flash and no-flash image pairs," *ACM Transactions on Graphics*, vol. 23, no. 3, p. 664, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1015706.1015777

[3] E. Eisemann and F. Durand, "Flash photography enhancement via intrinsic relighting," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 673–678, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015778

[4] M. Brown and D. G. Lowe, "Automatic Panoramic Image Stitching using Invariant Features," *International Journal of Computer Vision*, vol. 74, no. 1, pp. 59–73, 2006. [Online]. Available: http://www.springerlink.com/index/10.1007/s11263-006-0002-3

[5] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman, "Efficient marginal likelihood optimization in blind deconvolution," pp. 2657–2664, 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5995308

[6] R. Ng, M. Levoy, G. Duval, M. Horowitz, and P. Hanrahan, "Light Field Photography with a Hand-held Plenoptic Camera," *Main*, vol. 2, no. CTSR 2005-02, pp. 1–11, 2005. [Online]. Available: http://www.soe.ucsc.edu/classes/cmps290b/Fall05/readings/lfcamera-150dpi.pdf

[7] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," pp. 839–846, 1998. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=710815

[8] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid," *ACM Transactions on Graphics*, vol. 26, no. 3, p. 103, 2007. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1276377.1276506

[9] S. Paris and F. Durand, "A Fast Approximation of the Bilateral Filter Using a Signal Processing Approach," *International Journal of Computer*

Vision, vol. 81, no. 1, pp. 24–52, 2007. [Online]. Available: http://www.springerlink.com/index/10.1007/s11263-007-0110-8

[10] B. Weiss, "Fast median and bilateral filtering," *ACM Transactions on Graphics*, vol. 25, no. 3, p. 519, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1141911.1141918

[11] P. E. Debevec and J. Malik, "Recovering high dynamic range radiance maps from photographs," *Film*, vol. 31, no. 1, pp. 369–378, 1997. [Online]. Available: http://portal.acm.org/citation.cfm?id=1401174

[12] R. Fergus, B. Singh, A. Hertzmann, S. T. Roweis, and W. T. Freeman, "Removing camera shake from a single photograph," *ACM Transactions on Graphics*, vol. 25, no. 3, p. 787, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1141911.1141956

[13] A. Levin, "Blind Motion Deblurring Using Image Statistics," in *NIPS*, B. Schölkopf, J. C. Platt, and T. Hoffman, Eds. MIT Press, 2006, pp. 841–848.

[14] Q. Shan, J. Jia, and A. Agarwala, "High-quality motion deblurring from a single image," *ACM Transactions on Graphics*, vol. 27, no. 3, p. 1, 2008. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1360612.1360672

[15] S. Cho and S. Lee, "Fast motion deblurring," *ACM Transactions on Graphics*, vol. 28, no. 5, p. 1, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1618452.1618491

[16] N. Joshi, R. Szeliski, and D. J. Kriegman, "PSF estimation using sharp edge prediction," pp. 1–8, 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4587834

[17] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman, "Understanding and Evaluating Blind Deconvolution Algorithms," *IEEE Conference on Computer Vision and Pattern Recognition (2009)*, vol. 014, no. 2, pp. 1964–1971, 2009. [Online]. Available: http://eprints.pascal-network.org/archive/00005638/

[18] E. P. Bennett and L. McMillan, "Video enhancement using per-pixel virtual exposures," *ACM Transactions on Graphics*, vol. 24, no. 3, p. 845, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1073204.1073272

[19] H. Winnemöller, S. C. Olsen, and B. Gooch, "Real-time video abstraction," *ACM Transactions on Graphics*, vol. 25, no. 3, p. 1221, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1141911.1142018

[20] J. Xiao, H. Cheng, H. Sawhney, C. Rao, and M. Isnardi, "Bilateral filtering-based optical flow estimation with occlusion detection," *Computer VisionECCV 2006*, vol. 3951, pp. 211–224, 2006. [Online]. Available: http://www.springerlink.com/index/12h4u7801u223348.pdf