An Energy-Scalable Accelerator for Blind Image Deblurring

Priyanka Raina, *Student Member, IEEE*, Mehul Tikekar, *Member, IEEE*, and Anantha P. Chandrakasan, *Fellow, IEEE*

***Abstract*— Camera shake is a common cause of blur in cellphone camera images. Removing blur requires deconvolving the blurred image with a kernel, which is typically unknown and needs to be estimated from the blurred image. This kernel estimation is computationally intensive and takes several minutes on a CPU, which makes it unsuitable for mobile devices. This paper presents the first hardware accelerator for kernel estimation for image deblurring applications. Our approach, using a multi-resolution iteratively reweighted least squares deconvolution engine with DFT-based matrix multiplication, a high-throughput image correlator, and a high-speed selective update-based gradient projection solver, achieves a 78x reduction in kernel estimation runtime, and a 56x reduction in total deblurring time for a 1920 × 1080 image, enabling quick feedback to the user. Configurability in kernel size and number of iterations gives up to ten times energy scalability, allowing the system to trade off runtime with image quality. The test chip, fabricated in TSMC 40-nm CMOS technology, consumes 105 mJ for kernel estimation running at 83 MHz and 0.9 V, making it suitable for integration into mobile devices.**

***Index Terms*— Deconvolution, energy scalability, gradient projection solver, hardware accelerator, image deblurring.**

I. Introduction

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

Manuscript received November 21, 2016; revised February 19, 2017; accepted March 1, 2017. Date of publication March 13, 2017; date of current version June 22, 2017. This paper was approved by Guest Editor Wim Dehaene. *(Corresponding author: Priyanka Raina.)*

The authors are with the Massachusetts Institute of Technology, Cambridge, MA 02139 USA (e-mail: praina@mit.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSSC.2017.2682842

Recent research has shown energy and performance benefits of hardware accelerators for several computational photography and computer vision applications, such as high dynamic range and low-light imaging [3], obstacle detection [4], and deep learning-based object recognition [5]. However, image deblurring has so far been implemented only in software running on CPU/GPU platforms, which do not support real-time processing and have high energy consumption, making these implementations unsuitable for mobile devices.

II. Background

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

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

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

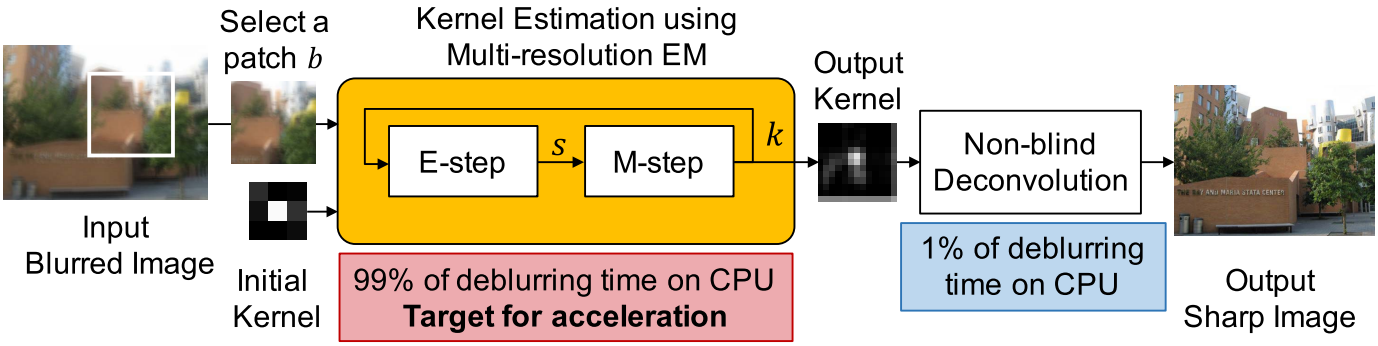


Fig. 1. EM-based blind image deblurring.

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

As shown in Fig. 1, the refined kernel obtained from the M-step is fed back into the E-step and the two steps are iterated multiple times until the desired number of iterations is performed. To avoid local optima, the EM iterations are carried out at successively higher resolutions of the patch size (32×32, 64×64, and 128×128), with the kernel at higher resolutions seeded from the results of the lower resolutions. We varied the size of the image patch used for kernel estimation from 256×256 to 32×32 in software, and observed that choosing a patch size smaller than 128×128 compromises the quality of the obtained kernel, and the resulting deblurred image has several ringing artifacts. The estimated kernel from a single patch is finally used for deconvolution of the full 1920×1080 blurred image. This deconvolution accounts for only 1% of the runtime, and is therefore done on a CPU using Gaussian or sparse prior-based approaches [6].

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

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

1) a *multi-resolution IRLS deconvolution engine* with DFT-based matrix multiplication for image refinement (Section III);
2) a *high-throughput image correlator* for computing the coefficient matrices of the kernel QP (Section IV);
3) a *high-speed selective update-based gradient projection solver* for solving the kernel QP (Section V).

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

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

III. Multi-Resolution IRLS Deconvolution Engine With DFT-Based Matrix Multiplication

Image refinement entails solving a minimization problem to get the sharp image s, but since the cost function to be minimized, $||b_\gamma - T_k s_\gamma||^2 + R(s_\gamma)$ is not quadratic because of the regularization term, and we use the IRLS approach [6]. This involves repeating the following two steps until convergence. In each iteration i

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

The linear system in each iteration is solved using a conjugate gradient (CG) solver. Here, the regularization weights penalize the gradients in the smooth regions more than in the edge regions, enforcing a sparse prior on the sharp image gradients. For a detailed derivation, see [1].

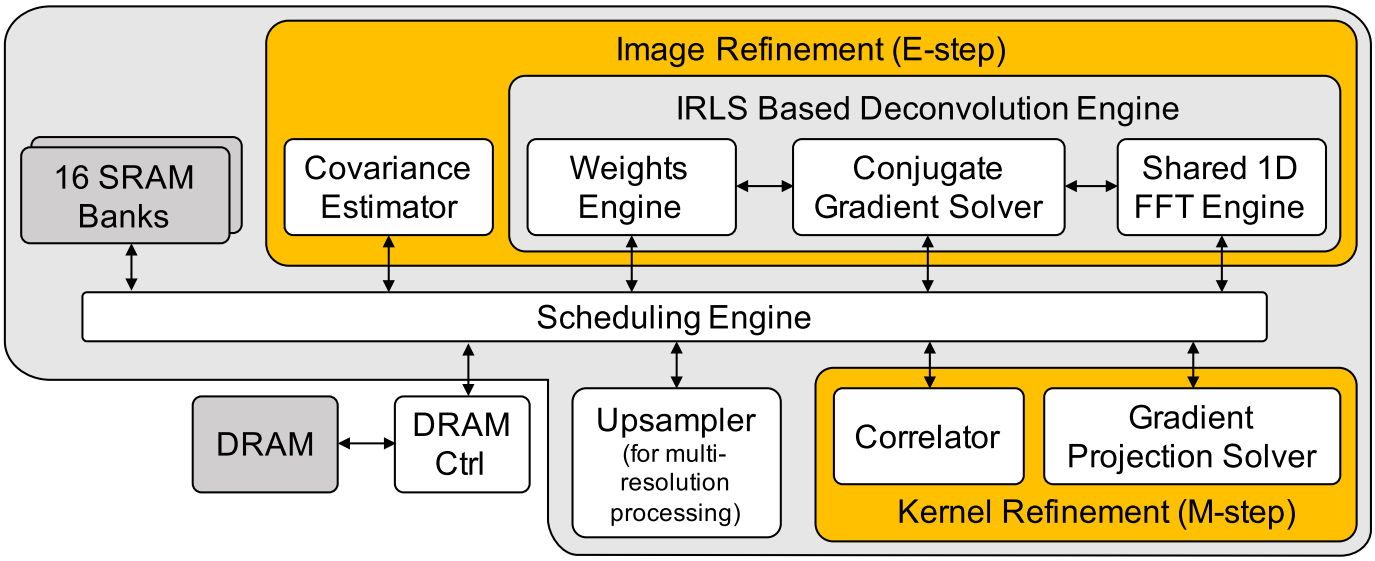


Fig. 2. Accelerator system architecture.

A. IRLS Optimizations

We propose the following optimizations to the IRLS-based deconvolution engine compared with [6] to reduce the number of floating point (FP) operations and area.

1) DFT-Based Matrix Multiplication: We observe that the most computationally intensive step in the IRLS deconvolution is the multiplication with $n^2 \times n^2$ matrix $T_k^T T_k$ in each CG iteration, where $n \times n$ is the size of the sharp image. This multiplication is equivalent to convolution with the auto-correlation of kernel k. Since k is large, we implement this convolution in the frequency domain as a multiplication with the squared magnitude of the 2-D DFT of k. This reduces the computational complexity of the matrix multiplication and gives an improvement in the number of floating point operations by 8.8 times for 128×128, 10.2 times for 64×64, and 12.2 times for 32×32 images and a 13×13 kernel compared with a spatial convolution-based approach.

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

IV. High-Throughput Image Correlator

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

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

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

A. Correlator Optimizations

We propose the following optimizations to the correlator to reduce the correlation execution time and buffering requirements compared with the software implementation of the same algorithm proposed in [1].

1) Diagonal Computation Reuse: We observe that along the diagonals of $\bar{A}_{k\gamma}$, the relative shift $(\Delta x, \Delta y) = (x_1 - x_2, y_1 - y_2)$ remains constant between the two shifted images $S_\gamma(x - x_1, y - y_1)$ and $S_\gamma(x - x_2, y - y_2)$ with absolute shifts (x_1, x_2) and (y_1, y_2), as shown in Fig. 4. Therefore, the same corresponding elements are multiplied, but the summation of the product is performed over different pixels. We propose doing this multiplication $S_\gamma(x, y) S_\gamma(x - \Delta x, y - \Delta y)$ only once for each relative shift $(\Delta x, \Delta y)$, and then using integral image [9] of the product to compute each entry along the diagonals. For example, for all the entries along the fourth diagonal shown in yellow in Fig. 4, the relative shift is $(1, 0)$. We compute the product $S_\gamma(x, y) S_\gamma(x - 1, y - 0)$

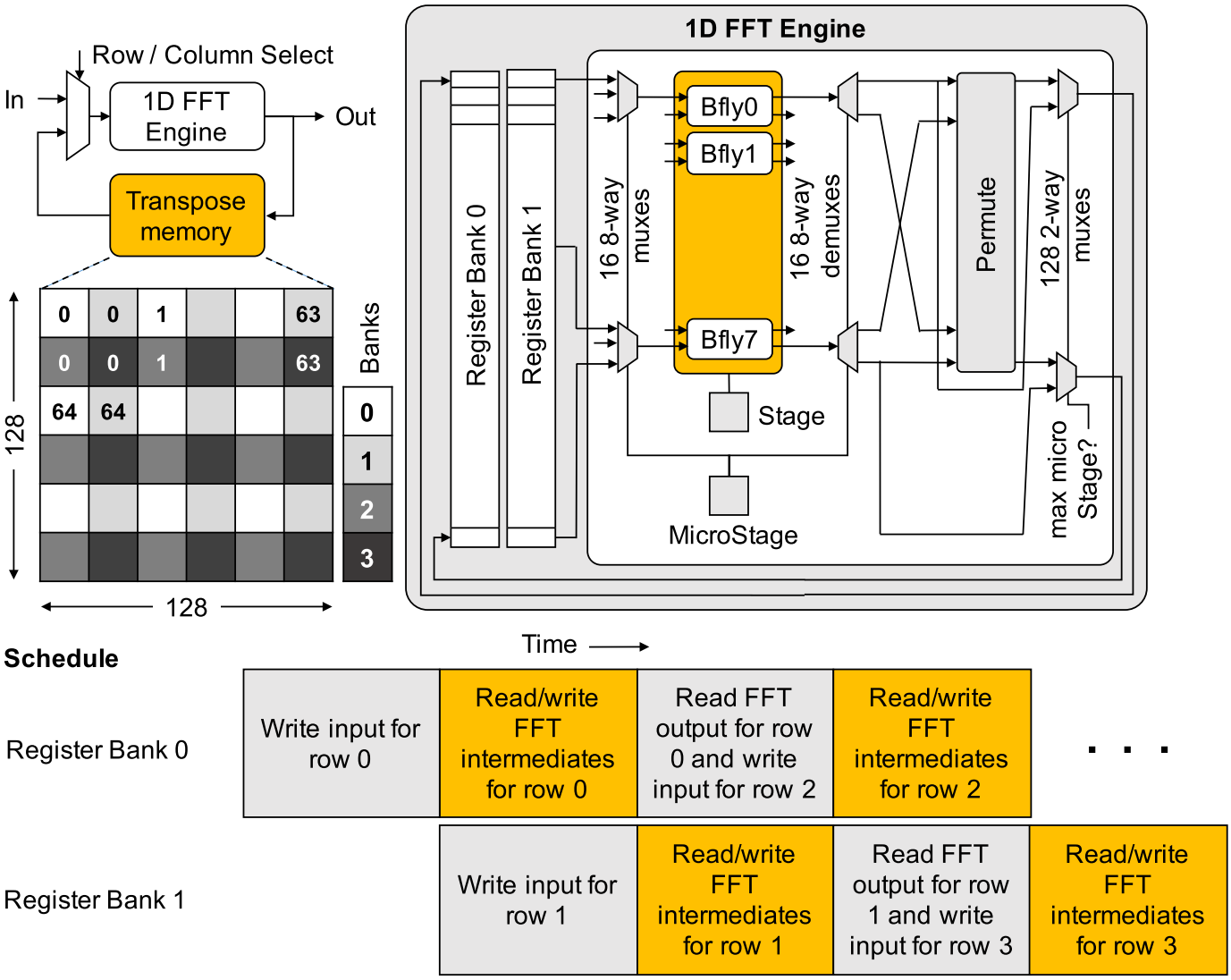


Fig. 3. Left: 2-D DFT with shared 1-D FFT and transpose memory. Right: reconfigurable 1-D FFT architecture. Bottom: schedule for register bank access for I/O and FFT computation.

and its integral image, and then, we accumulate the product over different rectangles shaded in gray to get the values for $\bar{A}_{k\gamma}(3,0)$, $\bar{A}_{k\gamma}(4,1)$, $\bar{A}_{k\gamma}(5,2)$, and so on in constant time with three operations for each of the elements of the integral image. This technique makes the computation of matrix $\bar{A}_{k\gamma}$, $O(m^2)$ rather than $O(m^4)$, providing a speedup of 42 times for computing the correlation for a typical 13×13 kernel.

2) Six-Parallel Correlator Architecture With Image Tiling: Diagonal computation reuse alone cannot meet the throughput requirements of the correlator. We further increase throughput by designing a highly parallel correlator architecture, which consists of an array of six processing elements (PEs) as shown in Fig. 5, each of which computes the correlation matrix elements for all absolute shifts that correspond to the same relative shift using diagonal computation reuse. The mapping of correlation computation to the PEs is shown in Fig. 6 for a toy kernel of size 3×3. The computation happens in two passes over the image, where in the first pass, the relative shift is positive, and in the second pass, the relative shift is negative.

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

3) Merged Correlation PE Architecture: Figs. 7 and 8 show the architecture of a correlation PE. Each PE is initialized with the input image size $(n \times n)$, the kernel size $(m \times m)$, and the 2-D relative shift $(\Delta x, \Delta y)$, and it calculates the

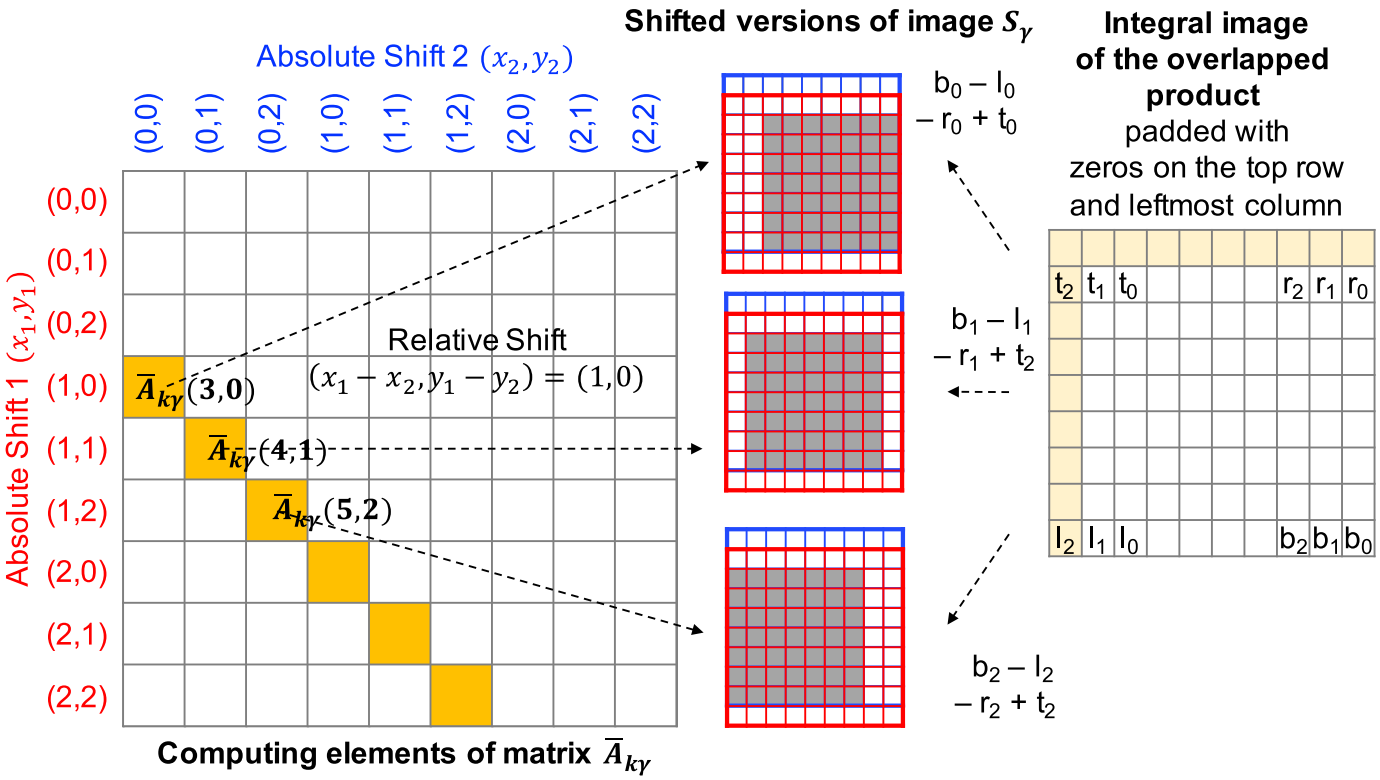


Fig. 4. Computation of correlation matrix $\bar{A}_{k\gamma}$ with diagonal computation reuse for a toy kernel of size 3×3. Along each diagonal, the same corresponding elements are multiplied, but are summed over a different area using the integral image of the overlapped product.

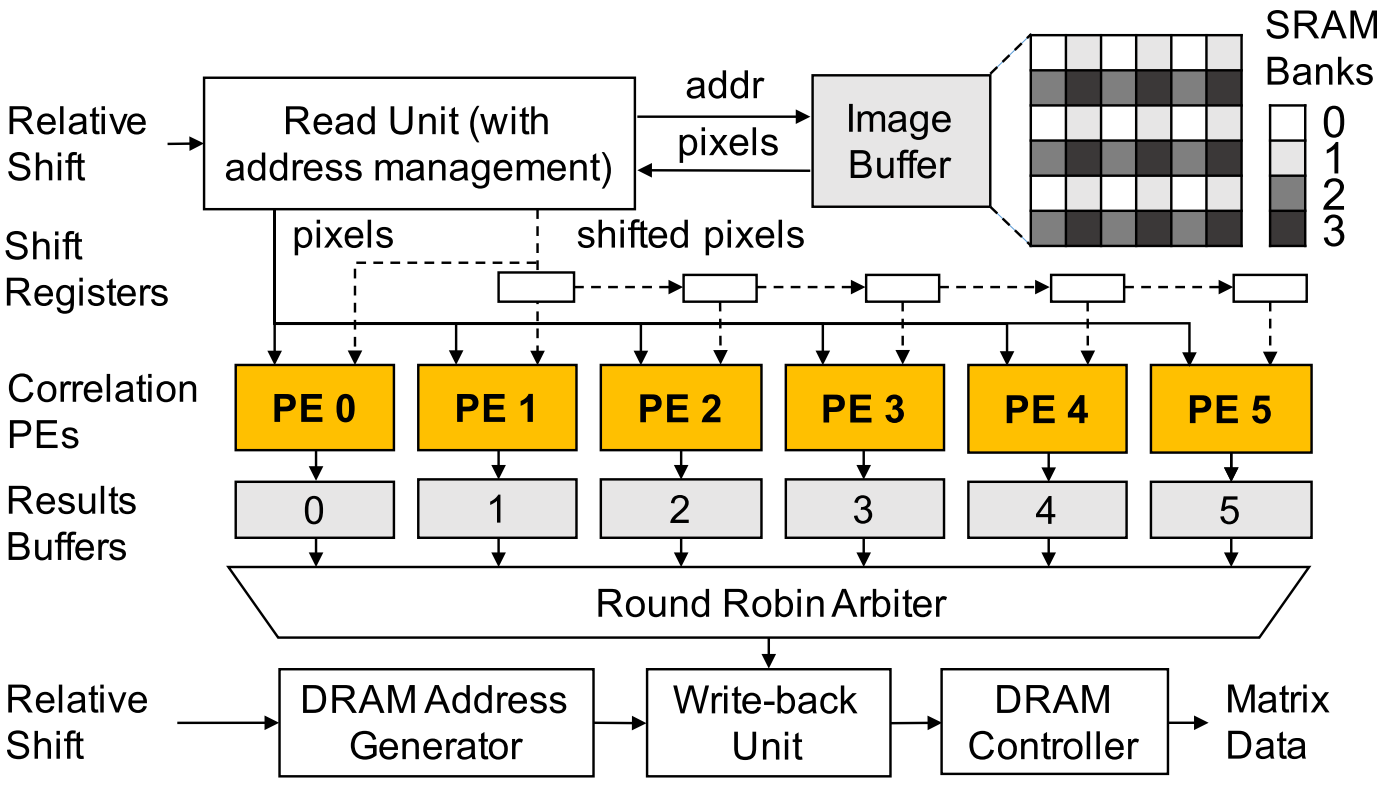


Fig. 5. Highly parallel correlator architecture with six PEs enabled by image tiling in the image buffer gives six times improvement in throughput.

correlation matrix elements for that relative shift, as shown in Fig. 7. One approach for computing correlation by the PEs would be to first multiply the corresponding pixels and compute the integral image by accumulating across columns followed by rows, and then to access four elements at a time from the integral image and use them to compute each

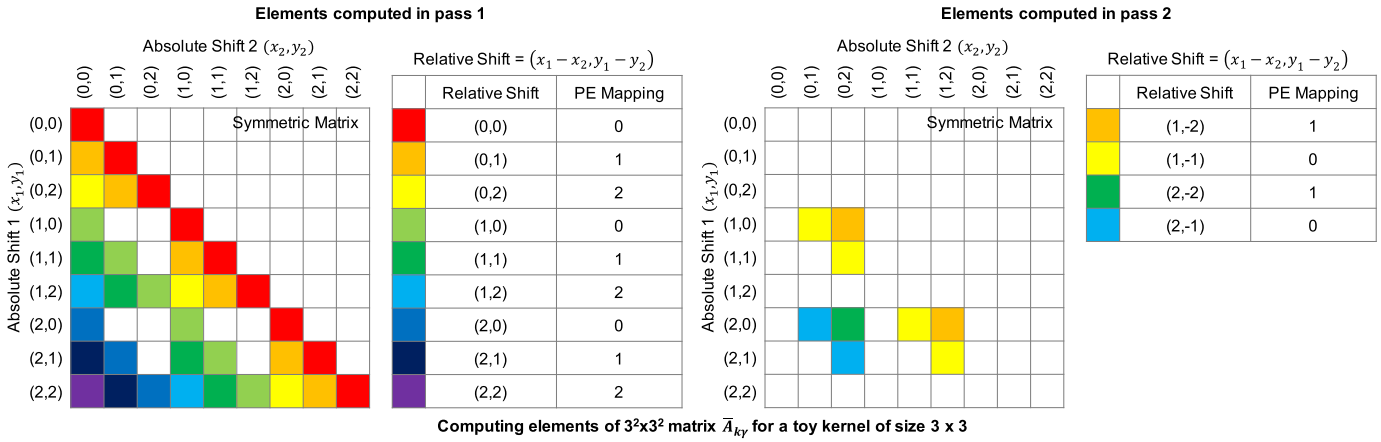


Fig. 6. Mapping of computation of correlation matrix elements to different PEs based on the relative shifts for a toy kernel of size 3×3, which leads to a $3^2 \times 3^2$ correlation matrix. Only the lower triangular part is computed, since the matrix is symmetric.

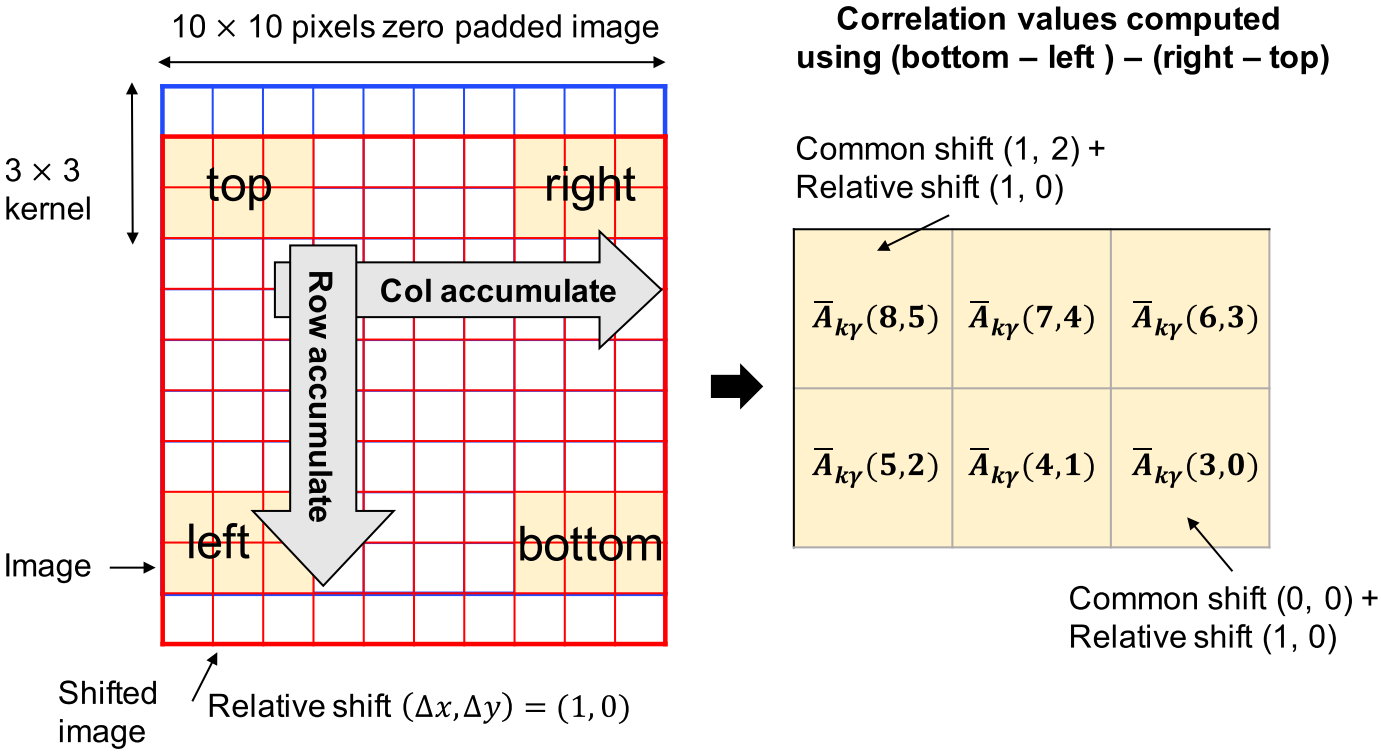


Fig. 7. Toy example to illustrate the integral image computation followed by matrix element calculation happening inside each correlation PE.

matrix element, as shown in Fig. 6. This approach is used several computer vision algorithms that require integral image computation, such as real-time object detection [9], robust feature extraction [10], and surface normal calculation [4]. This approach requires at least storing the entire $n \times n$ integral image, which in the worst case is 128×128. However, in our case, we only need to access a subset of elements from the integral image, and the locations of those elements are known *a priori*, so we can do better. To reduce the buffering requirements, we propose a different architecture that merges the integral image computation step with the subsequent step of calculating the matrix elements. Our approach reduces the buffering requirement to $m \times m + 3 * m \times 1$, where $m = 32$ in the worst case to accommodate the largest kernel, achieving a 14.6x reduction compared with the naive approach. Fig. 8 highlights the buffers in yellow, and the following paragraphs give the processing details.

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

The column subtractor results are accumulated across rows by the row accumulator and the results are written to accumulated row buffer (a). Accumulated row buffer (a) is copied

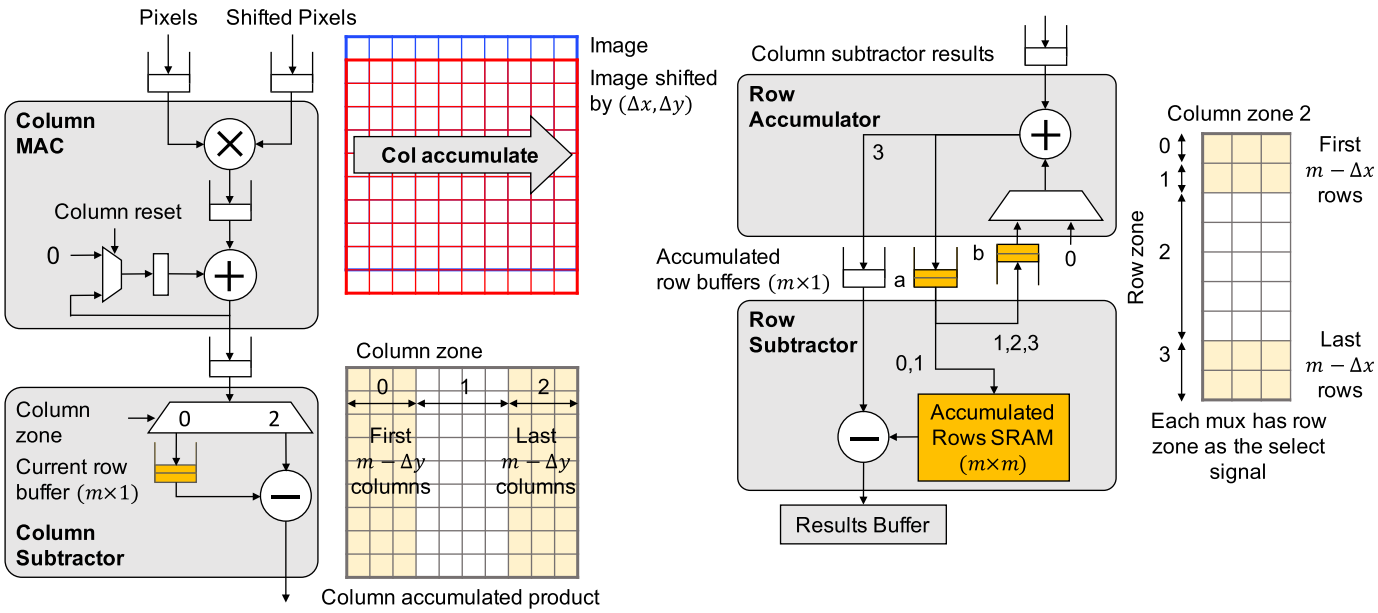


Fig. 8. Left: inside the correlation PE, the pixels and shifted pixels are multiplied together and accumulated along each row by the column MAC. The first $m - \Delta y$ accumulated columns are subtracted from the last $m - \Delta y$ accumulated columns and sent to the row accumulator. Right: accumulation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

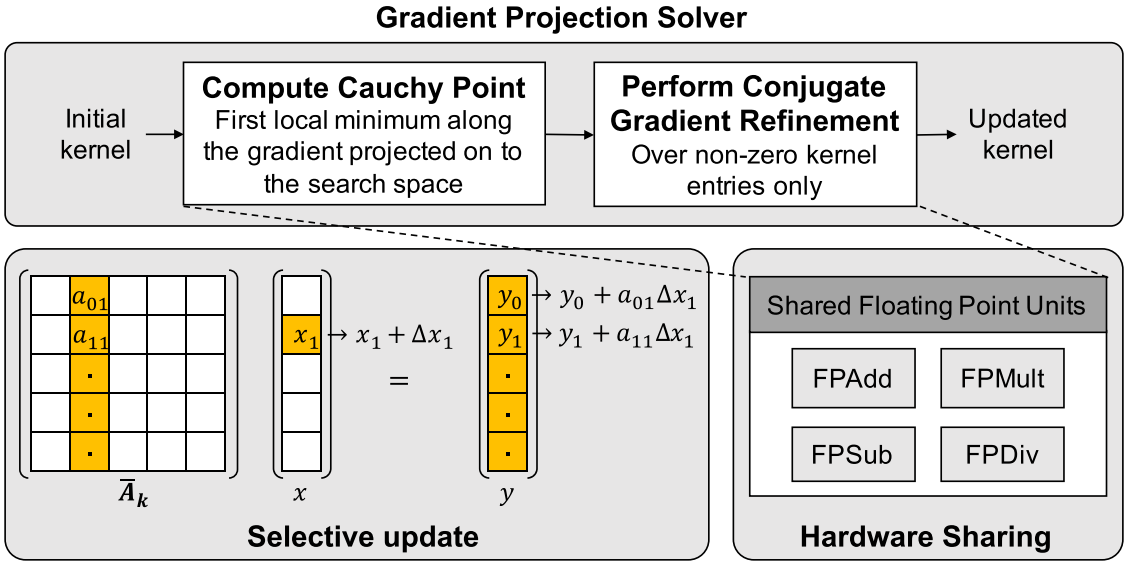


Fig. 9. Gradient projection solver with selective update and hardware sharing.

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

V. Selective Update-Based Gradient Projection Solver

The gradient projection solver [11] takes the kernel from the previous EM iteration, and the matrix $\bar{A}_k$ and the vector $\bar{b}_k$ computed by the correlator as inputs. It computes the refined kernel by minimizing the constrained QP, $J(k) = (1/2)k^T \bar{A}_k k + \bar{b}_k^T k$, such that $k \geq 0$. The constraints ensure that the kernel is a blur kernel. The solver refines the kernel by iteratively executing two sequential steps, as shown in Fig. 9:

1) Search along the steepest descent direction, that is the direction $-g$, where $g = \bar{A}_k k + \bar{b}_k$, from the current point k. Whenever a constraint is encountered, bend the

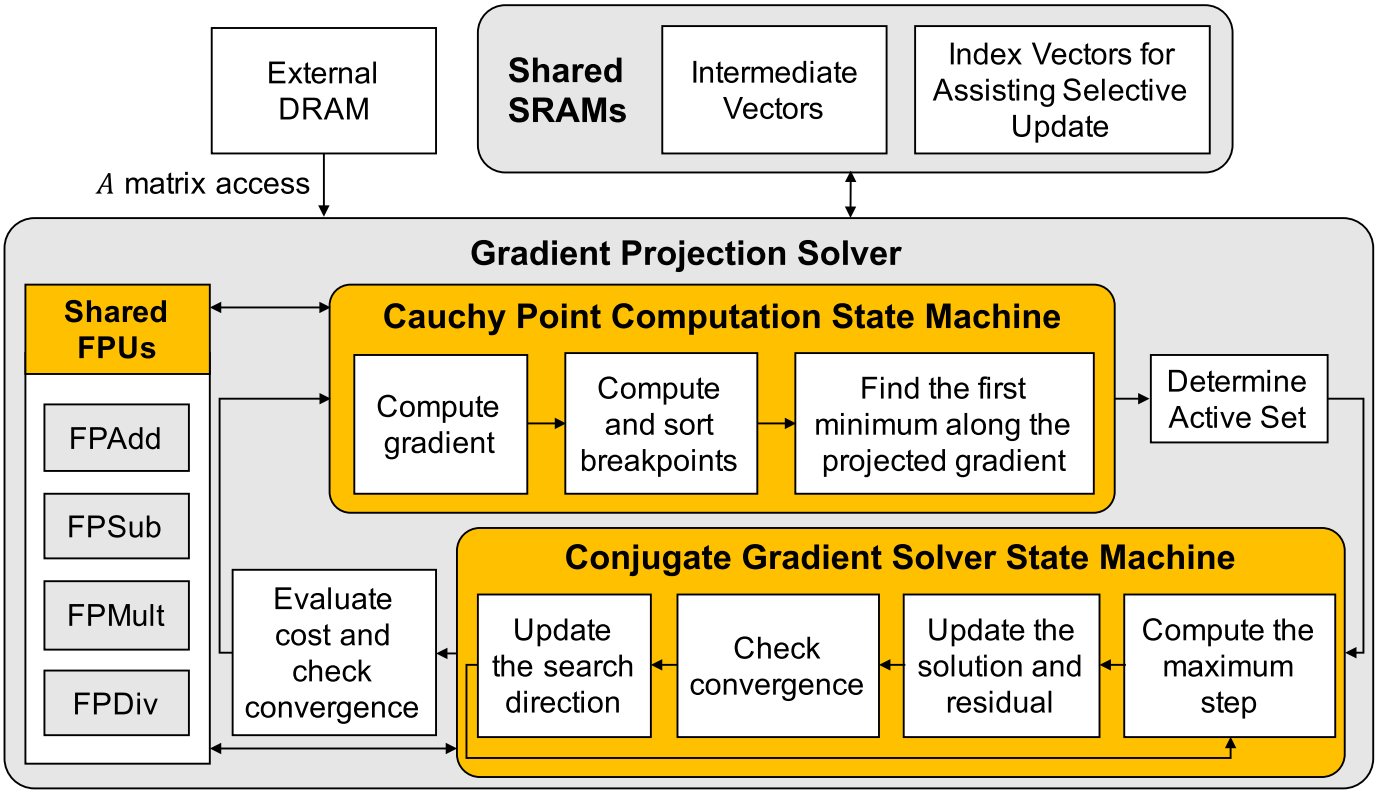


Fig. 10. Cauchy point computation and CG solver state machines.

Time →

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

Fig. 11. Scheduling of CG refinement steps over shared floating point arithmetic units. Yellow blocks denote the pipeline bubbles. Three dots denote that the processing happens over all vector indices.

search direction so that it stays feasible, and locate the first local minimizer of J along this piecewise linear path. The minimizer is called the Cauchy point k_c.

2) Explore the face of the feasible box on which the Cauchy point lies by solving a sub-problem, in which the value of k at the indices i at which the constraints are active (which means equal to zero in our case) is held fixed at k_i^c, using a CG solver, for faster convergence.

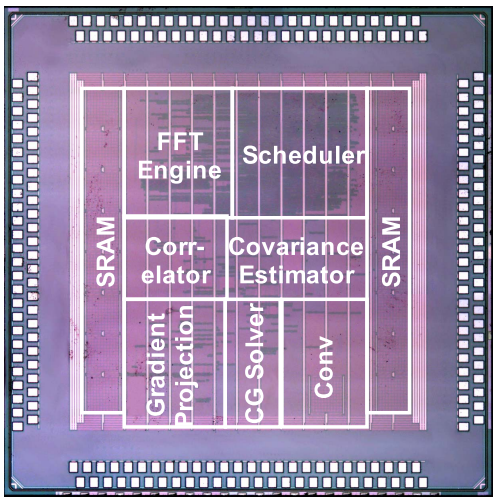
At the end of each iteration, the quadratic cost function is evaluated and compared with its value from the previous iteration; if the difference is small, the kernel is taken as final.

A. Gradient Projection Solver Optimizations

We perform the following optimizations to the gradient projection solver to ensure fast convergence and reduce energy compared with [11].

1) Selective Update: Matrix-vector multiplications with $\bar{A}_k$ dominate the computational time in the algorithm, as $\bar{A}_k$ can be very large (49×49 to 841×841 depending on the kernel size). However, we observe that during each iteration, the vector to be multiplied with $\bar{A}_k$ updates only at a few indices with respect to its value from the previous iteration. Our selective update algorithm (Fig. 9) converts matrix-vector multiplication into a few elementwise multiplications and additions on two vectors. For example, as shown in Fig. 9, if the vector x updates only at index 1 with respect to its value in the previous iteration, that is, it becomes $x_1 + \Delta x_1$ instead of x_1 in the previous iteration, we read only the second column of the $\bar{A}_k$ matrix instead of all columns to compute the new result vector y. This reduces the number of solver multiplies, adds, and DRAM accesses by 11 times on an average.

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



Chip Features

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

Fig. 12. Die photograph and chip features.

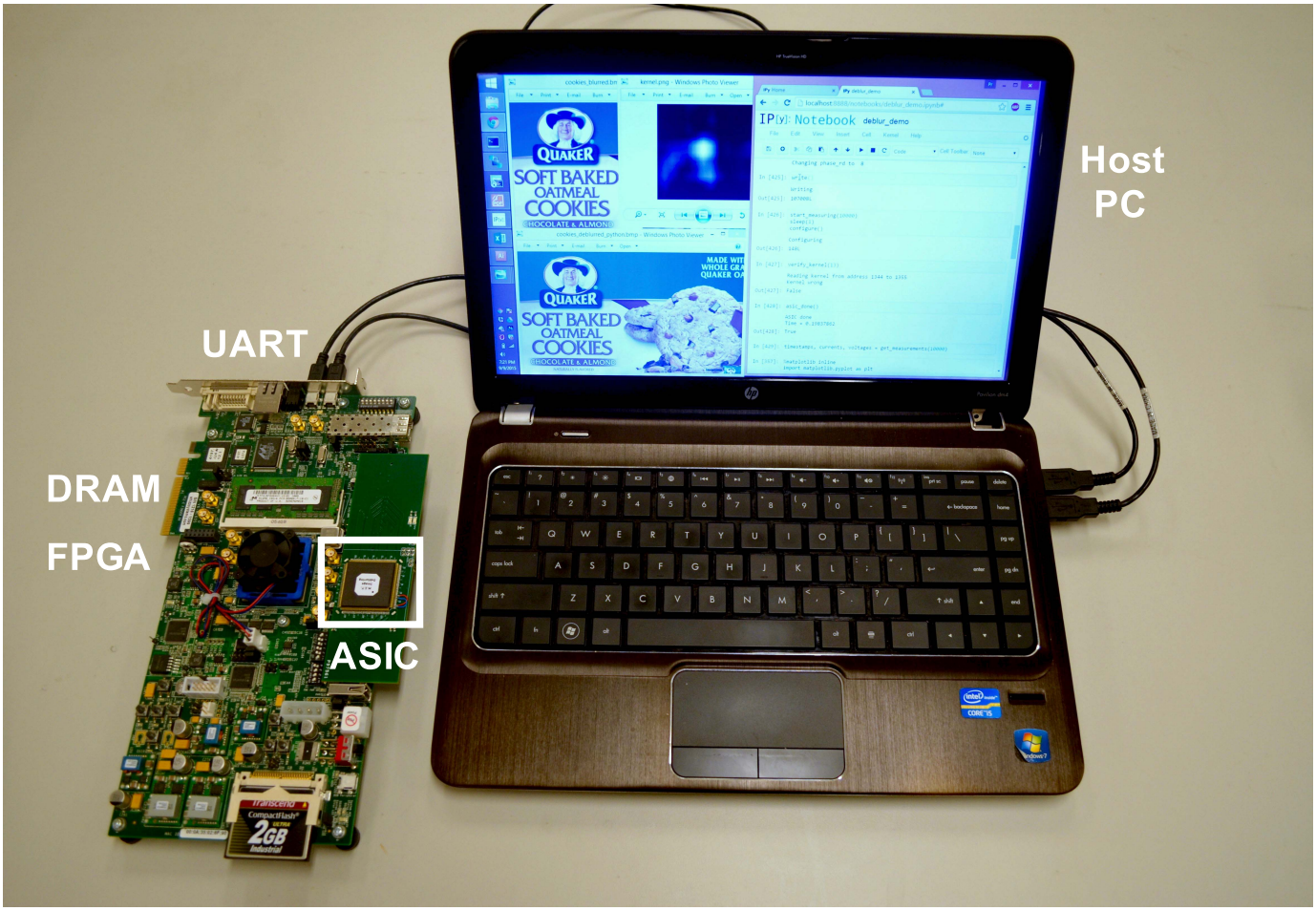


Fig. 13. Test setup for image deblurring accelerator. The chip is connected to Virtex-6 FPGA on Xilinx ML605 development board. The estimated kernel and the deblurred Full HD image (1920 × 1080) are displayed on the host PC.

B. Gradient Projection Solver Architecture

The gradient projection solver consists of schedulers for Cauchy point computation and CG solver as shown in Fig. 10, which are essentially hard-coded state machines that execute the steps of the algorithm on the shared floating point units. The steps that do not have a data dependence between them are executed in parallel, and the ones that do are executed sequentially. In this way, the scheduler extracts the maximum amount of parallelism given the I/O bandwidth constraints. Fig. 11 shows an example of the mapping of computation during three different sequential steps of CG refinement onto the shared arithmetic units. The utilization of the arithmetic units depends on the maximum available parallelism.

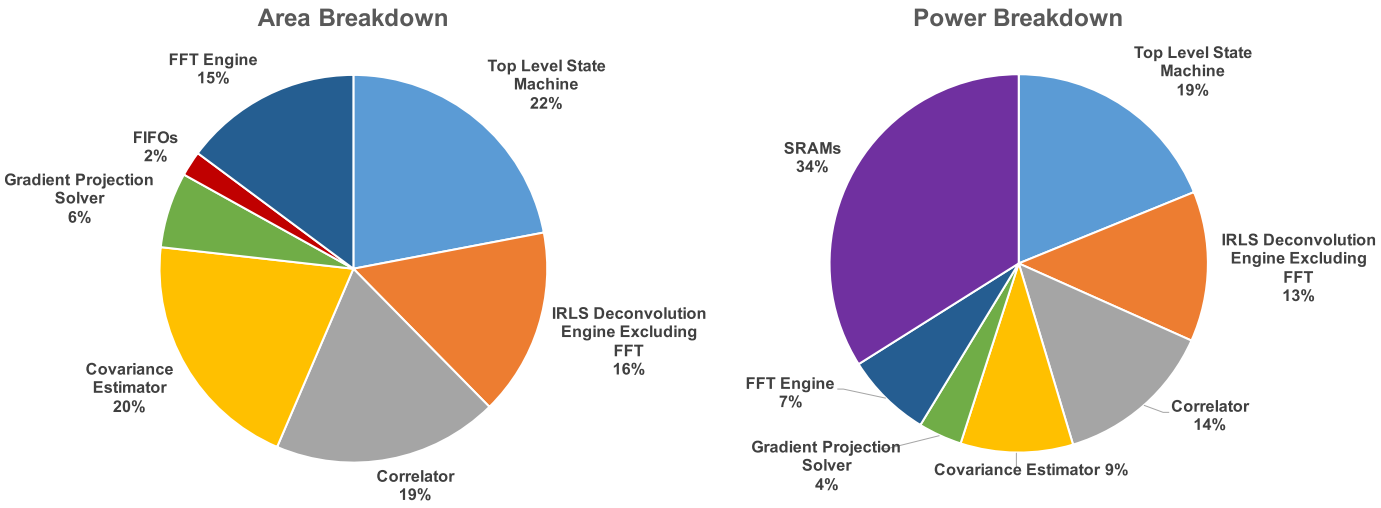


Fig. 14. Left: logic utilization for each processing block (total 2728 kgates). Right: total power breakdown for logic blocks and memory (a total of 59.5 mW).

Fig. 15. Test 1920 × 1080 blurred image, output kernel of size 13 × 13, and deblurred image.

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

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

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

1) We compute the projected gradient p, where $p_i = -g_i$ when $t_i > t^j$ and 0 otherwise.
2) Starting at the current k, the objective function till the next breakpoint can be written as a function of the step size Δt in the direction of the projected gradient p as $J(k + \Delta t p) = f_0 + f_1(\Delta t) + (1/2) f_2(\Delta t)^2$, where

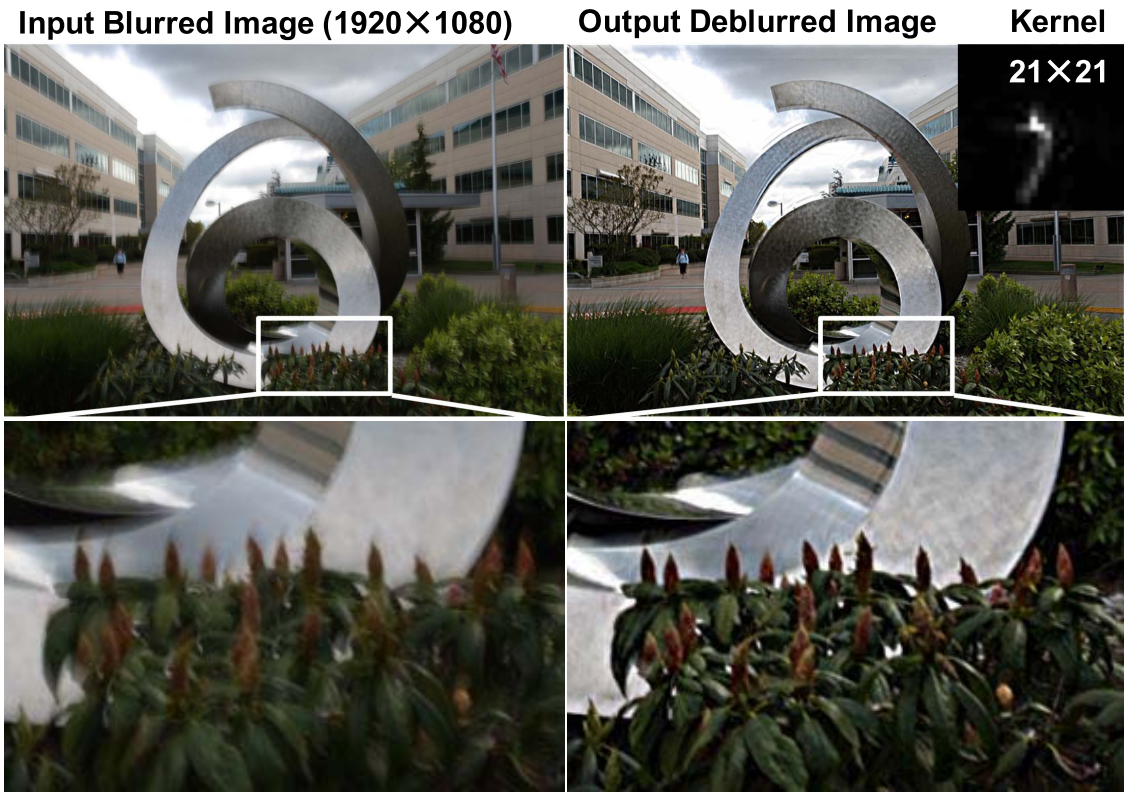


Fig. 16. Test 1920 × 1080 blurred image, output kernel of size 21 × 21, and deblurred image.

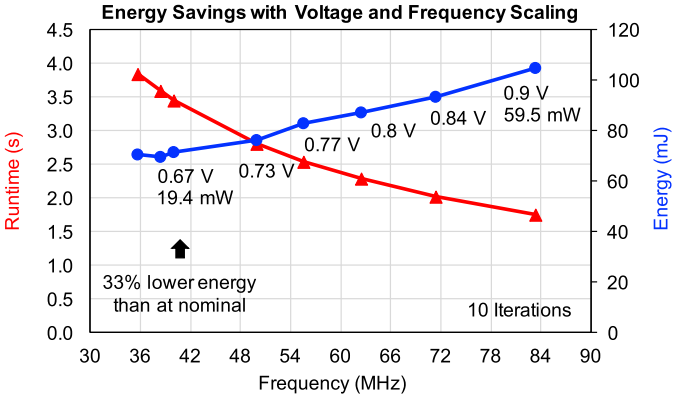


Fig. 17. By employing voltage and frequency scaling, the system obtains the minimum energy point at (0.67 V and 38 MHz), where the energy consumption is 33% lower than at nominal, and can be used for batch processing.

$f_1 = \bar{b}_k^T p + k^T \bar{A}_k p$ and $f_2 = p^T \bar{A}_k p$. We minimize this objective function with respect to the step size Δt, to get the optimal $\Delta t^* = -f_1/f_2$. The most computationally expensive operation here is matrix multiplication of $\bar{A}_k$ with p for computing f_1 and f_2. However, we observe that the projected gradient p updates only at a few indices at a time, depending on the number of breakpoints t_i that are less than or equal to t^j in each interval, and we use the selective update technique described earlier to reduce the number of computations while calculating $\bar{A}_k p$.

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

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

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

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

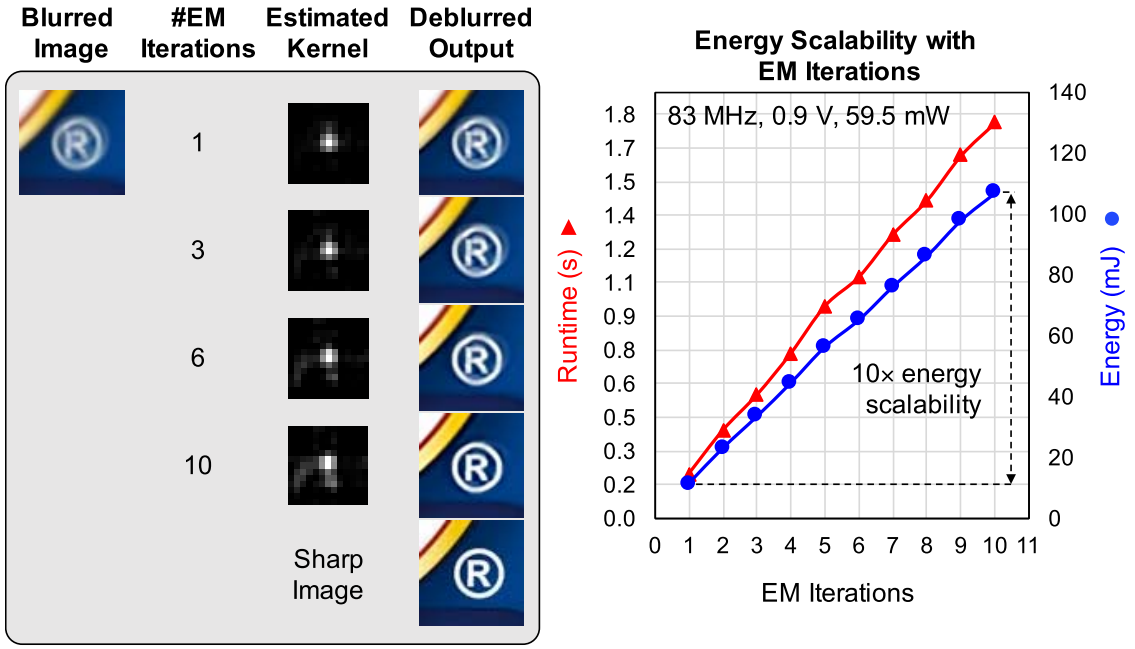


Fig. 18. Number of EM iterations can be tuned to trade off image quality with runtime giving ten times energy scalability.

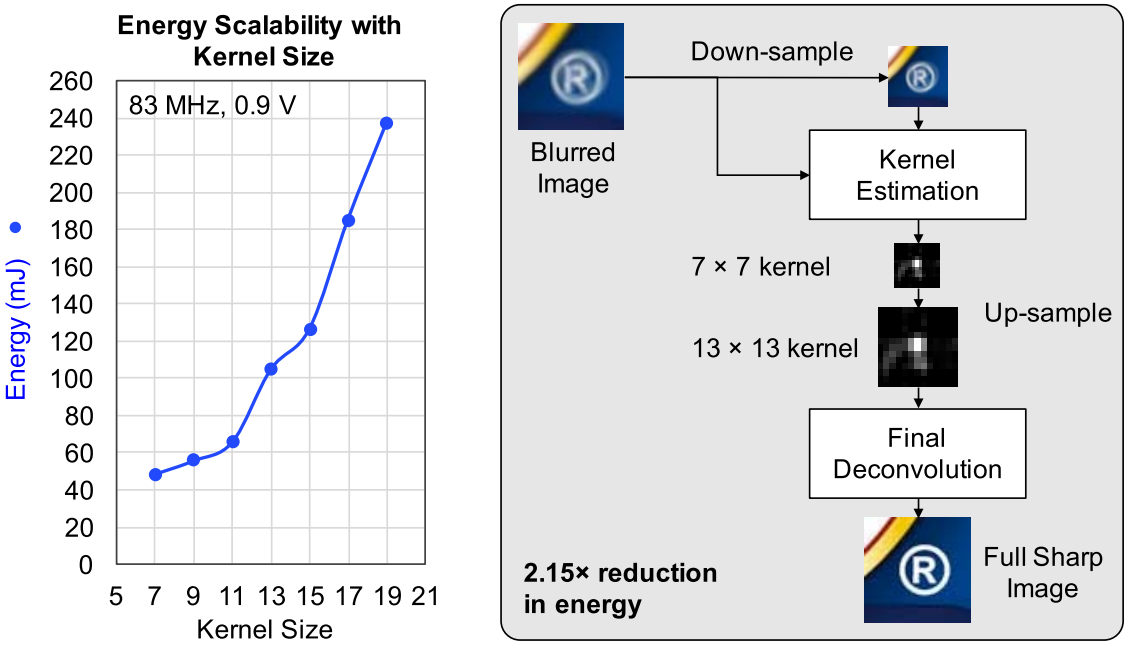


Fig. 19. Kernel size can also be tuned to achieve energy scalability.

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

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

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

   c) Compute the new residual $r_{cg} = r_{cg} - \alpha^* q_{cg}$ and check for convergence by comparing the norm of the new residual $\rho = r_{cg}^T r_{cg}$ with the norm of the

TABLE I
COMPARISON WITH THE STATE-OF-THE-ART ALGORITHMS ON DIFFERENT PLATFORMS

	Size			Time (s)		Energy (J)
Algorithm and Platform	Kernel	Patch	Image	**Kernel Estimation**	Full Deblurring	**Kernel Estimation**
This Work (based on [1] with Accelerator + CPU)	13×13	128× 128	1920× 1080	**1.70**	2.45	**0.105**
[1] on Intel Core i5	13×13	128× 128	1920× 1080	**134.00**	134.75	**467.000**
[1] on Samsung Exynos 5422 Cortex-A15	13×13	128× 128	1920× 1080	**816.00**	-	**2284.800**
[2] on NVIDIA Tesla C2050 GPU	15×15	-	441× 611	**169.70**	170.50	-

last residual ρ_{last}. If this difference is smaller than the tolerance, exit the loop.

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

4) Convergence Checking: After the completion of CG refinement, we have the updated solution k at all indices, but the updated gradient g at only the non-active indices. To compute the updated gradient at the active indices, we again use the selective update-based matrix multiplication algorithm, where we read in only a sub-matrix of $\bar{A}_k$ at the active rows and non-active columns and multiply it with the change in the solution at the non-active indices, to get the change in the gradient at the active indices. Since in the average case, only 50% of the indices are non-active, this results in DRAM accesses and FP operations for only 25% of the $\bar{A}_k$ matrix. Once we have the updated gradient, we also compute the new cost without re-reading the $\bar{A}_k$ matrix, using $J = (1/2)k^T(g + \bar{b}_k)$. We compare this cost with its value from the previous iteration, and if the difference is smaller than some tolerance, the solution k is taken as final, else the two steps are repeated.

VI. RESULTS

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

A. Runtime and Energy Reduction

The accelerator achieves runtime reduction by 78 times for kernel estimation with respect to our realization of [1] on an Intel i5 CPU for a 13×13 kernel, and by 56 times for complete deblurring of a 1920×1080 image, with final deconvolution performed on the CPU (Table I). Compared with our realization of [1] on a mobile processor (Cortex-A15), it achieves a 480x reduction in kernel estimation time. Also, compared with a related work [2] which implements deblurring on a GPU, this paper achieves a substantial reduction in kernel estimation time. Our kernel estimation time is independent of image size, while deconvolution time scales linearly.

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

B. Energy Scalability

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

VII. CONCLUSION

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

1) a multi-resolution IRLS-based deconvolution engine with DFT-based matrix multiplication which achieves at least 8.8 times reduction in the number of floating point operations;
2) a highly parallel image correlator with diagonal computation reuse and image tiling which achieves a speedup of two orders of magnitude over the baseline;
3) a selective update-based gradient projection solver which achieves 11 times increase in speed and 56% reduction in area compared with the baseline.

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

ACKNOWLEDGMENT

The authors would like to thank Foxconn Technology Group for sponsorship, TSMC University Shuttle Program for chip fabrication, and Prof. F. Durand and Prof. W. T. Freeman for valuable feedback.

REFERENCES

[1] A. Levin, Y. Weiss, F. Durand, and W. T. Freeman, "Efficient marginal likelihood optimization in blind deconvolution," in *Proc. IEEE CVPR*, Jun. 2011, pp. 2657–2664.

[2] M. Hirsch, C. J. Schuler, S. Harmeling, and B. Schölkopf, "Fast removal of non-uniform camera shake," in *Proc. IEEE Int. Conf. Comput. Vis.*, Nov. 2011, pp. 463–470.

[3] R. Rithe, P. Raina, N. Ickes, S. V. Tenneti, and A. P. Chandrakasan, "Reconfigurable processor for energy-efficient computational photography," *IEEE J. Solid-State Circuits*, vol. 48, no. 11, pp. 2908–2919, Nov. 2013.

[4] D. Jeon, N. Ickes, P. Raina, H. C. Wang, D. Rus, and A. P. Chandrakasan, "A 0.6 V 8 mW 3D vision processor for a navigation device for the visually impaired," in *IEEE Int. Solid-State Circuits Conf. (ISSCC) Dig. Tech. Papers*, Jan. 2016, pp. 416–417.

[5] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[6] A. Levin, R. Fergus, F. Durand, and W. T. Freeman, "Deconvolution using natural image priors," Massachusetts Inst. Technol. Comput. Sci. Artif. Intell. Lab., Cambridge, MA, USA, Tech. Rep. 3, 2007.

[7] P. Raina, M. Tikekar, and A. P. Chandrakasan, "An energy-scalable accelerator for blind image deblurring," in *Proc. IEEE Eur. Solid-State Circuits Conf. (ESSCIRC)*, Sep. 2016, pp. 113–116.

[8] C.-T. Huang, M. Tikekar, C. Juvekar, V. Sze, and A. Chandrakasan, "A 249 Mpixel/s HEVC video-decoder chip for quad full HD applications," in *IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 2013, pp. 162–163.

[9] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 1. Dec. 2001, pp. I-511–I-518.

[10] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (SURF)," *Comput. Vis. Image Understand.*, vol. 110, no. 3, pp. 346–359, 2008.

[11] P. H. Calamai and J. J. Moré, "Projected gradient methods for linearly constrained problems," *Math. Program.*, vol. 39, no. 1, pp. 93–116, 1987.

Priyanka Raina (S'17) received the B.Tech. degree in electrical engineering from IIT Delhi, New Delhi, India, in 2011, and the S.M. degree in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 2013, where she is currently pursuing the Ph.D. degree with Microsystems Technology Laboratories.

In 2013, she was with the Circuits Research Laboratory, Intel Labs, Hillsboro, OR, USA, where she was involved in designing a hardware accelerator for real-time video enhancement using multi-frame super-resolution. Her current research interests include design of energy-efficient circuits for computational photography, computer vision, and machine learning applications.

Ms. Raina received the Institute Silver Medal at IIT Delhi in 2011 and a Gold Medal at the Indian National Chemistry Olympiad in 2007.

Mehul Tikekar (S'10–M'15) received the B.Tech. degree in electrical engineering from IIT Bombay, Mumbai, India, in 2010, and the S.M. degree in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 2012, where he is currently pursuing the Ph.D. degree.

His current research interests include low-power system design and hardware-optimized video coding.

Mr. Tikekar was a recipient of the MIT Presidential Fellowship in 2011.

Anantha P. Chandrakasan (F'04) received the B.S., M.S., and Ph.D. degrees from the University of California at Berkeley, Berkeley, CA, USA, in 1989, 1990, and 1994, respectively, all in electrical engineering and computer sciences.

Since 1994, he has been with the Massachusetts Institute of Technology (MIT), Cambridge, MA, USA, where he is currently the Vannevar Bush Professor of Electrical Engineering and Computer Science. He was the Director of Microsystems Technology Laboratories, MIT, from 2006 to 2011. Since 2011, he has been the Head of the Electrical Engineering and Computer Science Department, MIT. He has co-authored the books *Low Power Digital CMOS Design* (Kluwer Academic, 1995), *Digital Integrated Circuits*—Second Edition (Pearson Prentice-Hall, 2003), and *Sub-Threshold Design for Ultra-Low Power Systems* (Springer, 2006). His current research interests include ultra-low-power circuit and system design, energy harvesting, energy efficient RF circuits, and hardware security.

Dr. Chandrakasan was a co-recipient of several awards, including the 2007 ISSCC Beatrice Winner Award for the Editorial Excellence and the ISSCC Jack Kilby Award for Outstanding Student Paper (2007, 2008, and 2009). He received the 2009 Semiconductor Industry Association University Researcher Award and the 2013 IEEE Donald O. Pederson Award in Solid-State Circuits. In 2015, he was elected to the National Academy of Engineering. He has served in various roles for the IEEE International Solid-State Circuits Conference (ISSCC), including the Program Chair, the Signal Processing Sub-Committee Chair, and the Technology Directions Sub-Committee Chair. He has been the Conference Chair of ISSCC since 2010.