Fast 2D Convolutions and Cross-Correlations Using Scalable Architectures

Cesar Carranza, *Member, IEEE*, Daniel Llamocca, *Senior Member, IEEE*, and Marios Pattichis, *Senior Member, IEEE*

***Abstract*— The manuscript describes fast and scalable architectures and associated algorithms for computing convolutions and cross-correlations. The basic idea is to map 2D convolutions and cross-correlations to a collection of 1D convolutions and cross-correlations in the transform domain. This is accomplished through the use of the discrete periodic radon transform for general kernels and the use of singular value decomposition -LU decompositions for low-rank kernels. The approach uses scalable architectures that can be fitted into modern FPGA and Zynq-SOC devices. Based on different types of available resources, for $P \times P$ blocks, 2D convolutions and cross-correlations can be computed in just $O(P)$ clock cycles up to $O(P^2)$ clock cycles. Thus, there is a trade-off between performance and required numbers and types of resources. We provide implementations of the proposed architectures using modern programmable devices (Virtex-7 and Zynq-SOC). Based on the amounts and types of required resources, we show that the proposed approaches significantly outperform current methods.**

***Index Terms*— Linear convolution, circular convolution, cross-correlation, discrete periodic radon transform, parallel architecture, scalable architecture, FPGA, SOC.**

I. INTRODUCTION

CONVOLUTIONS and cross-correlations have wide applications in image and video processing and imaging [1], [2]. The development of effective architectures and algorithms for computing convolutions and cross-correlations can be used in several applications (e.g., feature extraction [3], template matching [4], pattern recognition [5], edge detection, filtering, deconvolution, segmentation, and denoising [1]).

To support implementations in modern devices (e.g., FPGAs, PSOCs), we are also interested in scalable architectures. The basic idea is to make efficient use of hardware resources to deliver the best possible performance. For scalability, we investigate implementations that can be fitted within available resources.

A standard approach for developing efficient architectures for 2D convolutions and cross-correlations would be to build the systems based on 2D FFTs. As is well-known (e.g., see [6], [7]), for sufficiently large kernels, the use of 2D FFTs will give better results than direct approaches. Unfortunately, the direct implementation of 2D FFTs in hardware requires the use of complex-valued arithmetic units. As a result, the hardware scalability of using 2D FFTs is fundamentally limited by the number of 1D FFT processors that can be fitted in any given hardware device. We refer to [8]–[10] for details of the latest implementation of this approach. As shown in [8], performance can be improved by including up to 4 1D FFT processors. Beyond 4 1D FFT processors, performance stalls or even degrades due to I/O issues [8].

Modern FPGA and SOC devices are equipped with DSPs that can better facilitate the implementation of 2D FFTs. Thus, in modern FPGAs and SOCs, the scalability of the FFT-based methods is largely limited by the number of available DSPs. To provide fair comparisons, for our FPGA and SOC implementations, we compare our proposed approaches against the use of DSPs in FFT-based methods.

The development of $O(P)$ methods for 2D convolutions also poses significant I/O issues. For example, a sequential access through the pixels will require $O(P^2)$ clock cycles. In our proposed designs, we use parallel loads and stores that can move entire rows and columns of blocks of P-pixels into an array of registers in a single clock cycle. Furthermore, we rely on the use of parallelism and pipelined designs to ensure that each row of pixels is also processed in $O(P)$ clock cycles.

The number of clock cycles to compute the 1D FFT can be reduced from $O(P \log_2 P)$ to $O(P)$ using a fully-pipelined and parallel hardware implementation as documented in [10]. To provide fair comparisons, we will also consider 2D extensions of this work. Overall, due to the need to implement complex arithmetic operations for this approach, we have found that actual hardware resources remain relatively high for many reasonable values of P.

Alternatively, two-dimensional convolutions and cross-correlations can also be computed in the transform domain using the 2D Discrete Periodic Radon Transform (DPRT). The DPRT can be computed using summations along different directions [11], [12]. Similar to the FFT, the DPRT approach requires that we first take the DPRT of the image and the 2D kernel. Then, along each DPRT direction, we compute 1D circular convolutions/cross-correlations

Manuscript received May 28, 2016; revised January 20, 2017; accepted February 18, 2017. Date of publication March 5, 2017; date of current version March 27, 2017. This work was supported by the National Science Foundation under Grant NSF AWD CNS-1422031. The associate editor coordinating the review of this manuscript and approving it for publication was Dr. Christos Bouganis.

C. Carranza is with the Sección Electricidad y Electrónica, Pontificia Universidad Católica del Perú, Lima-32, Perú, and also with the Department of Electrical and Computer Engineering, The University of New Mexico, Albuquerque, NM 87131 USA (e-mail: acarran@pucp.edu.pe).

D. Llamocca is with the Electrical and Computer Engineering Department, Oakland University, Rochester, MI 48309 USA (e-mail: llamocca@oakland.edu).

M. Pattichis is with the Department of Electrical and Computer Engineering, The University of New Mexico, Albuquerque, NM 87131 USA (e-mail: pattichi@unm.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/TIP.2017.2678799

1057-7149 © 2017 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

between the DPRTs of the image and the 2D kernel. The 2D convolution/cross-correlation result can then be computed by taking the inverse DPRT of the previous result. Unlike the 2D FFT approach, the DPRT can be implemented with real-valued fixed-point additions. Furthermore, as shown in [12], we now have fast and scalable fixed-point architectures that can be implemented in FPGAs or SOCs that can compute DPRTs in $O(P)$ to $O(P^2)$ clock cycles depending on available hardware resources.

To implement fast and scalable convolutions and cross-correlations based on the DPRT, we also need to compute the separable 1D convolutions/cross-correlations in $O(P)$ and $O(P^2)$ clock cycles and address I/O issues when connecting with the DPRT blocks. For the fastest approach, we develop *FastConv* and *FastXCorr* based on the fast DPRT that can compute convolutions/cross-correlations in $O(P)$ clock cycles. Similarly, we develop *FastScaleConv* and *FastScaleXCorr* based on the scalable DPRT.

To provide balanced comparisons to other approaches, we also discuss resource requirements. Here, we restrict our discussion to the numbers of required additions, multipliers, and flip-flops. In general, we will classify and approach as one of requiring $O(P^2)$ or $O(P^3)$ resources if the number of additions, multipliers, or flip-flops grows as $O(P^2)$ or $O(P^3)$ respectively. We will provide more detailed comparisons in terms of the exact numbers of additions, multipliers, or flip-flops, DSPs, and other types of resources in a later section of the paper.

We also define scalability based on available resources. We are interested in convolution systems that can be scaled so as to fit into different device sizes. Then, the fastest methods refer to approaches that can compute 2D convolutions using the minimum number of clock cycles while requiring the maximum amount of resources. On the other hand, slower implementations will require fewer resources. Thus, we have a clear trade-off between performance and required resources.

In addition to comparisons against FFT based methods, we also consider spatial-domain methods based on systolic arrays [13]. The standard systolic array implementation of 1D convolutions computes an output every clock cycle. Without using separability, a direct extension of the 1D systolic array approach requires that we keep several image rows in memory [14], [15]. As a result, the application of non-separable systolic array implementations has been limited to relatively small kernels. Furthermore, a derivation of a $O(P)$ clock cycles approach based on systolic approaches leads to prohibitive hardware resource growth of $O(P^3)$ [14]. In comparison, hardware resourses in all of our proposed methods only grow as $O(P^2)$ or less.

In the spatial domain, we also have the relatively recent emergence of fast convolution using a sliding window [16]. At each image pixel, a sliding window of the same size as the kernel is applied to compute one output pixel of the convolved image [17]. This comes at a cost of using as many multipliers and adders as the coefficients in the kernel, and thus grows linearly with the number of coefficients in the kernel.

We also develop *FastRankConv*, a second family of fast and scalable architectures that represents an extension of the current systolic methods. Our approach is based on the use of separable approximations of non-separable kernels [18], [19]. The basic idea is to express non-separable kernels as a sum of of a small number of separable kernels. Then, scalable hardware implementations can be derived by controlling the number of efficient 1D processors.

Overall, we describe and implement two fast methods for computing 2D convolutions in $O(P)$ clock cycles. Both methods map 2D convolutions into a collection of 1D convolutions that are computed in $O(P)$ clock cycles. Each 1D convolution is computed in parallel using a row of multipliers followed by an adder tree. For *FastRankConv*, we approximate the 2D convolution kernel using a minimal number of 1D kernels that are applied along each row and each column. For *FastConv* and *FastScaleConv*, we first take the DPRT in $O(P)$ clock cycles using the fast DPRT, compute the 1D convolutions in the transformed domain, and then take the inverse DPRT in $O(P)$ clock cycles using the inverse DPRT.

We summarize the primary architectural elements of our design:

- ***An array of circular-shift-registers:*** The image data is processed using an array of circular shift registers.
- ***Fast memory:*** The memory array is implemented using a row of SRAMs where each SRAM stores a column of the image.
- ***Row-level parallel I/O:*** The scalable architectures load the image into memory using a sequence of parallel loads of rows of pixels. Thus, for an image with N rows, we can load the entire image into memory in N cycles.
- ***Row-level parallel and pipelined processing:*** The proposed scalable architectures are designed to process multiple rows at the same time. Thus, for FPGA and SOC implementations, the idea is to implement as many row-processing units as we can fit in the device. Then, each row-processor uses a pipelined architecture that produces results after each cycle after an initial latency.
- ***Fast transpositions:*** We significantly reduce the transposition overhead using an additional output memory array. The output memory array uses dual-port memories to allow us to write the output results and read intermediate values at the same time. Based on our proposed approach, we can read and write rows and columns in a single cycle as needed. Overall, in our pipelined design, the net effect is that transposition is performed during computation and will thus not require any additional cycles.

The scalability characteristics of our proposed architectures include:

- ***Performance scalability by controlling the number of row-processors in the DPRT and the 1D convolutions/ cross-correlations:*** We refer to [12] for the scalable DPRT implementation.
- ***Pareto optimality:*** We present Pareto-optimal designs in the sense that our family of architectures provide the fastest implementations based on available resources. In other words, additional resources always yield faster performance.
- ***Fast 2D convolutions and cross-correlations:*** *FastConv* and *FastXCorr* compute convolutions and

cross-correlations for $P \times P$ blocks in $O(P)$. For large images, the image can be broken into L separate blocks of size $P \times P$ and use an overlap-and-add approach to compute the final results. Thus, in the fastest case, we can compute convolutions and cross-correlations in just $O(L \cdot P)$ clock cycles. On the other hand, in the worst case scenario, with very limited resources, 2D convolutions and cross-correlations can be computed in $O(L \cdot P^2)$ clock cycles. Here, we use the term *large image* to refer to image sizes that require more on-chip resources than what is available.

The rest of the manuscript is organized as follows. The mathematical definitions for the DPRT, its inverse, and the transformation property of the DPRT are given in section II. The proposed approach is given in section III. Section IV presents the results. Conclusions and future work are given in section V.

II. Background

A. Basic Notation

Let $g(i, j)$ denote an image (or image block) of P_1 rows with P_2 pixels per row be of size $P_1 \times P_2$ with B bits per pixel. We index $g(i, j)$ using $0 \leq i \leq P_1 - 1$ and $0 \leq j \leq P_2 - 1$. We use h to denote the convolution kernel and assume a size of $Q_1 \times Q_2$ with C bits per pixel. We use $f(i, j)$ for the output of size $N_1 \times N_2$ where $N_1 = P_1 + Q_1 - 1$ and $N_2 = P_2 + Q_2 - 1$. For the case when $N_1 = N_2$ and $P_1 = P_2$, we simply use N and P throughout the text.

Typically, images tend to be much larger than the convolution or cross-correlation kernels. In such cases, the input image g will be broken into blocks that are equal to the size of the kernel. Thus, in the most typical scenario, we assume that the image and kernel blocks are of size $P \times P$. After linear convolution, the output image block is of size $N \times N$ where $N = 2P - 1$. To compute outputs over the entire image, we break the image into non-overlapping blocks and use overlap-and-add to produce the final results.

B. Separable Decomposition for Non-Separable Kernels

We begin with the 2D Z-transform of the convolution kernel h:

$$H(z_1, z_2) = \sum_{i=0}^{Q_1-1} \sum_{j=0}^{Q_2-1} h(i, j) z_1^{-i} z_2^{-j}. \tag{1}$$

To allow for separable decompositions, we consider a matrix re-formulation of (1) [19]:

$$H(z_1, z_2) = \mathbf{Z_1}^T \, \mathbf{H} \, \mathbf{Z_2} \tag{2}$$

where we have placed all of the filter coefficients in $\mathbf{H}$, and $\mathbf{Z_i} = [1 \, z_i^{-1} \, z_i^{-2} \, \ldots \, z_i^{-(Q_i-1)}]$ for $i = 1, 2$. Now that we have the filter coefficients in matrix form, we can consider separable matrix approximations to $\mathbf{H}$. First, consider the singular value decomposition (SVD) for $\mathbf{H}$: $\mathbf{H} = \mathbf{U}\Sigma\mathbf{V}^T$. Then, we can simplify $\mathbf{H}$ by zeroing out the smallest singular values of Σ. If we let $\Sigma_\mathbf{m}$ denote the resulting Σ after zeroing-out small singular values, we reconstruct an effective approximation to $\mathbf{H}$ using $\mathbf{H_r} = \mathbf{U}\Sigma_\mathbf{r}\mathbf{V}^T$ where we have kept the r larges singular values of $\mathbf{H}$. In this case, we use the LU decomposition of $\mathbf{H_r}$ to get [19]:

$$H_r(z_1, z_2) = \sum_{k=1}^{r} \left(\sum_{i=0}^{Q_1-1} l_{ki}^m z_1^i \right) \left(\sum_{j=0}^{Q_2-1} u_{jk} z_2^j \right) \tag{3}$$

where r also denotes the rank of $\mathbf{H_r}$. In (3), we have expressed the original 2D convolution into a sum of r separable 1D convolutions along the rows and columns. Furthermore, it is clear that the separable decomposition also applies to non-separable 2D kernels. Furthermore, in the simplest case we have $r = 1$ which eliminates the hardware required for accumulating the additions. We will not consider this case any further (see [20] for details).

C. The Discrete Periodic Radon Transform (DPRT)

We define the DPRT of f of size $N \times N$, N prime, using [21]:

$$F(m, d) = \begin{cases} \sum_{i=0}^{N-1} f(i, \langle d + mi \rangle_N), & 0 \leq m < N, \\ \sum_{j=0}^{N-1} f(d, j), & m = N, \end{cases} \tag{4}$$

where $d = 0, 1, \ldots, N - 1$, $m = 0, 1, \ldots, N$, and $< . >_N$ denotes the positive remainder when we perform integer division by N (e.g., $< 128 >_{127} = 1$). In (4), we have that m indexes the prime directions. Along each prime direction, we add up the pixels along each ray. In (4), d is used to index each the rays of each direction.

The inverse DPRT can be used to reconstruct f from the forward DPRT using:

$$f(i, j) = \frac{1}{N} \left[\sum_{m=0}^{N-1} F\left(m, \langle j - mi \rangle_N\right) - S + F(N, i) \right] \tag{5}$$

where:

$$S = \sum_{j=0}^{N-1} \sum_{i=0}^{N-1} f(i, j). \tag{6}$$

As noted in the definition, the size of the transform needs to be restricted to prime numbers. We do not impose this restriction directly to the input image block and kernel sizes, but to the result of the linear convolution of size $N_1 \times N_2$, with $N_1 = P_1 + Q_1 - 1$ and $N_2 = P_2 + Q_2 - 1$. Therefore, a minimal (or even none) zero padding is required if the input sizes are selected conveniently. There are several reasons for imposing this restriction. Most importantly though, for prime N, the DPRT provides the most efficient implementations by requiring the minimal number of $N + 1$ primal directions [22]. It is important to note that prime-numbered transforms have advantages in convolution applications. Here, just like for the Fast Fourier Transform (FFT), we can use zero-padding to extend the DPRT for computing convolutions in the transform domain. Unfortunately, when using the FFT with $N = 2^p$, zero-padding requires that we use FFTs with double the size of N. In this case, it is easy to see that the use

TABLE I

SUMMARY OF THE PROPOSED METHODS. SCALABILITY IS ACHIEVED BY VARYING THE NUMBER OF 1D CONVOLVERS (J) AND THE NUMBER OF ROWS PROCESSED IN PARALLEL IN THE DPRT AND INVERSE DPRT (H). THERE ARE SOME MINOR DIFFERENCES BETWEEN THE ARCHITECTURES THAT COMPUTE CROSS-CORRELATIONS AS OPPOSED TO CONVOLUTIONS. FURTHERMORE, CONVOLUTION SYSTEMS CAN IMPLEMENT CROSS-CORRELATIONS BY FLIPPING THE KERNEL OFFLINE OR IN REAL-TIME IN HARDWARE

Method	Hardware Components	Architecture	Algorithm	Section
FastConv / *FastXCorr*	1D Circular convolver, FDPRT[12], iFDPRT[12]	Fig. 1 (convolver), Fig. 5 (system) using FDPRT and iFDPRT	Fig. 2 (convolver), Fig. 4 (system) with $J = N + 1$ and $H = N$	III-A, III-B and III-C
FastScaleConv / *FastScaleXCorr*	1D Circular convolver, SFDPRT[12], iSFDPRT[12]	Fig. 1 (convolver), Fig. 5 (system) using SFDPRT and iSFDPRT	Fig. 2 (convolver), Fig. 4 (system) Scale parameters: J and H	III-A, III-B and III-C
FastRankConv / *FastRankXCorr*	1D Linear convolver, Custom SRAM	Fig. 9 (convolver), Fig. 8 (SRAM), Fig. 11 (system)	Fig. 10 (convolver), Fig. 12 (system)	III-D

of prime-numbered DPRTs is better since there are typically many prime numbers between 2^p and 2^{p+1}.

We refer to [12] for fast and scalable implementations of the DPRT and its inverse. In the fastest case, we can compute the full DPRT in just $2N + \lceil \log_2 N \rceil + 1$ clock cycles with $O(N^2)$ growth in resource usage. For the scalable DPRT implementation, we require $\lceil N/H \rceil (N + 3H + 3) + N + \lceil \log_2 H \rceil + 1$ cycles where H is used as the scalability parameter. We have a family of scalable DPRT implementation using $H = 2, \ldots, N$ with a resource usage that grows from $O(N)$ for the slowest case ($H = 2$) to $O(N^2)$ for the fastest case ($H = N$).

D. Circular Convolutions Using the DPRT

Consider the 2D circular convolution $f = g \otimes h$ given by:

$$f(k,l) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} g(i,j) h(\langle k-i \rangle_N, \langle l-j \rangle_N). \quad (7)$$

To define the DPRT convolution property, let m denote a prime direction and define the DPRTs along the m-direction using: $F_m(d) = F(m,d), G_m(d) = G(m,d), H_m(d) = H(m,d)$. We then have that the m-direction DPRTs are related through 1D dimensional circular convolution in the transform domain as given by [23]:

$$F_m(d) = \sum_{k=0}^{N-1} G_m(k) H_m(\langle d-k \rangle_N) \quad (8)$$

Thus, we can compute the result of 2D circular convolution in the transform domain using 1D circular convolutions along all of the prime directions as given by (8). After computing the DPRT of the result along each direction, we can then take an inverse DPRT to recover f.

III. METHODOLOGY

We provide a summary of the proposed methods in Table I. For general kernels, all methods are based on the DPRT and inverse DPRT. Here, the fastest methods (*FastConv*, *FastXCorr*) correspond to a simplification of the scalable methods (*FastScaleConv*, *FastScaleXCorr*). For low-rank kernels, we recommend the use of *FastRankConv* and *FastRankXCorr*. For the rest of the section, we begin with a description of the 1D convolver architecture that is shared by all methods (see section III-A). We then show how the 1D convolvers are integrated in the DPRT-based methods of sections III-B and III-C, and the low-rank decomposition methods of section III-D. In section III-E we describe the application of overlap-and-add for applying all of the methods to large images.

A. Computing 1D Circular Convolutions Using Circular Shifts

Let $F_m(d), G_m(d), H_m(d)$ denote the DPRTs of f, g, h along the m-th prime direction. We define a special flip operation $\breve{H}_m$ defined by:

$$\breve{H}_m(d) = H_m(N-1-d), \quad d \geq 0,$$

and the circular right shift (CRS) by n using H_m^n that is defined by:

$$H_m^n(d) = H_m(\langle d+n \rangle_N).$$

Then, start from (8) to derive a shifted representation of the circular convolution using:

$$\begin{aligned} F_m(d) &= \sum_{k=0}^{N-1} G_m(k)\, H_m(\langle d-k \rangle_N) \\ &= \sum_{k=0}^{N-1} G_m(k)\, H_m(\langle N-1-k+d+1 \rangle_N) \\ &= \sum_{k=0}^{N-1} G_m(k)\, H_m^{d+1}(N-1-k) \\ &= \sum_{k=0}^{N-1} G_m(k)\, \breve{H}_m^{d+1}(k). \end{aligned} \quad (9)$$

From (9), we can see that $F_m(d)$ can be expressed as the dot product between G_m and a flipped and circular right shifted by $d+1$ positions version of H_m (denoted as $\breve{H}_m^{d+1}$).

B. Fast 1D Circular Convolution Hardware Implementation

In this section, we derive a fast hardware implementation based on (9). We present the hardware architecture in Fig. 1, the associated algorithm in Fig. 2, and the timing diagram in Fig. 3.

We begin with the fast computation of 1D circular convolutions given in Fig. 2. Initially, we use parallel loads to transfer

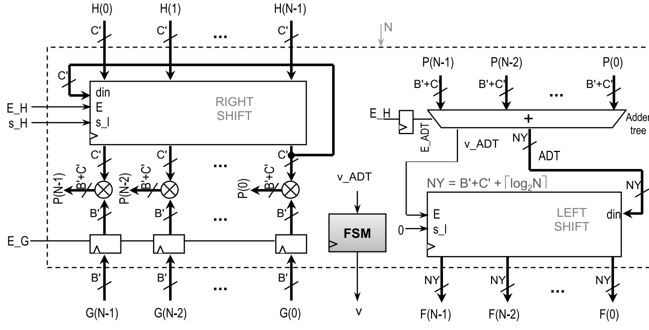

Fig. 1. Architecture for computing the 1D circular convolution $F_m = G_m \otimes H_m$. B' and C' represent the number of input bits of G and H respectively.

1: Parallel load $G = G_m$, flipped load $H = \breve{H}_m$
2: **for** d = $N - 1$ downto 0 **do**
3: Parallel mult. $P(k) = G(k)H(k),\ 0 \le k \le N - 1$.
4: Parallel add $F(d) = \sum_{k=0}^{N-1} P(k)$
5: CRS by one $H = H^1$
6: **end for**
7: Parallel output F

Fig. 2. Algorithm for computing the 1D circular convolution $F_m = G_m \otimes H_m$.

both of the DPRTs to the G and H registers in a single clock cycle. Note that flipping H_m into $\breve{H}_m$ is performed by simply wiring the inputs in reverse as shown in the upper register portion of Fig. 1. Starting with the last convolution output, we have a 3-step sequence of parallel multiplies, addition of the results, and a circular right shift to prepare for the next output (lines 3-5). The multiplications are performed in parallel in a single cycle using the parallel fixed-point multipliers of Fig. 1 and added using a pipelined tree structure in just $\lceil \log_2(N) \rceil$ clock cycles (e.g., see [12]). The resulting outputs are left-shifted in, one output sample at a time, into the output F register shown in the lower-right portion of Fig. 1. A single cycle is also needed to perform the circular right shift of H using the top-left register of Fig. 1.

To derive the timing requirements, refer back to Fig. 3. Using a fully pipelined approach, we begin working on the next output sample after the parallel multiplies. It is easy to see that after the initial latency for the first sample, we compute an output sample at every cycle. After adding the latency for the initial loads, the adder latency, and the final left shift, we have a total of just $N + \lceil \log_2(N) \rceil + 2$ clock cycles.

C. Fast and Scalable 2D Linear Convolutions and Cross-Correlations Using the DPRT

In this section, we develop the architectures, algorithms, bit requirements, and computational efficiency for 2D convolutions and cross-correlations. Most importantly, we discuss the scalability of the proposed approach that allows for the most efficient implementations based on available resources.

We begin with an analysis of the sequence of operations for computing fast and scalable 2D convolutions and cross-correlations as shown in Fig. 4. In the most efficient

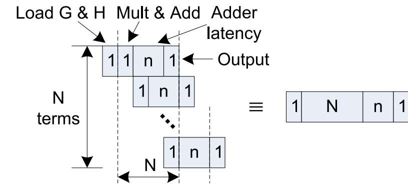

Fig. 3. Running time for the implementation of the fast architecture for computing one 1D circular convolution. In this diagram, time increases to the right. The number of clock cycles for computing each term of $F_m(d)$ is shown on each strip. The strip on the right represents the total running time. $n = \lceil \log_2 N \rceil$ represents the addition latency.

▷ For cross-correlation, apply:
▷ $\texttt{Flip}$ (h) and store
▷ the flipped version in memory.
1: Precompute/Compute
$H = \texttt{DPRT}\{\texttt{ZeroPad}\{h\}\}$
and store the results in memory.
2: Compute $G = \texttt{DPRT}\{\texttt{ZeroPad}\{g\}\}$
3: **for** p = 0 to $L - 1$ **do**
4: Compute J directions in parallel:
$F_{pJ+i} = G_{pJ+i} \otimes H_{pJ+i}, \quad \text{for } i = 0, \ldots, J - 1.$
5: **end for**
6: Compute $f = \texttt{DPRT}^{-1}\{F\}$

Fig. 4. Fast and scalable algorithm for computing 2D linear convolutions and cross-correlations between $g(i, j)$ and $h(i, j)$ using the architecture depicted in Fig. 5. $L = \lceil (N + 1)/J \rceil$.

implementation, the convolution kernel is available ahead of time. In this case, we can pre-compute the DPRT of the kernel and store it in memory as shown in the hardware architecture of Fig. 5. In Fig. 5, we provide a unifying architecture for implementing *FastScaleConv*, *FastScaleXCorr*, *FastConv*, and *FastXCorr*.

For adaptive filterbank applications, the DPRT of the zero-padded convolution kernel can be computed in real-time using the `SFDPRT_System` where the resulting DPRT is stored in (additional) memory. Alternatively, we can replicate the `SFDPRT_System` system for the kernel to avoid an increase of the running time. For computing cross-correlations, we need to undo the vertical and horizontal flips associated with convolution. This can be done by flipping the kernel along rows and columns as described in Fig. 4. Here, note that the horizontal and vertical flips are performed by the `SFDPRT_System` component during the loading of the kernel. An inverted `MODE` signal is used to control the `SFDPRT_System` to perform the needed flips. Vertical flips are implemented by switching the order of loading the rows. Thus, in a vertical flip, the last kernel row is loaded first and the first kernel row is loaded last. Horizontal flips are simply implemented by loading each row in reverse. Thus, in a horizontal flip, the last row element is loaded first and the first row element is loaded last. Overall, there is minimal overhead for implementing the horizontal and vertical flips.

Scalability is achieved by controlling (i) the number of 1D circular convolutions that can be computed in parallel

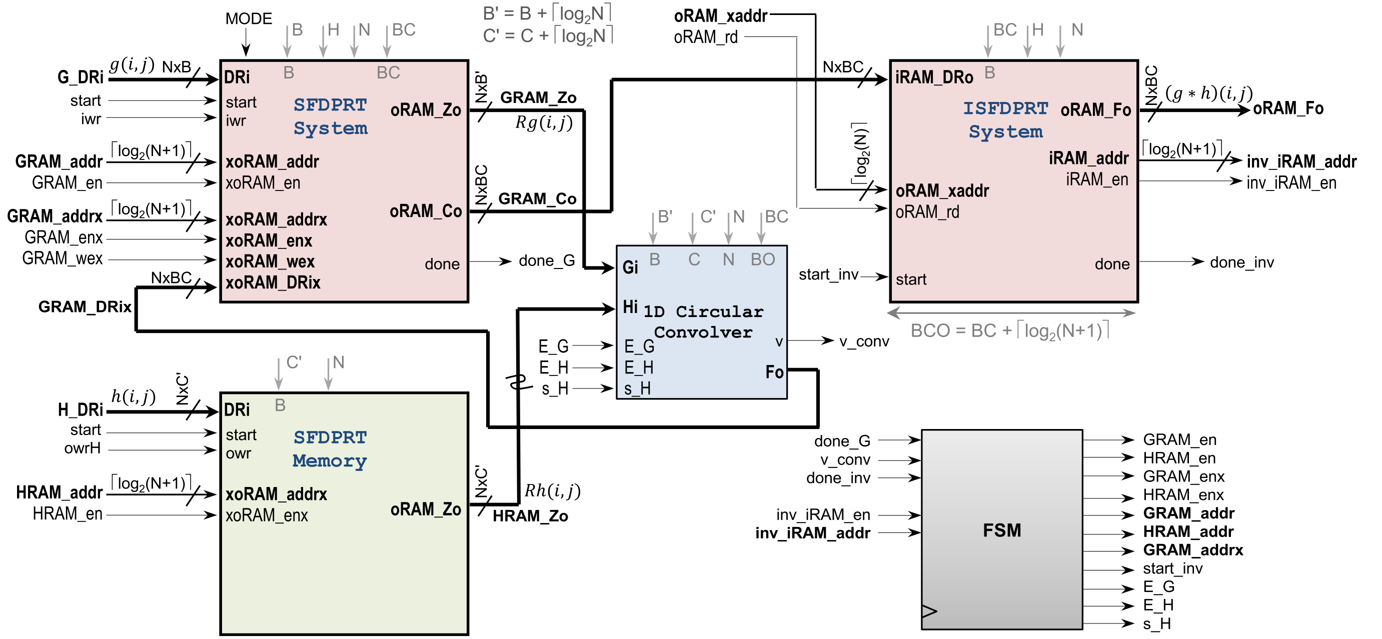

Fig. 5. *FastScaleConv* and *FastScaleXCorr*: Fast and scalable architecture system for computing 2D convolutions and cross-correlations based on the DPRT (also see Fig. 4). The forward DPRT is computed by SFDPRT. The inverse DPRT is computed by ISFDPRT. The linear convolver computes circular convolutions for J rows. The finite state machine (FSM) manages all the control signals (except for 'start' and 'iwr'). We use bold face letters to denote buses while convolution parameters are depicted in gray. Refer to section II-A for definitions of the basic convolution parameters. *FastConv* is a simplification of *FastScaleConv* for maximum performance (see text).

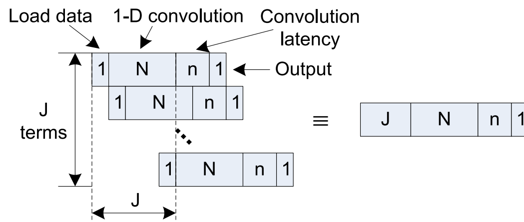

Fig. 6. Running time for computing J circular convolutions in parallel using J fast convolution blocks (see basic block structure in Fig. 1). In this diagram, time increases to the right. Here, it takes one cycle to perform a parallel load for each block. Overall, we require $J+N+n+1$ to compute everything, where $n = \lceil \log_2 N \rceil$ represents the addition latency.

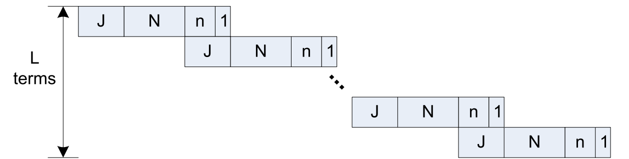

Fig. 7. Running time for computing $N+1$ 1D circular convolutions using J fast convolution blocks operating in parallel. In this diagram, time increases to the right. We need to reload the convolution blocks L times given by $L = \lceil (N+1)/J \rceil$. Each row shows the running time for performing J convolutions as described in Fig. 6.

(denoted by J), and (ii) the number of image rows that can be processed in parallel in the DPRT blocks (denoted by H as described in [12]). Following the computation of the 1D circular convolutions, an inverse DPRT is applied for computing the final result.

We also list bit requirements. For the setup, refer to the notation of section II-A. To compute exact convolutions, we need to zero pad to a prime number. We thus require $N = \texttt{NextPrime}(\max(P_1 + Q_1 - 1,\ P_2 + Q_2 - 1))$. Then, it is easy to see that we require (i) $B+n$ bits for the DPRT of g, $C+n$ bits for the DPRT of h where g uses B bits, h uses C bits, and $n = \lceil \log_2 N \rceil$ (also see [12]), (ii) $B+C+3n$ bits for the convolutions, and (iii) $B+C+4n$ bits just before the normalization step of the inverse DPRT [12], and $B+C+x$ bits for the final result, where x represents the additional bits used for precision after the division.

We next derive the computational complexity of our approach. From section II-C and [12], scalable DPRT computation requires $\lceil N/H \rceil (N+3H+3)+N+\lceil \log_2 H \rceil+1$ clock cycles that reduce to $2N + \lceil \log_2 N \rceil + 1$ clock cycles for the fast DPRT implementation. For computing the number of cycles required for the circular convolutions, refer to Figs. 6 and 7. As shown in Fig. 6, we require $J+N+n+1$ clock cycles to compute J convolutions in parallel where $n = \lceil \log_2 N \rceil$ represents the initial addition latency. To compute outputs for all of the $N+1$ required DPRT directions, we use all J parallel blocks of 1D convolutions for $L = \lceil (N+1)/J \rceil$ times. Depending on N, increasing J may not always provide for better solutions. There is a need to find optimal values for J. We refer to section III-F for determining the optimal values of J. Overall, we require a total of $L \cdot (J+N)+n+1$ clock cycles to compute all of the 1D convolutions. We provide a summary of the required resources for implementing the J 1D parallel convolution blocks in Table VIII.

Overall, based on the derived complexity, we have the fastest running time using $J = N+1$ parallel 1D convolutions

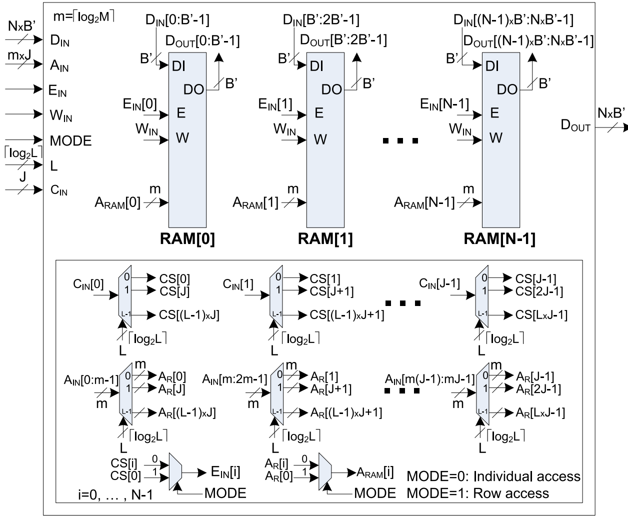

Fig. 8. Custom SRAM architecture for fast transpositions and memory access. The architecture allows for full-row (or full-column, i.e. transpose) read/write in a single clock cycle (`MODE=1`) and individual access to up to J SRAMs in a single clock cycle (`MODE=0`). The SRAM stores M rows (or columns) of N B'-bit per pixels. The basic architecture can be configured for different purposes as given in Table II.

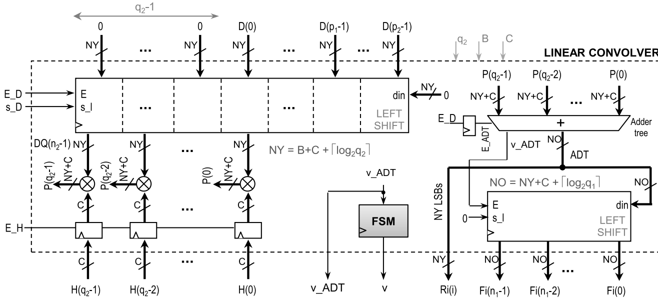

Fig. 9. Architecture for computing fast 1D linear convolution $F_i = D_i * H_i$. This block is the core for computing 2D convolutions with separable kernels. It computes 1D convolutions for two sizes: (i) D and H are of size p_2 and q_2 respectively, generating outputs of size $p_2 + q_2 - 1$, performed p_1 times (the rows of the 2D image or image block). (ii) D and H are of size p_1 and q_1 generating outputs of size $p_1 + q_1 - 1$, performed $p_2 + q_2 - 1$ times (the columns of the step (i) result). Refer to Fig. 11 (bottom part) for more details about the 2D convolution.

at just $2N + n + 2$ clock cycles with resource usage of $O(N^2)$ for flip-flops and full adders. For the minimum number of resources, we only use a $J = 1$ 1D convolution block that require $(N+1)^2 + n + 1$ clock cycles with the lowest resource usage $O(N)$ for flip-flops and full adders.

Following the 1D convolutions, we take the inverse DPRT using the `iSFDPRT_System` module. Similar to the forward DPRT, scalability is controlled by H, the number of image rows processed in parallel [12]. For this step, the input data uses $B + C + 3n$ bits per pixel. Depending on available resources, the inverse DPRT can be computed in just $2N + 5n + B + C + 2$ for the fast inverse DPRT with $O(N^2)$ resource usage (1-bit additions and flip-flops), or as slow as $\lceil N/2 \rceil (N + 2) + 4n + B + C + 4$ for $H = 2$ for just $O(N)$ resource usage [12].

1: **procedure** $\texttt{LinConv1D}(D, SG, SH, \texttt{MEM})$
2: Parallel load $GX[SH-1 : SG-1] = D$
3: Parallel load $GX[0 : SH-2] = 0$
4: **for** $s = 0$ to $SG - 1$ **do**
5: Parallel mult. $P(k) = GX[k]\,H[k]$ for $k = 0, \ldots, SH-1$
6: Parallel add $F[s] = \sum_{j=0}^{SH-1} P[j]$ and store or accumulate in `MEM`
7: CLS by one GX
8: **end for**
9: **end procedure**

Fig. 10. Algorithm for computing 1D linear convolution between D and the 1D kernel H (size=SH) and stores the results in F (size=SG). CLS refers to the circular left shift operation. GX represents the upper-left row of shift registers in Fig. 9. HX represents the lower-left row of registers in Fig. 9 that is pre-loaded with the 1D kernel (H). The output is stored in `MEM` = `MEM_TMP` for rows, or accumulated in `MEM` = `MEM_OUT` for columns.

TABLE II

SRAM MEMORY CONFIGURATIONS FOR MAXIMUM ACCURACY. ORIENTATION REFERS TO EACH SRAM HOLDING EITHER A FULL ROW OR COLUMN OF THE IMAGE. THE ACCUMULATE MODE NEEDS EXTERNAL ADDERS TO PERFORM THE ACCUMULATION AND DUAL-PORT SRAMS FOR FULL SPEED. B DENOTES THE NUMBER OF BITS OF THE INPUT IMAGE. C DENOTES THE NUMBER OF BITS USED FOR THE KERNEL COEFFICIENTS. WE HAVE $q1 = \lceil \log_2 Q1 \rceil$ AND $q2 = \lceil \log_2 Q2 \rceil$

SRAM	`MEM_IN`	`MEM_KER`	`MEM_TMP`	`MEM_OUT`
Quantity	$P2$	$Q2$	$P1$	$P2 + Q2 - 1$
Depth	$P1$	$2Q2$	$P2 + Q2 - 1$	$P1 + Q1 - 1$
Wordlength	B	C	$B + C + q2$	$B + 2C + q1 + q2 + r$
Function	g	h_R, h_C	g'	f
MODES	1	1	0/1	0/1
Orientation	Column	Column	Row	Column
WriteMode	Store	Store	Store	Accumulate

D. Fast and Scalable 2D Linear Convolution Using SVD-LU Decompositions (FastRankConv)

As described in section II-B, we can use a collection of 1D convolutions along the rows and columns to implement effective approximations to 2D convolutions with the inherent loss in accuracy due to zeroing the smaller singular values associated with the SVD decomposition. Unfortunately, direct approaches suffer from the need to implement two transpositions that require $O(N^2)$ clock cycles. In this subsection, we present a fast and scalable system that eliminates the need for transpositions and allows us to compute convolutions in $O(N)$ to $O(N^2)$ clock cycles with the addition of an intermediate custom memory.

As before, scalability is achieved by controlling J, the number of linear convolutions that are computed in parallel. The linear convolution blocks are similar to the circular convolution blocks except that the complexity is a function of the size of the convolution kernel only (see Fig. 9 and bottom part of Fig. 11). Then, in order to operate as fast as possible, we design a custom memory system that moves entire rows or columns to and from each linear convolver. The basic idea is to start by moving all of the rows into the J convolution

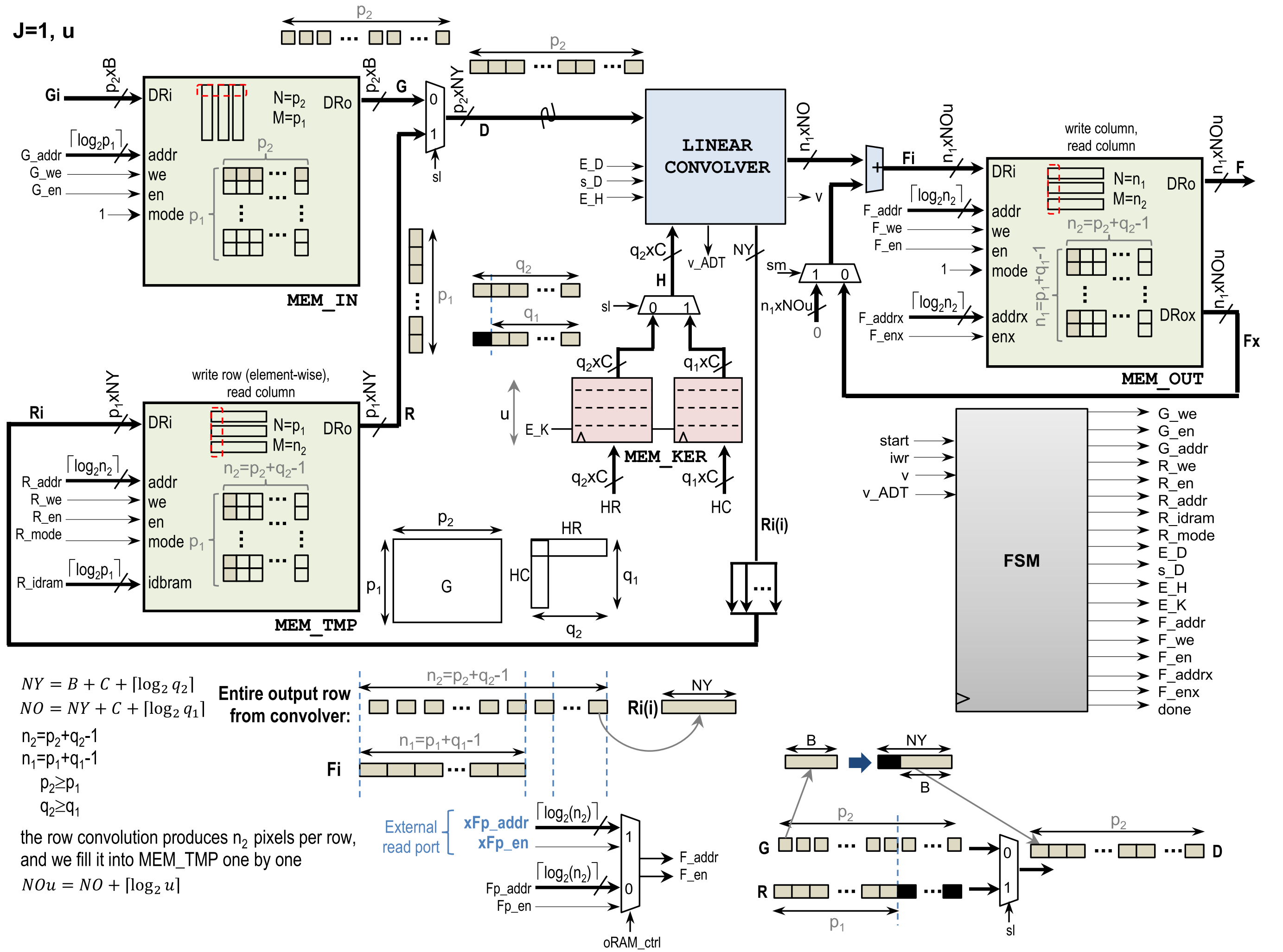

Fig. 11. *FastRankConv*: Fast and scalable 2D convolution system based on separable decompositions. Refer to subsection II-A for the notation and Fig. 8 for the memory architecture. The linear convolution blocks are very similar to the circular convolution blocks except that multiplications and additions are reduced to the size of the convolution kernel. The Bus width shown is the one for maximum accuracy. Also, note that the implementation of *FastRankCross* is not considered here since cross-correlation is the same as convolution with a flipped kernel, and flipping can be computed during pre-processing (prior to SVD and LU). The finite state machine is denoted by FSM. The Linear Convolver can be applied to several rows in parallel. The results are accumulated in MEM_OUT. Please refer to Fig. 12 for a full description of the algorithm and section II-A for definitions of the basic parameters (p1, q1, p2, q2, B, C).

blocks, store the convolution results in J SRAM memories so that the rows of the row-convolutions results correspond to the columns of the original image, and then perform row convolutions and store in J output SRAM memories. Thus, the need for the transpositions is completely avoided.

Then, for a single clock cycle, we use custom memories to (i) allow us to move entire rows and columns of blocks of pixels from memory to the convolution blocks and vice-versa, and (ii) allow direct access to J different SRAMs. We present the proposed custom SRAM architecture in Fig. 8, the full system architecture in Fig. 11 and the associated algorithms in Figs. 10 and 12. Refer to section II-A for the notation. We customize the basic SRAM architecture of Fig. 8 as given in Table II, so that in a single clock cycle: (a) MEM_IN provides a full row of the image, (b) MEM_KER provides the entire row or column filter coefficients, (c) MEM_TMP stores the results of convolution along each row, provides access to a full column of the results, and (d) MEM_OUT, accumulates the final result, adds up to $P2+Q2-1$ values of the convolved image (in a single clock cycle), and also provides a full row. The required resources are summarized in Table IX.

We also provide a summary of performance-resource requirements. Without loss of generality, we assume that $P2 \geq P1$, $Q2 \geq Q1$, and consequently $N2 \geq N1$. Furthermore, for the purposes of the analysis, assume full rank: $r = Q1$, and let $L_R = \lceil P1/J \rceil$ and $L_C = \lceil (P2+Q2-1)/J \rceil$. The total running time is given as the sum of clock cycles required for: (i) row processing: $r \cdot L_R \cdot (J+P2+Q2-1)$, (ii) column processing: $r \cdot L_C \cdot (J+P1+Q1-1)$, and (iii) the latency of the adder tree $\lceil \log_2 Q1 \rceil + 1$. To simplify the derivation, let $N = \max\{P2+Q2-1, P1+Q1-1\}$. Then, for $J=1$, we have minimum resource usage that grows as $O(N)$ with a running time of $O(N^2)$. For $J=N$, we have the fastest running time $O(N)$ with resource usage that grows as $O(N^2)$. Refer to

1: **for** q = 0 to $r - 1$ **do**
  ▷ Compute convolutions along the rows
2:   Parallel load row-kernel using
    $HX[0 : Q2 - 1] = h_{Rq}(j)$
3:   **for** $p = 0$ to $L_R - 1$ **do**
4:     Use J 1D-convolvers to compute:
      `LinConv1D` ($g_{p \cdot J+k}(j)$,
        $P2 + Q2 - 1$, $Q2$,
        `MEM_TMP`)
      for $k = 0, \ldots, J - 1$.
5:   **end for**
  ▷ Compute convolutions along the columns
6:   Parallel load column-kernel using
    $HX[0 : Q1 - 1] = h_{Cq}(j)$
7:   **for** $p = 0$ to $L_C - 1$ **do**
8:     Use J 1D-convolvers to compute:
      `LinConv1D` ($g'_{p \cdot J+k}(i)$,
        $P1 + Q1 - 1$, $Q1$,
        `MEM_OUT`)
      for $k = 0, \ldots, J - 1$
9:   **end for**
10: **end for**

Fig. 12. Algorithm for computing the 2D linear convolution between an image (or image block) $g(i, j)$ and the non-separable kernel $h(i, j)$ decomposed into r separable kernels. We use $h_R(i, j)$ for the row kernels and $h_C(i, j)$ for the column kernels. The results are computed in $g'(i, j)$. The row-convolution results are stored in `MEM_TMP`. The final output is accumulated in `MEM_OUT`.

Table II for detailed resource usage of the memories. We can further optimize the architecture parameters as described in section III-F. We will also provide more detailed comparisons in section IV.

E. Scalability for Large Images Using Overlap-Add

The convolution and cross-correlation kernels tend to be much smaller than the size of the input image. Thus, for much larger images, the best approach is to design the hardware architecture for the smaller kernels. We summarize the basic approach below.

The original image is subdivided into the smaller windows that are equal to the size of the kernel. Convolutions and cross-correlations can be computed for each block. Results from neighboring blocks must be added together. Furthermore, the final output is a concatenation of the results from each block.

The basic approach is very well understood. Furthermore, the approach can also be parallelized to use multiple hardware blocks. In what follows, we will simply assume that both the image (block) and the convolution/cross-correlation kernel size are of the same size. Furthermore, we will focus on the most common size when both the image (block) and the kernels are square.

F. Pareto Optimal Architectures

As we discussed earlier in this section, it is possible to use J that is sub-optimal. Here, we refer to an architecture as being sub-optimal in the Pareto sense (e.g., see [24]). Essentially, an architecture is considered to be Pareto-optimal if it provides the best possible performance for required resources. Thus, a Pareto optimal family of architectures will always produce better running time for more resources. To derive the set of Pareto-optimal solutions, recall that our scalable families of architectures may contain less than J rows for the last block of 1D convolutions. Thus, for *FastScaleConv* and *FastScaleX-cross*, to fully utilize available hardware resources, we require that the selected J values would satisfy $\langle N + 1\rangle_J = 0$. Similarly, for *FastRankConv*, we require that the selected J values simultaneously satisfy $\langle P1\rangle_J = 0$ and $\langle P2 + Q2 - 1\rangle_J = 0$.

IV. RESULTS

In this section, we provide extensive comparisons with prior methods to demonstrate the promise of the proposed methods. Here, we note that the proposed systems implement both convolutions and cross-correlations.

We compare our approach to relevant, state of the art, convolution systems by considering (i) serial systolic arrays [14] (SerSys), (ii) scalable and parallel systolic arrays [15] (ScaSys), (iii) sliding windows [25] (SliWin), and (iv) parallel and pipelined Fast Fourier Transform radix-2 [10] (FFTr2). Here, we do not consider methods based on Distributed arithmetic (DA) solutions since the internal ROM required for the DA operation grows exponentially with the kernel size, making them unsuitable for large kernels [26]. Furthermore, to provide fair comparisons, we are assuming that FFTr2 is based on the parallel use of the highly efficient 1D FFTs described in [10].

We will consider comparisons for different bitwidths. Here, we note that the required number of bits for maintaining full precision was developed in section III. Furthermore, we will provide results from full precision, the use of DSPs, and a limited bitwidth in the results. We are currently researching different methods for selecting different bitwidths for different stages of the algorithms.

As described earlier, the proposed architectures can compute both convolutions and cross-correlations. For *FastRankConv*, flipping the kernel can clearly be done during pre-processing, prior to SVD and LU computations. In what follows, we will present results for *FastConv*, *FastScaleConv*, and *FastRankConv*. Here, we note that *FastXCorr*, *FastScaleX-Corr*, and *FastRankXCorr* are minor variations of *FastConv*, *FastScaleConv*, and *FastRankConv*.

We describe the implementation setup in section IV-A. In section IV-A, we also describe alternative methods. We provide extensive comparisons in terms of performance and required hardware resources in section IV-B. FPGA and SOC implementations are described in section IV-C.

A. Implementation Setup

We consider convolutions with $P \times P$ kernels and image blocks where the output is of size $N \times N$ where $N = 2P - 1$. For section IV-B, we assume $B = 8$ bits for the input image pixels and $C = 12$ bits for the kernel coefficients. We use 12-bits for the outputs of the additions, multiplications, and the DPRT of section IV-B. We consider $C = 8$ bits for the kernel coefficients and full-precision for the outputs in the FPGA and SOC implementations of section IV-C. For the FFTr2, the computations are performed using 32-bit floating point units.

TABLE III

Comparison of the Performance of 2D Convolution and Cross-Correlations Architectures as a Function of Computational Resources. The Result Is of Size of $N \times N$, Where $N = 2P - 1$, P Represents the Input Image Size and Convolution Kernel Size, $n = \lceil \log_2 N \rceil$, $p = \lceil \log_2 P \rceil$, J Denotes the Number of Parallel 1D Circular Convolutions, and H Denotes the Number of Image Rows That Are Processed in Parallel by the DPRT. For ScaSys, P Needs to Be a Composite Number and It Is Assumed to be Given by $P = P_A \cdot P_B$. For FFTr2, $D = 2, 4$ Represents the Number of 1D FFT Units Running in Parallel. We Define: (i) $\mathtt{A_{ffb}}(a, b)$ to Be Number of Required Flip-Flops Inside the a-Operand of b Bits Adder Tree Including Input Buffers, (ii) $\mathtt{A_{ff}}()$ to Be the Same Number Without Accounting for Input Buffers, and (iii) $\mathtt{A_{FA}}()$ to Be the Equivalent Number of 1-bit Additions. To Interpret the Table, Note That $\mathtt{A_{ffb}}(.)$, $\mathtt{A_{ff}}(.)$, and $\mathtt{A_{FA}}(.)$ Grow Linearly as a Functions of N, and Can Be Computed Exactly Using the the Algorithm Given in the Appendix (Fig. 16). Instead of 12-bits, for the Number of Resources for a Different Number of Output Bits, Simply Replace 12 by the Desired Number of Bits

Method	Clock Cycles	Flip-flops (regs for FFTr2)	Additions	Multipliers	Memory
FastConv Proposed Fast Conv. (fixed point)	$6N + 5n + 17$	$(N+1)(36N + \mathtt{A_{ffb}}(N, 12))$ $+N(8N + \mathtt{A_{ff}}(N, 8))$ $+12N^2 + (N+1) \cdot \mathtt{A_{ff}}(N, 12)$ $+N(12+n)$	$(N+1) \cdot \mathtt{A_{FA}}(N, 12)$ $+N \cdot \mathtt{A_{FA}}(N, 8)$ $+(N+1)\mathtt{A_{FA}}(N, 12)$ $+N(12+n)$ 1-bit adds	$(N+1)N$ 12-bit fixed point mults	Ker: $12N(N{+}1)$
FastXCorr	Same as for *FastConv*. *FastXCorr* flips kernel prior to DPRT computations.				
FastScaleConv Proposed Scalable Convolution (fixed point)	$\lceil N/H \rceil (N + 3H + 3)$ $+N + \lceil \log_2 H \rceil + 1$ $+ \lceil (N+1)/J \rceil \cdot (J+N)$ $+n+1$ $+ \lceil N/H \rceil (N+H)$ $+2 \lceil \log_2 N \rceil + \lceil \log_2 H \rceil$ $+12+3$	$J \cdot (36N + \mathtt{A_{ffb}}(N, 12))$ $+N(8H + \mathtt{A_{ff}}(H, 8))$ $+12N(H+3)$ $+(N+1) \cdot \mathtt{A_{ff}}(H, 12)$	$J \cdot \mathtt{A_{FA}}(N, 12)$ $+N \cdot \mathtt{A_{FA}}(H, 8) + 12N$ $+(N+1) \cdot \mathtt{A_{FA}}(H, 12)$ $+2N(12+n)$ 1-bit adds	$J \cdot N$ 12-bit fixed point mults	$24N(N{+}1)$ Ker: $12N(N{+}1)$
FastScaleXCorr	Same as for *FastScaleConv*. *FastScaleXCorr* flips kernel prior to DPRT computations.				
FastRankConv Proposed Fast SVD-LU (fixed point)	$r \cdot (J+N)(\lceil P/J \rceil$ $+ \lceil N/J \rceil) + p + 1$	$J \cdot (36P + \mathtt{A_{ffb}}(P, 12))$	$J \cdot (\mathtt{A_{FA}}(P, 12) + 12)$ 1-bit adds	$J \cdot P$ 12-bit fixed point mults	$8P^2 +$ $12N(N +$ $P)$ Ker: $24P^2$
FastRankXCorr	Same as for *FastRankConv*. Kernel flipping implemented during pre-processing.				
SerSys [14] (fixed point)	$N^2 + 2P - 2$	$4P^3 + 34P^2 - 10P - 12$	$12P(P+1)$ 1-bit adds	P^2 12-bit fixed point mults	Ker: $12P^2$
ScaSys [15] (fixed point)	$\lceil N^2/P_A \rceil$ $+2P_A + P_B$ $+\lceil \log_2(P \cdot P_A) \rceil$	$P_A(20P^2 + \mathtt{A_{ffb}}(P_A P, 12))$ $+8P(P_A^2 + P_A - 1)$	$P_A(12\,P^2 +$ $\mathtt{A_{FA}}(P_A P, 12))$ 1-bit adds	$P_A \cdot P^2$ 12-bit fixed point mults	Ker: $12\,P_A \cdot P^2$
SliWin [25] (fixed point)	$N \cdot P + N^2$ $+2 \lceil \log_2 P \rceil + 1$	$20P^2 + \mathtt{A_{ffb}}(P^2, 12)$	$\mathtt{A_{FA}}(P^2, 12)$ 1-bit adds	P^2 12-bit fixed point mults	$8PN +$ $8P^2 +$ $12N^2$
FFTr2 [10] (32-bit floating point)	$(5N^2 + 4N)/D$	$D = 2:$ $(6N - 8)$ 32-bit registers. $D = 4:$ $(8N - 16)$ 32-bit registers.	$40D \cdot (\log_2 N + 1)$ 32-bit floating point adders.	$2D \cdot (1$ $+ \log_2 N)$ 32-bit float. point mults	$64N^2$ Ker: $32N^2$

For alternative image representations, we briefly refer to Table III and Fig. 16 that are covered in more detail in section IV-B. In Fig. 16 we provide an algorithm for computing the (i) number of flip-flops inside the adder tree and (ii) the equivalent number of 1-bit additions as functions of the number of bits per pixel D and the size of the final output image N. From Fig. 16, we can see that increasing the number of input bits from 8 to 24 will linearly increase the numbers of adder tree flip-flops and 1-bit additions but remain bounded above by $3\times$ the amounts presented here. In terms of the fixed-point multipliers, at 24-bits, we would also consider the use of highly-optimized, 32-bit floating point units. Furthermore, we note that there will be a minimal impact on running time. Overall, we note that our fixed-point implementations are most effective for the most commonly used, 8-bit inputs.

To enable comparisons between FFTr2 and fixed-point implementations, we make some simple, and realistic approximations for possible FPGA and SOC implementations. Based on Tables 2 and 3 from [27], in terms of 1-bit adders, we have that 32-bit floating point additions can be approximated as $10\times$ the cost of 32 1-bit fixed-point additions. Furthermore, to compare FFTr2 to all other implementations, we need to compare resources for 32-bit floating-point multiplication in terms of resources for 12-bit fixed-point multiplication. To provide realistic comparisons, we implemented complex, 32-bit floating point multiplications on the Virtex-7 (XC7VX1140). Without using DSPs, the multipliers require 650 LUTs and 32 flip-flops. When using DSPs, the multipliers require 34 LUTs, 32 flip-flops, and 2 DSPs. On the same FPGA, without using the DSPs, a 12-bit fixed point multiplier required just 148 LUTs. In terms of LUTs, we have: $650/148 \approx 4.4$. We thus approximate the cost of 32-bit floating point multiplications as equivalent to 4.4 times the cost of 12-bit fixed point multiplications.

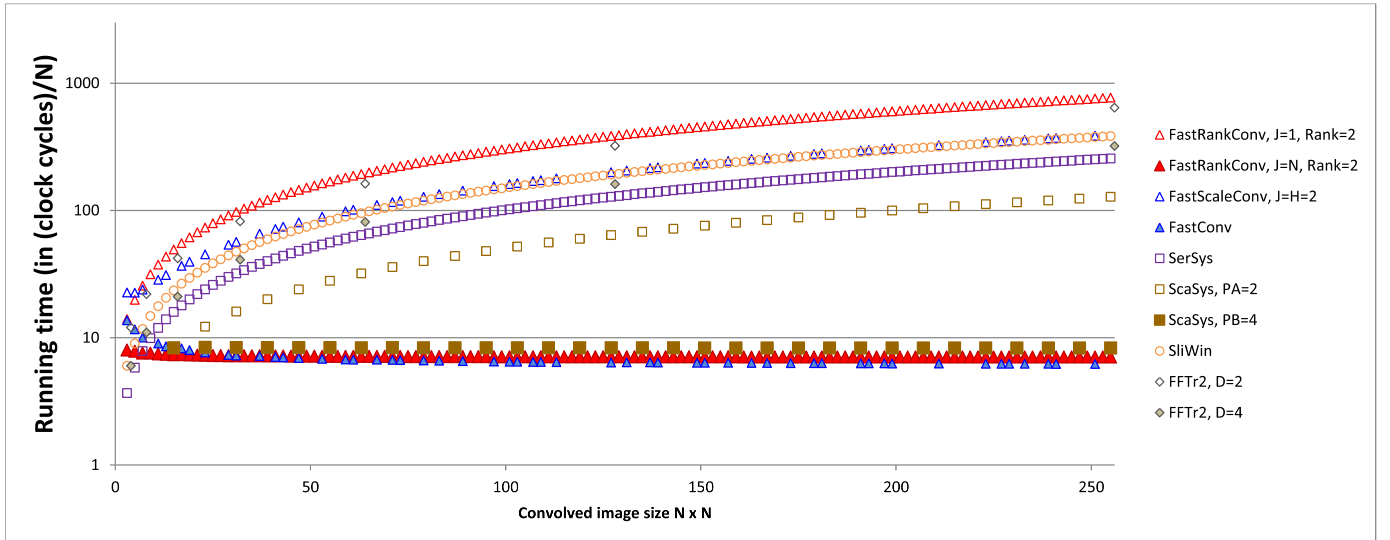

Fig. 13. Normalized running time in clock cycles versus output image (block) size (N). Here, the running time is the actual number of clock cycles divided by N. The plot refers to convolutions between $P \times P$ blocks ($N = 2P - 1$). For each implementation, as a function of N, we are assuming the use of the minimum amount of required resources. *FastConv* is the fastest followed closely by *FastRankConv* for rank=2 and $J = N$ (approximation). For $J = N + 1$ and $H = N$, *FastScaleConv* runs as fast as *FastConv* and thus, it is not repeated here. Methods with $O(N)$ execution times remain below 10 (implying $10N$ execution time). Methods with $O(N^2)$ execution times rise well above 10. Since $P = P_A \cdot P_B$, ScaSys implementations that achieve $O(N)$ execution time require $O(N^3)$ resources as opposed to $O(N^2)$ resources for the proposed methods (see Table III). Larger image sizes can also be handled using overlap-and-add.

For FFTr2, [10] does not provide detailed running time for the complete convolution of 2D images. To provide for fair comparisons, we consider an extension of FFTr2 using point-to-point multiplications using D 1D FFT cores. Then, in the fastest possible 2D implementation, we assume that it would take N^2/D additional clock cycles to implement the point to point complex multiplications.

As discussed earlier, for *FastScaleConv*, we achieve hardware scalability by varying H, the number of rows processed in parallel for the scalable DPRT, and J, which represents the number of 1D convolutions computed in parallel. Here, for $H = 2, 3, \ldots, N - 1$, we will simply set $J = H$ for a balanced approach towards both. Then, for $H = N$, we use $J = N + 1$ to provide the optimal solution using *FastConv*. For *FastRankConv*, we use r to denote the rank of the approximation.

There are some special restrictions on N. For the DPRT-based methods, N needs to be prime. For FFTr2, N is assumed to be a power of 2. For ScaSys, P needs to be a composite number ($N = 2P - 1$), and we thus assume that $P = P_A \cdot P_B$. We focus on the cases when $P_A = 2$ (slowest) and $P_B = 4$ (fastest), with an input buffer and fully pipelined additions. We do not include the case when $P_B = 2$ because the resource usage becomes prohibitive ($O(N^3)$). For SerSys, SliWin and *FastRankConv*, there is no restriction for P. When needed to change the size, we apply zero-padding.

B. MultiObjective Comparisons

A primary contribution of the manuscript is to provide convolution and cross-correlation architectures that are both fast and scalable. Because of scalability, for most reasonably-sized devices and convolution sizes, we can find possible implementations that can fit within it. On the other hand, there is great variability among different devices. Here, for an expanding range of P values, we show that our proposed architectures are optimal in the multi-objective sense. In other words, for any given level of fast performance (from $O(P)$ up to $O(P^2)$ clock cycles), the required architectures require fewer hardware resources.

In terms of transform size scalability, we note that there are several prime numbers between any two of powers of two. For example, from 4 to 256, we only have 7 possible sizes for FFTr2, compared to 53 prime numbers for *FastConv* and *FastScaleConv*, and no size restrictions for *FastRankConv*. As a result, for $P = 65$, $N = 129$ requires zero-padding to 256 for FFTr2 while it can be handled by a 131-sized DPRT associated with *FastConv* and *FastScaleConv*.

We present a comprehensive summary in terms of performance and resources in Table III. Table III lists performance in clock cycles, number of flip-flops, number of 1-bit additions (equivalent full-adders), number and type of multipliers, and SRAM requirements. The expressions in Table III are based on the derived running times and resources of section III using the bit-widths described in Sec. IV-A. The resources of Table III do not include control logic (e.g., for the Finite State Machine), or any logic needed for I/O interfacing. We will provide more details for FPGA and SOC implementations in section IV-C.

From Table III, we note the excellent performance of *FastConv*. When the amounts of required resources do not permit the full implementation of *FastConv*, *FastScaleConv* can be used to provide a trade-off between performance and required resources. Here, note that the complex expressions for *FastScaleConv* reduce to the ones for *FastConv* when

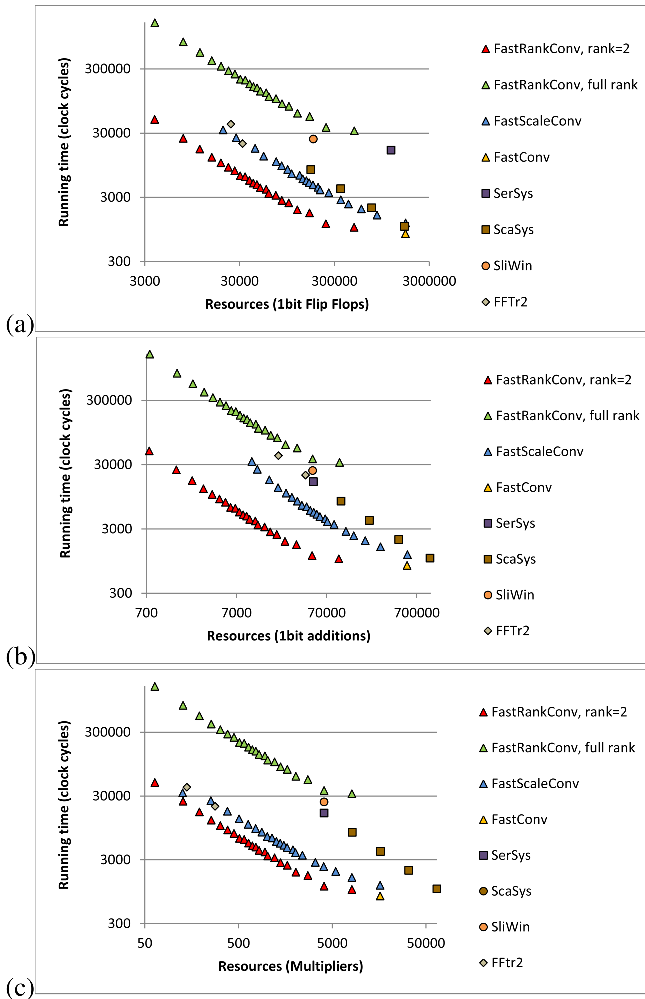

Fig. 14. Family of fast and scalable architectures for $N = 127$ ($N = 128$ for FFTr2). The plots refer to convolutions with 64×64 blocks ($P = 64$). For comparison purposes, we approximate the hardware cost for implementing 32-bit floating point additions and multiplications using fixed-point additions and 12-bit fixed-point multiplication as described in section IV-A. (a) Running time versus the required number of 1-bit flip-flops. (b) Running time versus the required number of 1-bit additions. (c) Running time versus the required number of equivalent multipliers.

FastScaleConv uses the maximum number of 1D convolvers ($J = N + 1$) and processes the maximum number of rows in the DPRT and iDPRT ($H = N$). Then, as we reduce H and J towards 1, the running performance grows from $O(P)$ to $O(P^2)$ clock cycles, and the numbers of resources are reduced from $O(P^2)$ to $O(P)$. Furthermore, to interpret the rest of the table, from $N = 2P-1$ we have that $P = (N+1)/2 = P_A \cdot P_B$, and note that $\mathtt{A_{ffb}}(.)$, $\mathtt{A_{ff}}(.)$, $\mathtt{A_{FA}}(.)$ grow linearly as functions of N. In what follows, we will further analyze performance in terms of running time (Fig. 13) and also provide multi-objective comparisons in terms of running time and required resources (Fig. 14).

Before we proceed with our analysis, we note that the results in Figs. 13 and 14 have been derived from Table III. On the other hand, we note that these results agree closely with our FPGA and SOC implementations that will be described in section IV-C. Furthermore, we are assuming that the data can be streamed to the FPGA or SOC at a rate that matches the computing rate. Here, we note that our assumption is not very restrictive since the bandwidth for a PCI express 3.x is about 16 GB/s and our FPGA or SOC implementations are running around 100MHz. However, for larger kernels, we may need custom hardware to stream data to the FPGA or SOC at high data rates.

We begin with a comparison of normalized execution times in Fig. 13. For Fig. 13, we divide the required number of clock cycles by N. To illustrate the range of possibilities, we consider the two extreme performance cases; architectures with quadratic and linear time complexity.

For quadratic time complexity ($O(N^2)$ clock cycles), we have scalable implementations derived by *FastScaleConv* for $J = H = 2$, and *FastRankConv* with $J = 1$, $r = 2$. Alternatively, we consider a scalable extension of FFTr2 for $D = 2, 4$, a scalable implementation of ScaSys, and the non-scalable implementations due to SliWin and SerSys.

FastConv provides the fastest performance at just $6N + 5n + 17$ clock cycles ($n = \log_2(N)$). For $J = N + 1$, *FastScaleConv* achieves the same performance as *FastConv*. For rank=2 approximations to the convolution kernel ($J = N$), *FastRankConv* approximates the performance of *FastConv*. In terms of related research, for $P_B = 4$, ScaSys achieves linear time-performance as well. On the other hand, from Table III, for linear time performance, using $P_A = P/P_B = P/4$, we can see that ScaSys's requirements grow as P^3 as opposed to P^2 growth for *FastConv* and *FastScaleConv*.

Due to the significant overhead associated with implementing floating point arithmetic for very large kernels, we do not consider the extreme case where N 1D FFTs can also yield linear performance (using a fast transposition, like the one we developed). However, we do note that for very large kernels, as given in Table III, the FFT based methods will most likely give the best results since the $N \log_2(N)$ growth in floating-point multipliers and additions will likely cost less than the N^2 growth of fixed-point multipliers and adders required by *FastConv*, *FastScaleConv*, and *FastRankConv*. We have certainly not found a study in the literature that demonstrated that such an approach was feasible. As we shall show next in our multi-objective comparisons, *FastConv*, *FastScaleConv*, and *FastRankConv* perform better than FFTr2 in realistic convolution kernels (e.g., for $N = 127$ and thus for lower N also). Furthermore, as we show later in this section, we can fit fast implementations of *FastConv*, *FastScaleConv*, and *FastRankConv* in current FPGAs and SOCs.

We provide detailed multi-objective comparisons in Fig. 14 for $N = 127$ ($N = 128$ for FFTr2). In Fig. 14 we show comparisons based for 1-bit FlipFlops (Fig. 14(a)), equivalent 1-bit Additions (Fig. 14(b)), and equivalent 12-bit fixed point Multipliers (Fig. 14(c)). Refer back to Table III for memory usage. To interpret the plots, note that each curve, (termed a Pareto front), represents a family of optimal implementations. The best results come from the Pareto fronts that are located in the lower-left. Within each Pareto front, the upper left point represents the implementation that requires the largest number of cycles (slowest) with the lowest number of required resources. Then, the lower-right point represent the implementation that requires the smallest number of cycles (fastest) with the maximum number of required resources. To enable more

TABLE IV

PERFORMANCE AND RESOURCE COMPARISONS FOR $N = 127$ (128 FOR FFTr2). HERE, WE HAVE CONVOLUTIONS BETWEEN 64×64 BLOCKS. FOR LINEAR-TIME IMPLEMENTATIONS, *FastConv* IS THE FASTEST AND SERVES AS THE REFERENCE DESIGN (ASSIGNED 1×). THE REMAINING IMPLEMENTATIONS ARE NORMALIZED BY THE CORRESPONDING RESOURCES REQUIRED BY *FastConv*. SIMILARLY, FOR QUADRATIC-TIME IMPLEMENTATIONS, *FastScaleConv* IS USED AS THE REFERENCE DESIGN. MEMORY REQUIREMENTS REFER TO SRAM BITS. ALSO, NOTE THAT THE REPORTED FFTr2 RESOURCES FOR ADDITIONS AND MULTIPLICATIONS REFER TO AN APPROXIMATION OF THE EQUIVALENT FIXED-POINT RESOURCES (REFER TO SECTION IV-A)

Implementations with linear running time, $J = 128$, $H = 127$, rank=2 (*FastRankConv*), $P_A = 16$ (ScaSys).

Method	Clock Cycles	Flip-flops	1-bit Additions	Multipliers	Memory
FastConv	810 (1×)	1687442 (1×)	548101 (1×)	16256 (1×)	195072 (1×)
FastRankConv	1023 (1.26×)	484632 (0.29×)	96012 (0.18×)	8128 (0.50×)	422156 (2.16×)
FastScaleConv	1195 (1.48×)	1689601 (1.00×)	552038 (1.01×)	16256 (1.00×)	585216 (1.39×)
ScaSys [15]	1054 (1.30×)	1645888 (0.98×)	982848 (1.79×)	65536 (4.03×)	786432 (4.03×)

Implementations with quadratic running time, $J = H = 4$, rank=2 (*FastRankConv*), $D = 4$ (FFTr2).

Method	Clock Cycles	Flip-flops	1-bit Additions	Multipliers	Memory
FastScaleConv	13093 (1×)	53888 (1×)	20309 (1×)	508 (1×)	585216 (1×)
FastRankConv	12583 (0.96×)	15264 (0.28×)	3024 (0.15×)	256 (0.50×)	422156 (0.72×)
SerSys [14]	16255 (1.24×)	1187188 (22×)	49908 (2.46×)	4096 (8.07×)	49152 (0.08×)
FFTr2 [10]	20608 (1.57×)	33256 (0.62×)	40960 (2.02×)	282 (0.56×)	1572864 (2.69×)
SliWin [25]	24270 (1.85×)	180212 (3.34×)	49140 (2.42×)	4096 (8.06×)	291340 (0.50×)

TABLE V

FULL-PRECISION IMPLEMENTATIONS OF *FastScaleConv* AND *FastConv* ON Zynq-SoC AND VIRTEX-7. EACH BRAM REPRESENTS UP TO 36 Kbits OF SRAM. EACH DSP REPRESENTS A MULTIPLIER. CLKS REFERS TO THE REQUIRED NUMBER OF CLOCK CYCLES AND BITS REFERS TO THE NUMBER OF BITS FOR REPRESENTING THE FINAL RESULT FOR 8-bit INPUTS AND 12-bit KERNELS. HERE, BITS $= B + C + n$ WHERE $B = C = 8$ bits AND $n = \lceil \log_2 N \rceil$. WE PRESENT A *FastConv* IMPLEMENTATION FOR $N = 37$ ON THE VIRTEX-7. IN TERMS OF RESOURCES, FOR THE ZYNQ-SOC (XC7Z100), WE HAVE 277400 LUTs, 755 BRAMs (36 Kb) AND 2020 DSPs. FOR THE VIRTEX-7 (XC7VX1140), WE HAVE 712000 LUTs, 1880 BRAMs (36 Kb), AND 3360 DSPs

Zynq-SoC							
N	**Bits**	**LUTs**	**BRAMs**	**DSPs**	**Clks**	**J**	**H**
7	25	4209	21	7	212	1	2
17	31	15166	68	17	799	1	2
41	34	47271	205	41	3817	1	2
41	34	225124	205	328	1094	8	8
97	37	202925	485	97	19811	1	2
109	37	239084	545	109	24869	1	2
113	37	241106	565	113	26683	1	2
Virtex-7							
N	**Bits**	**LUTs**	**BRAMs**	**DSPs**	**Clks**	**J**	**H**
7	25	4125	21	7	212	1	2
17	31	14950	68	17	799	1	2
37	***34***	***662352***	***185***	***1406***	***291***	***38***	**-**
41	34	46771	205	41	3817	1	2
41	34	617505	205	1312	658	32	32
127	37	321580	635	127	33536	1	2

TABLE VI

FULL-PRECISION AND SCALABLE FPGA IMPLEMENTATIONS OF *FastRankConv* (RANK=2) ON A VIRTEX-7 (XC7VX1140). THE RESOURCES REMAIN INDEPENDENT OF RANK SINCE WE ARE USING A FIXED A NUMBER OF BITS FOR ALL IMPLEMENTATIONS. RANK ONLY AFFECTS EXECUTION TIMES. REFER TO TABLE V FOR DEFINITIONS OF BRAM, DSPs, CLKS, AND RESOURCES FOR THE VIRTEX-7

P	Bits	LUTs	BRAMs	DSPs	Clks	J
7	31	1165	24	7	564	1
7	31	8859	24	49	124	7
17	35	3219	92	17	3406	1
17	35	59326	92	289	306	17
31	35	5737	169	31	11414	1
31	35	180645	169	961	558	31
41	37	8014	224	41	20015	1
41	37	353629	224	1681	739	41
53	37	9593	290	53	33503	1
53	37	513647	290	2809	955	53
61	37	14065	334	61	44415	1
67	39	16205	367	67	48903	1

direct comparisons, we also list specific numbers for some of the implementations in table IV.

Since they are the fastest, *FastConv* implementations are always in the lowest right portion in each plot. From table IV, we can see that *FastConv* only requires 25% of the multipliers and memory, and 56% of the addition resources required by ScaSys, while requiring only 77% of the clock-cycles. In terms of scalable approaches, the Pareto front for *FastRankConv* (rank=2), provide the best performance with minimum resources. The limited resources required by *FastRankConv* are also clearly documented in table IV. However, the use of rank=2 approximations may be inaccurate. On the other hand, the full-ranked *FastRankConv* requires the maximum amounts of resources to deliver the same performance.

Consistently, *FastScaleConv* provides the best scalable implementations without requiring low-rank. As seen from table IV, for the linear case, *FastScaleConv* is slightly more expensive in resources than *FastConv* and substantially less expensive than ScaSys. Overall, ScaSys ($P_B = 4$) implementations achieve the speed of *FastScaleConv* but require significantly more multipliers and adders. SerSys and SliWin require significantly more resources and are also much slower than *FastScaleConv*. Returning to table IV, for the quadratic case, *FastScaleConv* and *FastRankConv* (rank=2) are the fastest while requiring fewer adders and (equivalent) multipliers. In terms of memory requirements, our proposed scalable approaches do require more memory, still growing in the order of $O(N^2)$ which should not be a limitation with current technologies. On the other hand, with the exception of SliWin, only *FastConv*, *FastScaleConv* allow the kernel to change in running time. As a result, *FastConv* and *FastScaleConv* can also be used in cross-correlations with adaptive kernels, and adaptive filterbank applications.

TABLE VII

SELECTED FREQUENCIES (IN MEGAHERTZ) AND RUNNING TIME (IN MICROSECONDS) FOR FPGA IMPLEMENTATIONS USING VIRTEX-7 (XC7VX1140). REFER TO TABLES V AND VI FOR SETUP DETAILS

FastConv*, *FastScaleConv						
N	**J**	**H**	**f**	**RT**	**LUTs**	**DSPs**
7	1	2	112	1.9	4125	7
17	1	2	109	7.3	14950	17
37	38	-	113	2.6	662352	1406
41	1	2	105	36.4	46771	41
127	1	2	98	342.2	321580	127
FastRankConv						
P	**J**	**rank**	**f**	**RT**	**LUTs**	**DSPs**
7	1	2	109	5.2	1165	7
31	1	2	108	105.7	5737	31
53	1	2	108	310.2	9593	53
53	53	2	107	8.9	513647	2809

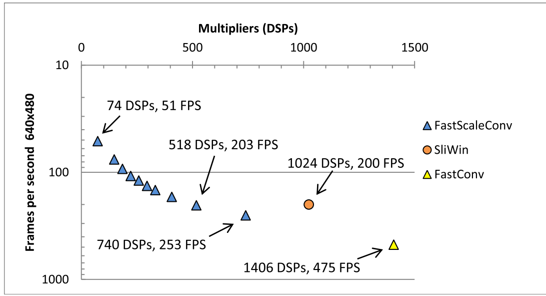

Fig. 15. Performance comparison between *SliWin* [25], *FastConv* and *FastScaleConv*. In terms of resources, we only count the number of DSPs which serves as a limiting factor for fitting the implementation in modern devices. To measure performance, we consider the number of Frames Per Second (FPS) to perform the convolution between an image of $480p$ (640×480) and a kernel of size 19×19. *SliWin* used a Stratix IV E530 with up to 1024 DSPs. *FastConv* and *FastScaleConv* used a Virtex-7 XC7VX1140, with up to 3360 DSPs and overlap-and-add with different block sizes. We note that it is fair to assume that both DSPs are equivalent, and expect that their amounts will not change with bit-widths for inputs up to 18 bits. At 200 FPS, *FastScaleConv* uses approximately 50% less DSPs than *SliWin*. On the other hand, *FastConv* is 2.4 times faster with just 40% more multipliers.

1: **procedure** $\texttt{Tree_Resources_WIB}(N, D)$
2: $\quad n = \lceil \log_2 N \rceil$
3: $\quad \texttt{A}_{\texttt{ffb}} = \texttt{A}_{\texttt{FA}} = 0$
4: $\quad a = N$
5: $\quad$ **for** $z = 1$ to n **do**
6: $\quad\quad r = \langle a \rangle_2$
7: $\quad\quad a = \lfloor a/2 \rfloor$
8: $\quad\quad \texttt{A}_{\texttt{FA}} = \texttt{A}_{\texttt{FA}} + a \cdot (D + z - 1)$
9: $\quad\quad a = a + r$
10: $\quad\quad \texttt{A}_{\texttt{ffb}} = \texttt{A}_{\texttt{ffb}} + a \cdot (D + z)$
11: $\quad$ **end for**
12: $\quad \texttt{A}_{\texttt{ffb}} = \texttt{A}_{\texttt{ffb}} + X \cdot D$ $\quad \triangleright$ With Input Buffers (WIB)
13: $\quad$ **return** $\texttt{A}_{\texttt{FA}}, \texttt{A}_{\texttt{ffb}}$
14: **end procedure**

Fig. 16. Required tree resources as a function of the zero padded image (N), and the number of bits per pixel (D). Refer to Table III for definitions of $\texttt{A}_{\texttt{ffb}}, \texttt{A}_{\texttt{FA}}$. To compute $\texttt{A}_{\texttt{ff}}$ for architectures that do not use input buffers, simply remove step 12 from the algorithm.

TABLE VIII

RESOURCE USAGE FOR DIFFERENT 1D CIRCULAR CONVOLUTIONS IMPLEMENTATIONS. HERE, WE HAVE TWO ZERO-PADDED IMAGES (OR IMAGE BLOCKS) g AND h OF SIZE $N \times N$ (INCLUDING ZERO-PADDED PIXELS), B AND C BITS PER PIXEL RESPECTIVELY AND $n = \lceil \log_2 N \rceil$. FOR THE ADDER TREE, WE DEFINE $\texttt{A}_{\texttt{ffb}}$ TO BE THE NUMBER OF REQUIRED FLIP-FLOPS INCLUDING INPUT BUFFERS, AND $\texttt{A}_{\texttt{FA}}$ TO BE THE NUMBER OF 1-bit ADDITIONS. $\texttt{A}_{\texttt{ffb}}$ AND $\texttt{A}_{\texttt{FA}}$ GROW LINEARLY WITH RESPECT TO N AND CAN BE COMPUTED USING THE ALGORITHM GIVEN IN FIG. 16. FOR THE MULTIPLIERS, WE NOTE THAT EACH ONE IS IMPLEMENTED USING TWO INPUTS OF SIZE $B + n$ AND $C + n$ BITS AND AN OUTPUT OF $B + C + 2n$ BITS. HERE, WE USE THE TERM "1-bit ADDITIONS" TO REFER TO THE NUMBER OF EQUIVALENT 1-bit FULL ADDERS

Block	Number of flip-flops	1-bit additions	Multipliers
Core	$N(2B + 2C + 5n)$ $+\texttt{A}_{\texttt{ffb}}(N, B + C + 2n)$	$\texttt{A}_{\texttt{FA}}(N, B + C + 2n)$	N
System	$J \cdot N(2B + 2C + 5n)$ $+J\texttt{A}_{\texttt{ffb}}(N, B + C + 2n)$	$J \cdot \texttt{A}_{\texttt{FA}}(N, B + C + 2n)$	$J \cdot N$

C. Full-Precision FPGA and SOC Implementations

In order to understand what can be fitted in modern devices, we consider full-precision implementations for 8-bit inputs and 12-bit kernels using the on-chip memory. Here, we assume that the image and convolution blocks are stored in the BRAMs inside the FPGA or SOC. For overlap-and-add implementations, we assume that the images will be stored in external memory and be transferred to the FPGA or SOC for processing. Here, we are not including any additional delays due to transferring the image from external memory to the FPGA or SOC. The proposed systems were implemented using current FPGA and SOC technologies (Virtex-7 and Zynq-SOC). For *FastScaleConv* and *FastConv*, for different N and J (the number of parallel 1D convolvers), we show different implementations in table V. For *FastRankConv*, by varying P and J, we present different implementations in table VI.

A collection of *FastScaleConv* architectures were successfully implemented for $N = 7$ to $N = 127$. For $N = 41$, a high-level of parallelism was achieved by computing the DPRT and inverse DPRT by parallel-processing $H = 32$ rows at a time through $J = 32$ 1D full-precision, pipelined convolvers also operating in parallel. In our full-precision example, the output images required 34 bits. For $N = 37$, we have a full precision implementation of *FastConv* that only requires 291 clock cycles by parallel processing 38 rows of the DPRT and inverse DPRT, and parallel computing 38 1D convolutions. From table V, we can see that implementations are limited by the number of available look-up tables. Thus, it is clear that larger values of N can be implemented by reducing the precision requirements.

As shown in table VI, *FastRankConv* makes a very efficient use of the DSPs while not requiring significant LUT resources. For example, for $P = 67$, *FastRankConv* only requires 16205 LUTs (out of 712000 LUTs). In comparison, *FastScaleConv* requirements for $N = 127$ (which approximates $2P - 1$), requires about 20 times more LUTs to deliver the full-accuracy results. Also for $P = 67$, *FastRankConv* with rank $r = 2$

TABLE IX

RESOURCE USAGE FOR DIFFERENT LINEAR CONVOLVERS IMPLEMENTATIONS. HERE, ALL THE QUANTITIES ARE GIVEN FOR MAXIMUM ACCURACY. REFER TO THE CAPTION OF TABLE VIII AND SECTION II-A FOR THE NOTATION

Block	Number of flip-flops	1-bit additions	Multipliers
Core	$N2 \cdot (B + C + q2) + Q2 \cdot C$ $+ \mathtt{A_{ffb}}(Q2, B + 2C + q2)$	$\mathtt{A_{FA}}(Q2, B + 2C + q2)$	$Q2$
System	$J \cdot (N2 \cdot (B + C + q2) + Q2 \cdot C$ $+ \mathtt{A_{ffb}}(Q2, B + 2C + q2))$	$J \cdot \mathtt{A_{FA}}(Q2, B + 2C + q2)$	$J \cdot Q2$

requires 48903 clock cycles, compared to 33507 clock cycles for *FastScaleConv* with $J = 1$ and $H = 2$ without any rank restrictions. Thus, as seen earlier, for low-rank kernels, *FastRankConv* is a good alternative to *FastScaleConv*. For higher ranks and general-purpose implementations, *FastScaleConv* is more preferable.

We also compare the results of the actual FPGA and SOC implementations of Tables V, VI, and VII with the predicted measurements of Table III. In terms of clock cycles, there is very little difference. Here, we note that, as described in section III, the number of clock cycles includes the number of cycles needed to load each image block row-by-row, processing, and final delay until the output is stored in the output memory. In terms of resources, we have found exact matches for multipliers (mapped to DSPs units), flip-flops and SRAM (mapped into BRAMs). In terms of the adders, we do have some additional combinational logic that is captured in the number of the LUTs. The additional logic is due to the use of muxes in the DPRT as reported in [12]. We note that the additional overhead due to the finite state machine (FSM) and ancillary logic did not exceed 1% of the total resources.

We provide running times for different implementations in Table VII. All of our implementations achieve a maximum frequency around 100MHz. For larger N, we have a slight decrease in frequency for the implementation of *FastScaleConv*. The decrease is due to an increase in propagation time for the larger muxes used in the DPRT blocks (see [12]). For $N = 37$ and $J = 38$, we have the highest frequency for *FastConv*. The highest frequency is due to the fact that the Fast DPRT component of *FastConv* is a simplified version of the scalable DPRT component of *FastScaleConv* [12]. The clock frequencies for *FastRankConv* remain fairly constant.

We also provide comparisons against implementations by alternative methods. To provide better comparisons, we compare the performance as a function of the number of DSPs that are required. Based on [25], we provide an optimized implementation for *SliWin*. The comparison is given in Fig.15 for images of size 640×480. From Fig. 15, we see that *FastConv* remains the fastest by far (2.3× the *SliWin* example). Furthermore, it is clear that *FastScaleConv* provides a nice Pareto front with several optimal implementations as a function of the number of available DSPs. Similar to *SliWin*, *FastScaleConv* with $H = 13$, $J = 14$ achieves around 200 FPS. However, for this implementation, *FastScaleConv* uses approximately 50% less DSPs.

By using the scalability of the proposed architectures, we can adjust H and J to ensure that the architecture fits into a given chip and also ensure that memory bandwidth requirements do not exceed what is available to us. The basic idea is to provide the proposed system with new convolution blocks after the required number of cycles. Fortunately, there is no bandwidth issue here. The required bandwidth is $O(J \cdot f)$ where f denotes the operating frequency. For $J = 2$ we have the slowest case and the required bandwidth that is $O(f)$. Based on our discussion, we use $J \approx P$ to require the highest bandwidth of $O(P \times f)$, where $P \times P$ is the block size. For instance, for a video of size 640×480 at 30 FPS, for $P = 19$, $J = H = 2$ (minimum resource usage), $f = 110MHz$, each frame is processed in 19ms (within the 33ms to achieve the 30FPS), and the required bandwidth is just $9.2MB/s$, which can be easily achieved. Thus, as for all 2D convolution methods, as the size of the block increases, our approach tends to be compute-bound.

V. CONCLUSIONS

The manuscript introduced fast and scalable architectures for computing 2D cross-correlations and convolutions. *FastConv* architectures deliver the best performance by computing convolutions in $O(P)$ clock cycles. The *FastScaleConv* family of architectures allows us to implement efficient architectures that can be restricted to the architectures of different devices. The *FastRankConv* family of architectures allows us to consider low-rank approximations that can significantly reduce the number of required resources. Overall, for the same level of performance, *FastRankConv* and *FastScaleConv* require significantly fewer hardware resources than alternative approaches.

REFERENCES

[1] A. C. Bovik, *The Essential Guide to Image Processing*. San Diego, CA, USA: Academic, 2009.

[2] A. C. Bovik, *The Essential Guide to Video Processing*. San Diego, CA, USA: Academic, 2009.

[3] M. S. Nixon and A. S. Aguado, *Feature Extraction and Image Processing for Computer Vision*, 3rd ed. London, U.K.: Academic, 2012.

[4] M. A. Anam and Y. Andreopoulos, "Throughput scaling of convolution for error-tolerant multimedia applications," *IEEE Trans. Multimedia*, vol. 14, no. 3, pp. 797–804, Jun. 2012.

[5] M. A. Anam, P. N. Whatmough, and Y. Andreopoulos, "Precision-energy-throughput scaling of generic matrix multiplication and convolution kernels via linear projections," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 24, no. 11, pp. 1860–1873, Nov. 2014.

[6] D. E. Dudgeon and R. M. Mersereau, *Multidimensional Digital Signal Processing*. Upper Saddle River, NJ, USA: Prentice-Hall, 1990.

[7] K. R. Rao, D.N. Kim, and J. J., Hwang, *Fast Fourier Transform-Algorithms and Applications*. New York, NY, USA: Springer Science & Business Media, 2011.

[8] I. S. Uzun, A. Amira, and A. Bouridane, "FPGA implementations of fast Fourier transforms for real-time signal and image processing," *IEE Proc.-Vis., Image Signal Process.*, vol. 152, no. 3, pp. 283–296, Jun. 2005.

[9] Xilinx. (Jul. 2012). *Logicore IP, Fast Fourier Transform V8.0*. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ds808 xfft.pdf

[10] M. Ayinala, M. Brown, and K. K. Parhi, "Pipelined parallel FFT architectures via folding transformation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 6, pp. 1068–1081, Jun. 2012.

[11] C. Carranza, D. Llamocca, and M. Pattichis, "The fast discrete periodic radon transform for prime sized images: Algorithm, architecture, and VLSI/FPGA implementation," in *Proc. IEEE Southwest Symp. Image Anal. Interpretation (SSIAI)*, Apr. 2014, pp. 169–172.

[12] C. Carranza, D. Llamocca, and M. Pattichis, "Fast and scalable computation of the forward and inverse discrete periodic radon transform," *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 119–133, Jan. 2016.
[13] H. T. Kung, "Why systolic architecture?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982.
[14] H.-K. Kwan and T. S. Okullo-Oballa, "2-D systolic arrays for realization of 2-D convolution," *IEEE Trans. Circuits Syst.*, vol. 37, no. 2, pp. 233–267, Feb. 1990.
[15] B. K. Mohanty and P. K. Meher, "Cost-effective novel flexible cell-level systolic architecture for high throughput implementation of 2-D FIR filters," *IEEE Proc.-Comput. Digit. Techn.*, vol. 143, no. 6, pp. 436–439, Nov. 1996.
[16] Y. Dong, Y. Dou, and J. Zhou, "Optimized generation of memory structure in compiling window operations onto reconfigurable hardware," in *International Workshop on Applied Reconfigurable Computing* (Reconfigurable Computing: Architectures, Tools and Applications). Berlin, Germany: Springer, 2007, pp. 110–121.
[17] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Dec. 2012, pp. 47–56.
[18] W.-S. Lu, H.-P. Wang, and A. Antoniou, "Design of two-dimensional FIR digital filters by using the singular-value decomposition," *IEEE Trans. Circuits Syst.*, vol. 37, no. 1, pp. 35–46, Jan. 1990.
[19] A. Antoniou, *Digital Signal Processing: Signals, Systems, and Filters*. New York, NY, USA: McGraw-Hill, 2005.
[20] D. Llamocca, C. Carranza, and M. Pattichis, "Separable fir filtering in fpga and gpu implementations: Energy, performance, and accuracy considerations," in *Proc. 11th Int. Conf. Field Program. Logic Appl. (FPL)*, Chania, Greece, Sep. 2011, pp. 363–368.
[21] T. Hsung, D. P. K. Lun, and W.-C. Siu, "The discrete periodic Radon transform," *IEEE Trans. Signal Process.*, vol. 44, no. 10, pp. 2651–2657, Oct. 1996.
[22] A. Kingston and D. S. Imants, "Projective transforms on periodic discrete image arrays," *Adv. Imag. Electron Phys.*, vol. 139, pp. 75–177, Dec. 2006.
[23] D. P. K. Lun, T.-C. Hsung, and W. C. Siu, "On the convolution property of a new discrete radon transform and its efficient inversion algorithm," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, vol. 3. Seattle, WA, USA, Apr. 1995, pp. 1892–1895.
[24] D. Llamocca and M. Pattichis, "Dynamic energy, performance, and accuracy optimization and management using automatically generated constraints for separable 2D fir filtering for digital video processing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, pp. 31:1–31:30, Dec. 2014. [Online]. Available: http://doi.acm.org/10.1145/2629623
[25] P. Cooke, J. Fowers, G. Brown, and G. Stitt, "A tradeoff analysis of fpgas, gpus, and multicores for sliding-window applications," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 1, pp. 2:1–2:24, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2659000
[26] P. K. Meher, S. Chandrasekaran, and A. Amira, "FPGA realization of FIR filters by efficient and flexible systolization using distributed arithmetic," *IEEE Trans. Signal Process.*, vol. 56, no. 7, pp. 3009–3017, Jul. 2008.
[27] A. Vera, M. Pattichis, and J. Lyke, "A dynamic dual fixed-point arithmetic architecture for FPGAs," *Int. J. Reconfigurable Computing*, vol. 2011, Jan. 2011, Art. no. 518602. [Online]. Available: http://ece.unm.edu/ivpcl/Publications/JOURNALS/2011/A%20Dynamic%20Dual%20Fixed%20Point%20Arithmetic%20Architecture%20for%20FPGA.pdf

Cesar Carranza received the B.Sc. degree in electrical engineering from Pontificia Universidad Católica del Perú in 1994, the M.Sc. degree in computer science from the Centro de Investigación Científica y de Educación Superior de Ensenada in 2010, and the M.Sc. degree in computer engineering and the Ph.D. degree (Hons.) in engineering from The University of New Mexico, Albuquerque, in 2016 and 2012, respectively. He is currently an Associate Professor with the Pontificia Universidad Católica del Perú. His current research interests include parallel algorithms for image processing, high performance hardware integration, and parallel computing.

Daniel Llamocca received the B.Sc. degree in electrical engineering from the Pontifical Catholic University of Peru in 2002, and the Ph.D. degree in computer engineering and the M.Sc. degree in electrical engineering from The University of New Mexico, Albuquerque, in 2012 and 2008, respectively.

He is currently an Assistant Professor with Oakland University. His research deals with the run-time automatic adaptation of hardware resources to timevarying constraints with the purpose of delivering the best hardware solution at any time. His current research interests include: reconfigurable computer architectures for signal, image, and video processing; high-performance architectures for computer arithmetic, communication, and embedded interfaces; embedded system design; and run-time partial reconfiguration techniques on FPGAs.

Marios Pattichis (M'99–SM'06) received the B.Sc. degree (Hons.) in computer science, the B.A. degree (Hons.) in mathematics, the M.S. degree in electrical engineering, and the Ph.D. degree in computer engineering from The University of Texas at Austin, in 1991, 1991, 1993, and 1998, respectively. He is currently a Professor with the Department of Electrical and Computer Engineering, The University of New Mexico (UNM), Albuquerque. His current research interests include digital image, video processing, communications, dynamically reconfigurable computer architectures, and biomedical and space image-processing applications.

Dr. Pattichis is currently a Senior Associate Editor of the IEEE SIGNAL PROCESSING LETTERS. He has been an Associate Editor of the IEEE TRANSACTIONS ON IMAGE PROCESSING, the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, and has also served as a Guest Associate Editor of the IEEE TRANSACTIONS ON INFORMATION TECHNOLOGY IN BIOMEDICINE. He was the General Chair at the 2008 IEEE Southwest Symposium on Image Analysis and Interpretation. He was a recipient of the 2016 Lawton-Ellis and the 2004 distinguished teaching awards from the Department of Electrical and Computer Engineering, UNM. For his development of the digital logic design laboratories at UNM, he was recognized by the Xilinx Corporation in 2003 and by the UNM School of Engineering's Harrison Faculty Excellence Award in 2006. He was a founding Co-PI of the Configurable Space Microsystems Innovations & Applications Center (COSMIAC) at UNM, where he is currently the Director of the Image and Video Processing and Communications Laboratory.