IEEE Open Journal of
Circuits and Systems

Received 29 June 2021; revised 25 September 2021; accepted 8 October 2021. Date of current version 9 December 2021.

*Digital Object Identifier 10.1109/OJCAS.2021.3123899*

# Implementing Convolutional Neural Networks Using Hartley Stochastic Computing With Adaptive Rate Feature Map Compression

**S. H. MOZAFARI (Student Member, IEEE), J. J. CLARK (Senior Member, IEEE), W. J. GROSS (Senior Member, IEEE), AND B. H. MEYER (Senior Member, IEEE)**

Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 0G4, Canada

This article was recommended by Associate Editor A. Bermak.

CORRESPONDING AUTHOR: S. H. MOZAFARI (e-mail: seyyed.mozafari@mail.mcgill.ca)

This work was supported in part by the Huawei Technologies Canada Inc., through the McGill Edge Intelligence Lab, and in part by FRQNT Postdoctoral Research Scholarship (B3X).

**ABSTRACT** Energy consumption and the latency of convolutional neural networks (CNNs) are two important factors that limit their applications specifically for embedded devices. Fourier-based frequency domain (FD) convolution is a promising low-cost alternative to conventional implementations in the spatial domain (SD) for CNNs. FD convolution performs its operation with point-wise multiplications. However, in CNNs, the overhead for the Fourier-based FD-convolution surpasses its computational saving for small filter sizes. In this work, we propose to implement convolutional layers in the FD using the Hartley transform (HT) instead of the Fourier transformation. We show that the HT can reduce the convolution delay and energy consumption even for small filters. With the HT of parameters, we replace convolution with point-wise multiplications. HT lets us compress input feature maps, in convolutional layers, before convolving them with filters. In this regard, we introduce two compression techniques: fixed-rate and adaptive-rate. In the fixed-rate compression, we select frequency domain input feature map (IFMap) coefficients with a constant pattern over all convolutional layers. However, for the adaptive-rate IFMap compression, the network, itself, learns to keep or discard coefficients, during training. Also, to optimize the hardware implementation of our methods (fixed- and adaptive-rate compressions), we utilize stochastic computing (SC) to perform the point-wise multiplications in the FD. In this regard, we re-formalize the HT to better match with SC. We show that, compared to conventional Fourier-based convolution, Hartley SC-based convolution can achieve 1.33x speedup, and energy is reduced by 23% on a Virtex 7 FPGA when we implement AlexNet over CIFAR-10 based on the fixed-rate compression. Also, we show that if we utilize the adaptive-rate compression, we receive 16% and 15% latency improvement and energy consumption reduction, respectively, compared to the fixed-rate method.

**INDEX TERMS** Deep neural networks, frequency domain transformation, hardware implementation, energy optimization, latency improvement, FPGA.

## I. INTRODUCTION

A WIDE range of applications, from convolutional neural networks (CNNs) to image compression tasks, utilize convolution. Convolution, due to its iterative matrix multiplications in the spatial domain (SD), is the most computation and energy hungry operation in vision related tasks [1]. For example, as you can see in Figure 1, more than 98% of the operations in LeNet, and AlexNet are in convolutional layers [2]. Consequently, researchers have introduced accelerators in the spatial and frequency domains, from application-specific hardware to GPUs, to optimize the latency and energy efficiency of convolution [3], [4].

Efficient convolution in the frequency domain (FD) is of particular interest. Researchers indicate, however, that for small filter sizes the cost of transformations (spatial to frequency, and vice versa) overwhelms the savings when implementing convolution in the FD [2], [4], [5]. Subsequently, separate methods have been developed for

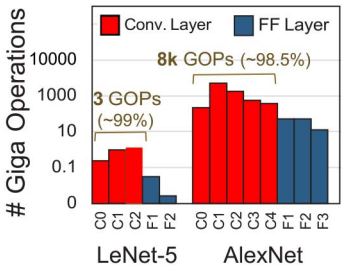


**FIGURE 1.** Number of operations in the different convolutional and fully connected layers of LeNet and AlexNet.

small and large filters. Winograd convolution [6] has been introduced to reduce the complexity of convolution for small filter sizes, while the Fast Fourier transform (FFT) addresses large filter sizes better. FFT-based convolution works with larger filter sizes better since the overhead that FFT has for the required domain transformations in convolution (FD→SD, and SD→FD) surpasses its computational saving for small filters [7]. To optimize FFT, researchers trade performance (latency) and area with the accuracy of complex multiplications [8]. However, such aggressive approximations limit the applications of these methods, and therefore, they have not been used widely in CNNs.

Like [9], [10], we propose to optimize convolution in CNNs by employing the Hartley transform (HT). However, we also (a) utilize the HT to compress CNN input feature maps (IFMaps), for all convolutional layers, improving CNN latency; and, (b) reformulating HT to target stochastic computing, further reducing CNN FPGA resource utilization and energy. We show that our proposed method can optimize CNNs even with small filter sizes and overcome the overhead of domain transformations required for FD convolution. To the best of our knowledge, our work is the first that utilizes the HT and SC to optimize CNNs with different filter sizes.

The Hartley transform [11] was developed to keep the main advantages of FFT, such as performing convolution based on point-wise multiplications, but without the addition complexity of working with imaginary numbers. The trade-off is that HT has been observed to require more computation than FFT.

One key advantage of our approach is that convolution can be performed directly on the HT compressed IFMaps. Frequency domain transformations have been widely used for image compression (such as JPEG) [12], and recent work has compressed data in the FD to optimize CNNs (e.g., [13]). Unfortunately, the FD transformations popular in compression (e.g., the cosine transform for JPEG) are not directly compatible with convolution [7] (they need extra processing for the transformed data before being used in the convolution).

Additionally, IFMap compression rate directly affects the speed of a CNN by reducing the number of operations, but it degrades accuracy [14]. To increase compression without reducing accuracy considerably, we can learn to drop which IFMap coefficients for convolutional layers besides learning weights in a neural network to optimize its latency and energy [15]. Many techniques have been developed that focus on reducing the number of weights of neural networks, at the possible expense of increased training cost [16], [17]. However, our method learns the weights for the FD transformed IFMap to optimize a CNN's speed and maintain its accuracy using the same loss function and gradient decent that are used for training the CNN.

A second key advantage of our approach is its suitability for use with stochastic computing. Stochastic computing [18] has also been used in the past to design low cost and energy CNN accelerators [19], [20]. However, SC-based operations suffer from two important problems: 1) low accuracy, and 2) long computation time [21]. Prior work has shown that CNNs can tolerate the low accuracy of SC [22]. Unfortunately, if we implement FFT-based convolution with SC [8], output feature map (OFMap) accuracy drops to the point that CNNs cannot learn from them, and SC latency overhead surpasses the speed-up gain from implementing convolution in the FD. To prevent such an accuracy drop, we utilize SC just for a part of FD convolution, point-wise matrix multiplications, and perform the rest via fixed-point operations.

The key contributions of this paper are:

- We introduce a stochastic based implementation of convolution based on the Hartley transform (HT).
- We introduce a fixed-rate HT compression method to approximate the IFMaps (in all CNN convolutional layers), reducing convolution latency and energy without degrading CNN accuracy.
- We introduce an adaptive-rate optimization for the HT-based compression technique to further improve latency and energy. To the best of our knowledge, we are the first to integrate this optimization into the training procedure, compressing IFMaps in each layer (at different rates, which these rates are chosen automatically by the neural network to optimize global accuracy and inference delay).
- We introduce a re-formalization of the HT that better fits with hardware, relying on a scaled, fixed-point look-up-table-based implementation that is well-suited to SC.

We demonstrate the benefits of our approach with HSC-CNN, a SC-based convolution accelerator that can be utilized for CNNs' inference. We use LeNet and AlexNet on MNIST and CIFAR-10, respectively, to validate our method against the spatial domain in terms of accuracy. We subsequently synthesize designs for FPGA for performance (latency) and energy analysis. Our results show that HSC-CNN significantly reduces inference time and energy compared with spatial- and FFT-based convolution even. We observe 1.21x and 1.28x latency improvement for HSC-CNN based on fixed-rate and adaptive-rate IFMap compression, respectively, compared to spatial LeNet with small filters (3x3) over MNIST without loss of accuracy, while FFT-convolution based LeNet performs 1.09x worst than spatial-LeNet.

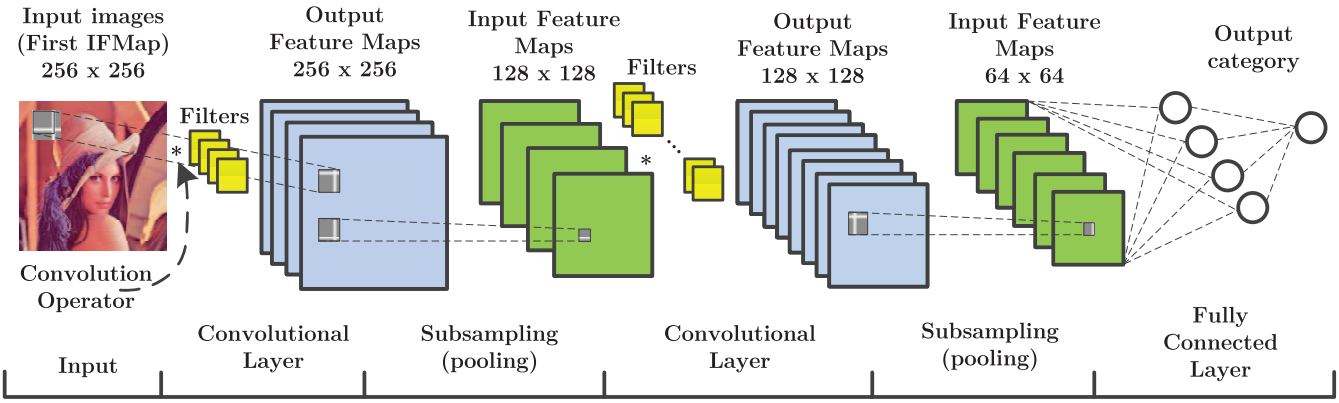


**FIGURE 2.** The architecture of an example convolutional neural network. An input image (first IFMap) is convolved with filters. Then, the result (OFMap) is sub-sampled using the pooling operation. This process repeats multiple times until the probability of input image belongs to a category is determined using a fully connected final layer.

## II. MOTIVATION AND BACKGROUND

### A. CONVOLUTIONAL NEURAL NETWORKS

CNNs are widely used in image and video processing [6]. As shown in Figure 2, a CNN consists of some convolutional, sub-sampling, and fully connected layers. CNNs gradually extract local features from feature maps of higher resolutions of images, and then they combine these features into more abstract feature maps with lower resolutions. This is done by alternating two types of layers: convolutional and sub-sampling. In convolutional layers, input feature maps (IFMaps) are convolved with filters. In sub-sampling layers pooling function is utilized to reduce the dimentionality of convolution's output. Also, note that a non-linear activation function could be applied on output feature maps (OFMaps) before pooling to add non-linear features to the network. The last few layers in the CNN use fully connected layers for the purpose of classification.

### B. CONVOLUTION

The convolution operation, in a convolutional layer of a CNN, can be represented with an input tensor $I \in \mathbb{R}^{c_{in} \times w_{in} \times h_{in}}$, and convolution filter weights $F \in \mathbb{R}^{c_{in} \times d \times d}$, where each filter is $d \times d$. $c_{in}$ represents the number of input channels and input feature maps. Each filter $c_{in}$ is convolved over the corresponding $w_{in} \times h_{in}$ input matrix, in the spatial domain (SD), as depicted at the top of Figure 3.

The convolution theorem [7] states that the dual of convolution in the SD is point-wise multiplication in the FD. Letting $H$ denote the Hartley transform, and $H^{-1}$ its inverse, we can compute convolutions between functions $I$ and $F$ as follows [11]:

$$I * F = H_{2\pi}^{-1}\left(\frac{\sqrt{2\pi}}{2} \odot H_{2\pi}(I) \odot \left[H_{2\pi}(F) + H_{-2\pi}(F)\right] + H_{-2\pi}(I) \odot \left[H_{2\pi}(F) - H_{-2\pi}(F)\right]\right), \quad (1)$$

where $\odot$ is element-wise-multiplication, and in $H_\omega$, $\omega$ is the HT angular frequency. Note that, as in the FFT, before utilizing the HT, we should make the size of IFMaps and filters equal and a power of two since these transformations work on square input matrices and are optimized when the size of matrices are a power of two [5]. In this regard, we add zero padding as depicted in Figure 3.

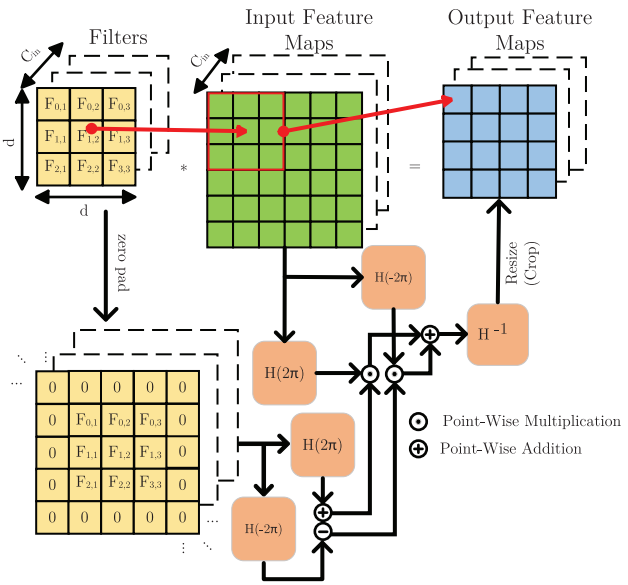


**FIGURE 3.** Convolution in the spatial- and frequency-domain (using the HT). Note that convolution operation is required in each convolution layer of CNNs.

The discrete Hartley-transform (DHT) for $N \times N$ matrix is

$$H_{\omega=2\pi}(k,l) = \frac{1}{N}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} f[m,n]cas\left(\frac{2\pi}{N}(km+ln)\right), \quad (2)$$

where $cas(x) = cos(x) + sin(x)$ [11]. As shown in Eq. (2), each element of an input matrix is multiplied with two coefficients. Not all of these coefficients are unique: $H(k+N, l+N) = H(k,l)$ as $cas(n \times 2\pi) = 1$. To implement the HT, we utilize the fast Hartley transform (FHT) which uses this symmetry to reduce the number of additions and multiplications compared to the DHT [23]. Also, note that

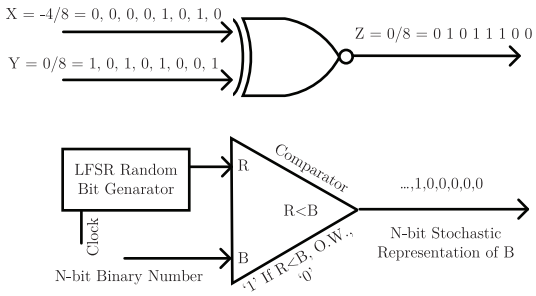


**FIGURE 4.** Stochastic multiplication and stochastic number generator.

the same transformation (Eq. (2)) is used for $H^{-1}$: we just need to substitute $f[m, n]$ with $H(k, l)$ in the formula [23].

### C. COMPRESSING IFMAPS

Natural images have a lot of redundancy, and therefore can be downsampled in the FD without affecting vision related tasks' accuracy such as image classification [24]. Moreover, it has been shown that IFMaps, which have properties similar to images, carry this redundancy throughout the first few layers in CNNs [25]. We hypothesize that this redundancy can be reduced without significantly increasing CNN error, while dramatically reducing CNN latency.

To evaluate this hypothesis we trained AlexNet over CIFAR-10. We train the network on compressed IFMaps, in all convolutional layers. We set the compression rate to 65% using the cosine transform (likewise JPEG [12]). As shown in Figure 5, an input image (in this case a tiger) keeps its image structure even-though it passes through multiple layers (C0-2). When we compress IFMaps, in all convolutional layers, with a constant rate, 65%, we observe that even-though the quality of IFMaps drops, we can still achieve the baseline accuracy for AlexNet over CIFAR-10 dataset (on average 64%). This experiment shows us a huge potential in terms of compressing IFMaps, in the FD, for CNNs.

However, the mentioned FD compression technique (based on the cosine transform), would not be useful for the SD networks since the networks' needed convolutions are being performed in the SD, and therefore, the compressed version of IFMaps in the FD cannot be used for convolution (JPEG's FD compression technique is not convolution compatible [7]). In sum, if we compress IFMaps in the FD and perform convolution in the same domain (FD), we could optimize CNNs considerably.

### D. STOCHASTIC COMPUTING

Stochastic computing (SC) has shown promising results for ultra low cost hardware implementation of different algorithms including CNNs [18], [26]. More specifically, SC is popular for performing low-cost, iterative, fixed-point multiplications [26]. As we observed in Section II-B, one of the main elements of convolution in the frequency domain is point-wise matrix multiplication (PwMM). Therefore, if we could utilize SC to perform PwMM, we could expect a

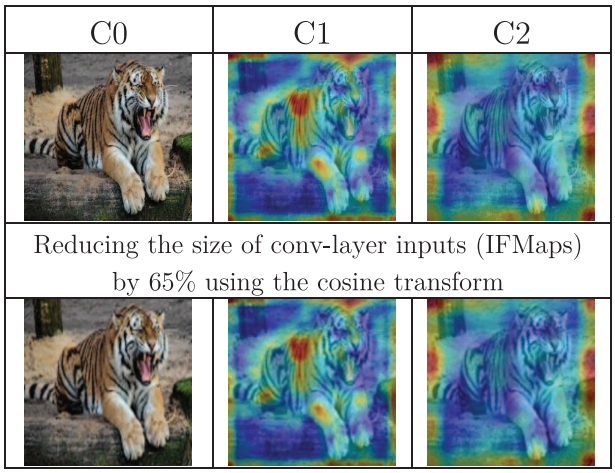


**FIGURE 5.** Visualizing IFMaps in the first three convolutional layers (C0-2) of AlexNet with(out) compression when a tiger image passes through the trained network with CIFAR-10. Note that for the sake of presentation, we made the size of all the layer IFMaps the same and we constructed the images from IFMaps' values by some scaling operations. (Red color sections, with different spectrum, show that from where and how each layer's filter extract features).

huge hardware cost reduction. However, converting values for use in SC with minimal overhead is challenging.

To utilize SC, first we should convert used numbers in an algorithm to a same range as stochastic numbers (SNs). Bipolar SNs represent values between $-1$, and 1. We can easily scale the range of all data in a CNN to $[-1, 1]$ to match it with SNs. For example, in CNNs, the filter weights are usually normalized, and between $[-1, 1]$ [22]. To convert feature maps to SNs, we use a normalization process: $w_s = \alpha \times (w - 128)$, where $\alpha$ is the scaling factor (in this case, $\alpha = \frac{1}{128}$). Therefore, $w_s \in [-1, 1)$.

We can represent SNs using streams of random bits [21], where the probability of having '1' in a long bit stream is determined by a value in the [0, 1] interval. However, to represent numbers in the $[-1, 1]$ interval, which is used in the bipolar representation, we need to utilize an encoding: a real number $x$ is processed by $P(X = 1) = \frac{x+1}{2}$. For example, we can represent 0.4 using 1011011101, as $P(X = 1) = \frac{0.4+1}{2} = \frac{7}{10}$. Multiplication is performed by a single XNOR gate in a bipolar representation [26]. This results in a significant reduction in the hardware cost compared to the conventional binary multiplier. A simple stochastic multiplication using a two-input XNOR gate is shown in Figure 4. Provided that X and Y are statistically independent (uncorrelated), a single XNOR gate can precisely compute $X \times Y$ in the bipolar representation [21].

To do SC operations, we need to convert binary numbers (BNs) to SNs and vice-versa. To do so, we need to utilize a stochastic number generator (SNG). As shown in Figure 4, a SNG often consists of a binary comparator, and a linear feedback shift register (LFSR) as the random bit generator [21]. For instance, to generate the corresponding SN representation of a fixed-point binary number $x = \frac{1}{2} = 0.100_2$, we take the fractional part and utilize an LFSR with $2^n$ stages, where $n$ is the number of fractional digits (in this case three). Then,

we compare the generated values from the LFSR with the fractional part of $x$, in this case we can represent $x$ with a bit-stream $S = 10101001$, where four '1's exist in an eight-bit length bit-stream ($P(X = 1) = \frac{4}{8} = \frac{1}{2}$). To represent numbers in SC via sufficiently independent bit streams (which is crucial in SC [21]), we should employ different LFSRs with different initial seeds. To convert an SN to BN, we just need to count the number of '1's in a bit-stream (a binary up-counter) [21].

## III. HARTLEY STOCHASTIC-BASED COMPUTING CNN WITH IFMAP COMPRESSION

### A. TRANSFORMING IFMAPS AND FILTERS FROM SD TO FD

We transform the range of all data in our method (Hartly stochastic-based computing CNN (HSC-CNN)) to $[-1, 1]$. In this regard, we scale and keep the range of IFMaps data to $[-1, 1]$ as discussed in Section II-D. Note that 1) it has been observed that if we keep the filter's values between $[-1, 1]$, it does not damage a CNN's accuracy [22], and 2) in the implementation, for each IFMap matrix we construct two matrices, one for the scaled data (which the range of is elements are between $-1$ and 1), and the other for the scaling factors of the matrix elements.

To perform FHT on the scaled input data (no matter if it is a Filter or an IFMap), we substitute every multiplication with addition. In this regard, first, we represent the scaled input image matrix values in the trigonometry numerical system. In other words, we apply the element-wise *arcsin* function on the scaled IFMaps and filters to calculate the degrees that correspond to the values of each element in the matrices. Therefore, we can represent any filter and input image values with $sin(x)$ function, where $x \in [0, \pi]$. Subsequently, we can represent $f[m, n]$ from Eq. (2) by $sin(x)$ after scaling, where $x = arcsin(f[m, n])$. As a result, we can expand the formula for the Eq. (2) as below (for a simpler formula representation we consider $\frac{2\pi}{N}(km + ln) = y$):

$$\begin{aligned} &sin(x) \times cas\left(\frac{2\pi}{N}(km + ln)\right) \\ &= sin(x) \times cos(y) + sin(x) \times sin(y) \\ &= \frac{1}{2}(sin(x + y) + sin(x - y) + cos(x - y) - cos(x + y)) \\ &= \frac{1}{2}\Bigg(sin(x + y) + sin(x - y) + sin\left(\frac{\pi}{2} - x + y\right) \\ &\qquad + sin\left(\frac{-\pi}{2} + x + y\right)\Bigg). \end{aligned} \tag{3}$$

On the other hand, we estimate the values for $sin(x)$ by a look-up table (LUT): we divide the $[0, \pi]$ interval into $2^p$ equal degrees, and we keep their corresponding *sine* values with $2^q$ bits in a fixed-point representation. $p$ and $q$ define the precision of estimating $sin(x)$ with the LUT, while they determine its memory usage as well. The final expressions in Eq. (3) are replaced by an indexing function that locates their angular interval and returns its corresponding LUT entry. For example, if $x = \frac{\pi}{4}$ and $y = \frac{\pi}{8}$, $sin(x+y) = LUT(index(64+32))$. Therefore, we can replace two fixed-point multiplications and one addition, for calculating each element in Eq. (2), with six integer (index) additions and three fixed-point additions as represented in Eq. (3), which is much more efficient (specifically in hardware). Note that th For more details regarding implementing $sin(x)$ function using the LUT method, and how to manage an index overflow issue in $sin(x + y)$, for example, you can refer to [27].

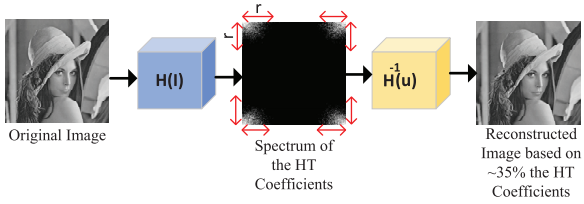


**FIGURE 6.** The process of compressing IFMaps (coefficient selection) using the HT in all convolutional layers. In this figure, just for an understandable presentation, we showed the compression process over the first layer's IFMap (input image) over a sample image: compressing an IFMap (image) by 65% has almost no effect in its quality (PSNR = 28.1 dB). Note that in the spectrum figure, the low frequency components are shown with lighter colors.

### B. COMPRESSING IFMAPS

#### 1) FIXED-RATE IFMAP COMPRESSION

It has been shown in literature that we can approximate the IFMaps before convolving them with kernels without reducing the accuracy of CNNs [28]. With this in mind, we compress IFMaps in the FD using the HT. The HT naturally provides a compression process that even-though is not as efficient as JPEG [12], however, it does not need an extra transformation to perform convolution with [11]. Note that standard JPEG works just with the cosine transform which is not convolution compatible [7].

Moreover, the HT compression is much memory efficient than the FT compression [7]. This is the case since HT produces one real coefficient matrix while FT needs two matrices to keep complex coefficients.

Also, as it is depicted in Figure 6, the HT separates the high and low frequency coefficients in the image's transformed matrix: it pushes low-frequency coefficients to the corners of the transformed matrix. Note that the low-frequency coefficients are more pronounced in the IFMaps' HT matrices, and therefore, they have the highest effect on the convolution's accuracy. To select the most important coefficients, with an inspiration from entropy coding method [7], we select four $r \times r$ sub-matrices from the four corners of a Hartely transformed matrix (we mask the transformed matrix), as is shown in Figure 6, and set the rest of the coefficients to zero. We add $r$ as a hyper-parameter to our CNN models. Even-though this process shrinks the IFMaps dramatically, it maintains the dimensionality of OFMaps after applying the inverse HT.

#### 2) ADAPTIVE-RATE IFMAP COMPRESSION

To compress IFMaps further, we propose to adaptive prune their coefficients in the FD. An IFMap matrix in the FD, $I$,

can be parameterized by associating a mask variable $m \in \{0, 1\}$. These binary parameters can be formed into a mask matrix, $T$, whose values indicate whether a coefficient is currently pruned or not. Then, the CNN's model optimization function can be described as:

$$\min_{I,T} L(I \odot T) \quad s.t. \quad T_{i,j} = f(I_{i,j}), \tag{4}$$

where $L()$ is the model's loss function, $\odot$ represents the element-wise product and $f()$ is the masking function which satisfies $f(I_{i,j}) = 1$ if coefficient $Y_{i,j}$ seems to be important in the current layer and 0 otherwise.

With an inspiration from the dynamic method in [29], we set two thresholds $a$ and $b$ to decide the mask values of coefficients for each layer. Also, we utilize an absolute value to evaluate the importance of each coefficient. Using function $f()$ as below, the coefficients are not pruned forever and have a chance to return during the training process.

$$f\left(I_{i,j}^{t+1}\right) = \begin{cases} 0 & \text{if} \quad a > \left|I_{i,j}^{t+1}\right| \\ T_{i,j}^{t} & \text{if} \quad a \leq \left|I_{i,j}^{t+1}\right| \leq b \\ 1 & \text{if} \quad b < \left|I_{i,j}^{t+1}\right| \end{cases} \tag{5}$$

The $a$ and $b$ should be set with respect to the distribution of coefficients in each layer, i.e.,

$$a = 0.9 \times (\mu + \gamma \times \sigma) \tag{6}$$

$$b = 1.1 \times (\mu + \gamma \times \sigma), \tag{7}$$

in which $\mu$ and $\sigma$ are the mean value and the standard deviation of all coefficients in one layer, respectively. $\gamma$ denotes the compression rate in each layer, by which the number of remaining coefficients in each layer is determined. Note that $\gamma$ is a parameter for the model which is being set during training.

### C. FREQUENCY DOMAIN CONVOLUTION BY POINT-WISE MATRIX MULTIPLICATION

We can perform convolution in the FD using the HT with just element-wise matrix operations: two PwMMs, two additions, and one subtraction (refer to Eq. (2)). We know that the HT is a real linear operator, and is symmetric [11]. Therefore, we can construct $\hat{Y}_{N\times N} = H_{-2\pi}(X)$ from $Y_{N\times N} = H_{2\pi}(X)$ with a simple constant matrix element re-mapping: $Y(i, j) = \hat{Y}(N - i, N - j)$, where $0 \leq i, j \leq N - 1$, and $N - 0 = 0$ are the indexes and the width for $Y$ and $\hat{Y}$, respectively. Subsequently, we can hard-code this matrix re-mapping in our implementation. As a result, we can implement convolution with the HT via 1) two HTs over $I$ and $F$, 2) two element-wise matrix re-mappings, 3) three matrix additions, and 4) two PwMMs. Also, note that there exist quite a few number of optimization techniques for FFT (e.g., Hermitian (conjugate) Symmetry, and kernel subsampling [7]). Most of them can be applied on the FHT as well [11]. However, investigating these methods is out of the scope for this paper.

We use stochastic computing for the PwMMs in the FD. SC naturally provides a very efficient implementation for a PwMM. This is the case since multiplications are independent in a PwMM, and therefore, we can use just two random bit generators (implemented in the form of LFSR) to produce the needed random bit streams for all the values in the two matrices that we want to multiply in a point-wise manner. Note that the need for multiple LFSRs for SC-based arithmetic functions is one of important bottle necks for SC implementations [21]. Moreover, we can utilize the fixed-point values in the *sine* LUT (see Section III-A) to generate the corresponding SNs (refer to Figure 4). This is the case since our proposed Hartley transform re-formalization and scaling regime keep the range of scaled Hartley-transformed values between $-1$ and $+1$. Therefore, without any further computation, we can utilize them to generate their corresponding SNs, and perform multiplication via SC. In sum, we utilize just two LFSRs, $2N^2$ comparators, and $N^2$ up-counters to implement a PwMM, which is much more efficient than fixed-point multiplications in terms of energy consumption. For more details regarding the hardware implementation of Hartley stochastic based convolution, you can refer to the Appendix.

### D. TRANSFORMING OFMAPS FROM FD TO SD

To transform back OFMap (the output of convolving IFMap and filter) from the frequency- to spatial-domain, we use the same transformation. This is the case since the HT is self-inverse and is a unitary operator. Therefore, the inverse HT is equivalent to the HT [11] (we utilize the same implementation approach as discussed in Section III-A).

Note that by implementing activation and pooling functions (refer to Section II-A) in the FD, we can implement the entire convolutional layer in the FD. This could save much computation by preventing repetitive domain transformation in each convolutional layer; and we would only need to transform OFMaps to the spatial domain before the fully connected layer(s) at the end of a the CNN architecture (see Figure 5). However, this approach needs to implement training in the FD as well. In other words, we need to learn weights and extract features in the frequency domain. Investigating these optimizations (i.e., implementing activation and pooling functions in the FD for HSC-CNN) is out of the scope of this paper and is the subject of our future work.

### E. CNN TRAINING WITH ADAPTIVE-RATE IFMAP COMPRESSION

When we utilize adaptive-rate IFMap compression, the training procedure must be changed accordingly. The procedure of training an L-layers CNN with IFMaps' pruning (masking) can be divided into three phases as illustrated in Figure 7: fully connected, back-propagation, and mask matrix and filter weights updates. Repeating these three phases, iteratively, results in training the CNN while pruning its IFMaps in all convolutional layers. Note that you can also find the algorithm for training in the Appendix.

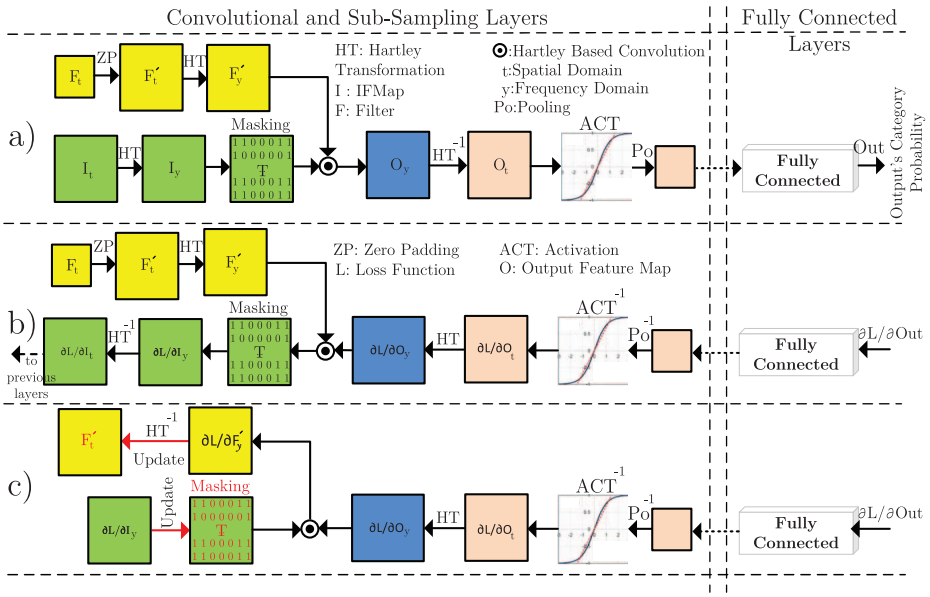


**FIGURE 7.** Training CNN in the FD: a) forward propagation, b) back propagation, and c) filter weights and mask matrix updates. In forward propagation, we, first, zero-pad filters ($F_y'$). Then, we transform IFMaps and filters to the FD using the Hartley transform. After the transformation, we mask IFMaps before convolving them with the transformed filters to generate OFMaps in the FD ($O_y$). Afterward, we transform $O_y$ back to the SD using $HT^{-1}$, and then pass $O_t$ through the activation and sub-sampling layer. This convolution and sub-sampling repeats multiple times until data reaches fully connected layers. In back propagation, we calculate the gradient of the loss from the output *Out* as $\frac{\delta(L)}{\delta(Out)}$. Then, for the loss to be propagated to the other layers, we use the Chain rule and find $\frac{\delta(L)}{\delta(O_y)}$, and $\frac{\delta(L)}{\delta(I_y)}$. Note that in the backpropagation we utilize the inverse operations such as the inverse of activation function ($ACT^{-1}$) and the inverse of pooling ($Po^{-1}$). Finally, in the update procedure, we update learnable parameters (filter weights and mask matrix coefficients). We perform this update based on the gradient that is propagated in back propagation and the current values for learnable parameters. In this regard, we calculate $\frac{\delta(L)}{\delta(F_y')}$, and then update filter weights in spatial domain ($F_t'$) using $HT^{-1}$.

As it is illustrated in Figure 7, in the fully connected phase, the input and weight filters are transformed into the FD using the HT. Afterward, a mask matrix is being applied on the transformed IFMap. Then, filter and masked IFMap are convolved in the FD. Consequently, a pooling and activation are applied before passing data to the next layers.

During back-propagation, after computing the OFMap gradient $\frac{\partial L}{\partial O_t}$, in the SD, a 2-D HT is directly used to obtain the gradient $\frac{\partial L}{\partial O_y}$ in the FD. Note that since HT is a linear transformation, the gradient in the FD is the HT transformation of the gradient in the SD. Then, the gradient is back propagated until first layer $\frac{\partial L}{\partial I_t}$, and is passed to previous layers.

During update phase, we update filter weights and mask coefficients. Also, we apply dynamic pruning (refer to Section III-B2) after updating the coefficients. Note that we update not only the remaining coefficients, but also the ones that are considered temporarily unimportant. Therefore, we can have the pruned parameters in previous training epochs to be returned.

## IV. EXPERIMENTAL RESULTS

To evaluate the accuracy, latency, and energy consumption of our approach, we use two networks, LeNet, AlexNet, over the MNIST and CIFAR-10 datasets, respectively. First, we apply the Hartley stochastic computing (HSC) convolution to CNNs to investigate how hyper-parameter choices in HSC-CNN (i.e., $p$, $q$, $r$, and $l$) affect network accuracy. We select the design that achieves the best latency without loss of accuracy compared with the original floating-point implementation using the fixed-rate IFMap compression method (refer to Section III-B1). Afterward, we fix the best set of hyper-parameters ($p$, $q$, and $l$) from the fixed-rate compression, and we re-train the network using the adaptive-rate IFMap compression method (refer to Section III-B2). Next, we synthesize these designs (adaptive-rate and fixed-rate), as well as conventional approaches for comparison, for FPGA implementation, and compare inference latency, inference energy, and FPGA resource utilization. Finally, we analyze the memory and computational complexity of just the fixed-rate HSC-convolution in comparison with spatial- and FFT-convolution.

Compressing IFMaps adaptively results in stochastic layer-wise compression rates which depend on training parameters (such as the number of epochs and learning rate). Therefore, the complexity of convolution based on adaptive-rate IFMaps compression cannot be formalized. However, we show that the complexity of convolution based on adaptive-rate IFMap compression is upper bounded by the fixed-rate one. Therefore, if fixed-rate IFMap compression works better than conventional methods in terms of computational and memory complexity, adaptive-rate compression always works even better.

### A. NETWORK ACCURACY

Figure 8 shows the accuracy and the relative speed-up of HSC-CNN over LeNet and AlexNet when we change the

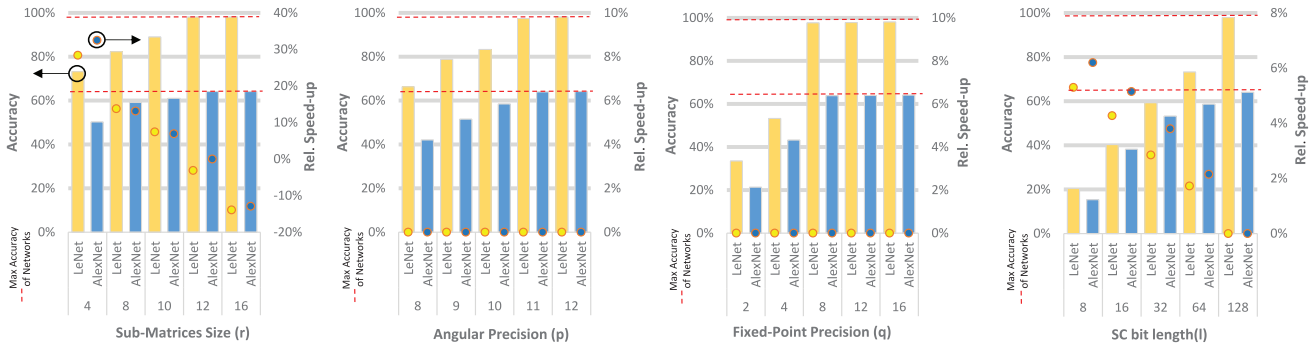


**FIGURE 8.** Accuracy (bars, left, in %), maximum average accuracy (red doted lines), and relative speed-up (dots, right, in %) comparison for different HSC-CNN configurations based on fixed-rate IFMap compression, $r = 12$ for AlexNet and $r = 11$ for LeNet, $p = 11$, $q = 8$, and $l = 128$ for both the networks). Note that there are some convolutional layers in these networks (LeNet and AlexNet) that their IFMap dimensions are less than $(2 \times r) \times (2 \times r)$; this translates to no compression for those layers.

hyper-parameters: angular precision ($p$), the *sine* function's fixed point precision ($q$), the size of selected sub-matrices for the four corners of IFMaps after HT ($r$), and the SC bit-stream lengths for representing the numbers for PwMMs ($l$). Note that 1) The reported accuracy of the networks can be improved by network-tuning. For example, we can apply hyper-parameter optimization or modify the network's architecture and/or utilize the knowledge transfer techniques such as model distillation [30]. However, investigating these methods is out of the scope of this paper. 2) We compare the configurations against a baseline system ($p = 11$, $q = 8$, and $l = 128$ for both the networks, and $r = 12$ for AlexNet and $r = 11$ for LeNet): whenever we change a hyper-parameter, we fix others to the baseline system's (default) values. 3) To reduce the number of hyper-parameters, we chose a same set of hyper-parameters (i.e., $p$, $q$, $r$, and $l$) for all convolutional layers of a network when we report the networks' accuracy.

### 1) ACCURACY UNDER FIXED-RATE COMPRESSION

Increasing $r$ increases model complexity, as it reduces IFMap compression. As a result, accuracy improves. For example, increasing $r$ from 4 to 12 results in 13.2% accuracy improvement for AlexNet, but a 34% increase in latency relatively. Note that LeNet almost reaches its maximum accuracy on MNIST (98.0%) when $r = 11$, while AlexNet needs at least $r = 12$ to reach 64.1% accuracy on average on CIFAR-10 (slightly better than the highest achievable accuracy utilizing full-precision—63.9%). r in AlexNet is bigger than in LeNet since LeNet has fewer convolutional layers compared to AlexNet. We hypothesize that the improvement in AlexNet's accuracy results from the regularization of HT compression. Also, note that sub-sampling a transformed IFMap w.r.t. $r$ could lead to $r$ values bigger than half of the IFMap's matrix size (refer to Figure 6). In this case, no compression is applied on that IFMap.

Angular precision ($p$) and fixed-point precision ($q$) do not affect network latency, but considerably affect network accuracy. This is the case since we utilize fixed-point operations and precision does not change the speed of these operations. However, operation precision radically affects network overall accuracy: reducing $q$ from eight to two bits reduces the accuracy of AlexNet from 64.1% to 21.3%.

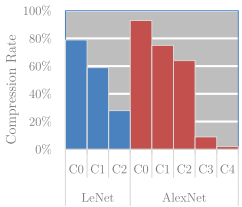


**FIGURE 9.** Adaptive compression rates for different convolution layers when $p = 11$, $q = 8$, and $l = 128$ for both the networks and $\gamma$ for each layer is being determined during training.

Stochastic number (SN) bit-length ($l$) significantly affects network accuracy and latency. For example, reducing the bit-length from 128 to 8 reduces the accuracy of LeNet more than 70%. PwMMs (which we implement using SC-multiplications) multiply two fixed-point numbers with magnitudes less than one, and therefore, the precision needed for the result of the multiplication becomes doubled of the original numbers. Also note that the speed of SC multiplication has an inverse linear relationship with the length of SNs. However, the overall effect of SN bit-length on latency is less than 6% when we increase the bit-length 16x, since PwMMs, which are the only operations made by SC, are relatively small compared to others (such as FHT and IFHT modules) in terms of computation.

### 2) ACCURACY UNDER ADAPTIVE-RATE IFMAP COMPRESSION

To evaluate the accuracy of networks in the presence of adaptive-rate IFMap compression, we fix hyper-parameters other than $r$ (i.e., $p$, $q$, and $l$) to those used to obtain the most optimized result using fixed-rate compression (refer to Section IV-A1 and Figure 8). Afterward, rather than tuning $r$, we perform the Bayesian hyper-parameter tuning for $\gamma$ for each layer based on the methodology introduced in Section III-B2.

In Figure 9, we show the IFMap compression rate that the adaptive-rate method reaches for each convolutional layer. Note that with these compression rates we do not observe any accuracy loss for the networks. These compression rates on different layers lead to dropping 78% and 58% of the

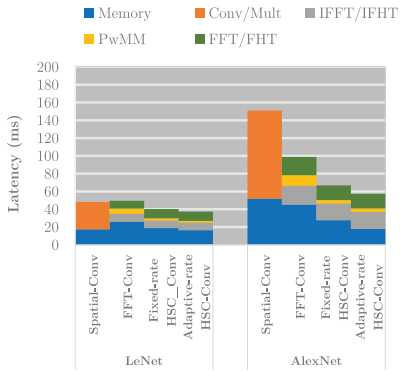


**FIGURE 10.** Latency for different convolution gate-level implementations. Note that for these results we set LeNet ($r = 11$ for fixed-rate, IFMap size = 32x32 and filter size = 5x5), AlexNet ($r = 12$ for fixed-rate, IFMap size = 224x224, and filter size = 11x11), and $p = 11$, $q = 8$, $l = 128$ for both the networks.

FD IFMaps coefficients for LeNet and AlexNet, respectively. These are considerably higher than the 46% and 41% of total compression rates for the fixed-rate method for LeNet and AlexNet, respectively. Also, we observe that the total compression rate for LeNet (78%) is more significant than AlexNet (58%). This originates from the fact that the adaptive-rate method works on the first convolutional layers better and cannot compress later layers (consider C3-4 from AlexNet in Figure 9). This is the case since as we pass through each convolutional layer, features that filters try to extract become integrated more to IFMap. Therefore, the compression rate for later layers is being reduced during training, otherwise, the network would lose the learned information from early layers in a way that training algorithm cannot converge. Also, note that higher compression rates come with the cost of more training time.

### B. HARDWARE IMPLEMENTATION

For energy and latency analysis, we implement the proposed accelerator on an FPGA. Taking inspiration from hardware implementations of the 2-D FFT and IFFT modules, we implement 2-D FHT and IFHT based on an array of 2-point 1-D FHT and IFHT implementations. For parallel fixed-point multiplications, we use only an array of 32 multipliers in hardware: we trade resource utilization for latency. For the sake of comparison, we also design a spatial-convolution module using a systolic MAC (multiply-accumulator) array [31]. To evaluate the cost of convolution in HSC-CNN, we developed hardware for spatial-, FFT-, and HSC-convolution in Verilog. Synthesis was performed using the Xilinx ISE targeting the Virtex-7 xc7s50 FPGA (28nm). The latency and energy consumption are reported using Xilinx ISE synthesis delay reports and XPower Analyzer, respectively.

#### 1) LATENCY

In Figure 10, we report the latency of different sub-functions (forward, PwMM, and inverse operations) in the spatial- and frequency-domain for the first convolutional layer in both the networks (LeNet and AlexNet). Even-though HSC-convolution utilizes SC and sequential bit streams, still it outperforms FFT-convolution by more than 33% for AlexNet. First, SC-based PwMM is much more efficient than the fixed-point PwMM; second, HSC-convolution performs PwMM on the compressed IFMaps which results in saving a considerable number of operations in HSC-convolution. Another interesting observation is that, as it is shown in Figure 10, FFT-convolution is slower than spatial-convolution for LeNet, while it is faster for AlexNet. This happens since the computational saving for performing convolution in the FD for small filter sizes (i.e., LeNet case) is less than its associate domain transformations overhead. However, as filters become bigger in a network (i.e., AlexNet case), the computational saving surpasses the domain transformations overhead. Consequently, convolution becomes faster in the FD than SD. Also, note that even though HSC-convolution needs those domain transformations, it outperforms spatial-convolution for both the networks since it compresses IFMaps before convolving them with filters.

We observe that compressing IFMaps with two different rates ($r = 11$ and $r = 12$ for LeNet and Alexnet, respectively) does not lead to a substantial decrease in latency compared to FFT-convolution. This is the case since we implement FHT and IFHT (the bottlenecks of HSC-convolution) in a butterfly architecture [32]: the same architecture that FFT and IFFT use. Butterfly architecture trades hardware resource to keep latency almost constant. Therefore, we observe that the latency for FFT and FHT as well as IFFT and IFHT are almost the same.

For a fair comparison in terms of memory latency, we keep the number of memory banks the same for FFT-convolution and HSC-convolution (for both adaptive- and fixed-rate implementations). In a case that we need parallel memory accesses beyond the provided memory throughput in the implementation, a controller translates those parallel accesses to sequential ones. We observe that most of latency saving for HSC-convolution comes from memory accesses. This is the case since HSC-convolution needs much less memory accesses compared to FFT-convolution and Spatial-convolution. HSC-convolution uses fewer coefficients to perform convolution.

We observe that if we utilize adaptive-rate IFMap compression, latency is reduced by 16% compared to the fixed-rate method. Also, latency is reduced more for AlexNet than LeNet. The filters and IFMaps for the first layer of AlexNet are larger than for LeNet, resulting in more opportunity for compression.

#### 2) HARDWARE RESOURCE UTILIZATION

As we see in Table 2, HSC-convolution requires fewer FPGA resources (e.g., 11% fewer memory (flip-flops) and 15% fewer slices for AlexNet), and 19% less energy for fixed-rate implementation compared to FFT-convolution. Even though our implementations for adaptive- and fixed-rate HSC-convolution implementations utilize the same resources,

**TABLE 1. Computation and memory demands for different convolution methods.**

| Approach | Operations | Computational Complexity | Memory Usage+ | | Arithmetic Operations* | |
|---|---|---|---|---|---|---|
| | | | Filter | IFMap/OFMap | Additions | FP-Mult/SC-Mult |
| Spatial-Convolution | CONV | $(n-k+1)^2.(k)^2$ | $(k)^2$ | $(n)^2+(n-k+1)^2$ | $\frac{1}{2}.(n-k+1)^2.(k)^2$ | $\frac{1}{2}.(n-k+1)^2.(k)^2$ |
| FFT-Convolution | Forward | $6.(n)^2.Log(n)$ | $(n)^2+2.(n)^2$ | $(n)^2+2.(n)^2$ | $2.(n)^2.Log(n)$ | $4.(n)^2.Log(n)$ |
| | PwMM | $6.(n)^2$ | $2.(n)^2$ | $2.(n)^2$ | $2.(n)^2$ | $4.(n)^2$ |
| | Inverse | $2.(n)^2.Log(n)$ | $2.(n)^2+2.(n)^2$ | $2.(n)^2+2.(n)^2$ | $(n)^2.Log(n)$ | $(n)^2.Log(n)$ |
| Fixed-rate HSC-Convolution (based on FHT) | Forward | $6.(n)^2.Log(n)$ | $(n)^2+2.(n)^2+(2)^p$ | $(n)^2+2.(n)^2+(2)^p$ | $6.(n)^2.Log(n)$ | - |
| | SC-based PwMM | $2.(n)^2+4.(r)^2$ | $(n)^2$ | $(n)^2$ | $2.(n)^2$ | $4.(r)^2$ |
| | Inverse | $6.(n)^2.Log(n)$ | $(n)^2+2.(n)^2$ | $(n)^2+2.(n)^2$ | $6.(n)^2.Log(n)$ | - |

+ Note that we, separately, report the memory usage for sub-functions (e.g., PwMM, Forward, etc.), while in efficient implementations, their memory allocations could be merged.

* Also, note that for HSC-Convolution. we do not report the needed arithmetic operations on the scaling matrices, while we report their memory overheads. This is the case since not necessarily for each addition we need to modify the corresponding scaling element for example. Also, note that the change in scaling matrices happen with bit-shift operations rather than additions or subtractions since the scaling factors always follow $(2)^x$.

n, k, and r represent the size of IFMap, filter, and selected sub-matrices for the four corners of transformed IFMap, respectively.

**TABLE 2. Resource utilization and energy consumption for different convolution (FPGA Virtex-7 xc7s50 implementation, and memory consumption is being represented using flip-flop counts). Note that for these results we set LeNet ($r$ = 11 for fixed-rate, IFMap size = 32x32 and filter size = 5x5), AlexNet ($r$ = 12 for fixed-rate, IFMap size = 224x224, and filter size = 11x11), and $p$ = 11, $q$ = 8, $l$ = 128 for both the networks.**

| Network | Convolution Type | Memory Banks (Flip-Flops) | Slices | Energy (mJ) |
|---|---|---|---|---|
| LeNet | Spatial-Convolution | 8,192 | 1,430 | 119 |
| | FFT-Convolution | 11,440 | 578 | 67 |
| | Fixed-rate HSC-Convolution | 9,745 | 453 | 42 |
| | Adaptive-rate HSC-Convolution | 9,745 | 453 | 37 |
| AlexNet | Spatial-Convolution | 30,389 | 3,730 | 412 |
| | FFT-Convolution | 41,998 | 2929 | 247 |
| | Fixed-rate HSC-Convolution | 37,478 | 2547 | 208 |
| | Adaptive-rate HSC-Convolution | 37,478 | 2547 | 176 |

we observe 15% additional energy savings for adaptive- compared to fixed-rate compression for AlexNet. When we apply adaptive IFMap compression, fewer memory accesses are needed for convolution, saving energy.

### C. COMPLEXITY ANALYSIS FOR FIXED-RATE HSC-CONVOLUTION

In Table 1, we summarize the computational complexity, order of used memory and the needed number of operations for the different approaches of convolution as a function of IFMap size ($n \times n$), and filter size ($k \times k$). Note that since the memory and computational complexities of adaptive-rate IFMap compression for HSC-convolution depend on the training procedure, and they vary based on the provided input data set and hyper-parameter tuning strategy, we do not analyze its complexity in this section.

As shown in Table 1, the computational complexity of fixed-rate HSC-convolution is the same as FFT-convolution [23]. We break down FFT-convolution and HSC-convolution into three consisting modules: 1) *forward* transformation, 2) point-wise matrix multiplication (*PwMM*), and 3) *inverse* transformation. In both *forward* and *inverse*, a 2-D FFT and FHT is being applied for FFT-convolution and HSC-convolution, respectively, which utilize butterfly data-path structures [7]. This butterfly structure reuses needed operations (additions and multiplications), and therefore, reduces the complexity from $n^2 \times n^2$ of 2D DFT and DHT (refer to Eq. (2)) to $2 \times n^2 \times Log(n)$ [7]. Note that for *forward*, two matrices (Filter and IFMap) are transformed into the FD, which each matrix transformation needs $2 \times n^2 \times Log(n)$ operations. Moreover, for *PwMM*, we need to multiply two matrices. Therefore, the complexity of this module is $4 \times n^2$ since this *PwMM* uses complex numbers for FFT-convolution, and two point-wise matrix additions and two *PwMMs* are needed for HSC-Convolution (refer to Figure 3). Also, we observe that the complexity of both FFT and FHT outperform spatial-convolution, especially when $k$ is close to $n$. Note that spatial-convolutions needs iterative point-wise matrix multiplications which makes its computational complexity quadratic [7] as shown in Table 1.

However, in terms of memory use, spatial-convolution consumes the least, while fixed-rate HSC-convolution needs less memory than FFT-convolution, even though we implement *sine* in memory. For the forward transformations, both FFT and FHT need $n^2$ memory cells to store a zero-padded filter and an IFMap. Also, both need $2 \times n^2$ memory cells to store their transformation coefficients; FHT requires additional storage for *sine* approximations, a $2^p$-entry table in memory. However, for the *PwMM* and the *inverse* operations (IFFT, and IFHT), fixed-rate HSC-convolution works on a single real matrix, while FFT-convolution utilizes two matrices (due to working with complex numbers). Ultimately, FFT-convolution utilizes $2 \times (n)^2 - 2^p$ more memory than FHT-convolution.

Spatial-convolution needs the most arithmetic operations, as it utilizes iterative point-wise matrix multiplications to perform convolution. However, fixed-rate HSC-convolution utilizes no fixed-point multiplications during frequency domain transformation, resulting in considerable

computational savings compared to FFT-convolution. The most important difference between these two convolution methods is their *PwMMs*: FFT-convolution utilizes $4 \times n^2$ fixed-point (FP) multiplications and $2 \times n^2$ additions (which are the needed operations for complex matrix multiplication), while fixed-rate HSC-convolution uses $2 \times (4 \times r)^2$ SC-multiplications which are very efficient in terms of implementations than fixed-point multiplications (refer to Figure 4), and $2 \times n^2$ decimal additions likewise FFT-convolution.

### D. COMPARISON WITH PAST WORK

Unfortunately, there are no appropriate FD past work to let us directly compare ours with them. This is the case since ours is the first to implement convolutions in CNNs using the frequency domain feature map approximation and stochastic computing. For example, Abtahi *et al.*, [33] showed an end-to-end FFT-based scalable FPGA accelerator for ResNet-20 via breaking down input matrices for convolution into multiple segments, and then generated block outputs are aligned and added to form a complete output. However, this method focuses on the required max memory usage of FFT-based convolution, and it does not do any computation optimization for the FD domain implementation of convolution. Therefore, our method (HSC-convolution) always wins, in terms of computation reduction, compared [33]. To reduce latency, [33] utilized a parallel processing elements (PE) architecture, results in the 65% latency improvement. However, this significant latency improvement is achievable by using the parallel PEs architecture, while HSC-convolution can utilize many PEs implementation on top of the computation reduction that it has due to its IFMap compression technique. For example, we can break-down the FD feature map spectrum (see Figure 6) into $r \times r$ sub-matrices from the four corners (refer to Section III-B1), and assign them to parallel PEs. As a result, we expect to receive better results than [33] if we implement HSC-convolution based on its proposed architecture.

As another past work, [20] utilized a combination of stochastic and deterministic computing design for low-cost, energy-efficient and yet accurate implementation of spatial domain CNNs. Like our work (HSC-convolution), Faraji *et al.*, [20] utilized a single gate to perform multiplications in the required PwMMs for convolution. Therefore, their hardware resource optimization is the same as ours in PwMMs. Note that [20] operates in the SD, while ours works in the FD, but both needs PwMMs. Faraji *et al.*, [20] claimed that they achieve 12.5% better critical path delay when they compare their implementation with a fully pipelined spatial-convolution used in LeNet. However, HSC-convolution reaches more than 21% latency improvement when it adaptively compresses IFMaps for convolution used in LeNet. Regarding the energy consumption, Faraji *et al.*, proposed approach works better since they used SC and a bit-stream based methods to implement convolution entirely. Consequently, they achieve more than 70% energy saving, while ours could reach to 64%.

### E. SUMMARY

By utilizing HSC-CNN, we can perform reduced precision, approximate operations ($p = 11$, $q = 8$, and $l = 128$), and achieve similar accuracy to with floating point computation for LeNet ($r = 11$) and AlexNet ($r = 12$). Implemented in hardware, this results in substantial latency and energy reduction, 33% and 28%, respectively compared to using FFT for AlexNet as an example. Even though the computational complexity of convolution with FFT and FHT are the same, our implementation using HSC results in simpler operations and less memory use.

Moreover, utilizing adaptive-rate IFMap compression lets us achieve higher compression rates: 22% and 17% for LeNet and AlexNet, respectively. These better compression regime translates to more optimization in terms of hardware and latency.

## V. RELATED WORK

To optimize convolution (specifically for big matrices), designers prefer to perform it in the FD [2], [5], [32]. There are a few number of spatial to frequency domain transformations that support convolution directly (e.g., Hartley, and Fourier) [7]: it means that these transformations do not need extra processes before performing convolution based on PwMM(s). The Fast Fourier transform (FFT) is proven to be the most efficient method for convolution in the FD [34]. However, FFT suffers from 1) complex hardware implementation structure (e.g., the need for multiple separate memory banks to keep real and imaginary coefficients), and 2) the need for complex multiplications (one complex multiplication requires four multiplications and two additions). Some work [8], [35] focuses on optimizing the FFT, for example, by approximating the operations of the transformation. However, we observe that if we follow these aggressive approximation techniques, the estimated result of a convolution cannot be utilized for CNNs (accuracy drops considerably). To address the complexity of FFT-based convolution, our method, which is based on the HT, utilizes just real numbers and operations [23]. Moreover, in the literature, there are some similar approaches that utilize the Hartley transform to reduce the complexity of CNNs by implementing them in the FD [9], [10]. However, 1) they did not utilize this transformation for optimizing CNNs with small filters, and 2) none of them addresses the high cost of needed point-wise matrix multiplications (PwMM) in the FD, while our HSC-CNN implementation does by 1) approximating feature maps in the FD, and 2) utilizing SC for PwMMs, respectively.

Some past work tried to reduce the needed computations for CNNs by compressing CNNs data using FD methods [13], [36]. For example, Liu *et al.*, [13] utilizes a lossy compression technique in the FD using discrete cosine transform (which is the base for JPEG compression), while [36]

uses JPEG compression to reduce the memory required for training a CNN. However, neither can utilize their used FD transformations to optimize convolution itself (they just compress the input data and weights). This is the case since the used FD transformations are not directly convolution compatible [7] (those transformations need extra operations to perform convolution in the FD). Our work utilizes the HT to not only compress the input feature maps (IFMaps), in covolutional layers of CNNs, without any computational overhead, but also optimizes convolution in the FD (i.e., performing convolution using a point-wise matrix multiplication (PwMM)).

To improve the performance (latency) of CNNs even more, past work tried to compress IFMaps in CNNs [14]. For example, [14], [37] proposed to compress images/IFMaps for a CNN using a frequency domain transformation (e.g., wavelet transform). However, these techniques cannot, adaptively, match the accuracy need for different convolutional layers during training, and therefore, they suffer from lower accuracy and higher computational need, while our method utilizes an adaptive-rate IFMaps pruning which results to more efficient networks. On the other hand, some techniques compressed IFMaps in the spatial domain using quantization schemes: fixed-point quantization [38], [39], binary quantization [40], [41], and power-of-two quantization [42]. Also, some others translated IFMaps' compression to weight pruning in CNNs [28], [43]: optimizing which values from IFMaps matrix can be pruned to cause the least accuracy drop and highest latency improvement. Therefore, they could utilize algorithms that address weight pruning optimization in neural networks (e.g., [16], [17], [44]). Even-though these spatial domain (SD) techniques are orthogonal to our method, they are inherently limited in finding more compact representations since they work in the SD, but our method compresses IFMaps in the FD.

Many efficient accelerator algorithms have been developed for CNNs [45], [46], and matrix convolution [8]. The first category of these accelerators try to reformulate convolution to make it more compatible with hardware. For example, Abtahi *et al.*, [33] proposed a method called FFT overlap and add convolution to limit the max memory required for the FFT hardware implementation. This FFT memory management technique coincidentally results in lower energy consumption and latency for CNNs. As another example, SPARCNet [47] makes CNNs' convolutional layers sparse, then, it introduces an efficient FPGA convolution implementation for them. These techniques are orthogonal to our proposed solution. Therefore, they could be utilized, together, to reach a higher optimization for the CNNs hardware implementation. The second category of CNN accelerators utilize approximation (i.e., low-bit width arithmetic units) since CNNs are naturally tolerant to bit-width variations [46]. This eliminates the need for costly full-precision arithmetic units. For instance, Judd *et al.*, [48] dynamically change the precision for the operations of a network to optimize its performance and energy consumption without loosing accuracy significantly. As another example, Yuan *et al.*, [8] focused on implementing the FFT using SC with an approximation method, which reduces the implementation's complexity significantly. However, we observed that utilizing fully SC circuits for FFT convolution does not necessarily lead to an optimized implementation as some operations, such as matrix multiplications, which needs many additions, cannot be implemented in SC efficiently. Our method addresses this problem by combining stochastic- and binary-computing.

**Algorithm 1:** How to Train a Multi-Layer CNN With Pruning IFMap in the FD

**Inputs:** training data, $\mathcal{M}$,
the reference model (i.e., the best fixed-rate model hyper-parameter: $(\hat{p}, \hat{q}, \hat{l})$), $\hat{\mathcal{W}}$,
the compression rate hyper-parameter, $\gamma$
**Output:** Updated binary mask $T$, and filter weights, $F$
**for** $l$=1,2,..., $L$ **do**
  Initialize $(p, q, l) \leftarrow (\hat{p}, \hat{q}, \hat{l})$, $T_l \leftarrow 1$
**end**
**for** *iter*=1,2,...,*max_iter* **do**
  Choose a minibatch of network input from $\mathcal{M}$
  **for** $l$=1,2,..., $L$ **do**
    Apply HT to the input matrix $I_t \rightarrow I_y$ and $F_t \rightarrow F_y$
    Forward propagation with $I_y$ and $F_y$ using convolution, activation and pooling operations.
  **end**
**end**
Loss calculation with $I \odot T$
Backward propagation and generate $\bigtriangledown L$
**for** $l$=1,2,..., $L$ **do**
  Compute the gradient w.r.t OFMap in the FD $\frac{\partial L}{\partial O_y}$
  Compute the gradient w.r.t IFMap in the FD $\frac{\partial L}{\partial I_y}$
  Using $\frac{\partial L}{\partial I_y}$ and $\frac{\partial L}{\partial O_y}$, compute gradients for filter weights in the FD, $\frac{\partial L}{\partial F_y}$
  Use $HT^{-1}$ to calculate $\frac{\partial L}{\partial F_t}$
  Using $\frac{\partial L}{\partial F_t}$, update $F_t$ by SGD method
  Update $T$ according to function $f()$ and the current $I$
**end**

## VI. CONCLUSION AND FUTURE WORK

In this work, we propose Hartley Stochastic-based convolution, a novel technique for optimizing the implementation of CNNs in hardware. Our method performs a frequency-domain compression of the IFMaps in each convolutional layer of a CNN, and implements convolution using SC, improving performance and energy efficiency. We first use the Hartley transform to perform a lossy compression on the IFMaps of each convolution layer in a CNN. In this regard, we introduce two compression methods: 1) fixed-rate, and 2) adaptive-rate. Then, we perform a reformulated Hartley-based convolution, replacing multiplications with additions. This new approach is a better match for stochastic computing hardware. By utilizing our introduced method, we achieve a considerable speedup (1.33x) and energy savings (37%) without reducing CNN accuracy based on fixed-rate IFMap compression. Also, we can reduce 16% and 15% latency and energy consumption more, respectively, when we utilize the adaptive-rate IFMap compression.

In this work, we did not use of the non-linearity of convolution in the frequency domain to optimize the FD-based convolutional layers more. This effect can be used as a

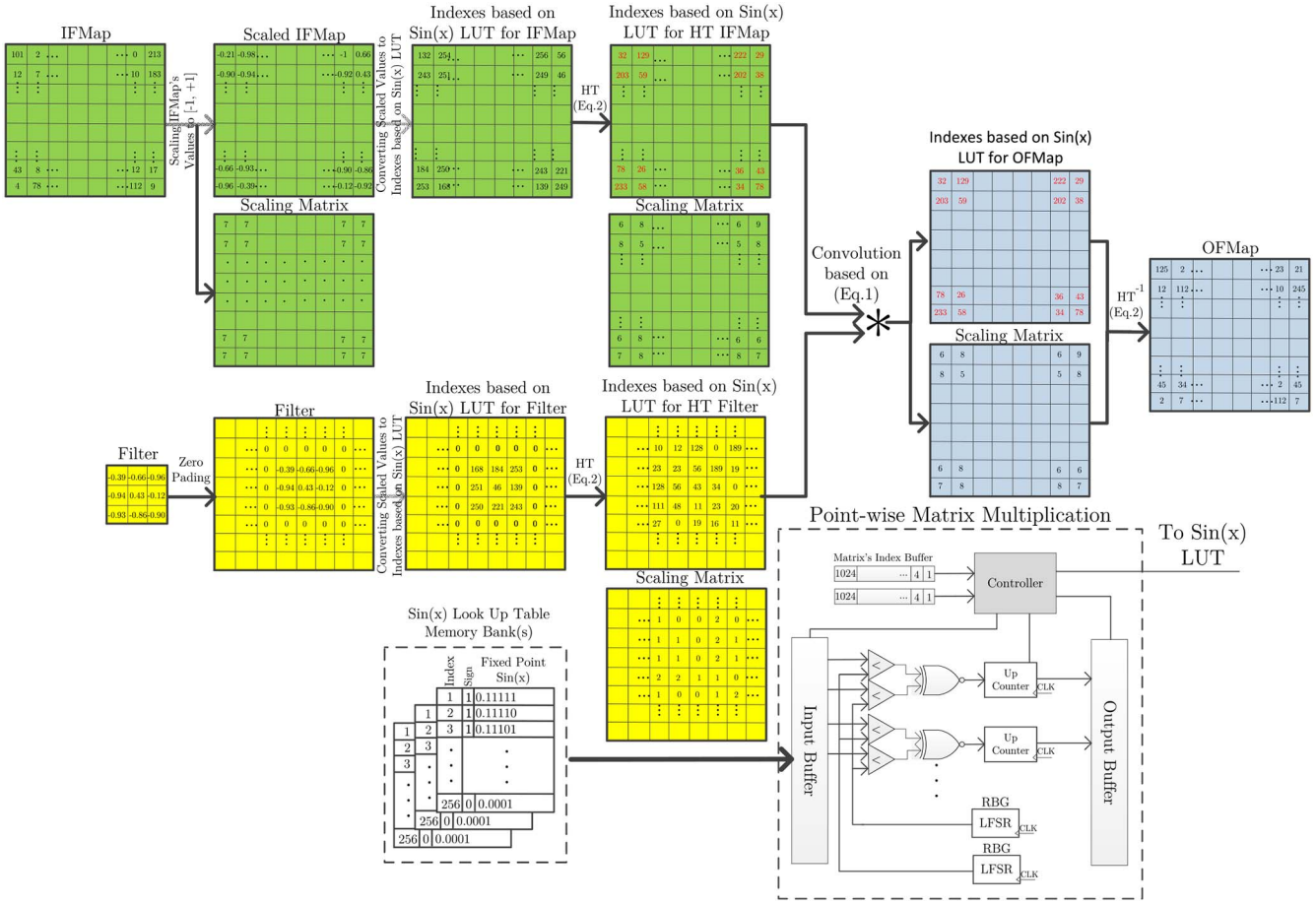


**FIGURE 11.** The dataflow and hardware implementation of HSC-convolution. First, IFMap matrix's values are scaled using the method which is described in Section II-D. This results into two matrices, scaled IFMap and scaling matrix. Then, the scaled IFMap is transformed to angular IFMap matrix using *arcsine* and its corresponding look-up table (refer to Section III-A). Afterwards, we transform the IFMap to the FD using Hartley transform (refer to Eq. (3)). On the other hand, we scale zero-padded filter values. Then, we transform them to the FD using HT. Consequently, we select a sub-set of transformed IFMap to convolve it with filter. We perform this based on point-wise matrix multiplication (PwMM), which is implemented using stochastic computing. In the end, the output of convolution (i.e., OFMap) is transformed back to the SD using $HT^{-1}$ and the angular look-up table. Moreover, you can see the micro-architecture for the PwMM unit. To perform PwMM, we load the indexes of the matrix elements that should be multiplied together in *matrix index buffers*, afterward the controller asks the memory bank(s) that save sin(x) look-up table to send the corresponding binary values to *input buffer*. Then, the corresponding SC bit streams for the values of matrix elements are generated using random bit generator (RBG) and comparators. Consequently, *XNOR* performs the point-wise multiplication and a up-counter transforms back the result to binary from SC. Subsequently, the binary output is saved to *output buffer* before being transferred to memory bank(s).

substitute for non-linear activation functions in the spatial domain, which could make HSC-CNNs even more efficient. Also, the spectral pooling that we introduced in this paper (refer to Section III-B1) could substitute for the pooling operation in the spatial domain if we implement the entire network in the frequency domain. Implementing activation and pooling functions in the frequency domain for the HSC-CNN could result considerable saving in computation as they eliminate the need for repetitive domain transformation in each convolutional layer.

## APPENDIX

### A. ALGORITHM FOR TRAINING A MULTI-LAYER CNN WITH ADAPTIVE-RATE IFMAP COMPRESSION (PRUNING) IN THE FD

See Algorithm 1.

### B. THE MICRO-ARCHITECTURE OF THE HARTLEY STOCHASTIC-BASED CONVOLUTION FOR CNNS

See Figure 11.

## REFERENCES


[1] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei, "Deep convolutional neural network architecture with reconfigurable computation patterns," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 8, pp. 2220–2233, Aug. 2017.

[2] J. H. Ko, B. Mudassar, T. Na, and S. Mukhopadhyay, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–59.

[3] N. Suda *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays*, Feb. 2016, pp. 16–25.

[4] V. Dhandapani and S. Ramachandran, "Area and power efficient DCT architecture for image compression," *EURASIP J. Adv. Signal Process.*, vol. 1, no. 1, p. 180, Dec. 2014.

[5] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun, "Fast convolutional nets with FBFFT: A GPU performance evaluation," Apr. 2015, *arXiv:1412.7580*.

[6] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," Nov. 2015, *arXiv:1509.09308*.

[7] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. London, U.K.: Pearson, 2008.

[8] B. Yuan, Y. Wang, and Z. Wang, "Area-efficient scaling-free DFT/FFT design using stochastic computing," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 63, no. 12, pp. 1131–1135, Dec. 2016.

[9] J. Li, S. You, and A. Robles-Kelly, "A frequency domain neural network for fast image super-resolution," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, 2018, pp. 1–8.
[10] S. Xue, W. Qiu, F. Liu, and X. Jin, "Faster image super-resolution by improved frequency-domain neural networks," *Signal Image Video Process.*, vol. 14, no. 2, pp. 257–265, 2020.
[11] R. N. Bracewell, *The Hartley Transform*, 1st ed. New York, NY, USA: Oxford Univ. Press, Jan. 1986.
[12] S. Svoboda and D. Barina, "New transforms for JPEG format," 2017, *arXiv:1705.03531*.
[13] Z. Liu, J. Xu, X. Peng, and R. Xiong, "Frequency-domain dynamic pruning for convolutional neural networks," in *Proc. Neural Inf. Process. Syst. (NeurIPS)*, Dec. 2018, pp. 1051–1061.
[14] E. Oyallon, E. Belilovsky, S. Zagoruyko, and M. Valko, "Compressing the input for CNNs with the first-order scattering transform," in *Proc. Computer Vision—ECCV*, Munich, Germany, 2018, pp. 305–320.
[15] E. Dupont, A. Goliński, M. Alizadeh, Y. W. Teh, and A. Doucet, "COIN: Compression with implicit neural representations," Mar. 2021, *arXiv:2103.03123*.
[16] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," Oct. 2015, *arXiv:1506.02626*.
[17] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," Feb. 2016, *arXiv:1510.00149*.
[18] A. Ardakani, F. Leduc-Primeau, N. Onizawa, T. Hanyu, and W. J. Gross, "VLSI implementation of deep neural network using integral stochastic computing," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 10, pp. 2688–2699, Oct. 2017.
[19] M. Alawad and M. Lin, "Stochastic-based deep convolutional networks with reconfigurable logic fabric," *IEEE Trans. Multi Scale Comput. Syst.*, vol. 2, no. 4, pp. 242–256, Oct. 2016.
[20] S. R. Faraji, M. H. Najafi, B. Li, D. J. Lilja, and K. Bazargan, "Energy-efficient convolutional neural networks with deterministic bit-stream processing," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Mar. 2019, pp. 1757–1762.
[21] A. Alaghi and J. P. Hayes, "Survey of stochastic computing," *ACM Trans. Embedded Comput. Syst.*, vol. 12, pp. 1–92, May 2013.
[22] R. Hojabr *et al.*, "SkippyNN: An embedded stochastic-computing accelerator for convolutional neural networks," in *Proc. ACM/IEEE Design Autom. Conf. (DAC)*, Jun. 2019, p. 132.
[23] G. Hou, "The fast hartley transform algorithm," *IEEE Trans. Comput.*, vol. C-36, no. 2, pp. 147–156, Feb. 1987.
[24] D. Forsyth and J. Ponce, *Computer Vision: A Modern Approach*, 2nd ed. Upper Saddle River, NJ, USA: Prentice-Hall, Nov. 2011.
[25] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–13.
[26] A. Alaghi, W. Qian, and J. P. Hayes, "The promise and challenge of stochastic computing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 8, pp. 1515–1531, Aug. 2018.
[27] C. Wilcox, M. Strout, and J. Bieman, "Tool support for software lookup table optimization," *Sci. Program.*, vol. 19, pp. 213–229, Dec. 2011.
[28] D. Gudovskiy, A. Hodgkinson, and L. Rigazio, "DNN feature map compression using learned representation over GF(2)," in *Proc. Comput. Vis.—ECCV Workshops*, 2019, pp. 502–516.
[29] Y. Guo, A. Yao, and Y. Chen, "Dynamic network surgery for efficient DNNs," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, Dec. 2016, pp. 1387–1395.
[30] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *Proc. Int. Conf. Neural Inf. Process. Syst. (NIPS)*, 2014, pp. 1–9.
[31] T. Na and S. Mukhopadhyay, "Speeding up convolutional neural network training with dynamic precision scaling and flexible multiplier-accumulator," in *Proc. Int. Symp. Low Power Elect. Design*, Aug. 2016, pp. 58–63.
[32] M. Mathieu, M. Henaff, and Y. LeCun, "Fast training of convolutional networks through FFTs," Mar. 2014, *arXiv:1312.5851*.
[33] T. Abtahi, C. Shea, A. Kulkarni, and T. Mohsenin, "Accelerating convolutional neural network with fft on embedded hardware," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 9, pp. 1737–1749, Dec. 2018.
[34] W. K. Pratt, *Introduction to Digital Image Processing*. Boca Raton, FL, USA: CRC Press, Sep. 2013.
[35] B. Yuan, Y. Wang, and Z. Wang, "Area-efficient error-resilient discrete Fourier transformation design using stochastic computing," in *Proc. 26th Ed. Great Lakes Symp. VLSI*, May 2016, pp. 33–38.
[36] R. D. Evans, L. Liu, and T. M. Aamodt, "JPEG-ACT: Accelerating deep learning via transform-based lossy compression," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 860–873.
[37] S. Fujieda, K. Takayama, and T. Hachisuka, "Wavelet convolutional neural networks for texture classification," Jul. 2017, *arXiv:1707.07394*.
[38] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," Feb. 2016, *arXiv:1604.03168*.
[39] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," Sep. 2015, *arXiv:1412.7024*.
[40] W. Tang, G. Hua, and L. Wang, "How to train a compact binary neural network with high accuracy?" in *Proc. 31st AAAI Conf. Artif. Intell.*, San Francisco, CA, USA, Feb. 2017, pp. 2625–2631.
[41] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-net: ImageNet classification using binary convolutional neural networks," Aug. 2016, *arXiv:1603.05279*.
[42] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional neural networks using logarithmic data representation," Mar. 2016, *arXiv:1603.01025*.
[43] T. Liang, L. Wang, S. Shi, and J. Glossner, "Dynamic runtime feature map pruning," Apr. 2019, *arXiv:1812.09922*.
[44] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through L0 regularization," Jun. 2018, *arXiv:1712.01312*.
[45] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 269–284, Feb. 2014.
[46] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 662–673.
[47] A. Page, A. Jafari, C. Shea, and T. Mohsenin, "Sparcnet: A hardware accelerator for efficient deployment of sparse convolutional networks," *J. Emerg. Technol. Comput. Syst.*, vol. 13, no. 3, pp. 1–32, May 2017.
[48] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–12.

**S. H. MOZAFARI** (Student Member, IEEE) received the B.Sc. degree in computer engineering from K. N. Toosi University of Technology, Tehran, Iran, in 2009, the M.Sc. degree in computer engineering from the Sharif University of Technology, Tehran, in 2011, and the Ph.D. degree in electrical and computer engineering from McGill University, Montreal, QC, Canada, where he is currently working as a Postdoctoral Fellow in electrical and computer engineering. His research interests include the area of hardware-aware deep neural network optimization, digital system design, and fault tolerant design for digital circuits.

**J. J. CLARK** (Senior Member, IEEE) received the B.A.Sc. and Ph.D. degrees in electrical engineering from the University of British Columbia in 1980 and 1985 respectively. In 1985, he joined the Division of Applied Sciences, Harvard University as a Postdoctoral Fellow and as a Instructor. In 1986, he was appointed as an Assistant Professor and then to the rank of Gordon McKay Associate Professor in 1990. In 1994, he joined the Nissan Cambridge Basic Research Lab as a Visiting Scientist. In 1996, he moved to McGill University joining the Department of Electrical Engineering as an Associate Professor. He was promoted to a Full Professor in 2004. From 2005 to 2011, he was an Associate Dean, an Academic with the Faculty of Engineering. From 2013 to 2019, he served as the Director of the McGill Research Centre for Intelligent Machines. Since 2020, he has been the Co-Director of McGill's Retail Innovation Lab. In 2014, he was honored by the Canadian Image Processing and Pattern Recognition Society with their annual Award for Research Excellence. He is the Co-General Chair for the 2021 International Conference on Computer Vision.

**W. J. GROSS** (Senior Member, IEEE) received the B.A.Sc. degree in electrical engineering from the University of Waterloo, Waterloo, ON, Canada, in 1996, and the M.A.Sc. and Ph.D. degrees from the University of Toronto, Toronto, ON, Canada, in 1999 and 2003, respectively.

He is currently a James McGill Professor and the Chair with the Department of Electrical and Computer Engineering, McGill University, Montreal, QC, Canada. His research interests are in the design and implementation of signal processing systems and custom computer architectures. He served as the Chair for the IEEE Signal Processing Society Technical Committee on Design and Implementation of Signal Processing Systems. He served as the General Co-Chair for IEEE/ACM NANOARCH 2021, IEEE GlobalSIP 2017 and IEEE SiPS 2017, and the Technical Program Co-Chair for SiPS 2012. He served as an Associate Editor for the IEEE TRANSACTIONS ON SIGNAL PROCESSING and as a senior area editor. He is a Licensed Professional Engineer in the Province of Ontario.

**B. H. MEYER** (Senior Member, IEEE) received the B.S. degree in electrical engineering, computer science and math from the University of Wisconsin–Madison in 2003, and the M.S. and Ph.D. degrees in electrical and computer engineering from Carnegie Mellon University in 2005 and 2009, respectively. He is an Associate Professor with the Department of Electrical and Computer Engineering, McGill University. After receiving his Ph.D., he worked as a Postdoctoral Research Associate with the Computer Science Department, University of Virginia. He has been on the Faculty with McGill since 2011. His research interests are diverse, but tend toward architectures and design algorithms for embedded computer systems; embedded system security and machine learning are new areas of interest. His research, published in more than 50 articles, has been recognized with Best Paper in Session awards at SRC TECHCON in 2007 and 2013, nominations for Best Paper at ASPDAC 2014 and GLSVLSI 2015, and the Best Paper Award at DFT 2015. He received the William and Rhea Seath Award in Engineering Innovation in 2018. He has served on the Technical Program Committees for CASES, CODES+ISSS, DAC, DATE, GLSVLSI, ICCAD, ISLPED, among others, and as the TPC Chair for AHS 2015 and RSP in 2019.