IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 30, 2022, accepted February 25, 2022, date of publication March 8, 2022, date of current version March 31, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3157325

Efficient, Geometry-Based Convolution

CHANON KHONGPRASONGSIRI, (Student Member, IEEE), WATCHARAPAN SUWANSANTISUK, (Member, IEEE), AND PINIT KUMHOM

Department of Electronic and Telecommunication Engineering, King Mongkut's University of Technology Thonburi, Bangkok 10140, Thailand

Corresponding author: Pinit Kumhom (pinit.kumhom@mail.kmutt.ac.th)

This work was supported by the King Mongkut's University of Technology Thonburi under the Petch Pra Jom Klao Master's Degree Research Scholarship. The work of Watcharapan Suwansantisuk was supported in part by the Faculty of Engineering, King Mongkut's University of Technology Thonburi, under the Research Strengthening Project.

ABSTRACT Several computationally intensive applications in machine learning, signal processing, and computer vision call for convolution between a fixed vector and each of the incoming vectors. Often, the convolution need not be exact because a subsequent processing unit, such as an activation function in a neuron network or a visual unit in image processing, can tolerate a computational error, hence allowing the optimization of the convolution algorithm. This paper develops a method of approximate convolution and quantifies its performance in software and hardware. The key idea is to take advantage of the known fixed vector, view a convolution as a dot product, and approximate the angles between the fixed vector and an incoming vector geometrically. We evaluate the proposed method in terms of the accuracy, running time complexity, and hardware power consumption on the field programmable gate array (FPGA) and application-specific integrated circuit (ASIC) hardware platforms. In a benchmark test, the accuracy of the approximate convolution is 3.7% lower than that of the exact convolution, a tolerable loss for machine learning and signal processing. The proposed method reduces the number of operations in the hardware, and reduces the power consumption of conventional convolution by approximately 20% and the existing approximate convolution by approximately 10%, while maintaining the same throughput and latency. We also test the proposed method on 2D convolution and convolutional neural network (CNN). The proposed method reduces complexity, power consumption for 2D convolution, and power consumption for CNN of the conventional method by approximately 22%, 25%, and 13%, respectively. The proposed method of approximate convolution trades off accuracy with running time complexity and hardware power consumption, and it has practical utility in computationally intensive tasks that tolerate a margin of convolutional error.

INDEX TERMS Convolution, dot product, power consumption, FPGA.

I. INTRODUCTION

Discrete-time convolution is an important operation in machine learning, signal processing, and computer vision [1]. For example, the neural network architecture AlexNet [1], [2] uses five embedded convolution layers for image recognition to achieve a recognition rate of 84.7%. ResNet-152 [3] uses 152 convolution layers to improve classification accuracy and achieve a recognition rate of 96.5%. The more convolution layers there are, the better the performance is, leading to a massive use of convolution operations.

Although extensive convolution improves performance in one area, it undesirably leads to high computational complexity [4] and power consumption. AlexNet takes 1.4 giga operations per second (GOPS) to process a single 224×224-pixel^2 image, while ResNet takes 22.6 GOPS [5]. This high computational complexity prevents massive convolution from taking place on ubiquitous small devices and Internet of Things (IoT) nodes that have limited computational resources. An extensive number of convolutions also leads to high power consumption, mainly due to memory access and processing elements in the hardware. To see the amount of power used, consider a typical example of massive signal filtering, which performs convolution 10 million times, each time between two 1000-element vectors. On hardware using 45 nm CMOS technology, one memory access consumes 5 pJ of power on a 32-bit SRAM cache and 640 pJ on DRAM [6]. Massive convolution is not possible on a fast SRAM cache because the power consumption of $(10M - 1000)(1000)(5\text{pJ}) \approx 0.005W$ on the

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

local memory is too large. In this example, the convolution must be performed on a slower DRAM, consuming $(10M - 1000)(1000)(640\text{pJ}) \approx 6.4W$ of power, an amount similar to CPU power usage [7]. High computational complexity and power consumption are bottlenecks in hardware and require a new design strategy for convolution.

Applications that require massive convolution have a common characteristic that can be exploited for efficiency and lower power consumption, namely, a tolerance for computational errors [8]–[10]. In image processing, a small visual error in an output image is acceptable and often undetectable to humans due to limitations in human perception. In deep-learning classification, exact computation of convolution at a neuron is less significant than the method of inference that is being used. Convolutional neural networks tolerate errors from approximated schemes [11] and accept computations with 4-bit precision without sacrificing inference accuracy [12]. Furthermore, many digital signal processing applications rely on probabilistic models, which have been designed to deal with noisy data. A trade-off between accuracy and hardware-software efficiency is the key to developing a method of convolution.

In this paper, we develop a method of approximate convolution that is sufficiently accurate and consumes low power. This method preprocesses the known, fixed vector in convolution to save subsequent computational effort, treats the convolution as a dot product between two vectors, and efficiently approximates the angles between the two vectors geometrically. The main contributions of this paper are the following:

- A method of approximate convolution that is suitable for massive error-tolerant applications;
- The hardware architecture of the proposed method for FPGA and ASIC platforms;
- A comparison of the proposed method with state-of-the-art methods in terms of accuracy, running time complexity, number of hardware operations, throughput, latency and power consumption.

The proposed method reduces the number of operations in hardware, reduces the power consumption by approximately 25%, and reduces the area by approximately 13%, while incurring a loss of approximately 3.8% in accuracy compared to conventional convolution. The proposed method has practical utility in machine learning, signal processing, and computer vision applications that require massive convolution and can tolerate computational errors.

The rest of the paper is organized as follows. We review the existing literature in Section II and give the problem statement in Section III. We develop the proposed method of approximate convolution in Section IV and describe the hardware implementation architectures for 1D and 2D convolutions in Section V. Finally, we evaluate and discuss the performance of the proposed method in Section VI, and conclude the important findings in Section VII.

II. RELATED WORK

Methods of reducing the computational complexity and power consumption of convolution can be broadly classified into two categories: exact convolution and approximate convolution. Exact convolutions are suitable for precise applications, which cannot tolerate errors, and they reduce power consumption by optimizing multiplications in the hardware. Approximate convolutions are suitable for error-tolerant applications, such as those focused on in this paper. We review the related work in both categories.

Exact convolutions divide a convolution operation into subproblems that involve the multiplication of the input vector with constants [13]. To optimize multiplication, related work [14]–[16] uses an adder graph tree structure that expresses addition, subtraction, and bit shifting. The nodes of the graph represent an adder or subtractor, and the edge weights belong to the bit shifts. These tree structures dominate the number of multiplications, while the number of addition and subtraction operations significantly increases, which is the main issue for constant multiplication. Kumm [14] obtains the optimal number of addition or subtraction operations by reducing the adder graph. To implement this design, Kumm *et al.* [16] uses a pipeline architecture to optimize the adder depth and timing constraints. A fast, exact multiplication results in a fast, exact convolution.

In addition to research work that optimizes a multiplication, several studies treat an exact convolution as a matrix-vector multiplication [17]–[25]. To reduce the complexity, a kernel expansion is used with the approximate Fastfood transform [18]. Other methods that use a fast convolution algorithm include the FFT convolution and Winograd convolution, which transform matrix multiplications into the Hadamard products [19]. Fast-convolution algorithms reduce the number of operations from $\Theta(n^2)$ to $\Theta(n \log_2 n)$ and incur some errors from the kernel transformation, where n is the number of rows and the number of columns of the convolution matrix for the Fastfood and FFT convolutions. For the Winograd convolution [25], many transformed matrices exist to provide fast results, although memory issues may occur when amortizing those matrices [21]–[24]. Another issue of the Winograd convolution is numerical instability [20], which occurs, for example, in an implementation [26] due to huge kernels. Fast convolution methods can be implemented in hardware using a parallel and pipeline architecture, which results in a high throughput. However, a drawback of fast convolution methods is in large power consumption. Related work in the category of exact convolution focuses on resource utilization and throughput.

Approximate convolutions use approximate adders or approximate multipliers in circuits. K. Du [27] introduces carry-select adders, which gain speed at the expense of circuit area and achieve mean relative error distances of approximately 2% to 10% compared to a 16-bit adder. Another architecture is the approximate full adder, which modifies the truth table of the full adder circuit [28], [29]. This architecture

occupies a small circuit area and reduces the delay of its throughput by approximately 50%, but it has an accuracy of approximately 60%. This level of accuracy is considered low and may cause errors to accumulate in a massive convolution operation. On the other hand, approximate multipliers can be achieved with one of these two approaches. The first approach is to approximate a partial product by using the K-Map for a 2 × 2 bit multiplier to create a 4 × 4 bit multiplier [30]. This approach also includes the approximate multiplication of [10], which generates the partial product by shifting the bits of a multiplicand and has an approximately 10% mean relative error. The second approach is to approximate a multiplier by using approximate compressors [31]–[40], such as the 4-2 approximate compressor. Although the accuracy of the approximate multiplier depends on the multiplication architecture, the mean relative error distance is often 5–15%, while the power reduction compared to exact multiplication is approximately 10–50%. In other words, the greater the power reduction is, the lower the accuracy of the multipliers.

Existing methods of approximate convolution are limited in a fundamental way. The number of addition and multiplication operations cannot be reduced due to the computation equation, which accounts for the restriction of power consumption improvement while maintaining the accuracy of the system. A method that addresses these limitations will break through the restriction on the number of operations, leading to power reduction while sacrificing little accuracy.

III. PROBLEM STATEMENT

We are interested in exploring a convolution implementation that yields better performance per watt by sacrificing some computational accuracy. The performance of an implementation of an algorithm can be measured by means of throughput and latency. To gain better performance, we usually have to pay a price in terms of the area and power consumption of the hardware. In general, for a given computation task, multiple algorithms usually exist by which correct computation results are produced with different performance levels. In turn, there exist various approaches for implementing a given algorithm on a chosen computation platform, resulting in a very large space of possible solutions. Hence, we need to scope down this solution space.

In this paper, we target our convolution implementation on an embedded computation platform whose computational resources are limited in terms of both area and available power, while the needs for high throughput and low latency continue to grow. One approach for overcoming these challenges is to implement the computation of interest as hardware, which is a digital system specifically designed to perform the computation problem. In particular, we target our design for an FPGA or ASIC chip.

With the target hardware setup, we are interested in convolution computation by the following definition. Let x and h denote two discrete-time functions acting as the input signal and the impulse response, respectively. We are interested in applying convolution in cases where the impulse response h has a finite number of tabs (weights) that are predetermined. The input signal x is a window of a streaming discrete-time signal. Moreover, this implementation is for applications that can tolerate some computational error at each sample point of the convolution. In fact, such errors are already inherited in the digital abstraction due to the finite representations of numbers. Furthermore, since we aim for an embedded implementation, we adopt an n-bit fixed-point representation in which the zero level is 2^{n-1} for each sample of h and x. Based on this setup, we define the computation task as follows.

TABLE 1. Vector $\mathbf{x}_k$ defined in Eq. (4).

k	$\mathbf{x}_k$
0	$\mathbf{x}_0=(x[0],0,\ldots,0)^T$
1	$\mathbf{x}_1=(x[1],x[0],\ldots,0)^T$
$\vdots$	$\vdots$
$N-1$	$\mathbf{x}_{N-1}=(x[N-1],x[N-2],\ldots,x[0])^T$
N	$\mathbf{x}_N=(x[N],x[N-1],\ldots,x[1])^T$
$\vdots$	$\vdots$
$M-1$	$\mathbf{x}_{M-1}=(x[M-1],\ldots,x[M-N])^T$
M	$\mathbf{x}_M=(0,x[M-1],\ldots,x[M-N+1])^T$
$\vdots$	$\vdots$
$M+N-1$	$\mathbf{x}_{M+N-1}=(0,0,\ldots,0,x[M-1])^T$

Let $x[j]$ denote the jth sample point of the streaming input x, for $j = 0, 1, \ldots, M-1$ and let $h[m]$ denote the mth coefficient (tab) of the filter impulse function h, for $m = 0, 1, 2, \ldots, N-1$. Without loss of generality, and by switching the computation task of interest, the convolution of the function x of M points with the function h of N points can be described by Eqs. 1 and 2:

$$\mathbf{y} = (y[0], y[1], \ldots, y[M-1])^T = x * h \quad (1)$$

$$y[k] = \sum_{j=0}^{N-1} x[j]h[k-j] = \sum_{j=0}^{N-1} x[k-j]h[j] = \mathbf{h} \cdot \mathbf{x}_k, \quad (2)$$

where $\mathbf{h}$ is a vector of size N that stores the coefficients (tabs) of the filter impulse function h (Eq. 3) and $\mathbf{x}_k$ is a vector storing the N points of data collected from the sliding window covering the input samples from point $k-N+1$ to k as described in Eq. (4), where $x[j] = 0$ for $j < 0$ and for $j \geq M$:

$$\mathbf{h} = (h[0], h[1] \ldots, h[N-1])^T \quad (3)$$

$$\mathbf{x}_k = (x[k], x[k-1], \ldots, x[k-N+1])^T . \quad (4)$$

Table 1 lists the vectors for M samples of input x; i.e., $x[k]$ for $k = 0, 1, \ldots, M+N-1$. Based on this setup, the proposed approach is described in the next section.

IV. PROPOSED METHOD

A. PROBLEM ANALYSIS

Let us consider the computation of consecutive convolution points $y[k]$ and $y[k+1]$ following Eq. (2). Note that $y[k]$ and $y[k+1]$ are the dot products of the same $\mathbf{h}$ with $\mathbf{x}_k$ and

$\mathbf{x_{k+1}}$, respectively. Since the data vectors $\mathbf{x}_k$ and $\mathbf{x_{k+1}}$ are the results of two consecutive sliding windows, the only element of x_{k+1} obtaining a new sample is $\mathbf{x}_{k+1}[N-1] = x_{k+1}$, while $\mathbf{x}_{k+1}[j] = \mathbf{x_k}[j+1]$, for $j = 0, \ldots, N-2$. However, this nice relationship does not help reduce the complexity of the dot product computation; i.e., it still requires a number of basic computations, multiplication and accumulation (MAC), on the order of $\Theta(N)$. In this paper, we aim to exploit this relationship to reduce the computational complexity by sacrificing some computation accuracy.

Our proposed idea is based on the geometric definition of the dot product following Eq. (5):

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{N} a_i b_i = |\mathbf{a}||\mathbf{b}| \cos\theta, \tag{5}$$

where $\mathbf{a}$ and $\mathbf{b}$ are two vectors of the same size. Applying the dot product's geometric interpretation to the computation of $y[k]$ and $y[k+1]$, we have

$$\begin{aligned} y[k] &= \mathbf{h} \cdot \mathbf{x}_k \\ &= |\mathbf{h}||\mathbf{x}_k| \cos\theta_k \end{aligned} \tag{6}$$

$$\begin{aligned} y[k+1] &= \mathbf{h} \cdot \mathbf{x}_{k+1} \\ &= |\mathbf{h}||\mathbf{x}_{k+1}| \cos\theta_{k+1}, \end{aligned} \tag{7}$$

where θ_k is the angle between vectors $\mathbf{x}_k$ and $\mathbf{h}$.

As $|\mathbf{h}|$ are precomputed because the vector $\mathbf{h}$ is known, we need to calculate $|\mathbf{x}_k|$, $\cos\theta_k$, $|\mathbf{x_{k+1}}|$, and $\cos\theta_{k+1}$. We can exploit the relationship of $\mathbf{x}_k$ and $\mathbf{x_{k+1}}$ in computing $|\mathbf{x}_k|$ and $|\mathbf{x_{k+1}}|$ as follows:

$$|\mathbf{x}_k|^2 = x[k]^2 + x[k-1]^2 + \ldots + x[k-N+1]^2 \tag{8}$$

$$\begin{aligned} |\mathbf{x_{k+1}}|^2 &= x[k+1]^2 + x[k]^2 + \ldots + x[k-N+2]^2 \\ &= |\mathbf{x}_k|^2 - x[k-N+1]^2 + x[k+1]^2. \end{aligned} \tag{9}$$

Due to the shifting property of $\mathbf{x}_k$ and $\mathbf{x}_{k+1}$, we can reduce the number of squares in computing the two magnitudes from $2N$ to $N+2$ by amortizing a large part of $|x_{k+1}|^2$ in $|x_k|^2$. Moreover, since we have to compute at least N consecutive dot products, we can reduce the square computations from N^2 to $3N-2$. Combined with the two multiplications among $|\mathbf{h}|$, $|\mathbf{x}_k|$, and $\cos\theta_k$ for each $y[k]$, the overall number of multiplications for computing N-point convolution is approximately $5N$ given that $\cos\theta_k$ is computed separately. Although this computation is on the same order as the original computation, the hardware architecture for implementing this computation is simple. This advantage, however, comes with the price that we need to compute the value of $\cos\theta_k$ for each computation of y_k. Next, we explain the proposed approach to tackle $\cos\theta_k$.

Finding $\cos\theta_k$ is a challenge when the vector's dimension is large. A known algorithm for finding θ is by using a rotation matrix [41]. However, this method is only practical in 2D space because finding a rotation matrix is difficult for a high-dimensional space. Moreover, the complexity of applying the rotation matrix to find θ is also high. Given that the coordinates of the two vectors are known, the best way to find the angle θ between the two vectors is to use the dot product's geometric definition as in Eq. (10):

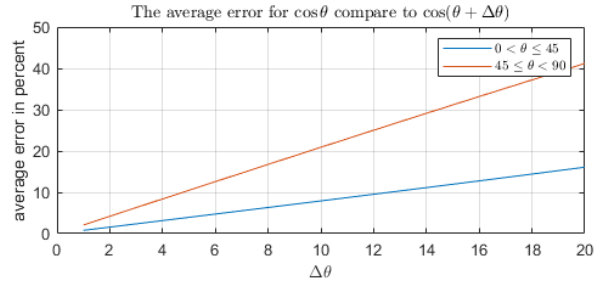


FIGURE 1. The percent error between $\cos\theta$ and $\cos(\theta + \Delta\theta)$.

$$\theta = \arccos\left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|}\right). \tag{10}$$

However, in our case, the dot product is what we want to compute. We need a clever method to apply Eq. (10).

We propose an approximate computation of $\cos\theta_k$. Based on Eq. (5), we observe the effects of the drift of θ on $\cos\theta$ by plotting the percent errors of $\cos\theta$ against the drift in θ denoted by $\Delta\theta$, as shown in Fig. 1. We partition the variable θ from 0° to 90° in two parts, $0° < \theta \leq 45°$ and $45 < \theta \leq 90°$. Since the rate of change in $\cos\theta$ increases as θ does, the error in $\cos\theta$ of the upper part of the angle is higher, as shown in Fig. 1. Nevertheless, if we can maintain the drift in θ at approximately 5 degrees, the average absolute difference for $\cos\theta$ and $\cos(\theta + \Delta\theta)$ is approximately 5% and 10% for the lower and higher ranges of θ, respectively. These results indicate that approximation of θ_k can tolerate higher errors.

Given that $\mathbf{h}$ is known, the two keys of our proposed implementation of convolution computation are as follows:

(1) we propose an efficient method for approximating θ_k so that it can be applied in computing convolution using the geometric definition of the dot product with the assumption that some computation error is allowed, and
(2) the computation of the magnitude of $\mathbf{x}_{k+1}$, the input vector for generating $y[k+1]$, is amortized due to the relation of $\mathbf{x}_{k+1}$ and $\mathbf{x}_k$.

We explain the proposed method for the θ_k approximation in the next subsection. Then, the overall convolution computation based on the geometric definition of the dot product is described.

B. VECTOR ANGLE APPROXIMATION

Given a constant vector $\mathbf{h}$ in a vector space of ℓ dimensions, we want to estimate the angle θ of vector $\mathbf{x}$ with reference to $\mathbf{h}$. Let us consider the case of vectors in the 3-dimensional vector space shown in Fig. 2 as an illustration of the proposed idea. To simplify the discussion and without loss of generality, we will assume that the coordinates of two vectors, the constant vector $\mathbf{h} = (h_1, h_2, h_3)^T$ and the input vector $\mathbf{x} = (x_1, x_2, x_3)^T$, lie in the positive cube. This assumption is generalized by the fact that we represent data using an n-bit unsigned number; i.e., all components of the two vectors,

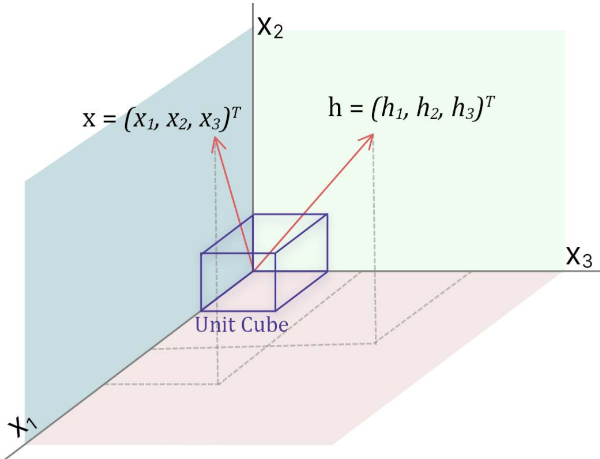


FIGURE 2. All possible cases of $\mathbf{x}_{\text{bin}}$ in a 3-dimensional vector space are seven nonzero vectors pointing to the corners of the positive unit cube.

$h_1, h_2, h_3, x_1, x_2,$ and x_3, are zero or positive. The proposed angle approximation method is based on the following ideas:

- We start with the idea of replacing $\mathbf{x}$ with a vector, denoted by $\hat{\mathbf{x}}$, that has the same direction as $\mathbf{x}$, by which the angle can be computed following Eq. (11):

$$\theta = \arccos\left(\frac{\hat{\mathbf{x}} \cdot \mathbf{h}}{|\mathbf{h}||\hat{\mathbf{x}}|}\right). \tag{11}$$

  The first step in this approach is to find $\hat{\mathbf{x}}$ such that the computation $\hat{\mathbf{x}} \cdot \mathbf{h}$ has less complexity.
- To this end, we propose using a binary vector of size N, denoted by $\mathbf{x}_{\text{bin}}$ which approximate $\hat{\mathbf{x}}$ and whose coordinates can be expressed as a 1-bit number in each dimension. Fig. 2 illustrates this idea for the case of 3D space. With this idea, the vector $\mathbf{x}$ is quantized to one of the seven nonzero corner vectors of the unit cube. Extending the idea to an N-dimensional space, $\mathbf{x}_{\text{bin}}$ can be generalized as Eq. (12).

$$\mathbf{x}_{\text{bin}} = (b_1, b_2, \cdots, b_N)^T, \quad b_j \in \{0, 1\} \tag{12}$$

$$|\mathbf{x}_{\text{bin}}| = \sqrt{\sum_{k=1}^{n} b_k^2} = \sqrt{\text{number of nonzero } b_k} \tag{13}$$

  Using $\mathbf{x}_{\text{bin}}$, we eliminate the multiplications in the computation of the dot product $\mathbf{x}_{\text{bin}} \cdot \mathbf{h}$ as described in Eq. 14 since the element b_j in dimension j of $\mathbf{x}_{\text{bin}}$ is either 0 or 1; i.e., the dot computation is just a sum of the elements j of $\mathbf{h}$ for all j such that $b_j = 1$. As a result, the approximation of θ, denoted by $\hat{\theta}(x_{\text{dot}})$, can be calculated following Eq. (15):

$$x_{\text{dot}} = \mathbf{x}_{\text{bin}} \cdot \mathbf{h} = \sum_{\forall j \text{ s.t. } b_j=1} \mathbf{h}[j] \tag{14}$$

$$\hat{\theta}(x_{\text{dot}}) = \arccos\left(\frac{x_{\text{dot}}}{|\mathbf{x}_{\text{bin}}||\mathbf{h}|}\right). \tag{15}$$

  For example, in the 3D space, $\mathbf{x}_{\text{bin}}$ will be one of the seven nonzero coordinates at the corners of the unit cube, as shown in Fig. 2; i.e., for the input vector $\mathbf{x} = (x_1, x_2, x_3)^T$, where x_j is encoded as an n-bit unsigned number, the estimated vector $\mathbf{x}_{\text{bin}}$ of $\mathbf{x}$ will be one of the seven nonzero 3-bit binary tuples $(b_1, b_2, b_3)^T$, which are $(0, 0, 1)^T$, $(0, 1, 0)^T$, $(0, 1, 1)^T$, $(1, 0, 0)^T$, $(1, 0, 1)^T$, $(1, 1, 0)^T$, and $(1, 1, 1)^T$. Note that for an ℓ-dimensional space, there are $2^\ell - 1$ possible cases of $\mathbf{x}_{\text{bin}}$, corresponding to the $2^\ell - 1$ nonzero combinations of the ℓ-bit $(b_1, b_2, \cdots, b_\ell)^T$. The idea is to quantize each vectors in the ℓ-dimensional space to one of the $2^\ell - 1$ vectors. Although this seems to be a rough estimate for the angle θ, it yields a much better estimate for $\cos\theta$, especially for $\theta \leq 45°$, because $\cos\theta$ changes slowly for small θ; i.e., with a limited n-bit representation of numbers, $\cos\theta$ is the same for a wide range of θ. Moreover, θ is quantized in the look-up table (LUT) implementation of $\cos\theta$, as shown in Fig. 3. However, this estimation method will have a greater effect for $\theta_k > 45°$.

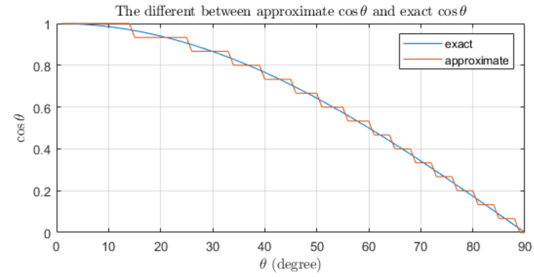


FIGURE 3. The approximate $\cos\theta$ and the exact $\cos\theta$.

- While using $\mathbf{x}_{\text{bin}}$ reduces the computational complexity of $x_{\text{dot}} = \mathbf{x}_{\text{bin}} \cdot \mathbf{h}$, we still need to deal with the computation of $\arccos\left(\frac{x_{\text{dot}}}{|\mathbf{x}_{\text{bin}}||\mathbf{h}|}\right)$. To do so, we rely on the fact that although $\mathbf{x}$ varies, the vector $\mathbf{h}$ is known and predetermined. Using this fact, we estimate θ by a linear function of x_{dot} as described in Eq. (16) using linear curve fitting to find the parameters P_1 and P_0 for all cases of x_{dot} given $\mathbf{h}$:

$$\hat{\theta}^{(1)} = P_1 x_{\text{dot}} + P_0. \tag{16}$$

  That is, for a given $\mathbf{h}$, we compute $\hat{\theta}(x_{\text{dot}})$ using Eq. (15) for all possible cases of $\mathbf{x}_{\text{bin}}$. Then, parameters P_1 and P_0 are determined using linear regression curve fitting, as shown in Fig. 4 for the case of random $\mathbf{h}$ with dimension $\ell = 20$.
- Note that the parameters P_1 and P_0 are calculated with reference to $\mathbf{x}_{\text{bin}}$, a vector whose direction estimates that of vector $\mathbf{x}$. Therefore, to further improve the estimation of θ, we take random samples of $\mathbf{x}$ and compute the actual θ using Eq. (11) and its estimate $\hat{\theta}^{(1)}$ using the parameters P_1 and P_0 from the previous step. We find that there are shifts of $\hat{\theta}^{(1)}$ from the actual θ. As a result, we compensate for this bias with the parameter B, which is the average of the drifts, resulting in the final equation for angle estimation in Eq. (17):

$$\hat{\theta} = \hat{\theta}^{(1)} - B. \tag{17}$$

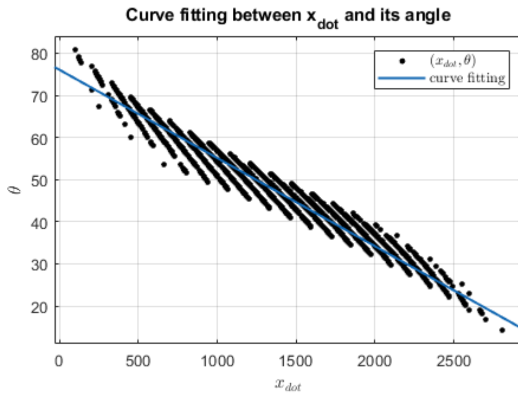


FIGURE 4. The line fitting between x_{dot} and $\hat{\theta}(x_{\text{dot}})$.

Algorithm 1 Algorithm for Estimating the Angle of Vector $\mathbf{x}$ With Reference to a Constant Vector $\mathbf{h}$

Require: Parameters P_1, P_0, and B are predetermined given $\mathbf{h}$

```
function FINDANGLE(x, P1, P0, B, ℓ)
    x_bin ← BinarizedX(x, ℓ)
    x_dot ← x_bin · h
    θ̂ ← P1 x_dot + P0 − B
    return θ̂
end function
```

Based on these ideas, the proposed algorithm for estimating the angle θ between a vector $\mathbf{x}$ and a constant vector $\mathbf{h}$ is described in Algorithm 1.

The next subsection explains the proposed method of finding $\mathbf{x}_{\text{bin}}$ used in the function BinarizedX in Algorithm 1.

C. FINDING $\mathbf{x}_{bin}$

Given $\mathbf{x} = (x_1, x_2, \ldots, x_\ell)^T$, where x_j is an n-bit unsigned number, we want to quantize x_j to the 1-bit b_j. Using the case of 3D space as an example, we want to quantize $\mathbf{x} = (x_1, x_2, x_3)^T$ to the nearest vector $(b_1, b_2, b_3)^T$. Our proposed method is as follows. First, we choose the dominant element by comparing the most significant bits of x_j, which depends on the position of the first nonzero bit of x_j starting from the left-most bit, bit $n_a - 1$. In other words, we find the significant bits of x_j by counting the number of leading zeros in the binary representation of $x_j = (x_j^{n-1} x_j^{n-2} \cdots x_j^0)_2$, where $x_j^m \in \{0, 1\}$ is bit m of x_j; i.e., the smaller the number of leading zeros, the more significant. Let us denote the leading zeros of x_j as lz_j, and let lz_m be the minimum of lz_j; i.e., $lz_m = \min_{\forall j} z_j$. Then, we find b_j of $\mathbf{x}_{\text{bin}}$ using the rule described in Eq. (18):

$$b_j = \begin{cases} 1 & \text{if } lz_j = lz_m \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

To illustrate the method, let us consider two examples in 3D space: $\mathbf{x}_1 = (56, 126, 34)^T$ and $\mathbf{x}_2 = (17, 45, 22)^T$ with an 8-bit binary representation of x_j. For $\mathbf{x}_1 = (56, 126, 34)^T$, the numbers of leading zeros in the 8-bit binary representations of $56 = 00111000_2$, $126 = 01111110_2$, and $34 = 00100010_2$ are 2, 1, and 2, respectively. Hence, the second element, x_2, is the only element of $\mathbf{x}_1$ with the most significant value ($lz_2 = lz_m = 1$). This results in $\mathbf{x}_{\text{bin}} = (0, 1, 0)^T$ for $\mathbf{x}_1$. For $\mathbf{x}_2 = (17, 31, 22)^T$, the leading zeros of $17 = 00010001_2$, $31 = 00011111_2$, and $22 = 00010110_2$ are all equal to 3, which results in $\mathbf{x}_{bin} = (1, 1, 1)^T$. The proposed method of quantization examines the portion of bits that affects the direction of the $\mathbf{x}$ the most.

Although, theoretically, this method maps the direction of $\mathbf{x}_k$ to one of $2^N - 1$ possible directions, the mapping does not perform well in practice for large N because using the same ℓz_m in Eq. (18) globally leads to some locally large x_j being quantized to 0. We address this issue by grouping the x_j's into a group of three. For example, given the 6-dimensional $\mathbf{x}_k = (x_1, x_2, x_3, x_4, x_5, x_6)^T = (56, 126, 34, 17, 45, 22)^T$ with an 8-bit representation of x_j, we find that its $\mathbf{x}_{\text{bin}}$ is $(0, 1, 0, 0, 0, 0)^T$, as $x_2 = 126$ dominates. This is not a good estimation of the direction of $\mathbf{x}_k$ because while the actual vector $\mathbf{x}_k$ has values in all dimensions, the estimated vector $\mathbf{x}_{\text{bin}}$ lies in one dimension. To mitigate the detrimental effect of large N, we propose to quantize each group of $\mathbf{x}_k$ separately. The number of elements in each group is 3 or small number, as a rule of thumb. For example, by separating the x_js into two groups of three, $\mathbf{x}_k^1 = (56, 126, 34)^T$ and $\mathbf{x}_k^2 = (17, 45, 22)^T$, where the superscript indicates the group number, we find that $\mathbf{x}_{\text{bin}} = (0, 1, 0, 0, 1, 0)^T$ as $\mathbf{x}_k^1$ and $\mathbf{x}_k^2$ are quantized to $(0, 1, 0)^T$ and $(0, 1, 0)^T$, respectively. The modified BinarizeX(x) for finding $\mathbf{x}_{\text{bin}}$ is described in Algorithm 2, where the lines beginning with % denote comments.

D. GEOMETRY-BASED APPROXIMATE CONVOLUTION

The geometry-based approximation of the dot product of vector $\mathbf{x}$ with a constant vector $\mathbf{h}$ is used as a core part of the proposed implementation of the convolution computation. Since the precomputed parameters P_1 and P_0 are based on generating all $2^\ell - 1$ cases of $\mathbf{x}_{\text{bin}}$ of size ℓ, we cannot practically apply the method to a large vector size. To address this limitation, we use the superposition property of the dot product computation; i.e., the dot product of two vectors of the same size is the sum of the dot products of their corresponding subvectors. Applying this property to compute $y[k] = \mathbf{h} \cdot \mathbf{x}_k$, we partition $\mathbf{h}$ and $\mathbf{x}_k$ into Q parts of size ℓ as described in Eq. (19) and (20), respectively, where $Q = \lfloor \frac{N}{\ell} \rfloor$ and with an additional final part of size $R = N - Q\ell$ if $N > Q\ell$:

$$\mathbf{h} = (\mathbf{h}_0, \mathbf{h}_1, \ldots, \mathbf{h}_{Q-1})^T \tag{19}$$
$$\mathbf{x}_k = (\mathbf{x}_{k,0}, \mathbf{x}_{k,1}, \ldots, \mathbf{x}_{k,Q-1})^T. \tag{20}$$

Then, $y[k] = \mathbf{h} \cdot \mathbf{x}_k$ can be described by Eq. (21):

$$y[k] = \mathbf{h} \cdot \mathbf{x}_k = \sum_{i=0}^{Q-1} \mathbf{h}_i \cdot \mathbf{x}_{k_i} + \mathbf{h}_Q \cdot \mathbf{x}_{k,Q} \tag{21}$$

Algorithm 2 Find $\mathbf{x_{bin}}$ of a Given $\mathbf{x}$

Require: All ℓ elements of $\mathbf{x}$ are n-bit unsigned numbers

```
1:  function BINARIZEDX(x, ℓ, n)
2:      G ← ⌊ℓ/3⌋
3:      % G is the number of groups
4:      R ← ℓ − G
5:      % R is the number of last group's elements
6:      x_bin ← (0, 0, ..., 0)^T
7:      for i ← 0 to G − 1 do
8:          lz_m ← n
9:          for j ← 1 to 3 do
10:             x_j ← x[i × 3 + j]
11:             Find leading zeros lz[j] of x_j
12:             if lz[j] < lz_m then
13:                 lz_m = lz[j]
14:             end if
15:         end for
16:         for j ← 1 to 3 do
17:             if lz[j] == lz_m then
18:                 x_bin[i × 3 + j] = 1
19:             end if
20:         end for
21:     end for
22:     lz_m ← n
23:     for j ← 1 to R do
24:         x_j ← x[G × 3 + j]
25:         Find leading zeros lz[j] of x_j
26:         if lz[j] < lz_m then
27:             lz_m = lz[j]
28:         end if
29:     end for
30:     for j ← 1 to R do
31:         if lz[j] == lz_m then
32:             x_bin[i × 3 + j] = 1
33:         end if
34:     end for
35:     return x_bin
36: end function
```

Here $\mathbf{h}_i$ and $\mathbf{x}_{k,i}$ are the vectors of ℓ elements taking from the ith part of $\mathbf{h}$ and $\mathbf{x}_k$, respectively, for $i = 0, 1, 2, \ldots, Q-1$:

$$\mathbf{h}_i = (\mathbf{h}[i\ell], \mathbf{h}[i\ell+1], \ldots, \mathbf{h}[(i+1)\ell-1])^T \tag{22}$$

$$\mathbf{x}_{k,i} = (\mathbf{x}[k+i\ell], \mathbf{x}[k+i\ell+1], \ldots, \mathbf{x}[k+(i+1)\ell-1])^T. \tag{23}$$

Vectors $\mathbf{h}_Q$ and $\mathbf{x}_{k,Q}$ contain the remaining R elements at the end of $\mathbf{h}$ and $\mathbf{x}_k$, respectively:

$$\mathbf{h}_Q = (\mathbf{h}[Q\ell], \mathbf{h}[Q\ell+1], \ldots, \mathbf{h}[N-1])^T \tag{24}$$

$$\mathbf{x}_{k,Q} = (\mathbf{x}[k+Q\ell], \mathbf{x}[k+Q\ell+1], \ldots, \mathbf{x}[k+N-1])^T. \tag{25}$$

As a convention, if $R = 0$, we define $\mathbf{h}_Q = \mathbf{x}_{k,Q} = 0$. By computing each dot product $\mathbf{h}_i \cdot \mathbf{xk}_i$ using the method proposed in the previous subsection, we can describe the dot product $\mathbf{h} \cdot \mathbf{x}_k$ for computing $y[k]$ in Algorithm 3.

Algorithm 3 Procedure for Computing the Proposed Geometry-Based Convolution

Require: $\mathbf{P}_1, \mathbf{P}_0, \mathbf{B}, \mathbf{hmag}$ are precomputed for a given $\mathbf{h}$

```
1:  function GEO_CONV(x, P_1, P_0, B, hmag, ℓ, N, M)
2:      Q ← ⌊N/ℓ⌋ and R ← N − Q × ℓ
3:      x_k = (0, 0, ..., 0)^T
4:      % fill initial x_k with N zeros
5:      xmag_p = (0, 0, ..., 0)^T
6:      % fill previous magnitude with ⌈N/ℓ⌉ zeros
7:      x_0 = (0, 0, ..., 0)^T
8:      % fill elements just being removed with ⌈N/ℓ⌉ zeros
9:      for k ← 0 to M − 1 do
10:         x_k[1 : N − 1] ← x_k[0 : N − 2]
11:         x_k[0] ← x[k]
12:         y[k] ← 0
13:         for i ← 0 to Q − 1 do
14:             xk ← x_k[i × Q : i × Q + ℓ − 1]
15:             % Compute |xk_i|
16:             x_1 ← xk[0]
17:             xm ← xmag_p[i] + x_1^2 − x_0[i]^2
18:             xmag_p[i] ← xm
19:             x_0[i] ← xk[ℓ − 1]
20:             % Find θ_i
21:             θ ← FingAngle(xk, P_1[i], P_0[i], B[i], ℓ)
22:             % Compute h_i · xk_i; combine it with y[k]
23:             yk ← hmag[i] × √xm × cos θ
24:             y[k] ← y[k] + yk
25:         end for
26:         if R > 0 then
27:             xk ← x_k[Q × ℓ : N − 1]
28:             x_1 ← xk[0]
29:             xm ← xmag_p[Q] + x_1^2 − x_0[Q]^2
30:             xmag_p[Q] ← xm
31:             x_0[Q] ← xk[R − 1]
32:             θ←FingAngle(xk, P_1[Q], P_0[Q], B[Q], R)
33:             yk ← hmag[Q] × √xm × cos θ
34:             y[k] ← y[k] + yk
35:         end if
36:     end for
37:     return y
38: end function
```

Next, we discuss Algorithm 3. For a given $\mathbf{h}$, the proposed method requires the precomputed parameters and $\mathbf{hmag}$, where $\mathbf{hmag}[i]$ stores $|\mathbf{h}_i|$, for $i = 0, 1, 2, \ldots, \lceil \frac{N}{\ell} \rceil$. For section i of size ℓ, its parameters, including $\mathbf{P}_1[i]$, $\mathbf{P}_0[i]$, and $\mathbf{B}[i]$, are precomputed using $\mathbf{h}_i$ following the method described in the previous subsection. With this setup, we arrange the computation of $y[k] = \mathbf{h} \cdot \mathbf{x}_k$ in three parts, as discuss next.

The first part computes the magnitude of $\mathbf{xk}_i$, subvector i of the input vector. As described in the problem analysis

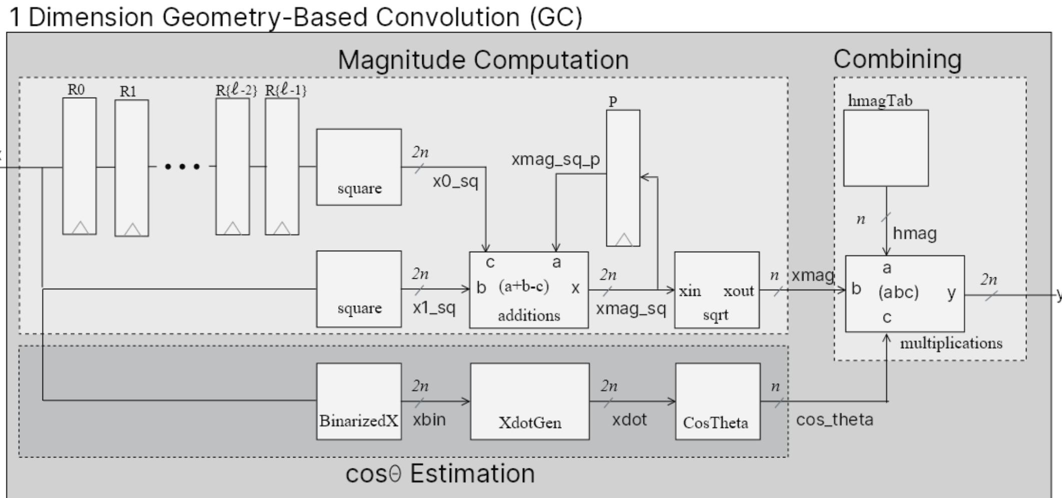


FIGURE 5. **Overall hardware architecture for the proposed convolution computation.**

section, we compute the square of the magnitude of $\mathbf{x}_{k+1}$ using the relationship of two consecutive input vectors, $\mathbf{x}_{k+1}$ and $\mathbf{x}_k$, representing respective sliding windows of the input samples. Since, in the proposed procedure, each input vector $\mathbf{x}_k$ is partitioned into Q parts of size ℓ, we have to compute $|\mathbf{xk}_i|^2$ of each part i at time k (line 14). Because $\mathbf{xk}_i$ at clock k and $k+1$ are two vectors from a sliding window of size ℓ, the computation of $|\mathbf{xk}_i|^2$ follows Eq. 9, in which we need (1) $|\mathbf{xk}_i|^2$ at time $k-1$, (2) the last element of $\mathbf{xk}_i$ at time $k-1$, and (3) the new element of $\mathbf{xk}_i$ at time k. In Algorithm 3, we use the variables (1) $\mathbf{xmag}_p$, (2) $\mathbf{x}_0$, and (3) x_1 to store these 3 values. The variables $\mathbf{xmag}_p$ and $\mathbf{x}_0$ are initialized to all zeros at the beginning and are updated at every step of k (lines 15 and 16) after their respective current values are computed. The variable x_1 is assigned to the first value of $\mathbf{xk}_i$ at time k, which is assigned at line 11.

The second part estimates the angle of vector $\mathbf{xk}_i$ using the function `FindAngle` described in Section IV-B. In the third part, the results from the first two parts are used to compute $yk = \mathbf{h}_i \cdot \mathbf{xk}$ using geometric definitions, and then, yk is combined with the current $\mathbf{y}[k]$. Note that the first two parts can be done in parallel, which is suitable for the hardware implementation described in Section V. Although the procedure computes $\mathbf{y}[k]$ for k from 0 to $M-1$, it can easily be adjusted to compute N additional samples to flush out all samples of $x * h$.

V. HARDWARE IMPLEMENTATION OF THE PROPOSED CONVOLUTION

Although, in general, the proposed approximate convolution computation can be implemented in many computation platforms, the proposed procedure is targeted for an embedded environment in which the computing resources and available power are limited. To this end, we propose a hardware architecture targeting FPGA or ASIC. Fig. 5 shows the overall datapath architecture of the hardware implementation for computing $\mathbf{x} \cdot \mathbf{h}$ of size ℓ. Note that the size ℓ is limited since in the computation of the parameters P_1, P_0, and B following the proposed method, we need to generate all $2^\ell - 1$ cases of $\mathbf{x}_{\text{bin}}$. We show in Section VI that the suitable ℓ is 20.

In this architecture, the streaming $x[k]$ at the input port `x` is fed to two parallel parts for computing $|\mathbf{x}_k|$, i.e., the signal `xmag`, and $\cos\theta_k$, i.e., the signal `cos_theta`. The results for these two parts are multiplied together with $|\mathbf{h}|$ in the final part to produce the streaming $y[k]$ at the output port `y`. All $|\mathbf{h}|$s are precomputed and stored in an LUT denoted as `hmagTab` in the diagram. If the size N of $\mathbf{h}$ (the number of tabs) is larger than ℓ, this hardware will be used to produce $\mathbf{xk}_i \cdot \mathbf{h}_i$, $i = 0, 1, \ldots, \tilde{Q} - 1$, where $\tilde{Q} = \lceil N/\ell \rceil$ is the number of sections. The results of the $\mathbf{xk}_i \cdot \mathbf{h}_i$s from all sections are added together to produce $y[k]$. This modular design allows us to implement the system based on the available resources. If we have sufficient resources, multiple instances of this architecture can be implemented to produce multiple $\mathbf{xk}_i \cdot \mathbf{h}_i$ in parallel. Otherwise, a single instance can be controlled to produce $y[k]$ every $\tilde{Q}$ cycles of the system clock. As a result, in this minimal hardware case, the maximum streaming rate is $f_c/\tilde{Q}$, where f_c is the system clock's frequency.

A. MAGNITUDE COMPUTATION HARDWARE

The computation of $\mathbf{x}_k$'s magnitude follows Eq. (9); i.e., $|\mathbf{x}_k|^2 = |\mathbf{x}_{k-1}|^2 + x[k]^2 - x[k-\ell]^2$, where $x[k]$, the sample entering the window, is the new element of $\mathbf{x}_k$ and $x[k-\ell]$, the element leaving the window, is the last element of $\mathbf{x}_{k-1}$. As shown in the upper part of Fig. 5, to implement the magnitude computation, we need the following components: (1) two multipliers or specially designed circuits of the square operation for computing $x[k]^2$ and $x[k-\ell]^2$, whose results are the signals `x1_sq` and `x0_sq`, respectively, (2) two adders for the operation of $a+b-c$, (3) a register for storing $|\mathbf{x}_{k-1}|^2$, the signal `xmag_sq_p`, (4) ℓ cascading n-bit registers for

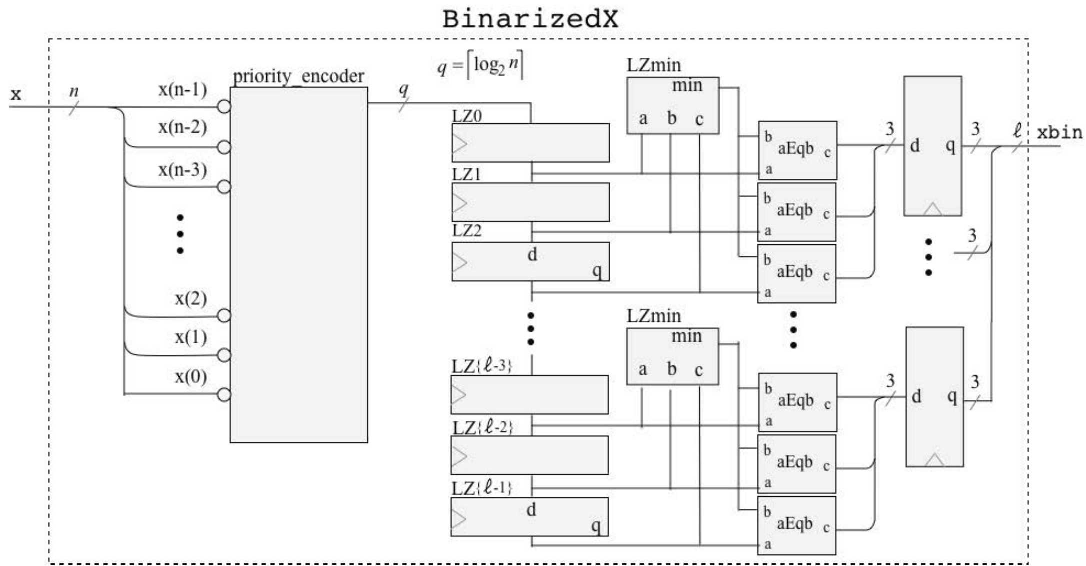


FIGURE 6. Hardware architecture of the `BinarizedX` module for computing $\mathbf{x}_{\text{bin}}$.

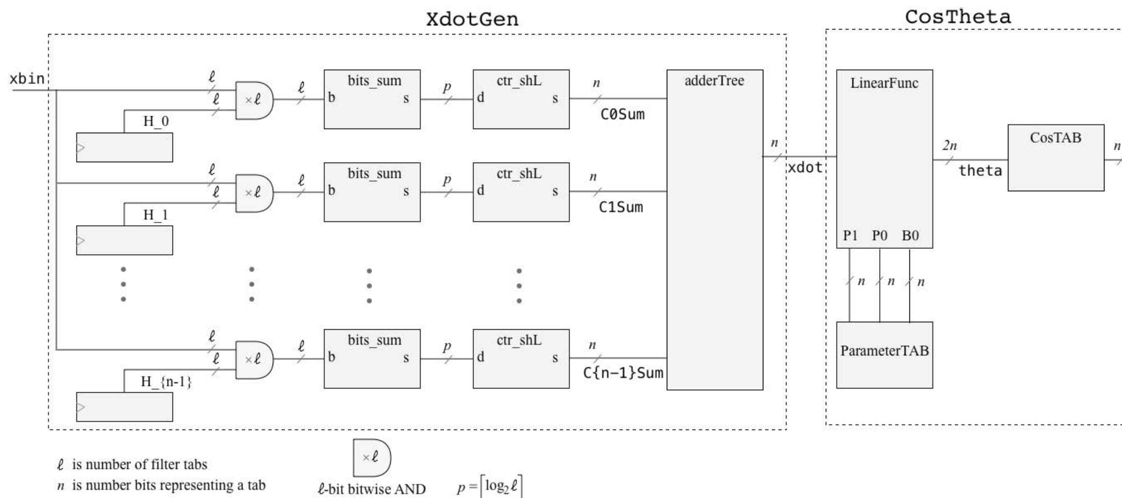


FIGURE 7. Hardware architecture for estimating $\cos\theta_k$ for a specific filter, h, where the middle block computes x_{dot}.

storing $x[k-\ell]$, the element leaving the window, and (5) a square root unit.

Note that the incoming data sample, $x[k]$, the signal `x`, is the newest element, whose square, `x1_sq`, is added with the previous magnitude, `xmag_sq_p`. It is also fed into the ℓ cascading n-bit registers, causing a delay of ℓ steps; i.e., its output is $x[k-\ell]$, whose square, `x0_sq`, is subtracted from the sum of `x1_sq` and `xmag_sq_p`, resulting in the square of the magnitude at cycle k in the signal `xmag_sq`. Finally, the sum of $|\mathbf{x}_{k-1}|^2$, $x[k]^2$, and $x[k-\ell]^2$ is fed into a square root unit for streaming out $|\mathbf{x}_k|$ in the signal `xmag`.

This architecture is suitable for implementation with the pipeline technique, allowing simple control of the system by simply pumping the data in and out at a constant rate. To this end, a pipeline design technique can be applied for a specific hardware target with the constraint that the number of stages of the pipeline matches that of the $\cos\theta$ implementation.

B. ANGLE ESTIMATION HARDWARE

To gain the benefits of the proposed convolution procedure, we need to design specific hardware for the approximation of $\cos\theta_k$. The first specific circuit shown in Fig. 6 generates a vector $\mathbf{x}_{\text{bin}}$ for each vector $\mathbf{x}_k$. The circuit takes a group of 3 consecutive samples that are converted to 3 respective bits of $\mathbf{x}_{\text{bin}}$. At the entry of the module, an n-input priority encoder circuit is adopted as the leading zero computation of the entering sample x, the $x[k]$ sample. Its result is then fed to the cascading registers, named `LZ0`, `LZ1`, `LZ2`, ..., `LZ{`$\ell-1$`}`. As the outputs of these registers are the leading zeros of consecutive samples, the minimum of the 3 registers' outputs is found by the module `LZmin`. Finally, the stored leading zeros are compared with their respective references from the `LZmin` modules to generate the binary elements of $\mathbf{x}_{\text{bin}}$ stored in the output registers.

The generated $\mathbf{x}_{\text{bin}}$ is fed to the second specific circuit shown in Fig. 7 for computing $x_{\text{dot}} = \mathbf{x}_{\text{bin}} \cdot \mathbf{h}$, denoted as `xdot`. For conventional implementation of the dot product, the number of additions required for the computation of x_{dot} is on the order of $\Theta(\ell)$. In the proposed implementation, the module, denoted as `XdotGen`, does the job in the order of $\Theta(n)$, where n is the number of bits representing $\mathbf{h}[j]$, an element of $\mathbf{h}$; i.e., the order of computation is constant with reference to the problem size, which is specified by N and M, the number of filter tabs, and the number of data samples. To explain the proposed implementation, let us define $\mathbf{x}_{\text{bin}}$ and $\mathbf{h}$ as follows:

$$\mathbf{x}_{\text{bin}} = (b_0, b_1, \ldots, b_{\ell-1})^T, \quad b_j \in \{0, 1\}$$

$$\mathbf{h} = (\mathbf{h}[0], \mathbf{h}[1], \ldots, \mathbf{h}[\ell-1])^T \tag{26}$$

$$\begin{aligned} \mathbf{h}[m] &= (a_{n-1}^{(m)} a_{n-2}^{(m)} \cdots a_0^{(m)})_{\text{base } 2}, \quad a_j^{(m)} \in \{0, 1\} \\ &= \sum_{j=0}^{n-1} a_j^{(m)} 2^j. \end{aligned} \tag{27}$$

Then, as described in Eq. (29), the computation of x_{dot} can be expressed as a summation of $c_j 2^j$, $j = 0, 1, \ldots, n-1$, where the coefficient $c_j = \sum_{m=0}^{\ell-1} b_m a_j^{(m)}$ depends on b_m of $\mathbf{x}_{\text{bin}}$ and $a_j^{(m)}$ of $\mathbf{h}[j]$:

$$\begin{aligned} x_{\text{dot}} &= \mathbf{x}_{\text{bin}} \cdot \mathbf{h} = \sum_{m=0}^{\ell-1} b_m \mathbf{h}[m] \\ &= \sum_{m=0}^{\ell-1} b_m \sum_{j=0}^{n-1} a_j^{(m)} 2^j \qquad (28) \\ &= \sum_{j=0}^{n-1} 2^j \sum_{m=0}^{\ell-1} b_m a_j^{(m)} \\ &= \sum_{j=0}^{n-1} c_j 2^j, \quad \text{where } c_j = \sum_{m=0}^{\ell-1} b_m a_j^{(m)}. \qquad (29) \end{aligned}$$

Since $a_j^{(m)}$ is known for all $m = 0, 1, \ldots, \ell - 1$, we can store all ℓs of the $a_j^{(m)}$s for each $j = 0, 1, \ldots, n-1$. In other words, storing $\mathbf{h}$ in an LUT gives us all the $a_j^{(m)}$s for a given filter, but we need to arrange the LUT by storing the m-bit $(a_j^{(m-1)}, a_j^{(m-2)}, \ldots, a_j^{(0)})$ in its address j.

Based on this analysis, we propose a design of a specific circuit, as shown in Fig. 7, that produces $c_j 2^j$, where $c_j = \sum_{m=0}^{\ell-1} b_m a_j^{(m)}$, for the input $\mathbf{x}_{\text{bin}}$, provided that $\mathbf{h}$ is known. This design is modular and is parameterized with ℓ and n. By storing bit j of all tabs as ℓ-bit data, $H_j = (a_j^{(\ell-1)}, a_j^{(\ell-2)}, \ldots, a_j^{(0)})$, in the respective register `H_j`, $j = 0, 1, \ldots, n-1$, a bitwise `AND` operation between `xbin` and `H_j` produces $b_m a_j^{(m)}$, $m = 0, 1, \ldots, \ell - 1$. Then, the module `bit_sum` in the diagram adds all bits of $b_m a_j^{(m)}$ to produce c_j. This is followed by the module `shL{j}`, which implements $c_j 2^j$ by shifting c_j to the left by j bits, which can be done by wiring for a given j. The results, denoted as `c0x1`, `c1x2`, ..., `c{n-1}x{w}` ($w = 2^{n-1}$), are added together in the module `adderTree`, which employs a tree structure of $n-1$ adders to implement $\sum_{j=0}^{n-1} c_j 2^j$, resulting in x_{dot}.

The estimated θ_k is produced by the module `LinearFunc`, which implements the computation of $P_1 x_{\text{dot}} + P_0 - B$, whereas P_1, P_0, and B for a given $\mathbf{h}$ are stored in the module `ParameterTAB`. Finally, $\cos\theta_k$ is produced by an LUT storing the cosine function.

C. APPLICATIONS TO 2D CONVOLUTION

Our hardware is considered a scalable array of processing elements (PEs), which are the base elements for building the applications. In this section, we explore applications of our hardware blocks to 2D convolution. First, we design a 2D engine based on our scalable hardware. Then, we apply the developed hardware to a state-of-the-art convolutional neural network.

1) TWO-DIMENSIONAL CONVOLUTION

To develop a 2D geometry-based convolution, we design a specialized, low-complexity, low-power-consumption hardware. Let x denote the input matrix of size $M \times N$ and w denote a filter of size $\tilde{n} \times \tilde{n}$, where $\tilde{n}$ is a positive, odd number, i.e., $\tilde{n} = 2a + 1$ for some non-negative integer a.[1] Elements of the output matrix y from the 2D convolution between x and w are given by

$$y(i, j) = \sum_{s=-a}^{a} \sum_{t=-a}^{a} w(s, t) x(i + s, j + t), \tag{30}$$

where i and j are row and column indices, respectively. As i and j vary, the filter's center visits every pixel of x once. The 2D convolution can be considered as a summation of 1D convolutions. Therefore, a hardware for 2D convolution can be obtained from our 1D-convolution hardware.

Based on our hardware in Fig. 5, the overall architecture for 2D convolution is shown in Fig. 8. The architecture distributes each row of the input matrix into the row of FIFO and parallelizes the computation by interleaving matrix rows over the PEs. Every PE stores a partition of weights, i.e., the convolution constants, in the BRAM and performs a computation of row convolution with those values. Finally, the results of each row from each convolution are added together using a parallel tree-structured hardware in the `addX` unit in the figure. The proposed architecture has `m` levels of parallelism, where `m` is the number of 1D convolutions, which is also the number of PEs parallelized in the engine. The notation `GC2(m)` in the figure denotes the overall 2D convolution engine. In the proposed architecture, the more PEs, the more levels of parallelization.

The number of PEs in the proposed architecture should be a multiple of $\tilde{n}$, i.e., `m` $= \beta\tilde{n}$, where β is the amount of output per cycle, so that the proposed hardware can simply

[1] The consideration that $\tilde{n}$ is an odd number is without loss of generality, since we can pad the filter w with zero elements to achieve the odd numbers of rows and columns.

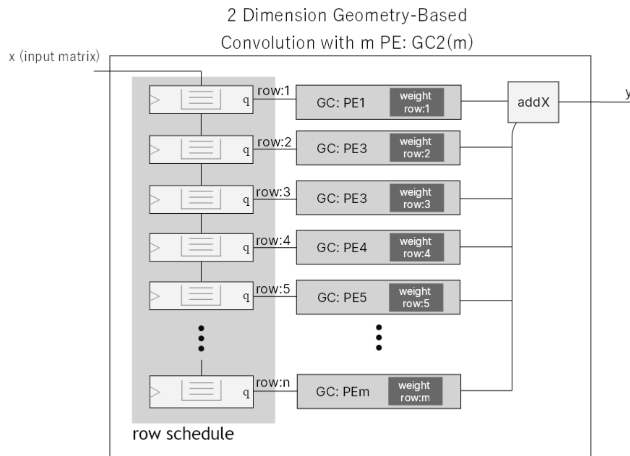


FIGURE 8. Hardware architecture for 2D convolution.

store the constant and schedule the data directly to each PE. For illustration, we let $\beta = 1$. To generate the first output row, the first m rows of the matrix, which are row 0 to row $m-1$, are streamed and split into each PE for computing the 1D convolution. After that, the 2D convolution is performed by adding those results together in the unit labelled `addX` in the figure. When the first output row is finished, the second output row is computed by moving the data to the FIFO queue, i.e., fetching row 1 to row m of the input matrix to each `GC` unit. In the remaining steps, the input matrix will be scheduled in the same fashion, by moving the input matrix row by row until the final row is added to the FIFO queue. This procedure is handled by the unit `row schedule` in Fig. 8.

2) CONVOLUTIONAL NEURAL NETWORK

This subsection applies 2D convolution to CNN models. Given that most CNN architectures use the convolution in a similar way, we select the AlexNet CNN model as an illustration. This subsection focuses on applications of our purposed method to complex CNNs.

We begin on how a 2D convolution is embedded in a CNN. For illustration, we consider a 4D kernel tensor K, where element $K_{i,j,k,l}$ is the connection between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns. Furthermore, we let V denote the input data, where element $V_{i,j,k}$ is the value of the input unit at row j, column k, and channel i.

We consider a simple case in which the output Z takes the same format as the input V and comes from a multichannel convolution:

$$Z_{i,j,k} = \sum_{l}\sum_{x}\sum_{y} V_{l,(j-1)s+x,(k-1)s+y} K_{i,l,x,y}, \quad (31)$$

where the summations cover all indices l, x and y. Parameter s is the stride of the convolution. Eq. (31) captures the main convolution operation in CNNs.

According to Eq. (31), each output $Z_{i,j,k}$ is the sum of multichannel 2D convolutions with respect to the number of filters in each layer and the stride s. To develop a convolution layer, we propose to use the hardware in Fig. 9 instead of the conventional convolution hardware, for less power consumption and less complexity. The hardware in Fig. 9 consists of many 2D convolution `GC2(m)` units.

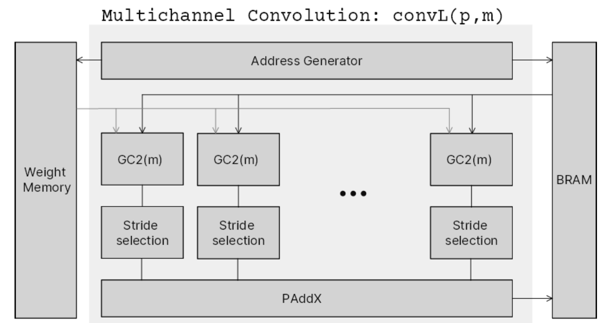


FIGURE 9. Hardware architecture for multichannel convolution with `p` parallelized `GC2` units.

In Fig. 9, the proposed architecture takes a stream of multiple layers of V, together with the configuration parameters, which consist of the current number of filters and the number of layers. Each channel of input V is interleaved into each PE in a scheme of 2D. Meanwhile, the PE clusters execute the convolutions. The weights and stride s are self-distributed to each PEs with respect to the sequence of computation. After that, the stride selection hardware is operated, for downsampling the convolution. The stride selection is built from the counter. Finally, the parameterized parallel adders `PAddX` are operated with respect to the final summation of the 2D convolutions, as indicated by Eq. (31). The final results of multichannel convolution are stored in the BRAM, and the hardware prepares to compute the convolution for the next layer.

To control the datapath, a conventional unrolling loop is used. The number of unrolling loops depends on the number of PEs in the field. The number of groups for parallel adders also depends on the number of filters according to Eq. (31). Moreover, the address generator block fetches the desire address to acquire the data from the BRAM. This core has two parameters: `m`, which is the number of 1D convolution `GC`; and `p`, which is the number of 2D convolution `GC2`. The core performs the operation `convL(p,m)` as shown in Fig. 9. Our hardware blocks are general and able to facilitate the architectures of 2D convolutions and CNNs.

VI. RESULTS AND DISCUSSION

The proposed approach is evaluated in three respects: computation accuracy, time complexity, and resources, measured in both area and power consumption.

A. ON ACCURACY

We evaluate the accuracy of the proposed approximate convolution in two steps. In the first step, we consider the errors of the approximated angle, $\hat{\theta}$, following Eq. (17), compared with the exact angle, θ, following Eq. (10). Then, the accuracy of

TABLE 2. Parameters used in the evaluations.

Parameters	Values used in the evaluations
Size of vector $\mathbf{h}$, N	(1) varies from 4 to 20 (2) varies from 4 to 100
Types of $\mathbf{h}$	Uniform random, Gaussian filter, Low-pass filter, High-pass filter
Size of vector $\mathbf{x}$, M	400
Types of filter, $\mathbf{x}$	Uniform random, multiple-frequency signals
No. of samples	10,000
Size of sub-vector, ℓ	20
Data representation	8-bit unsigned numbers,

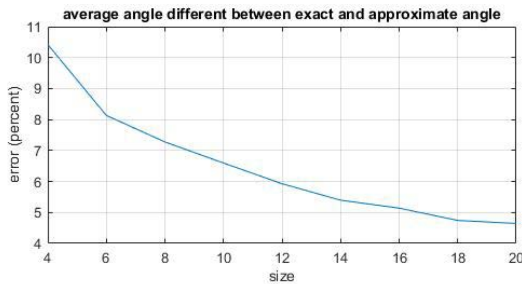


FIGURE 10. The average difference between the exact and approximate angles with 10,000 pairs of vectors.

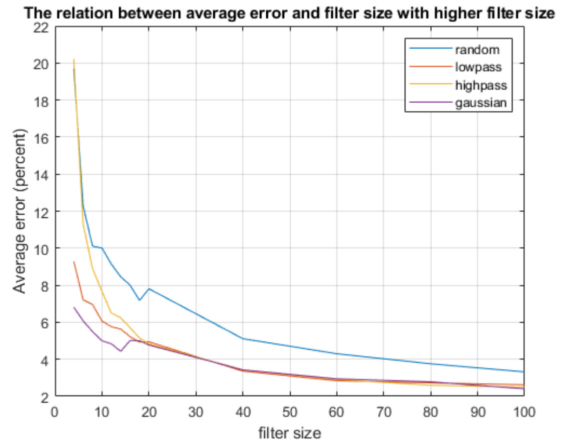


FIGURE 11. The average error with a random representative impulse response with any dimension when compared to the exact convolution.

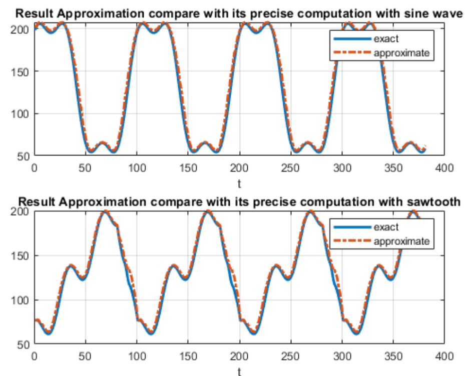


FIGURE 12. Output signals from the low-pass filter using the proposed approximated convolution compared with those using the exact convolution.

the approximate convolution results is considered. Both evaluations are performed with the parameters shown in Table 2.

For the first result, the effect of vector-to-angle approximation is considered using a random vector as $\mathbf{h}$, the impulse response. The percent angle errors for the vector size from 4 to 20 are plotted in Fig. 10. As expected, the percent error decreases for larger vector sizes as the space to which $\mathbf{x}_{\text{bin}}$ is quantized grows exponentially with respect to N, the vector size. The important finding in this evaluation is that the error is saturated for sizes greater than 18. These results are used to choose an appropriate value for ℓ in our proposed method, as the price for a large value of ℓ is in the process of predetermining the values of P_1, P_0, and B. Based on these results, we choose $\ell = 20$.

Using the chosen $\ell = 20$, we evaluate the accuracy of the proposed convolution approximation for 4 different kinds of filters, a uniform random filter, a Gaussian filter, a low-pass filter, and a high-pass filter with vector size N varying from 4 to 100. Using randomly generated samples of an input vector $\mathbf{x}$ of size 400, we compute the average absolute error of the approximate convolution using the equation below:

$$\texttt{conv_error} = \frac{|\mathbf{h} * \mathbf{x} - \texttt{approx}(\mathbf{h} * \mathbf{x})|}{\mathbf{h} * \mathbf{x}}, \tag{32}$$

where $\mathbf{h} * \mathbf{x}$ is the precise convolution and $\texttt{approx}(\mathbf{h} * \mathbf{x})$ is its approximation using the proposed method.

Fig. 11 shows the plot of the average absolute errors for various filter types, with the filter size varying from 4 to 100. In this plot, for each filter size, we average the percent error using 200 samples of $\mathbf{h}$, each of which is evaluated with 1000 samples of $\mathbf{x}$. The plot shows a very good result, as the average error of the proposed method is approximately 5% for specific filter types of size greater than 20. Moreover, the larger the size is, the smaller the error. For the random filter type, the error is larger than that for a specific type due its randomness. Hence, using this as an upper bound of error, the proposed approximation achieves less than 5% error for any filter of size greater than 50.

To evaluate the effect of the approximation on the quality of the output signal, we feed a sinusoidal signal with the fundamental frequency and its third harmonic and a sawtooth with the 3rd harmonic sinusoidal signal to a Chebyshev low-pass filter with the order of 20. Fig. 12 shows the output signals from the filter obtained using the proposed approximated convolution compared with those using the actual convolution. Although the results reveal some local distortions of the approximated output from its precise version, the maximum errors are 17% and 14%, while the average errors are less than 4.5% and less than 3.7% for the sinusoidal and sawtooth signals, respectively.

Furthermore, we also evaluate and report the accuracy in Table 3 among the various types of approximate

TABLE 3. The convolution error of the approximate convolution among various designs.

Approximate Algorithm	Accuracy		
	ER	NMED	MRED (%)
Momeni [32]	0.93	1.63×10^{-3}	9.03
Venka [33]	0.63	1.25×10^{-3}	1.3
Akbar [36]	0.84	2.93×10^{-3}	4.82
Sabetz [37]	0.98	1.93×10^{-3}	9.52
Ahma [38]	0.77	1.47×10^{-3}	1.7
Yang1 [34]	0.04	4.79×10^{-5}	0.02
Yang2 [34]	0.2	2.50×10^{-4}	0.2
Yang3 [34]	0.28	3.68×10^{-4}	0.3
Lin [35]	0.04	9.16×10^{-5}	0.04
Ranjbar1 [39]	0.88	2.15×10^{-3}	4.92
Ranjbar2 [39]	0.9	1.88×10^{-3}	4.29
Rangbar3 [39]	0.88	2.15×10^{-3}	4.78
Kong [40]	0.04	4.79×10^{-5}	0.024
Proposed	0.89	1.05×10^{-3}	3.8

FIGURE 13. Example images from exact (left) and approximate (right) convolutions with Gaussian filter.

FIGURE 14. Example images from exact (left) and approximate (right) convolutions with Laplacian of Gaussian filter. The colors are inverted from black to white and white to black for visual clarity.

multiplication on the same comparable test case. Evaluation metrics include the error rate (ER), the normalized mean error distance (NMED) and the percentage of mean relative error distance (MRED) [42]–[45]. The ER is the rate of $\mathbf{h} * \mathbf{x} - \texttt{approx}(\mathbf{h} * \mathbf{x}) \neq 0$. The NMED is the average of $|\mathbf{h} * \mathbf{x} - \texttt{approx}(\mathbf{h} * \mathbf{x})|$ divided by $(2^n - 1)^2$. The lower these evaluation metrics, the more accurate the approximation method. From the table, the proposed method performs just above average among the other types for both NMED and MRED. This is to confirm that the proposed method is suitable for convolution tasks.

Next, we quantify the effect of approximate 2D convolution on image filtering, using the following procedure. We apply two types of filters to images in the benchmark datasets. The two types of filters are the Gaussian filter, which represents a low-pass filter, and the Laplacian-of-Gaussian filter, which represents a high-pass filter. The benchmark datasets are the MNIST [46], CIFAR [47], and InDoor Scene [48] datasets. Each type of filter is applied twice—first, using an exact 2D convolution and, second, using the approximate 2D convolution proposed by the paper— to each benchmark image. We use the peak signal-to-noise ratio (PSNR) to measure the difference between a filtered image from an exact convolution and the counterpart from an approximate 2D convolution, where $\texttt{PSNR} = 20\log(255) - 10\log(\text{the mean square error})$ dB. The PSNR of at least 20 dB [49] generally indicates a reasonable agreement between the exact and the approximate filtered images.

TABLE 4. The PSNR between exact and approximate filtered images.

Dataset	Gaussian	Laplacian of Gaussian
MNIST	20.08	18.79
Indoor Scene	33.64	27.8
CIFAR	25.65	22.68

Table 4 shows the average PSNR of each dataset with respect to each filter type. The PSNRs are reasonably large, indicating an agreement between the exact and the approximate convolutions. For a given dataset, the PSNR due to the Gaussian filter is larger than the PSNR due to the Laplacian-of-Gaussian filter. When using the approximate 2D convolution, the Gaussian filter produces images of better quality than the Laplacian-of-Gaussian filter does. Figs. 13 and 14 show examples of filtered images from the exact and approximate 2D convolutions, for the Gaussian filter and the Laplacian-of-Gaussian filter, respectively. In each figure, the PSNR between the exact-convolution and the approximate-convolution images is approximately 25 dB. Visually, the two images in each figure are indistinguishable by eye. The proposed method of approximate 2D convolution attains a reasonable level of accuracy.

We further evaluate performance of the approximate 2D convolution in a neural network. As explained in Section V-C2, we modify the convolutional layer, replacing the exact 2D convolution with the proposed geometry-based approximate convolution. For illustration, the datasets are the ImageNet and MNIST datasets. Table 5 summarizes classification accuracy of the proposed method, compared with the exact convolution.

From the table, classification accuracy of the approximate convolution is smaller than that of the exact convolution, as expected. The difference in classification accuracies of the exact and approximate 2D convolutions is between 5 to 15 percentage points for the MNIST dataset and between 15 to 40 percentage points for the ImageNet dataset. The loss in classification accuracy depends the number of convolutional layers that are being approximated. The MNIST dataset has only 10 classes, which simplify the tasks of training the AlexCNN model and of classifying a sample. On the other hand, the ImageNet dataset has thousands of classes, which are more complex to classify, hence reducing the classification accuracy when the convolution is approximated. A loss

TABLE 5. Classification accuracy of the proposed method compared with the exact computation for the AlexNet CNN model.

Dataset	Exact Convolution	Proposed Approximate Convolution	Loss
MNIST	98%	89–94%	5–12%
ImageNet	56%	34–50%	13–40%

in classification accuracy is offset by a gain from simple hardware and efficient power consumption, as discussed in the next section.

B. ON PERFORMANCE

In this section, we evaluate the number of basic operations, including addition, multiplication, and square root, needed in the proposed convolution implementation, compared with the conventional convolution and other implementations. Furthermore, we evaluate the throughput, which is the amount of output in a given period. To be specific, the complexity comes from the convolution computation only because the complexity of the preprocessing step is fixed for the given, fixed impulse response and diminished in a long run as the size N of the streaming input is large. Another reason to safely omit the complexity of the preprocessing is that preprocessing can be computed before the convolution implementation.

Recall that N and M denote the sizes of vectors $\mathbf{h}$ and $\mathbf{x}$, respectively. For the proposed method, to compute M points of $\mathbf{h} * \mathbf{x}$, there are three parts of the computation as follows:

(1) For the computation of $|\mathbf{x}|^2$, the proposed method needs N multiplications and $N-1$ additions for computing $|\mathbf{x_1}|^2 = \sum_{j=1}^{N} x_j^2$. Then, for $k = 2, 3, \ldots, M$, only one multiplication and 2 additions are needed for each $|\mathbf{x}_k|^2$. In total, the proposed method needs $N+M$ multiplications and $N+2M-3$ additions for computing $|\mathbf{x}|^2$ in all M points of $\mathbf{h} * \mathbf{x}$.

(2) For the computation of $\cos\theta_k$, $k = 1, 2, \ldots, M$, the proposed method eliminates the multiplications by using $\mathbf{x}_{\text{bin}}$, the binarized $\mathbf{x}_k$. Furthermore, we reduce the number of additions in computing $\mathbf{x}_{\text{bin}} \cdot \mathbf{h}$ with a specific circuit, as described in Section V-B. With the direct implementation, $\mathbf{x}_{\text{bin}} \cdot \mathbf{h}$ needs $N_{avg}M$ additions, where N_{avg} is the average number of nonzero bits in $\mathbf{x}_{\text{bin}}$. However, with the proposed hardware implementation, the number of additions is a constant, which is equal to the number of bits of the data representation. This reduction is traded off with two additional circuits: (1) a circuit for generating $\mathbf{x}_{\text{bin}}$ (Fig. 6) and (2) a circuit for determining the total weight of each bit (Fig. 7).

(3) Since the proposed method computes the square of y_k based on the computation of $|\mathbf{h}|^2|\mathbf{x}_k|^2\cos\theta_k$, it needs $2M$ multiplications and M square root operations in total.

In total, for each computation of $\mathbf{h} * \mathbf{x}$, the proposed method needs $N + (2+w)M$ additions (w is the number of bits representing h_j), $N+3M$ multiplications, and M square root operations.

TABLE 6. Number of basic operations of proposed convolution procedure comparing with conventional convolution and other approximation procedure.

Procedure	Operations	Operation Counts
Proposed Method	Addition	$N+10M$
	Multiplication	$N+3M$
	Square Root	M
Conventional Convolution	Addition	$M(N-1)$
	Multiplication	MN
	Square Root	0
M. Kumm et al. [16]	Addition	$\alpha MN - M$
	Multiplication	0
	Square Root	0
Approximate Convolution with [32]- [40]	Addition	$M(N-1)$
	Multiplication	MN
	Square Root	0
Fastfood and FFT Convolution [18]	Addition	$M\log_2 M$
	Multiplication	$(M/2)\log_2 M$
	Square Root	0
Winograd Convolution [19]	Addition	$M(N-1)$
	Multiplication	$N+3M$
	Square Root	0

Table 6 also compares the number of basic operations of the proposed method with those of the conventional convolution procedure with constant multiplication enhancement, conventional convolution with approximate multiplication, and the Fastfood approximation method [18]. Given N, the size of vector $\mathbf{x}$, and M, the size of vector $\mathbf{x}$, the proposed method provides linear growth of $\Theta(N+M)$ for both addition and multiplication compared with $\Theta(MN)$ of the conventional and $\Theta(M\log_2 M)$ of Fastfood approximations, FFT and Winograd. The convolution with constant multiplication enhancement has only polynomial growth in addition, which is $\Theta(MN)$, and the convolution with approximate multiplication has the same results as the conventional method because it has the same operation. However, the proposed method needs additional square root computations with the growth of $\Theta(M)$.

The proposed method and the Winograd convolution have the same number of the multiplication operations, asymptotically. However, the number of addition operations is smaller in the proposed method. The number of multiplication operations for the Winograd convolution can be reduced to $M+N-1$ by storing the transform matrix in memory, an approach that could cause a memory issue in the embedded system. In our design, we store three more values of parameters than the conventional method does, leading to less memory consumption than the Winograd. At the same level of time complexity, our method is easier and simpler to implement than the Winograd convolution.

In addition to time complexity, we evaluate the throughput of the core. Firstly, the throughput and latency of 1D and 2D convolutions are investigated. The throughput depends

TABLE 7. The power consumption of the FPGA implementations in mW of the proposed and conventional methods.

Filter size (N)	Proposed method's Power				Conventional method's Power				M. Kumm et al. [16]	Winograd Convolution [24]
	signal	logic	dsp	total	signal	logic	dsp	total		
5	21	9	3	33	20	4	6	30	27.2	30
10	21	9	3	33	20	4	11	35	33	35
15	22	10	3	35	20	4	16	40	38	40
20	22	10	3	35	20	4	20	44	40	42

TABLE 8. The resource utilization of the proposed method and the conventional method.

filter size	Proposed Method			Conventional method			M. Kumm [16]			Winograd [24]		
	LUT	FF	DSPs	LUT	FF	DSPs	LUT	FF	DSPs	LUT	FF	DSPs
5	874	153	3	200	82	5	349	108	0	280	108	4
10	901	153	3	220	82	10	458	140	0	332	140	4
15	928	153	3	250	82	15	501	156	0	406	156	4
20	954	153	3	280	82	20	556	179	0	495	179	4

TABLE 9. Experimental results of approximate convolution when synthesized with 45 nm standard CMOS cells.

Algorithm	Resource Utilization	
	Area (μm^2)	Power (μW)
Conventional	6157	803.76
M. Kumm [16]	3832–6080	598–835
Approximate		
Momeni [32]	5169.77	706.36
Venka [33]	5166.76	702.38
Akbar [36]	5047.18	695.93
Sabetz [37]	4846.58	686.69
Ahma [38]	4927.24	686.46
Yang1 [34]	6134.77	816.09
Yang2 [34]	6085.78	808.06
Yang3 [34]	5802.12	791.88
Lin [35]	6029.05	807.71
Ranjbar1 [39]	5252.12	716.56
Ranjbar2 [39]	5100.72	699.25
Rangbar3 [39]	5136.82	696.91
Kong [40]	5332.80	718.6
Proposed	5337.14	602.16

on the system clock, which is one sample per clock in our hardware architecture and other computational designs. The latency depends on the size of vector $\mathbf{h}$, which generates each sample of the output $\mathbf{y}$ according to Eq. (2). From Eq. (2), the hardware needs to wait until the current data $\mathbf{x}_k$ were completely streamed before it can stream the new window of data $\mathbf{x}_{k+1}$. Therefore, the throughputs of the proposed method and the conventional method are equal to one sample per clock. The latencies of the proposed method and the conventional method are equal to the system clock interval multiplied by the size of vector $\mathbf{h}$. The proposed method is on par with the conventional method in terms of the throughput and latency.

Furthermore, we investigate the effect of our proposed method on the execution time of a CNN. The total execution time, $\mathtt{T}_{\mathrm{CNN}}$, of the complete CNN is given by

$$\mathtt{T}_{\mathrm{CNN}} = \underbrace{\sum_{i=1}^{n_{\mathrm{CL}}} \mathtt{T}_{\mathrm{CL}_i}}_{\mathtt{T}_{\mathrm{CL}}} + \sum_{i=1}^{n_{\mathrm{FL}}} \mathtt{T}_{\mathrm{FL}_i} + \mathtt{T}_{\mathrm{MEM}}. \tag{33}$$

Here, $\mathtt{T}_{\mathrm{CL},i}$ is the execution time of the ith convolutional layer, n_{CL} is the total number of convolutional layers, $\mathtt{T}_{\mathrm{FL},i}$ is the execution time of the ith fully connected layer, n_{FL} is the total number of fully connected layers, and $\mathtt{T}_{\mathrm{MEM}}$ is the memory loading time for loading an image or the results from the previous layer. To quantify an effect of approximate convolution, we consider only the total execution time of the convolutional layers, $\mathtt{T}_{\mathrm{CL}}$, in our proposed method.

The number $\mathtt{CY}$ of cycles for each convolutional layer is determined by the number of 2D convolutions:

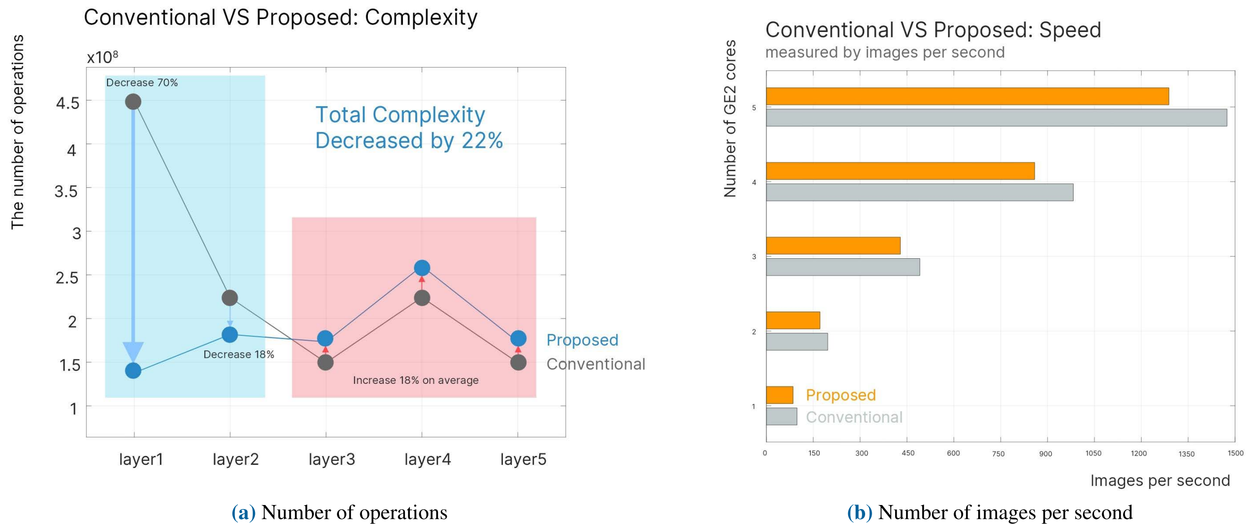
$$\mathtt{CY} = \frac{\mathtt{nConv} \times \mathtt{Is} \times \mathtt{Ks}}{\mathtt{m} \times \mathtt{p} \times \mathtt{s}}, \tag{34}$$

where $\mathtt{nConv}$ is a product between the number k of kernels and the number l of channels, i.e., $\mathtt{nConv} = k \times l$. In the above equation, $\mathtt{Is}$ is the size of input, $\mathtt{s}$ is the stride, $\mathtt{Ks}$ is the size of kernel, $\mathtt{m}$ is the number of 1-dimensional engines, and $\mathtt{p}$ is the number of 2-dimensional engines in the core. The total execution time of the convolutional layers equals

$$\mathtt{T}_{\mathrm{CL}} = \frac{\mathtt{CY}}{\mathtt{clk\ freq}} + \mathtt{mem\ time}, \tag{35}$$

which is a sum of the convolution execution time and the memory consumption time.

To compare the execution times of the CNN, the conventional, exact convolutions are also implemented on the same hardware architecture. The differences between the exact and approximate convolutions appears in the stride configuration. Our proposed method stalls on the number of strides, multiplied with the system clock to acquire the input. In contrast, the conventional convolution does not stall, leading to performance degradation. According to Eq. (34), our method has a constant value of stride equal to one, but the conventional method follows the CNN architecture.



(a) Number of operations

(b) Number of images per second

FIGURE 15. Complexity and performance comparison between the proposed method and the conventional method, measured by (a) the number of operations and (b) the number of images per second.

Fig. 15a shows the complexity reduction of the proposed method compared with the conventional method. The proposed method can reduce the complexity of the conventional method by 70% and 18% approximately in the first two layers, highlighted blue in the figure, respectively. In the last three layers, the complexity increases 18% approximately. Overall, the proposed method reduces the total complexity of the conventional method by 22% approximately.

We further test the proposed architecture on a classic benchmark CNN, the AlexNet, and measure how many images the proposed architecture can produce in one second, compared to the conventional convolution. The AlexNet takes 227×227-pixel^2 images as input. In the test, the size of kernels, channels, and input for each convolution layers are standard and follow the specifications of the AlexNet CNN model. The maximum size of kernel in AlexNet is 11×11. Hence, the choice of $\mathtt{m} = 15$ simplifies the distribution of the data to the core and is suitable in the proposed architecture.

Fig. 15b shows the number of images processed in one second for the conventional and proposed methods. The conventional method processes approximately 10–15% more number of images in one second than the proposed method does. A decrease in speed, measured by images/second, in the proposed method is a trade off for better power consumption and complexity. The decrease in speed is caused by a change in the input structure, namely the stride size, as described in the next section.

C. ON RESOURCES

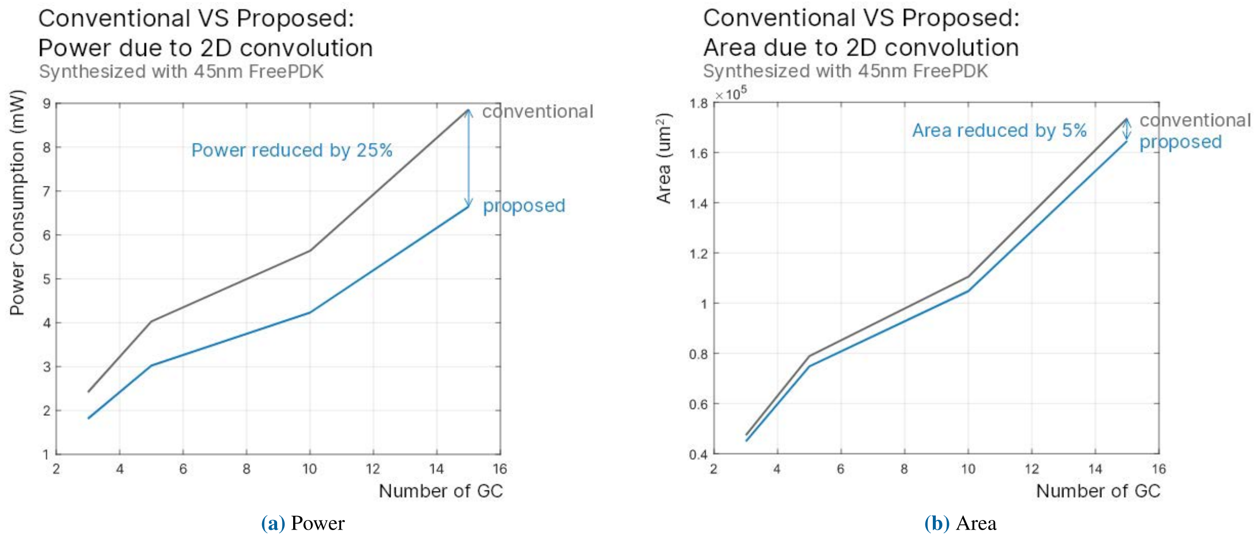
In this section, we evaluate the cost of achieving the performance level in terms of the operation counts described in the previous section. As the proposed convolution procedure has been targeted as a hardware implementation on either a programmable hardware chip, an FPGA chip, or an ASIC chip, the cost is measured in terms of area and power. The FPGA and ASIC chips in our evaluation have the clock frequency of 100 MHz. For the FPGA implementation, we tested the hardware implementation described in Section V on the Zynq 7z010 FPGA fabric [50]. After verifying the correction of the implementation, we measured its power consumption using the tools provided with the development kit and recorded the implementation's area in terms of resource utilization, as shown in Tables 7 and 8, respectively.

For the power consumption issue, the results show that for a large practical filter size, the proposed method can reduce the total power consumption by approximately 20% compared with that of the conventional method. Because the proposed hardware architecture is designed to handle any size N of $\mathbf{h}$ by arranging the computation in multiple blocks of 20, the power needed to operate the proposed hardware is approximately the same for any filter size $N \geq 20$. However, the energy consumption will grow with respect to the vector sizes N and M. For the proposed implementation, this growth of energy is linear, following the growth of the operation counts discussed in the previous section.

For resource utilization, the implementations are coded in HDL and synthesized by the Xilinx Synthesis Tool that comes with Vivado 2021.2. The results show that the proposed hardware utilizes more hardware resources than other implementations. These results are as expected due to the tradeoff we make to reduce the power consumption.

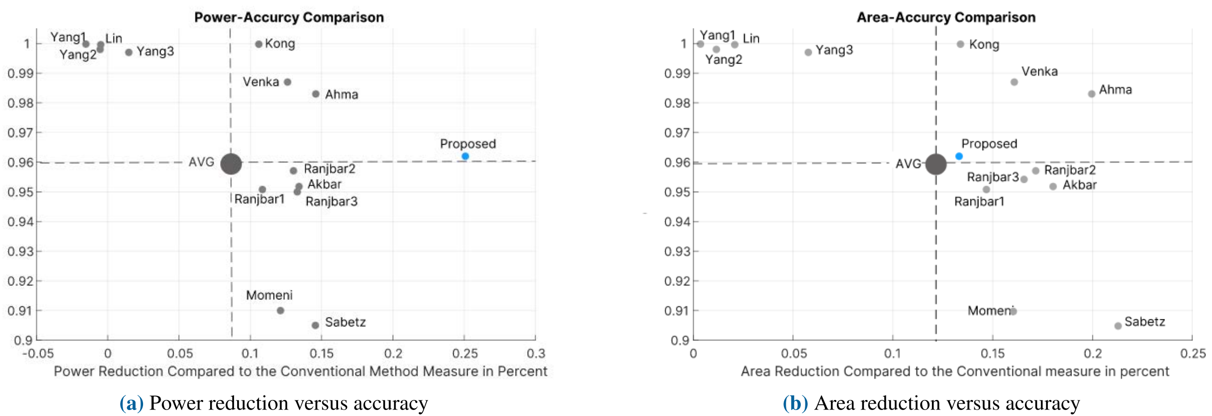
However, in ASIC, we can design the proposed hardware architecture with significantly less area because all circuits are specific, while in FPGA, all resources are prebuilt so that they can be programmed. To illustrate this point, we synthesize our proposed method and the conventional version targeting an open-source 45-nm [51] (FreePDK45) PDK standard cell using Synopsys DC compiler for the purpose of evaluating the area and power consumption of the proposed hardware in ASIC, as shown in Table 9.

According to Table 9, for the power measurement, the proposed method operates at 602.16 μW compared with the



(a) Power **(b)** Area

FIGURE 16. The power and area comparison between the proposed and the conventional methods on the 2D convolution.



(a) Power reduction versus accuracy **(b)** Area reduction versus accuracy

FIGURE 17. The comparison between the proposed and the existing methods.

803.76 μW spent by the conventional counterpart, which is an approximately 25% difference. For the area, the ASIC synthesis results show that the proposed hardware footprint ($5337\mu m^2$) is approximately 86% the size of that of the conventional method ($6157\mu m^2$), which means our proposed method needs approximately 14% less area. These results show that we can achieve a lower cost in terms of hardware cost, area and power consumption in ASIC. This is possibly because the proposed hardware, which requires specific circuits in the estimation of $\cos\theta_k$, is more suitable for specific circuit implementations in ASIC than the programmable hardware in FPGA. Comparing to other approximate methods, the proposed method consumes less power, approximately 10%–15% less, while utilizes more area with the same accuracy, which is the result of operation reduction. In addition, the proposed hardware is designed using the pipeline technique, which allows the streaming of results in every clock cycle.

Next, we measure the resource and power utilization for specific applications. All designs are synthesized under the same environment, which is the Synopsys DC compiler with a 45nm FreePDK. Firstly, we begin with the 2D hardware GC2. Its utilization is determined by the number of GC units and the number of FIFOs used to buffer each row of the input image. The number of FIFOs in the proposed method depends on the size of the image filter, in order to simplify the process of data scheduling. We test the proposed method with difference filter sizes, while keeping the input image size fixed at 227×227-pixel^2, which is the same image size in our CNN evaluation. The result is shown in Fig. 16. From the figure, our proposed method reduces the area and power consumption of the conventional method by approximately 5% and 25%, respectively. Finally, we measure the utilization and power consumption of the CNN engine. The AlexNet CNN model is used for testing the architecture, with $\mathtt{m} = 15$ and $\mathtt{p} = 10$.

TABLE 10. Experiment of CNN execution with AlexNet model based on the conventional and proposed convolution methods.

Design	area (mm^2)	power (mW)
Conventional	2.61	132.92
Proposed	2.46	114.72

The result is shown in Table 10. The proposed design has a smaller area and consumes less power than the conventional method. Power consumption of the proposed method is 13% lower than the conventional method. The proposed method outperforms the conventional method in terms of the area and power consumption.

For three different matrices, the power-accuracy and area-accuracy tradeoff graphs in Fig. 17a and Fig. 17b show that the proposed design has a good tradeoff for both area and power. For the area-accuracy tradeoff, there are some designs that use less area and gain more accuracy. Nevertheless, our design is above average among the designs. Additionally, our proposed method has the best tradeoff for power consumption among the various approximate multipliers. Moreover, for applications in image filtering and neural networks, the accuracy of the proposed method is acceptable. Our proposed method can reduce the number of operations while trading off accuracy and throughput with the power consumption. Operation reductions directly affect power consumption. These characteristics make our proposed method a favorable choice for an energy-efficient design.

VII. CONCLUSION

In this article, the implementation of convolution computation to achieve less power consumption with the limited resources of embedded applications along with better performance is proposed. Trading off the computation accuracy, we propose an approximate convolution procedure based on the geometric interpretation of the dot product by approximation of $\cos\theta$. All state-of-the-art convolution methods, approximate convolution methods, and the proposed method are implemented on FPGA and the 45 nm FreePDK CMOS process with Vivado 2021.2 and Synopsys DC compiler to investigate the area and power utilization. The experimental results indicate that our design has a 25% power reduction and area reduction compared with exact convolution and an approximately 10%–15% power optimization compared with other approximate methods. We also verify that our method supports 2D convolutions and has an acceptable classification accuracy for CNN models, where we take the AlexNet CNN model as a benchmark example. We also quantify the time complexity of the proposed method. The proposed approximate convolution and the corresponding hardware architecture have applications to inference and signal processing tasks that tolerate some errors in the convolution and require low power consumption.

ACKNOWLEDGMENT

The authors would like to thank Silicon Craft Technology Public Company Ltd. and Synopsys for the Synopsys DC Compiler's license.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25. Red Hook, NY, USA: Curran Associates, 2012.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[4] D. H. Wolper and W. G. Macready, "No free lunch theorems for optimization," *IEEE Trans. Evol. Comput.*, vol. 1, no. 1, pp. 67–82, Apr. 1997.

[5] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64270–64277, 2018.

[6] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," 2015, *arXiv:1510.00149*.

[7] W. Dargie, "A stochastic model for estimating the power consumption of a processor," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1311–1322, May 2015.

[8] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *Proc. IEEE Eur. Test Symp.*, May 2013, pp. 1–6.

[9] R. Nair, "Big data needs approximate computing: Technical perspective," *Commun. ACM*, vol. 58, no. 1, p. 104, Dec. 2014, doi: 10.1145/2688072.

[10] Z. Liu, A. Yazdanbakhsh, T. Park, H. Esmaeilzadeh, and N. S. Kim, "SiMul: An algorithm-driven approximate multiplier design for machine learning," *IEEE Micro*, vol. 38, no. 4, pp. 50–59, Jul. 2018.

[11] C. Khongprasongsiri, W. Suwansantisuk, and P. Kumhom, "An investigation of multiplication error tolerances in CNN and SIFT," *Proc. SPIE*, vol. 11049, pp. 714–718, Mar. 2019.

[12] B. Zhuang, C. Shen, M. Tan, L. Liu, and I. Reid, "Towards effective low-bitwidth convolutional neural networks," 2017, *arXiv:1711.00205*.

[13] F. Johansson, "Faster arbitrary-precision dot product and matrix multiplication," 2019, *arXiv:1901.04289*.

[14] M. Kumm, "Optimal constant multiplication using integer linear programming," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 65, no. 5, pp. 567–571, May 2018.

[15] F. de Dinechin, S.-I. Filip, M. Kumm, and L. Forget, "Table-based versus shift- and-add constant multipliers for FPGAs," in *Proc. IEEE 26th Symp. Comput. Arithmetic (ARITH)*, Kyoto, Japan, Jun. 2019, pp. 1–8. [Online]. Available: https://hal.inria.fr/hal-02147078

[16] M. Kumm, M. Hardieck, and P. Zipf, "Optimization of constant matrix multiplication with low power and high throughput," *IEEE Trans. Comput.*, vol. 66, no. 12, pp. 2072–2080, Dec. 2017.

[17] R. Appuswamy, T. Nayak, J. Arthur, S. Esser, P. Merolla, J. Mckinstry, T. Melano, M. Flickner, and D. Modha, "Structured convolution matrices for energy-efficient deep learning," 2016, *arXiv:1606.02407*.

[18] Q. V. Le, T. Sarlos, and A. J. Smola, "Fastfood: Approximate kernel expansions in loglinear time," 2014, *arXiv:1408.3060*.

[19] G. Tong and L. Huang, "Fast convolution based on Winograd minimum filtering: Introduction and development," 2021, *arXiv:2111.00977*.

[20] V. Y. Pan, "How bad are Vandermonde matrices?" *SIAM J. Matrix Anal. Appl.*, vol. 37, no. 2, pp. 676–694, 2016, doi: 10.1137/15M1030170.

[21] L. Meng and J. Brothers, "Efficient Winograd convolution via integer arithmetic," 2019, *arXiv:1901.01965*.

[22] Y. Zhao, D. Wang, and L. Wang, "Convolution accelerator designs using fast algorithms," *Algorithms*, vol. 12, no. 5, p. 112, May 2019. [Online]. Available: https://www.mdpi.com/1999-4893/12/5/112

[23] B. Barabasz, A. Anderson, K. M. Soodhalter, and D. Gregg, "Error analysis and improving the accuracy of Winograd convolution for deep neural networks," 2018, *arXiv:1803.10986*.

[24] Y. Huang, J. Shen, Z. Wang, M. Wen, and C. Zhang, "A high-efficiency FPGA-based accelerator for convolutional neural networks using Winograd algorithm," *J. Phys., Conf.*, vol. 1026, May 2018, Art. no. 012019, doi: 10.1088/1742-6596/1026/1/012019.

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

[26] S. Fox, D. Boland, and P. Leong, "FPGA fastfood—A high speed systolic implementation of a large scale online kernel method," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, Feb. 2018, pp. 279–284, doi: 10.1145/3174243.3174271.

[27] K. Du, P. Varman, and K. Mohanram, "High performance reliable variable latency carry select addition," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2012, pp. 1257–1262.

[28] H. Waris, C. Wang, and W. Liu, "High-performance approximate half and full adder cells using nand logic gate," *IEICE Electron. Exp.*, vol. 16, no. 6, Jan. 2019, Art. no. 20190043.

[29] V. Gupta, D. Mohapatra, A. Raghunathan, and K. Roy, "Low-power digital signal processing using approximate adders," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 1, pp. 124–137, Jan. 2013.

[30] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *Proc. 24th Int. Conf. VLSI Design*, Jan. 2011, pp. 346–351.

[31] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2014, pp. 1–4.

[32] A. Momeni, J. Han, P. Montuschi, and F. Lombardi, "Design and analysis of approximate compressors for multiplication," *IEEE Trans. Comput.*, vol. 64, no. 4, pp. 984–994, Apr. 2015.

[33] S. Venkatachalam and S.-B. Ko, "Design of power and area efficient approximate multipliers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 5, pp. 1782–1786, May 2017.

[34] Z. Yang, J. Han, and F. Lombardi, "Approximate compressors for error-resilient multiplier design," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst. (DFTS)*, Oct. 2015, pp. 183–186.

[35] C.-H. Lin and I.-C. Lin, "High accuracy approximate multiplier with error correction," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Oct. 2013, pp. 33–38.

[36] O. Akbari, M. Kamal, A. Afzali-Kusha, and M. Pedram, "Dual-quality 4:2 compressors for utilizing in dynamic accuracy configurable multipliers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 4, pp. 1352–1361, Apr. 2017.

[37] F. Sabetzadeh, M. H. Moaiyeri, and M. Ahmadinejad, "A majority-based imprecise multiplier for ultra-efficient approximate image multiplication," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 66, no. 11, pp. 4200–4208, Nov. 2019.

[38] M. Ahmadinejad, M. H. Moaiyeri, and F. Sabetzadeh, "Energy and area efficient imprecise compressors for approximate multiplication at nanoscale," *AEU Int. J. Electron. Commun.*, vol. 110, Oct. 2019, Art. no. 152859. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1434841119304923

[39] F. Ranjbar, Y. Forghani, and D. Bahrepour, "High performance 8-bit approximate multiplier using novel 4:2 approximate compressors for fast image processing," *Int. J. Integr. Eng.*, vol. 10, no. 1, pp. 1–20, Apr. 2018.

[40] T. Kong and S. Li, "Design and analysis of approximate 4–2 compressors for high-accuracy multipliers," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 29, no. 10, pp. 1771–1781, Oct. 2021.

[41] M. Garrido, P. Källström, M. Kumm, and O. Gustafsson, "CORDIC II: A new improved CORDIC algorithm," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 63, no. 2, pp. 186–190, Feb. 2016.

[42] H. Jiang, C. Liu, N. Maheshwari, F. Lombardi, and J. Han, "A comparative evaluation of approximate multipliers," in *Proc. IEEE Int. Symp. Nanosc. Architectures (NANOARCH)*, Jul. 2016, pp. 191–196.

[43] D. Esposito, A. G. M. Strollo, E. Napoli, D. De Caro, and N. Petra, "Approximate multipliers based on new approximate compressors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 12, pp. 4169–4182, Dec. 2018.

[44] V. Leon, G. Zervakis, S. Xydis, D. Soudris, and K. Pekmestzi, "Walking through the energy-error Pareto frontier of approximate multipliers," *IEEE Micro*, vol. 38, no. 4, pp. 40–49, Jul. 2018.

[45] J. Liang, J. Han, and F. Lombardi, "New metrics for the reliability of approximate and probabilistic adders," *IEEE Trans. Comput.*, vol. 62, no. 9, pp. 1760–1771, Sep. 2013.

[46] L. Deng, "The MNIST database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 141–142, Nov. 2012.

[47] A. Krizhevsky, "Learning multiple layers of features from tiny images," Univ. Toronto, Toronto, ON, Canada, Tech. Rep., 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

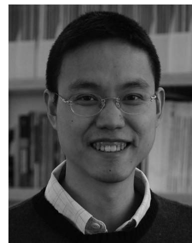
[48] A. Quattoni and A. Torralba, "Recognizing indoor scenes," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 413–420.

[49] X. Li and J. Cai, "Robust transmission of JPEG2000 encoded images over packet loss channels," in *Proc. IEEE Int. Conf. Multimedia Expo*, Jul. 2007, pp. 947–950.

[50] C. Holland. (Mar. 2011). *Xilinx Provides Details on ARM-Based Devices*. [Online]. Available: https://www.embedded.com/xilinx-provides-detailson- arm-based-devices/

[51] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "FreePDK: An open-source variation-aware design kit," in *Proc. IEEE Int. Conf. Microelectronic Syst. Educ.*, Jun. 2007, pp. 173–174.

CHANON KHONGPRASONGSIRI (Student Member, IEEE) received the B.Eng. degree in electronic and telecommunication engineering from the King Mongkut's University of Technology Thonburi, Thailand, in 2018, where he is currently pursuing the master's degree. His research interests include the Internet of Things and their applications, digital systems design and implementation, and signal and image processing.



WATCHARAPAN SUWANSANTISUK (Member, IEEE) received the B.S. degrees in electrical and computer engineering, and computer science from Carnegie Mellon University, PA, USA, in 2002, and the M.S. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, in 2004 and 2012, respectively.

He is currently an Assistant Professor with the King Mongkut's University of Technology Thonburi (KMUTT), Thailand. Before joining KMUTT, he spent summers at the University of Bologna, Italy, as a Visiting Research Scholar and at Alcatel-Lucent Bells Laboratory, NJ, USA, as a Research Intern. His main research interests include wireless communications, synchronization, and statistical signal processing.

Dr. Suwansantisuk serves on the technical program committees for various international conferences and served as the Symposium Co-Chair for the IEEE Global Communications Conference, in 2015. He received the Leonard G. Abraham Prize in the field of communications systems from the IEEE Communications Society, in 2011, jointly with Prof. M. Chiani and Prof. M. Win; and the Best Paper Award from the IEEE RIVF International Conference on Computing and Communication Technologies, in 2016, jointly with N. Chedoloh.

PINIT KUMHOM received the B.Eng. degree in electrical engineering from the King Mongkut's University of Technology Thonburi, Thailand, in 1988, and the Ph.D. degree in electrical and computer engineering from Drexel University, PA, USA, in 2000. He is currently an Assistant Professor with the King Mongkut's University of Technology Thonburi. His research interests include the Internet of Things and their applications, digital systems design and implementation, and signal and image processing.