Received September 20, 2021, accepted September 29, 2021, date of publication October 4, 2021, date of current version October 15, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3117335

Development of Floating-Point MAC Engine for 2-D Convolution of Image

AJAY KUMAR SAHU, VISHNUMURTHY KEDLAYA K., AND SUBRAMANYA G. NAYAK, (Member, IEEE)
Department of Electronics and Communication Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal 576104, India
Corresponding author: Subramanya G. Nayak (gs.nayak@manipal.edu)

ABSTRACT In the emerging trend of Graphics Processing Architecture, IEEE 754-2008 Floating point numbers are being widely used. Convolution is one of the standard operations in image processing applications, and because of its computationally intensive nature, an appropriate and efficient image processing architecture is of great need. This paper proposes a single-precision Floating Point MAC engine to accelerate the sliding window algorithm for the 2-D convolution of image. The engine uses a modified algorithm for virtual zero-padding that saves memory space, and it also provides configurable parameters to specify filter and image size. A low power multiplier with reduced dynamic power, specifically when operating on pixels and a faster increment by one circuit based on AND-EXOR gate structures, has been proposed to improve the MAC architecture. Finally, the paper shows the post-synthesis power dissipation, area estimate, and the quality comparison of the image obtained from the RTL Simulation of the proposed architecture.

INDEX TERMS 2-D image convolution, low-power multiplier, increment by one, convolution accelerator.

I. INTRODUCTION

Various imaging applications such as object detection, classification, and visual search requires image filtering. Among these applications, visual search involves a massive amount of calculations for feature detection in the input image. In such scenarios, rather than using software (SW) implementations that are not capable of providing real-time performance, hardware (HW) implementations of image filtering can be more performance optimized for real-time processing of higher resolution images.

Over the years, several designs have been proposed but are tailored for Gaussian kernels only. These methods and designs cannot be used for applications that work on kernels other than Gaussian kernels. The design presented in [1] is compatible with such scenarios as it does not recur to the separability property of the Gaussian kernels but requires a complex arrangement of SRAM modules. This complexity grows as the dimensions of the tile to be processed increase. The HW convolution module in [2] is confined to only 3×3 kernels. Its convolution operation relies on a tree-based structure. However, it is not flexible to incorporate convolution of any other size; neither such structure is suitable for parallelism in the convolution of an image. The architecture proposed in [3] is designed for kernels of fixed coefficients and fixed no. of input sets; it is well suited when processing continuous data streams. Moreover, the implementation is in FPGA, which does not reveal the design's actual throughput and power efficiency.

To address the challenges discussed above, it is crucial to design an architecture that can support a parallel computation of convolution operation while reducing the data movement from off-chip and should support the different sizes of the kernel. This paper contributes to the HW design of the single-precision floating-point based MAC engine, a hardware accelerator for image convolution. It is a standalone module that can be integrated into a microcontroller or SOC system for image processing applications. The aim is to develop an architecture that can independently perform 2-D convolution without relying on the host system and operate with kernels of any size rather than having a fixed-size kernel buffer. The parallel arrangement of functional units and linear indexing of the pixel data & kernel weights using a small micro-code processor (MCP) has achieved this.

While being a separate unit, it allows overcoming the limitations of SW-based image convolution. The 2-D convolution is computed in a loop-wise manner to save power. We show that the proposed sliding window algorithm can retain the

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

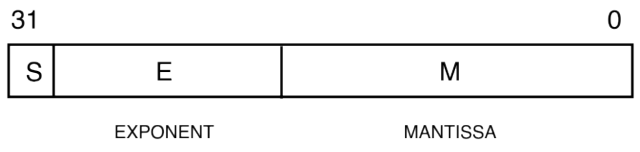

FIGURE 1. IEEE-754 single-precision floating-point format.

boundary pixels without the need for padding. The proposed algorithm also addresses the additional local memory requirement for HW implementation of convolution, specifically when trying to exploit parallelism in the filtering process. The data movement is reduced by mapping the entire image into the internal register files (RF) of the functional units (FU). The hardware has configurable parameters that support different shapes in convolution. Meanwhile, an application-specific multiplier with reduced dynamic power and a faster increment by one circuit has also been presented to optimize the FU.

A. CHALLENGES IN IMAGE PROCESSING HW DESIGN

Several workloads, such as Image Processing, require specialized hardware to meet expected performance. However, such responsive hardware comes at the cost of flexibility, and sometimes power is sacrificed. There is a need to create a specialized HW that offers reasonable performance, power efficiency, and flexibility. Image processing calls for such specialized HW since CPUs are not optimized for such a high level of data parallelism. The traditional solution followed to exploit this data parallelism is an implementation using SIMD Units which is highly flexible but way slower. The alternative solution to this is using GPUs which provide great flexibility and are more performant than SIMD units; however, they consume too much power. Another alternative solution is deploying ASIC accelerators which will be very performant and power-efficient at the cost of flexibility since ASIC can implement only one algorithm. An Engine is an architecture with reasonable performance, flexibility, and power for a specific algorithm.

B. FLOATING POINT FORMAT

Floating-point data is commonly used in scientific images, representing measurements using 32 or 64 bits per pixel. In the rest of literature 32-bit Floating Point (FP32) is used which is a IEEE-754 standard format [4] comprised of a sign bit (S), 8 exponent bits (E) and 23 mantissa bits (M). Figure 1 shows the internals of FP32 and its equivalent value is calculated as shown in equation (1)

$$\text{Value} = (-1)^{S} \times 2^{E} \times (1.M) \tag{1}$$

C. INTEGER VS FLOATING POINT FUNCTIONAL UNIT FOR IMAGE PROCESSING

Several works have been dedicated to integer functional units for image processing that processes 8-bit grayscale channel, 24b-bit RGB channels, and 32-bit RGBA channels. However, very few pieces of literature are available related to floating-point image processing hardware. Floating-point pixels have application in HDR Imaging in mobile cameras, satellite imaging as elevation of the land surface has different dynamic ranges. The filter being used can also be categorized into integer-order and fractional-order filters. Fractional order filters collect more image details in their high-frequency region than integer ones and provide better noise sensitivity. Due to this very advantage, floating-point image processing is also considered a suitable option for biomedical applications. Floating-point images can use 32-bit floats or 16-bit half-precision format, but half-precision does not have as much dynamic range as required in practical applications. Floating-point pixels have some advantages over Integer pixels but have extra memory overhead.

II. RELATED WORKS

Image features and weights can be compressed and stored in 16 bit Floating Point (FP16) [5]; however, for optimum calculation error, it is always suitable to perform computation in single-precision floating-point (FP32) rather than half precision (FP16) [6] by decompressing FP16 to FP32. FPGA based Fixed point Convolution accelerator [7] gives a comparison among Strassen, Winograd, Strassen–Winograd, which requires fewer computation resources to accelerate convolution. The algorithm uses tree-based convolution, which is more power efficient than the conventional algorithm but compromises with CNN's flexibility. Another FPGA based Floating point convolution module [2] also uses tree based structure. The module processes FP32 features and weights by quantizing them to BF16, which is generally used in machine learning and the convolution module achieves better error distribution than computation in FP16. However, BF16 based system has lesser precision. Convolution on a SIMD architecture based on a smart-tiling scheme that uses four accumulator registers in PE instead of one is proposed [8]. The architecture deals with fixed-point data type and showcases the implementation of 4 layer CNN-based detection. FPGA based implementation of a convolution system for floating-point pixels and fractional order filter is shown in [9]. It uses a line buffer to hold the consecutive rows of pixels in an image.

An ultra-low-power fused multiply-add unit has been proposed in [10], which uses a multi-speculative kogge-stone adder and presents new digital logic circuits for 2's complement. The multi-speculative adder divides the addition into fragments and assumes carry-in is '0' to each section and later corrects any possible mispredictions using the new CCTA. The convolution engine proposed in [11] enables data reuse and works on reducing data transfer overheads. It uses a line buffer and several convolution slices to accelerate the algorithm.

Some existing work [12], [13], and [14] has proposed methods and techniques to reduce the dynamic power consumption of multiplier core; however, these methods use

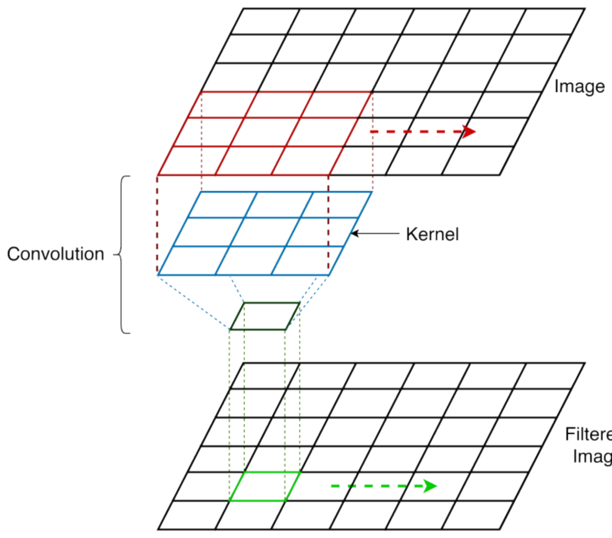

FIGURE 2. **Sliding window algorithm based 2-D convolution.**

modified gate structures, which are not compatible with a semi-custom ASIC design.

III. ARCHITECTURE

A. CONVOLUTION OPERATION

A 2-D convolution collects necessary data from image pixels; mathematically, it is an element-wise multiplication of image kernel (filter weights) and input feature map (pixel data). This process of extracting essential features in image pixels is called Convolution, a fundamental building block of any Convolutional neural network. Convolution operator is denoted by '$*$', equation (2) & (3) expresses the entire convolution operation.

$$y[m,n] = x[m,n] * h[m,n] \tag{2}$$

$$y[m,n] = \sum_{j=-\infty}^{\infty}\sum_{i=-\infty}^{\infty} x[i,j] \times h[m-i, n-j] \tag{3}$$

where x is function from image location to pixel value, h is filter applied to the image, and y is filtered image.

There are multiple ways to compute image convolution, such as sliding window transform, shift-multiply add Fourier transform. A pictorial representation of the Sliding-window algorithm is shown in Figure 2. The stencil of kernel matrix slides over the image matrix so that the target pixel appears precisely in the center and then performs multiply-accumulate for that particular region; this goes on till the whole image area is covered.

The convolution is indeterminate at the boundaries of the image, an $i \times i$ image, and $k \times k$ kernel produces an $(i - k + 1) \times (i - k + 1)$ image. These can be prevented by padding the image with a layer of zeros or sometimes any arbitrary value. To prevent the loss of pixels at the boundary of the image, the Algorithm in Algorithm 1 is modified to incorporate virtual zero padding. It does not actually use zeros for padding; instead arranges the remaining pixels of the zero-padded $[I]_{k\times k}$ matrix with its corresponding kernel coefficients for the current window.

Algorithm 1 Sliding Window for 2-D Convolution

With an Image matrix $[I]_{i\times i}$ and Kernel Matrix $[K]_{k} \times_{k}$

```
for cr in range (0, orows-1)
    for cc in range (0, ocols-1)
        for i in range (cr, krows + cr-1)
            for j in range (cc, kcols + cc-1)

                op[cr][cc] + = ip[i][j] ×kernel[i-cr][j-cc];
```

where, orows = irows - krows + 1;
ocols = icols - kcols + 1,
irows, icols, krows, kcols are the configurable parameters

Now the convolution of $i \times i$ images and $k \times k$ kernel with p level of padding produces an $(i - k + 2p + 1) \times (i - k + 2p + 1)$ image. A typical HW implementation of convolution with padding would require storing this padding value as we use image partitioning; the no. of padding values that need to be stored increases. The proposed modified algorithm shown in Algorithm 2 brings power saving as well since there is no need for storing zeros for padding and wasting the operation cycle in the calculation of redundant element whose value is going to be zero. Table 1 shows the memory depth required in different scenarios; from Table 1, it is evident that Algorithm 2 can save significant local memory space for HW-based convolution.

B. FLOATING MAC UNIT (FPMAC)

FPMAC composes the data path of the engine. It is responsible for performing a two-stage pipelined multiplication & accumulation and producing an output pixel. A signal from controller controls the magnitude of this accumulation, and it has two accumulator registers to support continuous multiplication & accumulation also shown in Figure 3. The use of a second accumulator allows the accumulation of the new incoming window data without breaking the flow of MAC operation. The MAC is fed with the kernel weights and the input pixel data coming from the register file. After each $k \times k$ cycle, the target pixel surrounded by $k^2 - 1$ pixels is produced into a filtered pixel. FU performs convolution in a multiplication and accumulation manner till it covers every element of the current window, the result is saved in one of the accumulation registers, and for the next target pixel, the other accumulator register becomes active.

C. FLOATING POINT ADDER

The following are the steps to be pursued to add two single precision floating point numbers, block diagram of FP Adder is show in Figure 4. Let X and Y be the two numbers to be added.

i. First, the exponents $e_x$and e_y are compared and the tentative exponent is the larger of these two.

Algorithm 2 Modified Sliding Window With Virtual Zero Padding

With an Image matrix $[I]_{i\times i}$ and Kernel Matrix $[K]_{k\times k}$

```
for cr in range (0, irows-1)
    for cc in range (0, icols-1)
        for i in range (cr-p×b, krows + cr-p-a-1)
            for j in range (cc -p×d, kcols + cc-p-c-1)

                op[cr][cc] + = ip[i][j] ×kernel[i-cr+p][j-cc+p];
```

Where p is number of padding level

$$a = \begin{Bmatrix} 1, cr == irows - 1 \\ 0, otherwise \end{Bmatrix}, \quad c = \begin{Bmatrix} 1, cc == icols - 1 \\ 0, otherwise \end{Bmatrix}$$

$$b = \begin{Bmatrix} 0, cr == 0 \\ 1, otherwise \end{Bmatrix}, \quad d = \begin{Bmatrix} 0, cc == 0 \\ 1, otherwise \end{Bmatrix}$$

TABLE 1. Memory Space requirement for image with padding and without padding.

	Memory Depth
Without padding	i^2
With padding	$i^2 + 4i + 4$
A 16 × 16 image without padding	256
A 16 × 16 image with Padding	256 + 68
A 16 × 16 image partitioned into 4 equal blocks with Padding	256 + 144

ii. The mantissa with smaller exponent is shifted right by 'e_x - e_y' position and set as 2nd operand to addition/subtraction while the other mantissa is set as 1st operand.

iii. The residual mantissa bits left out due to shifting are used for guard bit, rounding bit and sticky bit (GRS) computation.

iv. The aligned mantissas are added or subtracted depending upon the sign of both operands and sign of result is set to the sign of largest exponent operand.

v. The sum is normalized in two cases:

- When there is carry out of significand addition, then the m_r is shifted right by one position through the GRS bits and the tentative exponent is incremented by one.
- When there is cancellation in significand, in this case the number leading zeros 'z' in the tentative significand is counted or anticipated and the tentative significand is shifted left by 'z' position where as tentative exponent is set to $e_r - z$. Accordingly G, R, S bits also gets modified.

vi. The guard bit, rounding bit and sticky bit changes during normalization based on carry out bit of addition. And at last rounding logic decides the rounding status of mantissa.

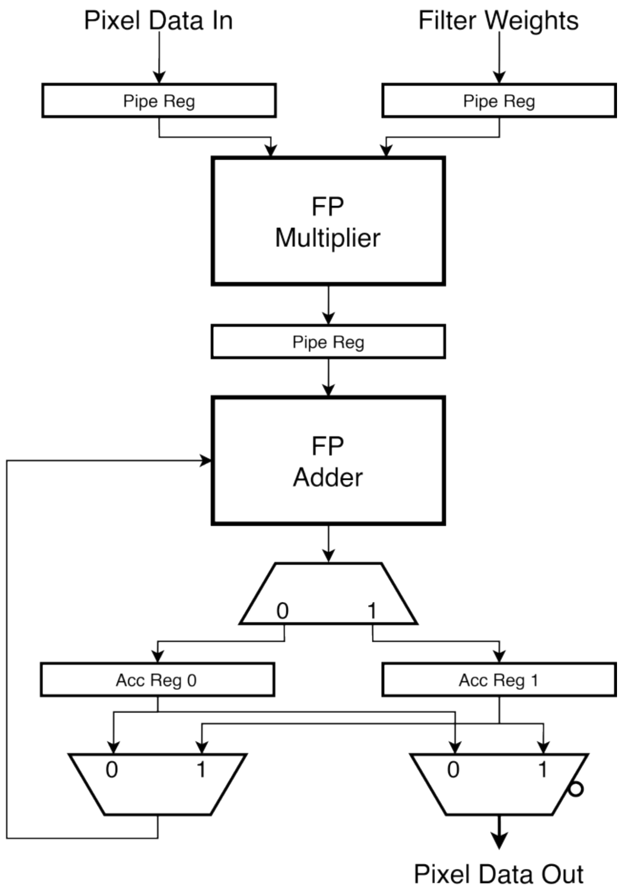

FIGURE 3. Architecture of floating point MAC unit.

The component honors overflow and underflow, and accordingly settles the output value. One of the common operations in floating-point arithmetic is increment by one used for normalization or rounding purposes. Increment by One is part of the critical path and hence should be fast. A typical way to increment a binary number is to use a carry look-ahead adder but it offers a great amount of delay.

Nevertheless, there is a more efficient way to do this. One way to accomplish increment by one is to use binary to the excess-1 converter, but it only works for fewer bits and will face higher propagation delay for many bits. The design in [4] suggests using a carry-propagate adder for this. Another way is trailing one detection of the input word, then special encoding followed by partial complementing. Special encoding encodes the results of trailing one detector to generate a string 'S'. All the bit positions in 'S' which follows trailing zero along with trailing zero position itself are set to one. This work proposes a new method that works by producing a mask string 'S' at first, without the need of trailing one detector and then passing it through ex-or gates along with input bits for partial complementing. This complete method of incrementing a 24-bit binary number by one is shown in Figure 5.

A string expression to determine the one-hot representation is mentioned in [15], where only those bits in string expres-

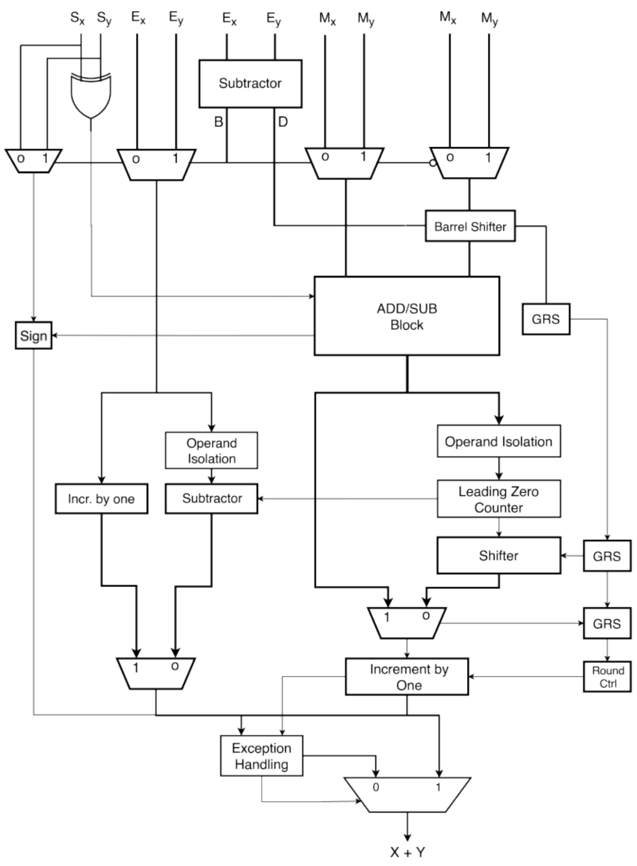

FIGURE 4. Single precision floating point adder.

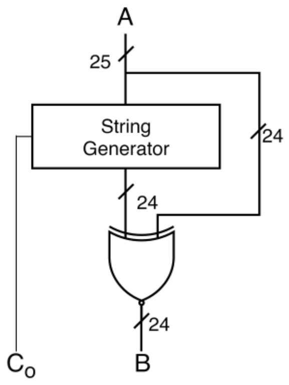

FIGURE 5. Proposed increment by one circuit.

sion that follow the leading one are set to '1'. To derive the string S required for proposed circuit, the logical expression illustrated in [15] is modified, the ith bit of S denoted as S_i, $0 \leq i \leq n\text{-}1$ is defined as follows:

$$S_i = A_{n-1} \bullet A_{n-2} \bullet \ldots\ldots. \bullet A_{i+2} \bullet A_{i+1} \bullet C_0 \qquad (4)$$

where • denotes the logical AND operation. A_i is the input to increment by one circuit. The string S thus can be defined as:

$$\begin{aligned} S_0 &= C_0 \\ S_i &= 1, \quad \text{if } i \leq R \\ &\quad 0, \quad \text{otherwise} \end{aligned} \qquad (5)$$

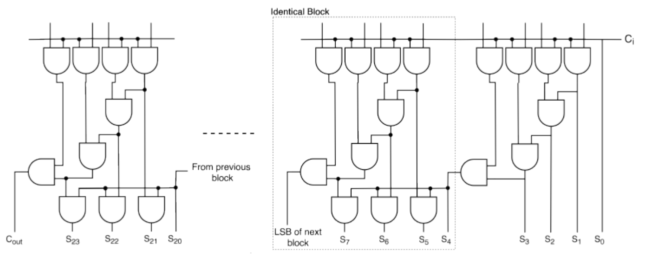

FIGURE 6. Proposed 24-bit string generator for increment by one circuit.

TABLE 2. Power dissipation of increment by one circuit.

	Power (μW)	Delay (ns)	PDP (mJ)	Area (μm^2)
Conventional Circuit [4]	4.530	4.934	22.05	159.899
Proposed Circuit	5.664	1.662	8.96	181.573

where R-1 is the position of trailing one in binary word A and C_0 is carry-in (control bit). Translating the equation (4) & (5) into the gate-level circuit will result in a long chain of AND gates. It can be avoided when the AND gate chain is broken into identical blocks, each generating 4-bits of string S as shown in Figure 6, and the output of each block is isolated using the MSB from the previous block.

The advantage of the latter method is that it buffers the input word when the incremented value is not desired and eliminates the need for a multiplexer in the normalization or rounding section. Table 2 shows proposed method is significantly faster than the other suggested method in [4].

D. FLOATING POINT MULTIPLIER

Below are the steps to be followed to multiply two single precision floating point numbers, block diagram of FP multiplier is shown in Figure 7. Let X and Y be the two numbers to be multiplied.

i. The exponents of both operands are added and then subtracted with bias '127'.
ii. The significand $1.m_x$ and $1.m_y$ are multiplied and the product of two significands will be less than '4'. The lower 23-bits are forwarded as pre-normalized mantissa and rest of the bits (residual products) are used for sticky bit computation. Few bits are used for guard and round bits.
iii. As the step of normalization the Least Significant Bit (LSB) of product is checked for bit '1'and if so pre-normalized mantissa is shifted right by one position. In parallel of Normalization, rounding logic decides the rounding status of mantissa.
iv. The carry out bit from rounding of normalized mantissa, along with LSB of product decides whether to increment the tentative exponent or not.
v. The value is also checked for overflow before normalization and rounding while it is checked for underflow after rounding.

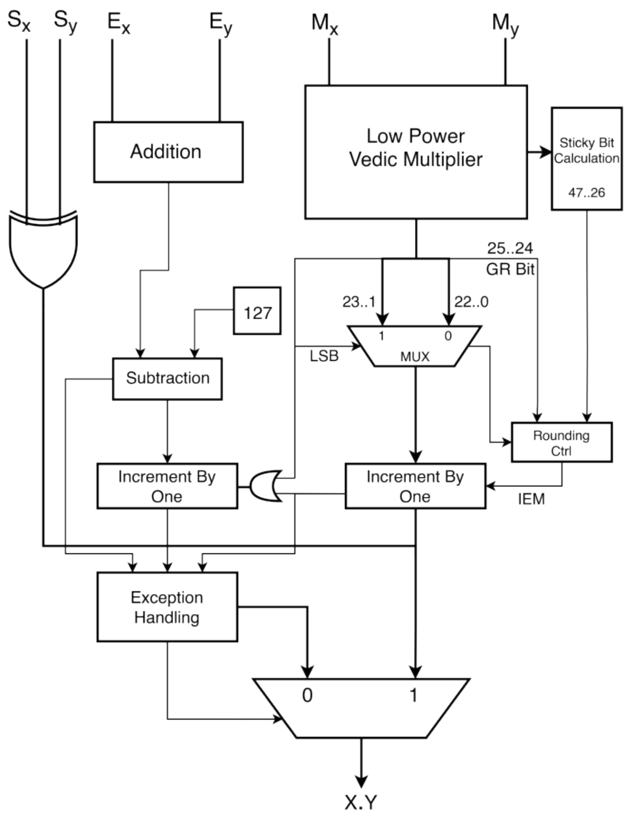

FIGURE 7. Single precision floating point multiplier.

The significand multiplication itself is fixed-point multiplication and has a significant share in the power consumption of MAC. The relatively close elements in the image matrix are almost equal in value and do not require to be fully multiplied again and again. Based on this observation, this work proposes a low power multiplier (LPM) specific to the image processing application.

From the population graph of the 7×7 pixel matrix shown in Figure 8(c), it can be observed that the consecutive pixels have a small dynamic range, and the upper bits of these pixels toggle infrequently; the flat line curve in the graph depicts this exactly. This infrequent change gives an opportunity where the i^{th} pixel is checked for its dynamic range with respect to the $i+1^{th}$ pixel. If the upper bits of these pixels are equal, the multiplier disables switching in the higher section of the multiplier using operand isolation. The block diagram of this method is shown in Figure 9. The structure of the Vedic Multiplier based on Urdhva Triyakbhyam consists of several sections, each working independently on the multiplication of different bit combinations of two operands; this allows disabling the unused section of multiplier dynamically, thereby saving significant power without affecting the working of the rest of the multiplier.

The block diagram in Figure 9 depicts the implementation of LPM using the Vedic Multiplication method, where a 2-stage line buffer for both the operands compares the two consecutive operands and saves significant power as shown in Table 3 without affecting the delay of the data path.

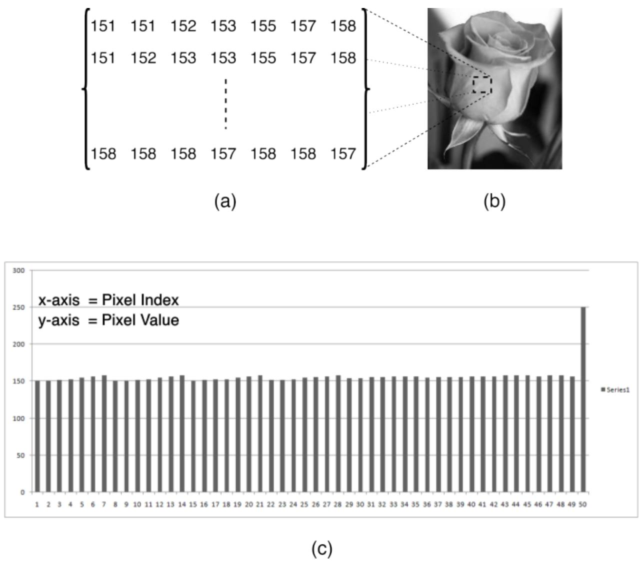

FIGURE 8. (a) A 7 × 7 pixel matrix, (b) A grayscale Image, (c) Graph of consecutive image pixels of a small region.

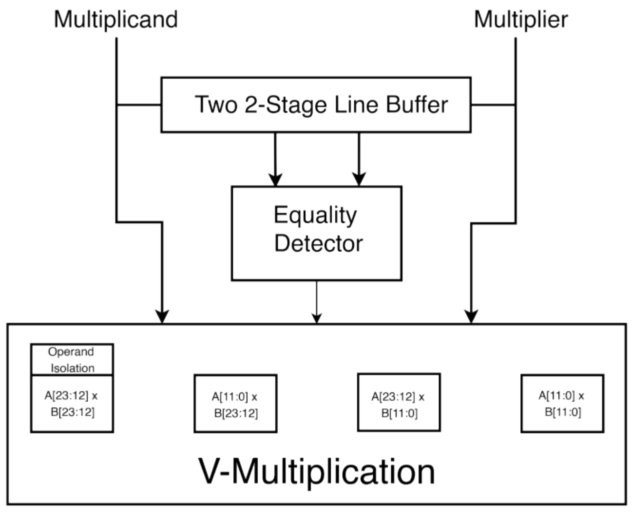

FIGURE 9. Proposed low power vedic multiplier.

TABLE 3. Power dissipation of 24-bit vedic multiplier for mantissa.

	Power (μW)	Delay (ns)	PDP (mJ)	Area (μm^2)
Conventional Circuit [16]	617.509	15.084	9262	7355
Proposed Circuit	527.176	15.529	8168	7564

E. ENGINE ARCITECTURE

The engine uses FPMAC as the basis of the functional Unit whose operation is controlled by the controller. The MAC Engine architecture is shown in Figure 10. To accelerate the

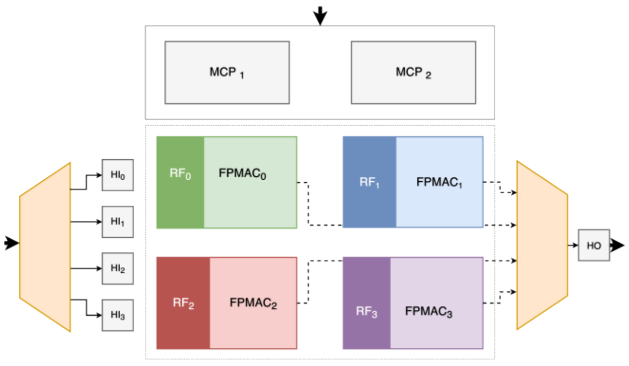

FIGURE 10. Architecture of FP MAC engine.

sliding window algorithm, the architecture employs an array of 2^n functional units, each working in parallel, where 'n' is the scalable factor for the array. The parallelism of the 2-D convolution on an image is based on a basic approach of partitioning the image [17] into blocks of rows and columns, each of these blocks forming a square matrix. The architecture shown in Figure 10 partitions the image into 4 blocks. For a given image, each FPMAC out of 2^n functional units performs the 2-D convolution only on a single block, thereby exploiting parallelism and acceleration of image convolution.

F. MEMORY SYSTEM

The primary concern of Algorithms 1 & 2 is that the degree of data replication of input pixels is very high, which leads to complex and costly memory access patterns when the image pixels are stored off-chip. Inside the functional unit, a deep internal memory (register file) is merged directly with FPMAC to avoid the frequent data travel from the host memory; this will reduce the interconnect power related to multiple memory access of the overlapped data when pixels are stored off-chip. This tight coupling of register file with FU will come alive at the physical design of the architecture. And the power reduction in data access will be achieved without the need for data/pixel reuse but at the cost of a larger internal storage unit. However, various existing designs also have large on-chip/internal memory to minimize the enrgy.

To support the image partitioning within the engine, pixel transfer from external memory to the engine and within the engine plays a crucial role. The data written into the external DRAM by the host CPU is collected one by one at the engine via a buffer. The pixel data distribution among the functional units analogous to time division de-multiplexing (TDDM). Using this method helps eliminate the data hindrance to the other functional units, which otherwise would occur if other functional units are written into only after the writing of image block into the first functional unit has finished. Respective data to each functional unit are transferred one by one in consecutive cycles via data handlers that monitor each register file's write addresses. To implement the TDDM like data distribution, a four-level nested for-loop as mentioned in proposed Algorithm 3 is used which can also be viewed as

Algorithm 3 Image Partitioning Algorithm

```
Ina Image matrix $[I]_{i\times i}$

for p in range (0, t-1)
    for q in range (0, t-1)
        for fu_r in range (0, $\log_2$(fu)-1)
            for fu_c in range (0, $\log_2$(fu)-1)

                read = (rows$\times$ (t$\times$fu_r + p)) + (t$\times$fu_c + q);
```

Where $t = \dfrac{i}{\log_2(fu)}$,

fu is number of functional units/ no. of blocks of partitioned image. 'fu' should be 4 or 16 or 64.

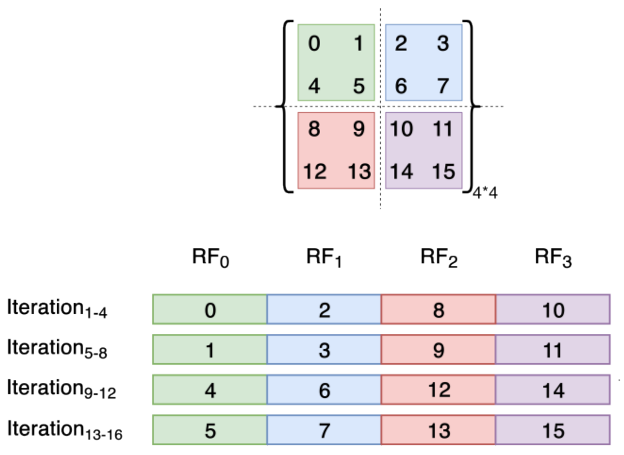

FIGURE 11. Pixel data distribution of 4 × 4 matrix using Algorithm 3.

image partitioning algorithm. Figure 11 shows how the pixel data for a 4 × 4 image is partitioned and distributed among the functional units using the mentioned algorithm.

Each element of the incoming image block is flattened into a single row vector and stored sequentially into vertical register file, also shown in Figure 12; this makes indexing the desired pixel easier. The other advantage of this method is that the convolution is not fixed to particular kernel size but instead it can support convolution with any size of the kernel. The maximum size of the kernel will be decided at the design time by describing the depth of the register file for storing the kernel. The data access pattern of content stored in this RF follows the modified sliding window algorithm mentioned in Algorithm 2. As discussed in Section III.A this algorithm does not require padding values to be stored,. Each processed pixel value from the 2^n units is picked up and multiplexed out to the external memory via a data handler.

G. MICROCODE PROCESSOR

The host processor can implement the modified convolution algorithm presented in Section III.A. Rather than relying on the host processor to compute memory addressing offsets for register files, the MAC Engine exploits a microcode

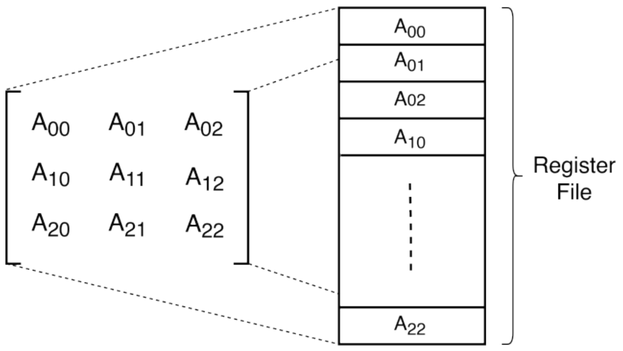

FIGURE 12. Matrix storage.

TABLE 4. PSNR comparison of images.

	PSNR	PSNR Error
RTL Simulation	4.2013	0.4857
MATLAB	3.7156	

processor similar to one presented in [18]. The microcode processor (MCP) is built as a part of this image convolution system. Information related to image size & kernel size is received from the host system, and its sets various flag to indicate completion of the image read operation and convolution. The microcode processor architecture is based on a small set of Instructions that implements the nested loops of the algorithm.

The implementation of this ISA is a single cycle execution architecture that has 8-bit instructions such as MOV, ADD, INCREMENT, JUMP, and COMPARE. The corresponding opcodes related to Algorithms 2 & 3 are stored in the two separate ROMs. Two different processors fetch these opcodes each cycle and perform desirable operations to obtain the address values for the external and internal memory. The whole operation of the MAC engine is orchestrated by these two MCPs. MCP 1 controls the data flow from external memory to internal memory and maps the partitioned image block to its respective functional unit, whereas MCP 2 controls data flow in the internal engine, from RF to FPMAC.

IV. EXPERIMENTAL RESULTS

To experiment with the engine's image processing, the engine has been subjected to RTL simulation. The pixels of a 128×128 single-channel image and 3×3 kernel for emboss operation are initialized into a RAM. The engine takes all the pixels along with kernel weights and write backs the filtered pixels to the RAM. Figure 13 shows the different versions of an image obtained to analyse the quality of processed pixels. The quality metric comparison is done in Table 4, PSNR of the sample obtained from Verilog RTL simulation is 4.2013, and the PSNR of one obtained from MATLAB is 3.7156. The embossed image obtained from the RTL simulation has better PSNR and detail compared to the one obtained from MATLAB.

Table 5 reports the area and power estimate of the design synthesized in a generic gpdk-45nm technology library in

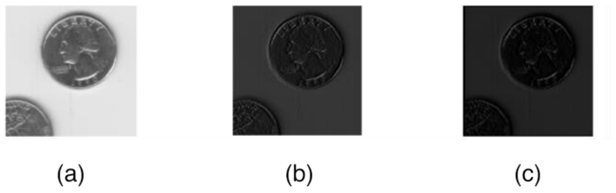

FIGURE 13. (a) Original image, (b) Image obtained from RTL simulation, (c) Image obtained from MATLAB.

TABLE 5. Power dissipation of MAC engine.

Parameters	Proposed MAC Engine	Convolution Module [2]
Design Structure	Parallel	Tree-based
Power (mW)	188.6	-
Total Area (μm^2)	79.35×10^3	71.58×10^3
Data Format	FP32	BF16
Application	Image Filtering	-

the slow corner where the compared work has been synthesized in 90nm technology. For this purpose we instantiated the design with four functional units each having four $64K \times 4B$ memory for image and a $32 \times 4B$ of memory for kernel. Architecture with these resources can process any image of size up to 512×512 with any of filter of size up to 5×5.

V. CONCLUSION

A convolution module modified at the algorithmic and architectural level is presented in this paper to improve the HW-based convolution. The previous works related to HW implementation of convolution do not show the image results for a floating point pixel data. Those literatures also do not discuss the padding in image convolution. The overall architecture of MAC engine is asynchronous and can work in complete autonomy when given Direct Memory Access to the main memory by the host system. The image can be stored either off-chip or directly mapped into internal memory (register files); this can save a lot of transaction power and demonstrate better utilization of internal memory.

The advantage of this approach in this work is that exploiting parallelism becomes easy, and significant dynamic power can be saved using the low power multiplier proposed in Section III.D. This paper shows how the 2-D image convolution can be virtually padded with zeros, partitioned, accelerated at the HW level, and that the resulting image will have better noise performance when using floating-point image data.

REFERENCES

[1] F.-C. Huang, S.-Y. Huang, J.-W. Ker, and Y.-C. Chen, "High-performance SIFT hardware accelerator for real-time image feature extraction," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 3, pp. 340–351, Mar. 2012.
[2] Y. Zhao, D. Wang, and L. Wang, "Convolution accelerator designs using fast algorithms," *Algorithms*, vol. 12, no. 5, p. 112, May 2019.
[3] G. Licciardo, C. Cappetta, and L. Di Benedetto, "Design of a convolutional two-dimensional filter in FPGA for image processing applications," *Computers*, vol. 6, no. 2, p. 19, May 2017.

[4] J.-M. Müller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefévre, G. Melquiond, N. Revol, and S. Torres, *Handbook of Floating-Point Arithmetic*. Basel, Switzerland: Birkhäuser Science, 2018.

[5] C. Manders, F. Farbiz, and S. Mann, "A compression method for arbitrary precision floating-point images," in *Proc. IEEE Int. Conf. Image Process.*, Oct. 2007, pp. IV-165–IV-168.

[6] M. Seznec, N. Gac, A. Ferrari, and F. Orieux, "A study on convolution using half-precision floating-point numbers on GPU for radio astronomy deconvolution," in *Proc. IEEE Int. Workshop Signal Process. Syst. (SiPS)*, Oct. 2018, pp. 170–175.

[7] J. Li, X. Zhou, B. Wang, H. Shen, and F. Ran, "Design of efficient floating-point convolution module for embedded system," *Electronics*, vol. 10, no. 4, p. 467, Feb. 2021.

[8] T. Geng, L. Waeijen, M. Peemen, H. Corporaal, and Y. He, "MacSim: A MAC-enabled high-performance low-power SIMD architecture," in *Proc. Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2016, pp. 160–167.

[9] O. H. Moustafa and S. M. Ismail, "FPGA-based floating point fractional order image edge detection," in *Proc. 15th Int. Comput. Eng. Conf. (ICENCO)*, Dec. 2019, pp. 91–94.

[10] A. A. Del Barrio, N. Bagherzadeh, and R. Hermida, "Ultra-low-power adder stage design for exascale floating point units," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 3s, pp. 1–24, Mar. 2014.

[11] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, Jun. 2013, pp. 24–35.

[12] R. Mudassir, M. Anis, and J. Jaffari, "Switching activity reduction in low power booth multiplier," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2008, pp. 3306–3309.

[13] T. Ahn and K. Choi, "Dynamic operand interchange for low power," *Electron. Lett.*, vol. 33, no. 25, pp. 2118–2120, Dec. 1997.

[14] J.-F. Lin, C.-Y. Chan, and S.-W. Yu, "Novel low voltage and low power array multiplier design for IoT applications," *Electronics*, vol. 8, no. 12, p. 1429, Nov. 2019.

[15] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis, and D. Nikolos, "Low-power leading-zero counting and anticipation logic for high-speed floating point units," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 7, pp. 837–850, Jul. 2008.

[16] A. Deepa and C. N. Marimuthu, "Design of a high speed vedic multiplier and square architecture based on Yavadunam Sutra," *Sādhanā*, vol. 44, no. 9, pp. 1–10, Aug. 2019.

[17] D. G. Bailey, "Image processing," in *Design for Embedded Image Processing on FPGAs*, vol. 8. Hoboken, NJ, USA: Wiley, 2011, ch. 1, sec. 1, pp. 14–17.

[18] F. Conti, P. D. Schiavone, and L. Benini, "XNOR neural engine: A hardware accelerator IP for 21.6-fJ/op binary neural network inference," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2940–2951, Nov. 2018.

AJAY KUMAR SAHU received the B.E. degree in electronics and telecommunication engineering from MIT, Manipal, where he is currently pursuing the M.Tech. degree in microelectronics. He has published one paper during his B.E. His research interests include digital logic design, processor architecture, and ASIC design.

VISHNUMURTHY KEDLAYA K. received the B.E. degree in electronics and communication engineering and the M.Tech. degree in digital electronics and advanced communication engineering. He is currently working as an Assistant Professor—Selection Grade with the Department of Electronics and Communication Engineering, Manipal Institute of Technology, Manipal. His research interests include digital systems and application.

SUBRAMANYA G. NAYAK (Member, IEEE) received the B.E. degree in electronics and communication engineering, the M.Tech. degree in biomedical engineering, and the Ph.D. degree in electrical and electronics engineering. He is currently working as a Professor with the Department of Electronics and Communication Engineering, Manipal Institute of Technology, Manipal. His research interests include digital systems and processor architecture design and applications.