IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received January 14, 2022, accepted January 26, 2022, date of publication February 7, 2022, date of current version February 16, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3149505

Hardware Acceleration of a Generalized Fast 2-D Convolution Method for Deep Neural Networks

ANAAM ANSARI, (Graduate Student Member, IEEE), AND TOKUNBO OGUNFUNMI, (Senior Member, IEEE)
Department of Electrical and Computer Engineering, Santa Clara University, Santa Clara, CA 95053, USA
Corresponding author: Tokunbo Ogunfunmi (togunfunmi@scu.edu)

This work was supported by the School of Engineering, Santa Clara University, Dean Excellence in Research Award.

ABSTRACT The hardware acceleration of Deep Neural Networks (DNN) is a highly effective and viable solution for running them on mobile devices. The power of DNNs is now available at the edge in a compact and power-efficient form factor with the aid of hardware acceleration. In this paper, we introduce an architecture that uses a generalized method called Single Partial Product 2-Dimensional Convolution (SPP2D Convolution) which calculates a 2-D convolution in a fast and expedient manner. We demonstrate that the SPP2D architecture prevents the re-fetching of input weights for the calculation of partial products, and it can calculate the output of any input size and kernel with low latency and high throughput compared to other popular techniques. SPP2D based architecture can reduce the memory access and execution time related to input reuse by at least **three** times in comparison with the work done in Ardakani *et al.* (2018) and approximately **nine** times that of the standard sliding window approach. We have implemented the generalized SPP2D architecture on the Xilinx KC705 Kintex-7 evaluation board to illustrate that the new SPP2D algorithm is well-suited for the hardware acceleration of DNNs. We implemented LeNet-5 and VGGNet-16 using the SPP2D architecture. We demonstrate that the SPP2D based LeNet-5 has a high throughput of **5 GOP/s** and **14.8 GOP/s/W** and **42 GOP/s/W** for the convolution operation using the SPP2D IP. Our LeNet-5 design achieves a similar throughput to Zhou and Jiang (2015) however using **3.3**× fewer DSPs and an even smaller memory and lookup table (LUT) footprint. The SPP2D based VGGNet-16 network has a latency of **91.3 ms** which is **79**%, **97**%, **17**% and **95**% less than contemporary designs respectively, while running at a low power of **298 mW** which is similar to the power level of these designs. The total processing time of our design with a parallelism factor of nine is **3.93** secs and it is **70**% less than that in Ardakani *et al.* (2018) and **24**% less than that in Panchbhaiyye and Ogunfunmi (2021). The SPP2D based LeNet-5 and VGGNet-16 accelerators provide a low-latency design with reduced memory access thus leading to a low-power design. As a result, SPP2D convolution is very well suited for hardware acceleration of DNNs.

INDEX TERMS SPP2D, convolution, convolutional neural networks (CNN), deep neural network (DNN), hardware accelerator, processing engine, LeNet-5 , VGGNet-16 .

I. INTRODUCTION

In recent years Deep Neural Networks (DNN) have become ubiquitous and therefore there are many variants that exist to accomplish a myriad of applications. These tasks range from object classification to natural language processing (NLP). The robustness of DNNs to distortions and simple geometric transformations makes them highly effective for processing images [1].

DNNs are becoming increasingly large and complex. Therefore, the hardware architectures that implement them need to be commensurate with their growth while delivering the cutting-edge throughput with minimum latency. The demand for low power and low memory access has also become more challenging with their growth. Hardware acceleration of DNNs on FPGAs and ASICs are more energy efficient and portable than GPU implementations (at least for inference) [2].

Hardware imposes tremendous limitations on the design of DNNs owing to its high computational complexity.

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

Low latency concerns and the need for a high memory-access bandwidth also pose challenges in CNN accelerator designs. It is challenging to reap the complete benefits of logic resources within the architecture even after implementing pipe-lining and time-efficient designs leading to a sophisticated architecture being under-utilized. Implementing DNN architectures involves, meeting the peak performance in the fa ce of the aforementioned limitations. Therefore, the hardware implementation of DNN calls for a huge design exploration.

II. RELATED WORK

There are hardware acceleration solutions for GPUs [3],CPUs [4]–[6] FPGAs [7]–[15] and ASIC platforms [16]–[19]. Recently there has been a migration of inference applications to FPGA devices. FPGAs provide a flexible sandbox for development and deployment of DNNs for inference. This provides an excellent platform for testing rapidly evolving DNNs [20]. FPGAs are also cost effective in terms of energy savings. They deliver better performance for the same amount of energy spent to perform DNN computations compared with GPUs [21]. For all these reasons, FPGA hardware acceleration is appropriate for inference. The CNN accelerator described in [7] is a combination of hardware and software design processes. In [17] the authors explored the design space of the available loop blocking parameters using the roof-line model. They determined the optimum loop unrolling parameters based on design space exploration. The loop unrolling factors are variable in nature, and the authors select a fixed unroll loop parameter that tolerates a degradation of 5% in throughput and performance. In [22] we focused on increasing the computational throughput using a novel processing unit design called nested processing element (NPE) using the variable unroll parameters calculated in [17] to achieve a throughput enhancement of 94%. Eyeriss [18] is a design in which the authors limit external memory accesses using a memory hierarchy. For this purpose, they used local scratch pads and global buffers. They achieved energy efficiency by adopting a reuse scheme known as row-stationary. Energy efficiency is the result of reducing external data access.

There are many designs that approach the problem from both the software and hardware fronts. Studies such [23] have investigated the effect of quantization on the accuracy of models. Optimizations such as quantization minimize the storage of input feature maps and weights so that they can fit on the on-chip memory [24]–[26]. Our work in [27] explored the effect of fixed-point quantization on the accuracy of DNNs. The pruning and quantization of weights results in sparse networks. Sparse networks generate associative data that is inconvenient to handle. Their increased complexity makes it difficult to use traditional methods such as matrix multiplication and calls for tailored solutions. Therefore, there is a need for simple solutions that can exploit existing dense computation techniques. All of the techniques described above aim to reduce the number of memory transfers in creative ways.

A commonly used operation in Deep Neural Networks is 2-D convolution. The number of calculations required to implement a deep neural network is of the order of millions [28], [29]. Many solutions such as [30]–[33] exploit sparseness to overcome this impediment. Works such as [34]–[41] reduce the inputs and weights to binary values leading to a smaller memory footprint and less memory transfer. Works such as [31], [42] use Winograd's filter technique to calculate convolutions [16]–[19] in a fast manner. A fast Fourier transform can also be used to calculate the convolutions in an expedient manner [43].

The convolution operation is at the heart of deep neural network and therefore research in alternate ways to implement this operation is relevant and opportune owing to the rising trend of deep learning. There are several methods for performing the convolution operation. Convolutions can be implemented using matrix multiplication and vector multiplication [2]. However, convolution using matrix multiplication introduces redundant operations by converting the input matrix into a Toeplitz matrix and convolutions using vector multiplication take a long time if done serially or require large memory transfers if the operation is carried out in parallel. Solutions such as [18] propose the reuse of input weights to avoid fetching them from the off-chip memory. Reuse is an important mechanism in implementing DNNs as the traditional sliding window method without any reuse results in fetching some input pixels and weights multiple times. In [16], the authors presented an architecture that aimed to reduce the latency of networks by implementing the reuse of input pixels. In [44] the authors designed a convolution method that removed redundant multiplication operations from both 1-D and 2-D convolution computations at the cost of increased addition operations. In [45], we performed a redundancy analysis of the weights in order to avoid repeatedly sending similar data from off-chip to on-chip. In [46], we implemented an architecture to perform SPP2D convolution. The design was able to perform 2D convolution of an input of size 5×5 with kernel 3×3 in approximately 9 clock cycles. Although, being fast the design had several limitations. We addressed those limitations to deliver a complete SPP2D convolution-based CNN architecture in this work.

In this paper we present an SPP2D based architecture on Xilinx's KC705 Kintex 7 evaluation board. We implemented the LeNet-5 and VGGNet-16 networks using the SPP2D architecture. This design can implement a 2-D convolution of an input of any size with a kernel of any size. Our architecture design addresses the two prominent concerns in this area of research - computational complexity and latency and power consumption due to memory movement. In this work, we contribute to the following:

- We have designed an architecture based on SPP2D convolution which has a generalized and improved processing engine that can compute the outputs of a 2-D convolutions by avoiding the re-fetching of any input for the calculation of any partial product.

- The SPP2D architecture supports the convolution of an input of any size and a kernel of any size which leads to a network-agnostic architecture design.
- We implemented the LeNet-5 and VGGNet-16 network using SPP2D architecture. We implemented VGGNet-16 using a parallelism factor of 1 and 9. The LeNet-5 design handled the same workload as [47] with a smaller resource footprint and higher throughput than in [48], in which the authors presented a design with a reduced number of parameters. The SPP2D based VGGNet-16 design achieved a low total latency of 91.3 ms which is lower than [11]–[13], [16], [18], [19] at 298 mW for a parallelism factor of 9.

This promising concept helps resolve latency issues previously experienced and observed in other popular designs such as [16], because it avoids the re-fetching of input pixels for the calculation of partial products. This results in the low overall number of operations required to compute a frame.

The remainder of this paper is organized as follows: In Section III we explain the concept of the Single Partial Product 2-D Convolution and its analysis in Section IV. We explain the hardware architecture in Section V and the implementation process of LeNet-5 and VGGNet-16 in Section VI. This is followed by the results and conclusion in Sections VII and SectionVIII.

III. SINGLE PARTIAL PRODUCTION 2-D CONVOLUTION

A. 2-D CONVOLUTION OPERATION

The 2-D convolution operation can be described using a sliding window operation. Consider an input I of size $N \times N$ and a kernel W of size $k \times k$. We can describe the output O of size $M \times M$ where $M = N - k$ as the output of the convolution of input I and kernel k with a stride of 1 (Figure 1). There is a lot of information that can be gleaned about the movement of data during the sliding window operation. By analyzing, the interaction of the input pixels' with the kernel elements, we can understand how many times an input is required for calculating the output. This has broad implications in terms of memory requirements and dealing with the burden of data movement in the case of the CNN architecture design process.

B. FREQUENCY OF REUSE

The convolution operation involves the reuse of the input pixels in calculation of partial products. We define the number of times an input pixel is required to calculate the output as the frequency of reuse of that input pixel. The reuse of the input pixels in calculating the output pixels influences the design of memory traffic infrastructure in hardware for acceleration. We denote the frequency of reuse as **N(x)** where **x** denotes the frequency itself. **Note: Here N(x) is not the same as N which is the number of input pixels**. The frequency of reuse for all the pixels in a 5×5 input convolved by a kernel of size 3×3 is shown in Figure 2a. There are some key takeaways such as the maximum frequency of reuse is **N(9)** and the minimum frequency of reuse is **N(1)** for an input of 5×5 and

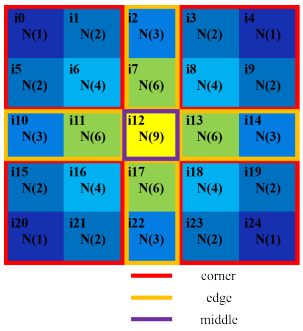
```
{All Rows of the Output}
for (x = 0; x < M; x++) do
  {All Columns of the Output}
  for (y = 0; y < M; y++) do
    {kernel rows}
    for (i = 0; i < k; i++) do
      {kernel columns}
      for (j = 0; i < k; j++) do
        {Accumulation of output pixels}
        O_{x,y} += I_{x+i,y+j} * W_{i,j}
      end for
    end for
  end for
end for
```

FIGURE 1. 2-D convolution: Sliding window operation.

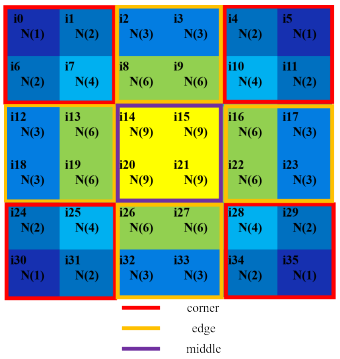
kernel of size 3×3. The frequency of reuse are arranged in a pattern across in the input as demonstrated in all the examples in Figure 2. The highest frequency **N(9)** is in the center of the input - $i12$ in Figure 2a. The frequency of reuse for an input of 6×6 and kernel of size 3×3 is shown in Figure 2b. The maximum frequency is **N(9)** and the minimum frequency is **N(1)**. The maximum frequency of reuse pixels lie in the "**middle**" of the input and all the other lie on the boundary. Consider two more inputs of size - 9×9 and 10×10 and kernel of size 5×5 (Figure 2c and Figure 2d). The maximum frequency of reuse is **N(25)** and the minimum frequency of reuse is **N(1)**. Some conclusions can be drawn from all the above examples:

- This pattern of frequency of reuse is universal for a kernel of size k and an input of size greater than or equal to $(2 * k - 1) \times (2 * k - 1)$ with a **stride of 1**.
- The input pixels can be classified into three major regions- "**middle**", "**edge**"and "**corner**".
- The minimum frequency of reuse is **N(1)** and the maximum frequency of reuse is **N($\mathbf{k^2}$)** for a kernel of size $k \times k$ and an input of size greater than or equal to $(2 * k - 1) \times (2 * k - 1)$ (Figure 3).
- The maximum frequency always lies in the "**middle**"
- All other frequencies are present at the periphery of the input spanning $(k - 1)$ pixels along the entire boundary where k is the kernel dimension.
- The "**corner**"pixels are fixed in number. As the size of the input increases, it has no effect on the number of these pixels. The "**edge**"and "**middle**"pixels grow with the increase in input size (Figure 2b and Figure 2d). For example, in Table 1, we can see how the "**edge**"and "**corner**"pixels grow in number with the increase in size of input and the "**corner**"pixels remain fixed.

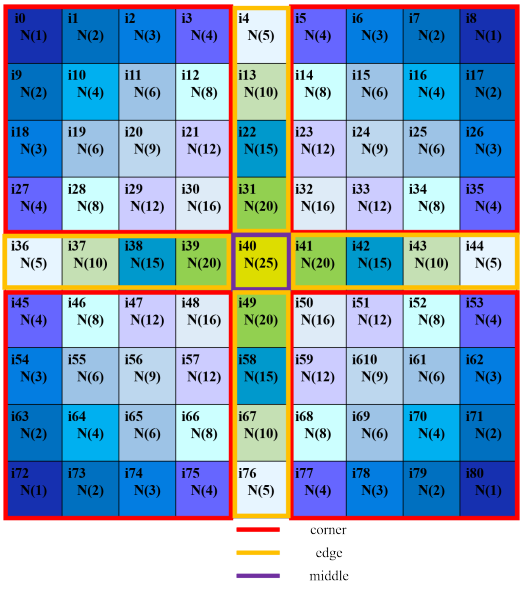
Figure 3 shows how the number of pixels in each region can be determined and Table 2 summarizes the number of input pixels that correspond to each region ("**middle**", "**edge**" and "**corner**") for convolution between an input of any size with a kernel of any size. The size of the "**middle**"



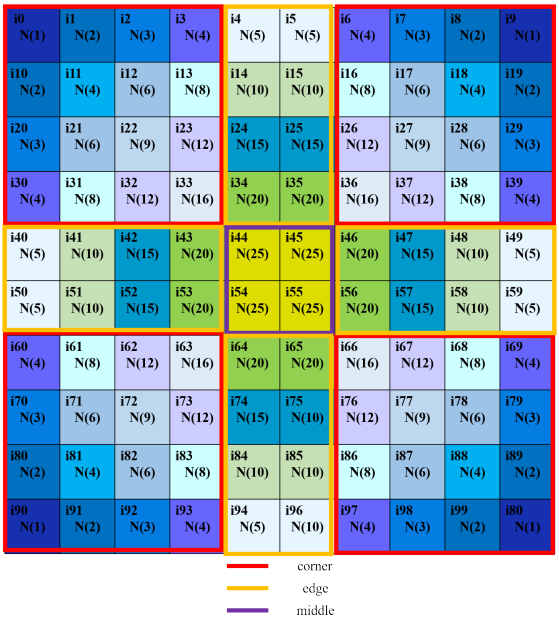
(a) Frequency of reuse- input 5×5 and kernel 3×3



(b) Frequency of reuse- input 6×6 and kernel 3×3.



(c) Frequency of reuse- input 9×9 and kernel 5×5..



(d) Frequency of reuse- input 10×10 and kernel 5×5..

FIGURE 2. Conventional sliding window convolution.

TABLE 1. Number of inputs with the frequency N(9), N(6), N(3), N(4), N(2), N(1) for various input sizes and a kernel of size 3 × 3.

	"middle"	"edge"	"edge"	"corner"	"corner"	"corner"
input size	N(9)	N(6)	N(3)	N(4)	N(1)	N(2)
5	1	4	4	4	4	8
6	4	8	8	4	4	8
7	9	12	12	4	4	8
10	36	24	24	4	4	8
14	100	40	40	4	4	8
28	576	96	96	4	4	8
56	2704	208	208	4	4	8
112	11664	432	432	4	4	8
224	48400	880	880	4	4	8

region is given by $(N - 2(k-1))^2$. The "**edge**" pixels are given by $(N - 2(k-1))(k-1)$ along each edge and there are four edges and the four sets of "**corner**"pixels are given as $(k-1)^2$, where N is the size of the input and k is the size of the kernel.

The work done in [16] deploys the reuse of input pixels to combat the refetching of input pixels for the calculation of

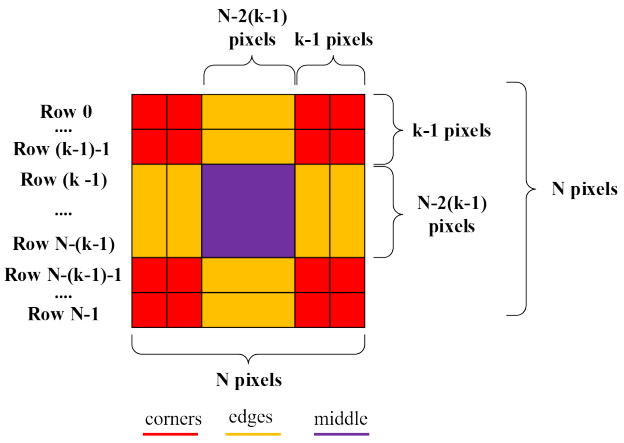


FIGURE 3. Number of input in classified regions - ("middle", "edge" and "corner").

partial products. In their design, they calculated the output by preserving the partial products calculated between two

TABLE 2. Number of input in based on region classification.

region	No. of input pixels
middle	$(N-2(k-1))^2$
edges	$4(N-2(k-1))(k-1$
corners	$4(k-1)^2$

consecutive outputs and only calculating the new partial products introduced due to the stride. They reused input pixels in their design, however there is still room for reuse. This paper also reported a large amount of latency as the size of the input increased. Based on our observations of the input pixels and their frequency of reuse, we can avoid completely re-fetching the input pixels. We designed an architecture that exploits this strategy to avoid the re-fetching of the input pixels, and extract the maximum use of the input pixel while it is in the on-chip memory or buffers. The strategy involves calculating all partial products that an input would generate while the input pixel is in the buffer or on-chip memory before it is discarded to the off-chip memory. The process of simultaneous generation of partial products is described in the following paragraph.

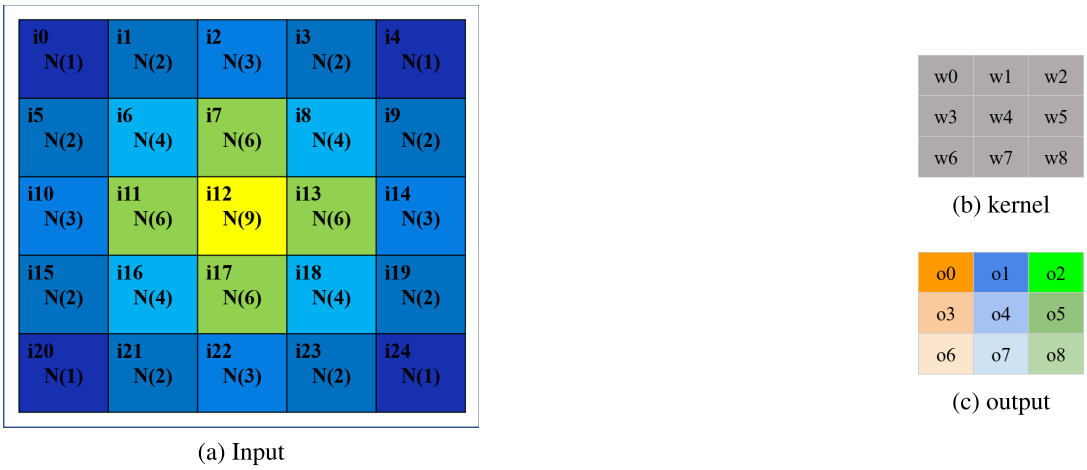
Following the example of an input of size 5×5 being convolved with a kernel of size 3×3 (Figure 4), we organize the input pixels in the descending order of their frequency of reuse and multiply them with the respective kernel weights necessary to produce their respective partial products as shown in Table 3. We can aggregate the partial products being generated to arrive at the output theoretically at $N \times N$ iterations where N is the size of the input. The partial product aggregation process is described in the Table 3. In Table 3 we arrive at the output in 25 aggregation iterations. There are a few design critical takeaways from Table 3. The first takeaway is that 9 weights are used with 9 multipliers and input pixel $i12$ occupies all these multipliers while all the other input pixels leave gaps or not occupy all the multipliers. The second takeaway from Table 3 is that there are complementary sets of inputs that can occupy all 9 multipliers, and they are highlighted with similar colors. The complementary sets are as follows: ($i7$, $i22$), ($i2$, $i17$), ($i11$, $i14$), ($i6$, $i19$, $i21$, $i24$), ($i1$, $i4$, $i16$, $i19$), ($i10$, $i13$), ($i5$, $i8$, $i20$, $i23$), ($i0$, $i3$, $i15$, $i18$). The complementary sets of inputs help engage all 9 multipliers. This gives us an opportunity to combine the input with complementary sets and parse them into 9 multipliers with weights. This is presented in Table 4. In this manner, we can engage all the 9 multipliers corresponding to the number of weight kernels. The combined process of simultaneous generation of partial products and combining the inputs to the complementary set helps us reduce the aggregation cycles from 25 to 9. The color scheme described in Table 4 represents the aggregation pattern. The accumulation of all the similar colors in the Table 4 provides the output. For example the accumulation of partial products- $w0i0$, $w1i1$, $w2i2$, $w3i5$, $w4i6$, $w5i7$, $w6i10$, $w7i11$ and $w8i12$ gives the output pixel $o0$ (Figure 4)

C. TYPES OF INPUT PIXELS

Examining the pattern of frequency of reuse for inputs of different sizes we can conclude that we can classify the pixels into 3 broad categories - "**middle**", "**edge**"and "**corner**".In addition to the broader classification shown in Figure 3, we divided the pixels into finer categories. This gives rise to the $(2k-1)^2$ category of pixels that is 25 types of pixels for an input of any size and kernel of size 3×3 and 81 types of pixels for an input of any size and kernel of size 5×5. Figure 5 depicts the categories of the input pixels $T0$ to $T24$. The pixel of type $T12$ lies at the center of the input. The types of input pixels are classified to facilitate the organization of the input data for hardware processing. Pixel-type classification enables us to have a second level of discrimination of pixels in addition to the frequency of reuse. As the size of the input increases (and the kernel is 3×3) the number of pixels of $T12$ with frequency **N**(**9**) also increases. This type of pixel needs to be combined with all 9 weights. Therefore, the $T12$ type pixel can be classified as the "**middle**". A large chunk of the input must be multiplied by all the 9 weights. The second major categories of input pixels are - $T2$, $T7$ which are **N**(**3**) and **N**(**6**) pixels respectively, $T17$ and $T22$ which are **N**(**6**) and **N**(**3**) respectively, $T10$ and $T11$ which are **N**(**3**) and **N**(**6**) pixels respectively and $T13$ and $T14$ which are **N**(**6**) and **N**(**3**) pixels respectively. These pixels can be classified as the "**edge**". The other pixels that belong to the "**corner**"are $T0$, $T4$, $T20$ and $T24$. They have frequency **N**(**1**). Pixels $T6$, $T8$, $T16$ and $T18$ have frequency **N**(**4**) and pixels having frequency **N**(**2**) are $T1$, $T3$, $T5$, $T9$, $T15$, $T19$, $T21$ and $T23$. The complementary types of pixels are $T12$, {$T2$, $T17$}, {$T7$, $T22$}, {$T10$, $T13$}, {$T11$, $T14$}, {$T6$, $T9$, $T21$, $T24$}, {$T5$, $T8$, $T20$, $T23$}, {$T1$, $T4$, $T16$, $T19$}, {$T0$, $T3$, $T15$, $T18$}. The number of pixel types changes with the kernel size. As stated earlier, the number of pixel types depends on the size of the kernel k and is given by $(2k-1)^2$.

D. SPP2D CONVOLUTION OPERATION

We can explain SPP2D Convolution using an example. Consider an input of size 5×5 and a kernel of size 3×3 described in Figures 6a and 6b. The convolution operation is done with a stride of 1. We highlight the input pixels using a color scheme that denotes the frequency of reuse. The kernel weights are used in an unfolded manner, and they can be aligned in the form of a vector multiplier. The input pixels are combined with the kernel weight in the optimized order as follows:{$i0$, $i3$, $i15$, $i18$},{$i1$, $i4$, $i16$, $i19$}, {$i2$, $i17$},{$i5$, $i8$, $i20$, $i23$}, {$i6$, $i9$, $i21$, $i24$}, {$i7$, $i22$}, {$i10$, $i13$}, {$i11$, $i14$}, $i12$. This is illustrated in Figure 7a. Once the partial products are generated they must be sorted into their various output pixels. The aggregation order is given by the color scheme described in Figure 7b. For example, The output $o0$ is the aggregation of **sum{**(3×1), (8×1), (7×3), (2×3), (2×2), (5×2), (5×1), (8×1), (6×2)**}** is 77 as seen in Figure 6c. The final output shown in Figure 6c is the result of the convolution of the input matrix (Figure 6a) and kernel (Figure 6b).



(a) Input

(b) kernel

(c) output

FIGURE 4. Convolution example of an input of size 5 × 5 and kernel of size 3 × 3.

TABLE 3. Input stream based on frequency of reuse.

aggregate iterations	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
	N(9)	N(6)	N(6)	N(6)	N(6)	N(3)	N3)	N(3)	N(3)	N(4)	N(4)	N(4)	N(4)	N(2)	N(2)	N(2)	N(2)	N(2)	N(2)	N(2)	N(2)	N(1)	N(1)	N(1)	N(1)
weights	i12	i7	i11	i17	i13	i2	i14	i22	i10	i6	i8	i18	i16	i1	i13	i5	i9	i15	i19	i21	i23	i0	i4	i20	i24
w0	w0i12	w0i7	w0i11			w0i2			w0i10	w0i6				w0i1		w0i5						w0i0			
w1	w1i12	w1i7	w1i11		w1i13	w1i2				w1i6	w1i8			w1i1	w1i3										
w2	w2i12	w2i7			w2i13	w2i2	w2i14				w2i8				w2i3		w2i9						w2i4		
w3	w3i12	w3i7	w3i11	w3i17					w3i10	w3i6			w3i16			w3i5		w3i15							
w4	w4i12	w4i7	w4i11	w4i17	w4i13					w4i6	w4i8	w4i18	w4i16												
w5	w5i12	w5i7		w5i17	w5i13		w5i14				w5i8	w5i18					w5i9		w5i19						
w6	w6i12		w6i11	w6i17				w6i22	w6i10				w6i16					w6i15		w6i21			w6i20	
w7	w7i12		w7i11	w7i17	w7i13			w7i22				w7i18	w7i16						w8i19	w7i21	w7i23				
w8	w8i12			w8i17	w8i13		w8i14	w8i22				w8i18									w8i23			w8i24	

TABLE 4. Optimized input stream.

aggregate cycles	1	2	3	4	5	6	7	8	9
weights	i0 i3 i15 i18	i1 i4 i16 i19	i2 i17	i5 i8 i20 i23	i6 i9 i21 i24	i7 i22	i10 i13	i11 i14	i12
w0	w0i0	w0i1	w0i2	w0i5	w0i6	w0i7	w0i10	w0i11	w0i12
w1	w1i3	w1i1	w1i2	w1i8	w1i6	w1i7	w1i13	w1i11	w1i12
w2	w2i3	w2i4	w2i2	w2i8	w2i9	w2i7	w2i13	w2i14	w2i12
w3	w3i15	w3i16	w3i17	w3i5	w3i6	w3i7	w3i10	w3ii11	w3i12
w4	w4i18	w4i16	w4i17	w4i8	w4i6	w4i7	w4i13	w4i11	w4i12
w5	w5i18	w5i19	w5i17	w5i8	w5i9	w5i7	w5i13	w5i14	w5i12
w6	w6i15	w6i16	w6i17	w6i20	w6i21	w6i22	w6i10	w6i11	w6i12
w7	w7i18	w7i16	w7i17	w7i23	w7i21	w7i22	w7i13	w7i11	w7i12
w8	w8i18	w8i19	w8i17	w8i23	w8i24	w8i22	w8i13	w8i14	w8i12

IV. ANALYSIS OF THE SPP2D

We can calculate the number of times the inputs will be operated by the sliding window technique by aggregating the frequency of reuse for each input pixel. This can be a strong estimate of the number of cycles required to perform the convolution operation using the sliding-window technique. This is shown in Table 5 - columns (a and d). The authors in [16] described an equation to calculate the number of clock cycles required to perform the convolution operation using their architecture. It is described in columns (b and e) in Table 5. Following the example of an input of size 5 × 5 being convolved with a kernel of size 3 × 3 we can analyze the number of clock cycles required to complete the SPP2D convolution based on the size of the kernel. We obtained the clock cycle estimate using the following formula: No. of clock cycles for SPP2D = No of inputs with **N(9)** + $\frac{1}{2}$×(No. of inputs with **N(6)**+ No of inputs with **N(3)**) + $\frac{1}{4}$× (No. of inputs with **N(4)** + No of inputs with **N(2)** + No of inputs with **N(1)**). We break down equation 1 into its constituent literals (equations 2,3,4). The number of inputs with frequency of reuse **N(9)** ("**middle**"pixels) is given in Equation 2. In Equation 3,

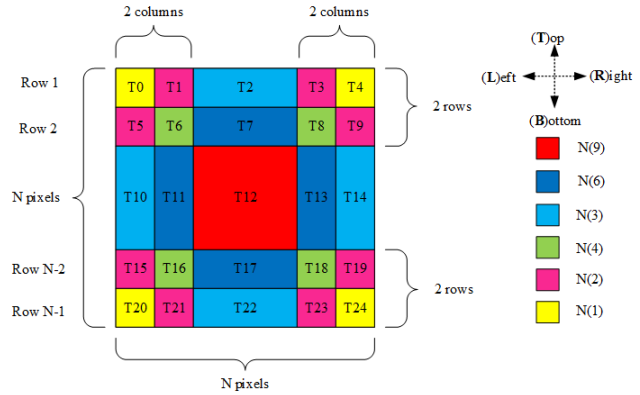
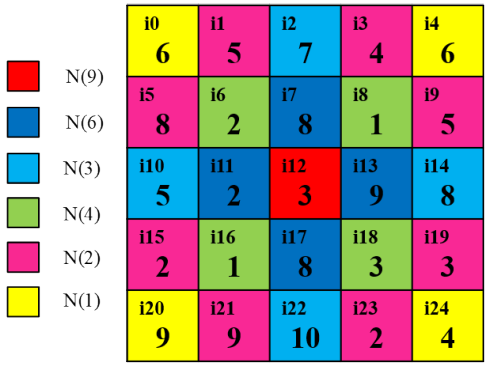
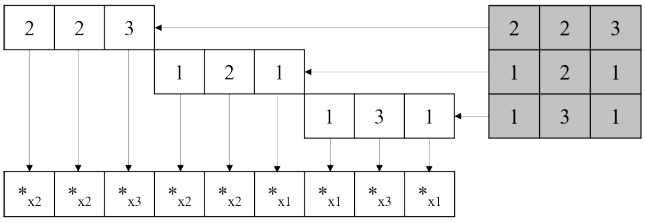


FIGURE 5. Conventional sliding window convolution.



(a) 5 × 5 input example



(b) The kernel is unfolded into an array of multipliers

77	75	93
69	68	82
81	98	85

(c) The computed output

FIGURE 6. The example input and the kernel.

we use the factor $\frac{1}{2}$ since inputs with frequency **N(6)** and **N(3)** (“**edge**”pixels) are used together therefore, we need to divide the sum of them by 2 as we use them together. In Equation 4, we use the factor $\frac{1}{4}$ since inputs having frequency $N(4)$, $N(2)$ and $N(1)$ (“**corner**”pixles) are used together and therefore we need to divide their number by 4 as we use them 4 at a time (N(2) is used 2× in a corner arrangement). Finally, the number of clock cycles for the SPP2D convolution is given by summing Equations 2, 3 and 4. The total number of clock

i18+i3+i15+i0	6	4	4	2	3	3	2	3	3
i16+i1+i19+i4	5	5	6	1	1	3	1	1	3
i17 +i2	7	7	7	8	8	8	8	8	8
i8+i5 +i23+i20	8	1	1	8	1	1	9	2	2
i6+i19+i21+i24	2	2	5	2	2	5	9	9	4
i7 +i22	8	8	8	8	8	8	10	10	10
i13 + i10	5	9	9	5	9	9	5	9	9
i11 +i 14	2	2	8	2	2	8	2	2	8
i12	3	3	3	3	3	3	3	3	3

(a) The partial products obtained by using the optimized input stream

i0+i3+i15+I18	6	4	4	2	3	3	2	3	3
i1+i4+i16+i19	5	5	6	1	1	3	1	1	3
i2 +i17	7	7	7	8	8	8	8	8	8
i5+i8 +i20+i23	8	1	1	8	1	1	9	2	2
i6+i19+i21+i24	2	2	5	2	2	5	9	9	4
i7 +i22	8	8	8	8	8	8	10	10	10
i10 + i13	5	9	9	5	9	9	5	9	9
i11 +i14	2	2	8	2	2	8	2	2	8
i12	3	3	3	3	3	3	3	3	3

(b) Sort the partial products generated into their respective outputs

FIGURE 7. The partial products produced and sorted into outputs.

cycles is given by Equation 5. We can calculate the number of clock cycles for an SPP2D convolution of input of any size with a kernel of any size and a stride of 1 using Equation 5. We can compare the number of clock cycles of the SPP2D convolution using the sliding window convolution technique and the convolution operation in the architecture described in [16] in Table 5. We have demonstrated that the number of clock cycles it takes to perform convolution on 5 × 5 with 3 × 3 takes 9 clock cycles to generate an output of size 3 × 3 (column c in Table 5) and it takes 25 clock cycles if the output is of size 5 × 5 (column f in Table 5). This can be scaled for inputs with higher dimensions such as the input dimensions of the layers of VGGNet-16 network. Columns a, b, c indicate the number of clock cycles required to perform a convolution operation on the input of a given dimension using the sliding window, [16], and SPP2D convolution **(w/o padding)** and columns d, e, f indicate the number of clock cycles required by the sliding window, [16], and SPP2D convolution **(with padding)**. From Table 5 we see that the SPP2D convolution requires approximately 3× (columns (b/c), (e/f)) less clock cycles than [16] and 9× (columns (a/c), (d/f)) less clock cycles than the sliding window technique for both operation types - w/o padding and with padding.

$$\begin{aligned}&\text{SPP2D Cycles}\\&= \text{No of inputs with } \mathbf{N(9)}\\&\quad + \frac{1}{2}(\text{No. of inputs with } \mathbf{N(6)} + \text{No of inputs with } \mathbf{N(3)})\\&\quad + \frac{1}{4}(\text{No. of inputs with } \mathbf{N(4)} + \text{No of inputs with } \mathbf{N(2)}\\&\quad + \text{No of inputs with } \mathbf{N(1)}) \qquad (1)\end{aligned}$$

Lets break down equation 1 into its constituent literals

$$\begin{aligned}&\text{No of inputs with } \mathbf{N(9)}\\&\quad = (N - 2(k-1))^2 \qquad (2)\end{aligned}$$

TABLE 5. Theoretical number of clock cycles needed for various input sizes inputs.

	without padding							with padding						
	a	b	c	a-c	a/c	b-c	b/c	d	e	f	d-f	d/f	e-f	e/f
input size	Sliding Window (w/o padding)	[16] (w/o padding)	SPP2D Conv w/o padding					Sliding Window (w padding)	[16] (w padding)	SPP2D Conv (w padding)				
5	81	45	9	72	9.00	36	5.00	225	105	25	200	9.00	80	4.20
6	144	72	16	128	9.00	56	4.50	324	144	36	288	9.00	108	4.00
7	225	105	25	200	9.00	80	4.20	441	189	49	392	9.00	140	3.86
10	576	240	64	512	9.00	176	3.75	900	360	100	800	9.00	260	3.60
14	1296	504	144	1152	9.00	360	3.50	1764	672	196	1568	9.00	476	3.43
28	6084	2184	676	5408	9.00	1508	3.23	7056	2520	784	6272	9.00	1736	3.21
56	26244	9072	2916	23328	9.00	6156	3.11	28224	9744	3136	25088	9.00	6608	3.11
112	108900	36960	12100	96800	9.00	24860	3.05	112896	38304	12544	100352	9.00	25760	3.05
224	443556	149184	49284	394272	9.00	99900	3.03	451584	151872	50176	401408	9.00	101696	3.03

$$\frac{1}{2}(\text{No. of inputs with } \mathbf{N(6)} + \text{No of inputs with } \mathbf{N(3)}) = 2(N-2(k-1))(k-1) \quad (3)$$

$$\frac{1}{4}(\text{No. of inputs with } \mathbf{N(4)} + \text{No of inputs with } \mathbf{N(2)} + \text{No of inputs with } \mathbf{N(1)}) = (k-1)^2 \quad (4)$$

$$\begin{aligned}\text{SPPD clock cycles} &= (N-2(k-1))^2 \\ &\quad +2(N-2(k-1))(k-1)+(k-1)^2 \\ &= ((N-2(k-1)+(k-1))^2 \\ &= (N-k-3)^2 \end{aligned} \quad (5)$$

where N is the dimension of the input and $\mathbf{N(x)}$ is the frequency of reuse.

V. HARDWARE ARCHITECTURE

The architecture of the SPP2D engine is described in Figure 8. The design is implemented on Xilinx's Kintex KC705 Board. The architecture design has many features that are common to most computer architectures such as pipelining etc. However, it also has some custom components that are designed to facilitate SPP2D convolution. It has input and weight buffers to store part of the input and kernel weights on the board. They are read from external memory which is the off-chip memory. The **Input stream** block organizes the input pixels in the order which is optimal for processing them. It classifies inputs of all sizes into the types of pixels (which are $T0$ to $T24$ for a kernel of size 3×3 and $T0$ to $T81$ for a kernel of size 5×5).We have implemented VGGNet-16 and LeNet-5 which have kernels of size 3×3 and 5×5 respectively. The **Mux Array** takes the input pixels as they are delivered in the form of pixels of type- $T0$ to $T24$ and selects the complementary sets from them. The output of the **Mux Array** is fed into the **Multiplier** along with the weights from the **Weight Buffer**. The **Decoder** block sorts the output of the **Multiplier**.After sorting, the partial products are accumulated in the **Accumulator** block.After accumulation,they get stored in the **Output Buffer** and then eventually back to **External Memory**.

A. INPUT STREAM

The input stream block of the architecture controls the data-flow and parsing of the input pixels. The input was divided into rows and the order in which it needs to be

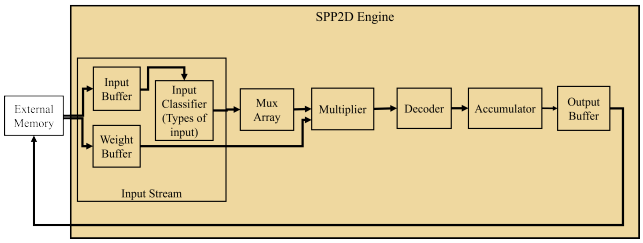
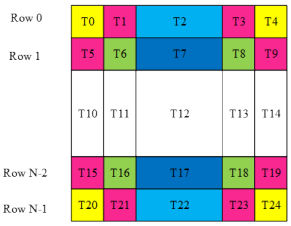


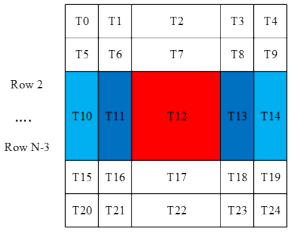
FIGURE 8. SPP2D engine architecture with pipelining.

conveyed to subsequent blocks is determined. To understand the order of conveyance we need to understand the properties of the rows of the input which is being processed by a kernel of size 3×3. There are N rows in an input spanning from 0 to $N-1$. As discussed in Section III-C input pixels form complementary type sets. The input rows can also have similar properties. The pixels in row 0 are complementary to the pixels in the row $N-2$ (Figure 9a). The pixels in the rows 1 and $N-1$ form complementary sets. Pixel sets $\{T2, T17\}$, $\{T0, T3, T15, T18\}$,$\{T1, T4, T16, T19\}$ from row 0 and $N-2$ are complementary sets. Similarly, rows 1 and $N-1$ are complementary rows and pixels - $\{T7, T22\}$, $\{T5, T8, T20, T23\}$ and $\{T6, T9, T21, T24\}$ are complementary pixels. Rows 2 to $N-3$ onwards we see that the only complementary pixel sets are $\{T10, T13\}$ and $\{T11, T14\}$ (Figure 6b). The pixel type $T12$ is a pixel which according to SPP2D convolution combines with all weights and occupies all multipliers and doesn't require a complementary pixel. The pixels are fetched using re-ordered rows seen in (Figure 9c). There are $(2(k-1))$ complementary rows for an input being convolved by a kernel of size k. The arrangement would be similar if we were to look an input convolved by a kernel of size 5×5. We would first calculate all the complementary rows and then the remaining rows.

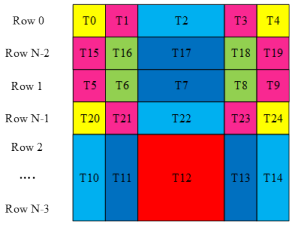
From among the 25 types of pixels that exist for an input being convolved by a kernel of size 3×3 there are pixels that are fixed in number regardless of the size of the input and there are pixels that grow and are variable with the size of input. The corner pixels remain fixed - $\{T0, T3, T15, T18\}$, $\{T1, T4, T16, T19\}$, $\{T5, T8, T20, T23\}$, $\{T6, T9, T21, T24\}$. These are the pixels that have frequency of reuse $\mathbf{N(4)}$, $\mathbf{N(2)}$, $\mathbf{N(1)}$ in this example. The remaining pixels can vary with the size of the input and have frequencies of



(a) Rows 0 and N-2 are complementary and 1 and N-1 are complementary



(b) Rows 2 to N-3 are the majority rows in the input that are sent after 1,2,N-2,N-1



(c) The rows are read in the following order.

FIGURE 9. Input stream for a kernel of size 3 × 3.

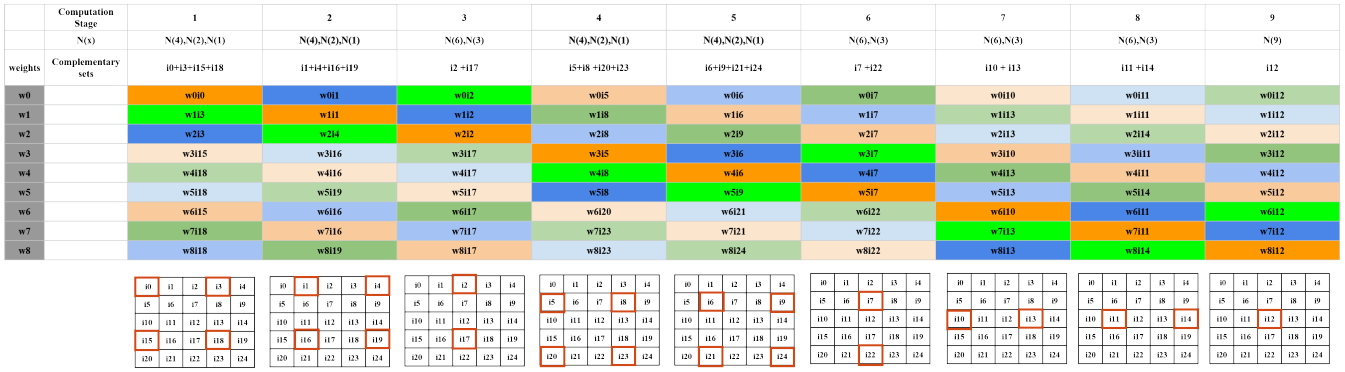
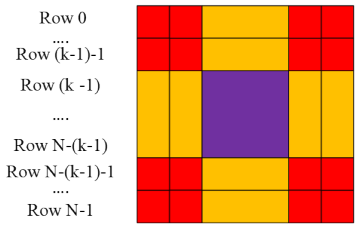
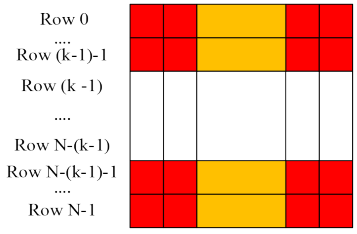


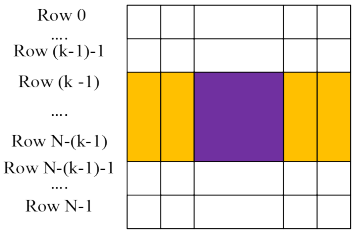
FIGURE 10. Input parsing order for SPP2D convolution.



(a) Rows 0 and N-2 are complementary



(b) Rows and 1 and N-1 are complementary

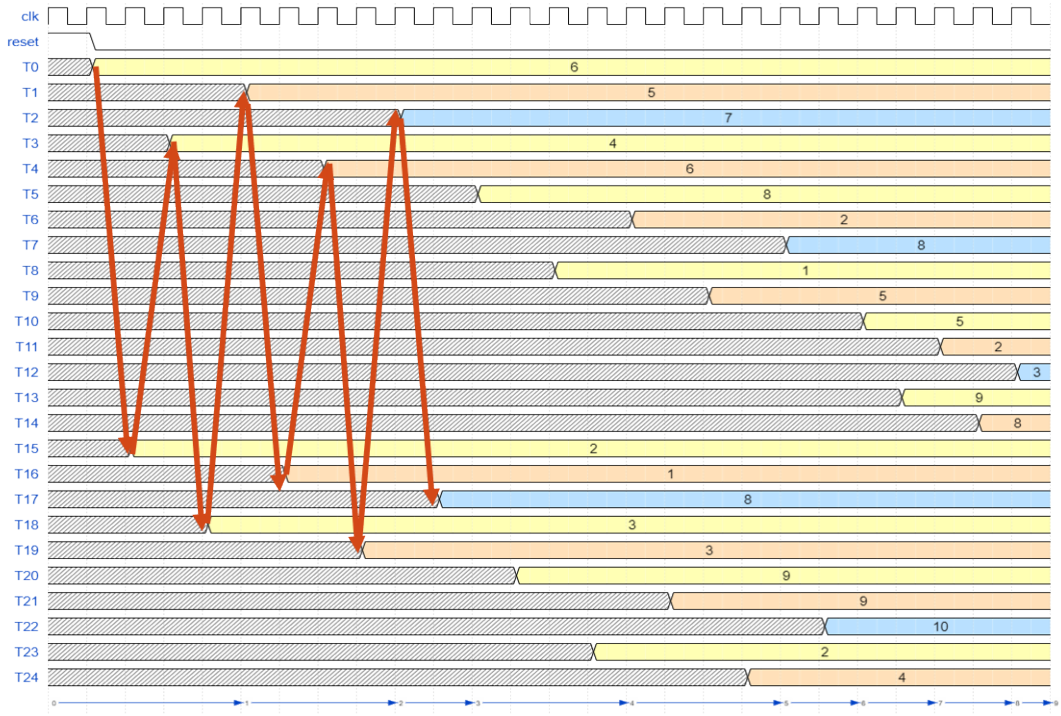


(c) The rows from 2 to N-3 are the majority rows in the input that are sent after 0,1,N-2,N-1

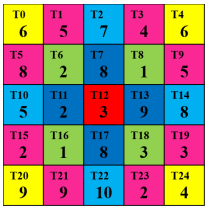
FIGURE 11. General input stream for a kernel of any size.

N(3), **N(6)** and **N(9)**. The input pixels have a property of being fixed or variable. The strategy for parsing the input pixels is based on the property of fixed or variable input pixels. We align the inputs ensuring that the rows $0, N-2, 1, N-1$ are processed first making sure that the fixed number of pixel are processed first then the variable number of pixels. Once we process rows $0, N-2, 1, N-1$, we can process rows 2 to $N-3$. The input parsing scheme is illustrated in Figure 9c. Figure 10 shows the optimized input stream order for an input of 5×5 and kernel of 3×3. The order of the new optimized stream is based on using rows with fixed pixels first then the rows with variable type of pixels used last. The parsing scheme is similar for pixels within a row. The fixed pixels are used first followed by the variable pixels. It takes 9 compute stages to finally arrive at the output. The strategy of processing a fixed number of pixels followed by the variable number of pixels in the complementary rows can be adopted to process an input that is convolved with a kernel of any size. Consider an input of any size that is convolved with a kernel of any size (Figure 11a), the rows 0,1,$N-2$ and $N-1$ are processed first (Figure 11b) and then the rows 2 to $N-3$ (Figure 11c). For these rows, the fixed pixels are processed first followed by the variable pixels. Therefore, making the process of organizing input data generalized irrespective of the size of the input and the kernel with which it is being convolved.

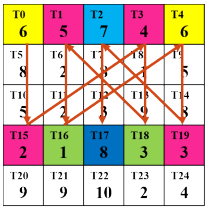
The Input stream organization works by classifying the pixels into types of pixels as shown in Figure 5. Given any size of input, the input stream is organized into 25 categories for a kernel of size 3×3. The input is classified into $(2k-1)^2$ for a kernel size of k. In Figure 12a, we see an example of a 5×5 input (Figure 12b) classified into the 25 types of pixels.



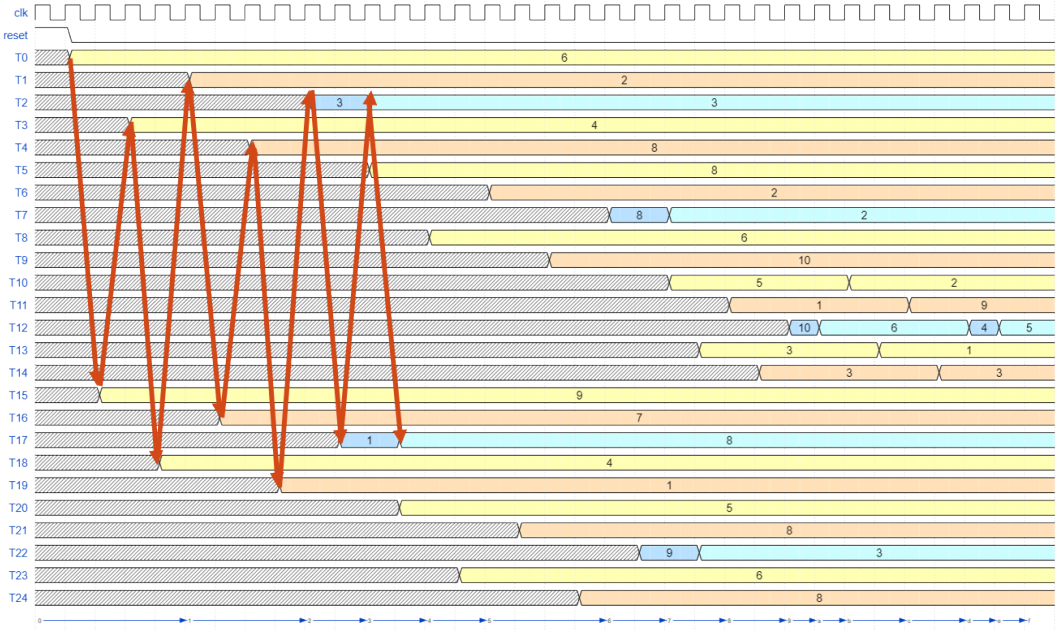
(a) Waveform describing the input stream of input 5×5



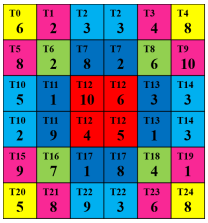
(b) input 5×5



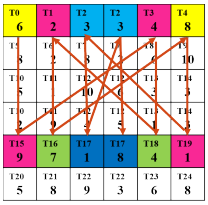
(c) Complementary rows 0 and N-2



(d) Waveform describing the input stream of input 6×6



(e) input 6×6



(f) Complementary rows 0 and N-2

FIGURE 12. Input stream.

The parsing of complementary rows 0 and $N-2$ of input of size 5×5 is described in Figure 12a and 12c. In Figure 12d we see the input of size 6×6 (Figure 12e) classified into the 25 types of pixels. The parsing of complementary rows 0 and $N-2$ of input of size 6×6 is described in Figure 12d and 12f. The input pixels are parsed in a generalized manner. The pixels stream corresponding to the compute stages defined in Figure 10 and it can be seen in Figure 12a. There are 9 compute stages for an input of size 5×5. As the input size increases, the compute stages will have echoes as types of pixels that are variable in nature grow. Looking at Figure 12d we can see that compute stages 1 and 2 are followed by compute stage 3 and 4. Compute stages 3 and 4 process complementary pixel set $\{T2, T17\}$. There are 2 pairs of pixels in this complementary set. This repeats for compute stage 7 and 8. Furthermore, rows 3 and 4 have pixels that have multiples of a type of pixel $T10$, $T11$, $T12$, $T13$ and $T14$. Therefore we can see the entire row echoing in computation.

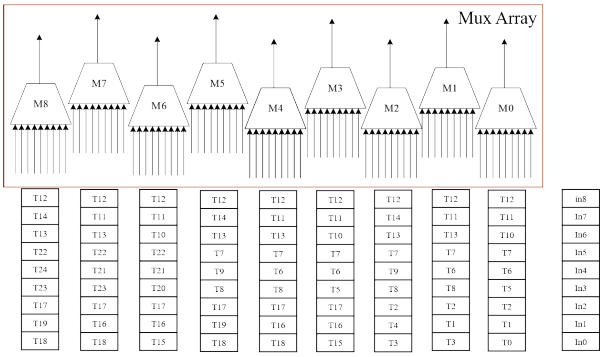


FIGURE 13. Mux array.

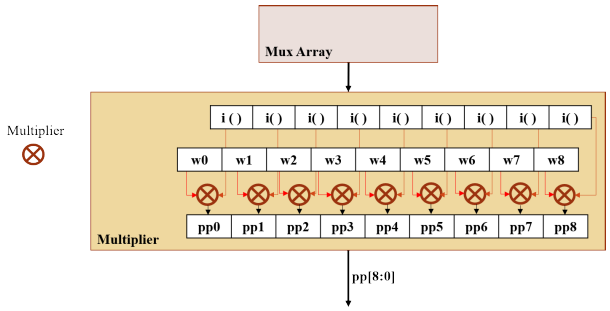


FIGURE 14. Multipliers.

The compute stages 8, 9, a and b are echoed as c, d, e, f. These stages process complementary pixel sets $\{T10, T13\}$, $\{T11, T14\}$ and pixel $T12$.

B. MUX ARRAY

The mux array processes its input only if all the inputs are present and available. The number of multiplexers in the mux array is given by k^2 where k is the kernel size. In general, we design the mux array based on the largest kernel size in the architecture. The mux array as an example of the input of 5×5 being convolved with a kernel of 3×3 has nine 9 to 1 muliplexers. The output of the mux array delivers the input pixels to the multipliers. The inputs to the all the muxes in the Mux Array are $in0$ to $in8$ as seen in Figure 13. The order in which the inputs to the mux are delivered is described using the optimized input stream described in Table 4 and also depicted in Figure 13.

Consider we have the pixels from complementary rows 0 and $N-2$ (Figures 9a) available at the output of the input stream block. These pixels are arranged at the $in0$ inputs of all the muxes of the Mux Array (Figure 13). The next set of complementary rows are arranged at the input $in1$ of all the muxes of the Mux Array. The entire arrangement of the Mux Array is based on the parsing strategy described in Section V-A. The design is such that we can send a select signal to the Mux Array which sends all the mux inputs in the ascending order.

C. MULITIPLIER

The multiplier block shown in Figure 14 is fairly straightforward. Its function is to accept the input pixels delivered by the mux array and multiply them by the weights. The partial products generated are combined in a bus for sorting and sent to the decoder block. Booth multipliers were used to perform the multiplication.

D. DECODER AND ACCUMULATOR

The decoder component of the architecture sorts the outputs generated by the multiplier. Once the partial products are generated they must be combined with relevant partial products for each output pixels. The sorting process is described in Figure 7b. The hardware used to sort the output of the convolution of an input of size 5×5 with a kernel of size 3×3 is described in Figure 15. It consists of nine 9 to 1 multiplexers that accept all nine partial products (pp[8:0]) generated using the multipliers. The select signal for each of these muxes are provided by the input stream block. These mux select codes are known due to the analysis done in Section III-D. Once the partial products are sorted into z_0 to z_8 they can be aggregated to obtain the final outputs o_0 to o_8.

If the input size increases, the output size will also increase and thus sorting the partial products becomes more complicated. Consider, an input that generates an output of pixels higher than 3×3. In Figure 16, we see that compute stages-1,2 and 3 which are also described in Figure 10 generate a pattern of partial product delivery to the designated output locations. We can see that compute stages 1, 2 lead to partial products that occupy the output corners. These positions are fixed for these compute stages. However, compute stage 3 can be variable and the output pixels will slide along the columns so that it can accommodate pixels generated with complementary sets $\{T2, T17\}$ (Figure 16). This is similar for compute stages 4,5 and 6 (Figure 17). Once we have dealt with rows 0, 1, $N-1$ and $N-2$, we process rows 2 to $N-3$. Rows 2 to $N-3$ engage compute stages 7, 8 and 9 (along with echoes). These rows contain pixels that can grow in number with the size of the input. For compute stage 7 and 8 (and their echoes) we populate output pixels on the edge (Figure 18). Therefore, output pixel for compute stages 7 and 8 slide row wise. For compute stage 9 (and its echoes) the partial products are sliding along both rows and columns. This pattern holds for any size input being processed by a kernel of any size. Consider an input of any size and kernel of any size the sort pattern for rows 0 and $N-2$ is given in Figure 19. The hardware accelerator for input of any size and kernel of any size produces (k^2) partial products given at a time (pp_0 to $pp_{(k^2-1)}$). We see that the fixed pixels generate partial products that will go to a fixed location and the variable partial products will contribute to variable locations. The sorting pattern for rows 1 and $N-1$ is given in Figure 20. The sorting pattern for rows 2 to $N-3$ is given in Figure 21.

VI. IMPLEMENTATION

We implemented LeNet-5 [49] and VGGNet-16 using SPP2D architecture on the Xilinx's Kintex KC705 development

TABLE 6. Throughput and performance for LeNet-5 implemented using SPP2D architecture.

	Input Size	Kernel Size	Dimension	Parameters	MACs	clock cycles	Execution Time (ms)	Performance	GOP/s	GOP/s/W #	GOP/s/W *
Conv1	28	5	1x28x28	500	288000	11520	5.76E-02	17361.11	5	14.8	42.7
Conv2	12	5	20x12x12	25000	1600000	64000	3.20E-1	3125	5	14.8	42.7
Conv1, Conv2					1888000	75520	3.78E-01	2648.31	5	14.8	42.7
F1	4	500	500x1	400000	400000	400000	2	500	0.2	0.59	1.71
F2	1	10	10x1	5000	5000	5000	2.5E-02	40000	0.2	0.59	1.71
F1, F2					405000	405000	2.03	493.83	0.2	0.59	1.71
Conv1, Conv2, F1, F2					2293000	480520	2.4	3142.13	0.954	2.83	8.16

Parallelism Factor = 1,* Parallelism Factor = 9

TABLE 7. Throughput and performance for VGGNet-16 implemented using SPP2D Architecture.

										Parallelism = 1		Parallelism = 9	
	Input Size	Channel Number	Kernel size	Kernel number	Bias	Parameters	Memory Access (MB)	Layer Latency (s)	Total Operations	Execution time (s)	ops/s	Execution time (s)	ops/s
Layer Conv 1_1	224	3	3	64	64	1792	1.52E+05	1.51E-03	8.67E+07	1.94E-01	4.46E+08	2.29E-02	3.78E+09
Layer Conv 1_2	224	64	3	64	64	36928	3.25E+06	3.21E-02	1.85E+09	4.14E+00	4.47E+08	4.89E-01	3.784E+09
Layer Conv 2_1	112	64	3	128	128	73856	8.77E+05	8.04E-03	9.25E+08	2.06E+00	4.48E+08	2.36E-01	3.91E+09
Layer Conv 2_2	112	128	3	128	128	147584	1.75E+06	1.61E-02	1.85E+09	4.13E+00	4.48E+08	4.73E-01	3.912E+09
Layer Conv3_1	56	128	3	256	256	295168	6.97E+05	4.04E-03	9.26E+08	2.06E+00	4.49E+08	2.33E-01	3.980E+09
Layer Conv3_2	56	256	3	256	256	590080	1.39E+06	8.06E-03	1.85E+09	4.12E+00	4.49E+08	4.65E-01	3.980E+09
Layer Conv3_3	56	256	3	256	256	590080	1.39E+06	8.06E-03	1.85E+09	4.12E+00	4.49E+08	4.65E-01	3.980E+09
Layer Conv4_1	28	256	3	512	512	1180160	1.38E+06	2.07E-03	9.30E+08	2.07E+00	4.50E+08	2.32E-01	4.014E+09
Layer Conv4_2	28	512	3	512	512	2359808	2.76E+06	4.08E-03	1.86E+09	4.14E+00	4.50E+08	4.63E-01	4.014E+09
Layer Conv4_3	28	512	3	512	512	2359808	2.76E+06	4.08E-03	1.86E+09	4.14E+00	4.50E+08	4.63E-01	4.014E+09
Layer Conv 5_1	14	512	3	512	512	2359808	2.46E+06	1.07E-03	4.72E+08	1.05E+00	4.50E+08	1.18E-01	4.013E+09
Layer Conv 5_2	14	512	3	512	512	2359808	2.46E+06	1.07E-03	4.72E+08	1.05E+00	4.50E+08	1.18E-01	4.013E+09
Layer Conv5_3	14	512	3	512	512	2359808	2.46E+06	1.07E-03	4.72E+08	1.05E+00	4.50E+08	1.18E-01	4.013E+09
Total Conv						1.47E+07	2.38E+07	9.13E-02	1.54E+10	3.43E+01	5.83E+09	3.90E+00	
FC 1	7x7x512	25088	1	4096	4096	102764544	1.03E+08	1.70E-05	1.03E+08	2.28E-01	4.50E+08	2.54E-02	4.047E+09
FC 2	1x4096	4096	1	4096	4096	16781312	1.68E+07	3.01E-06	1.68E+07	3.73E-02	4.50E+08	4.15E-03	4.047E+09
FC 3	1x4097	4096	1	1000	1000	4097000	4.10E+06	8.00E-07	4.10E+06	9.11E-03	4.50E+08	1.01E-03	4.047E+09
Total FC						1.24E+08	1.24E+08	2.08E-05	1.24E+08	2.75E-01	1.35E+09	3.05E-02	
Total Conv + Total FC						1.38E+08	1.47E+08	9.13E-02	1.55E+10	3.46E+01	4.49E+08	3.93E+00	3.96E+09

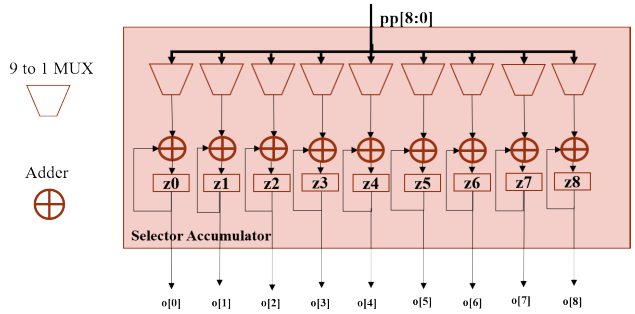


FIGURE 15. Sort and accumulator blocks.

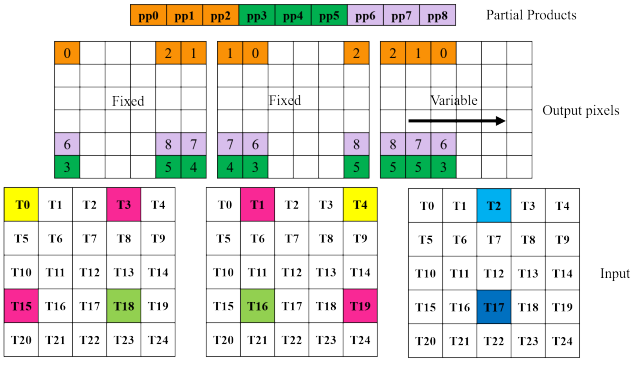


FIGURE 16. Sort for compute stages 1, 2 and 3 (Rows 0 and N-2).

board. We implemented the LeNet-5 design without any parallelism as it a small network. The VGGNet-16 network was implemented both with and without parallelism.

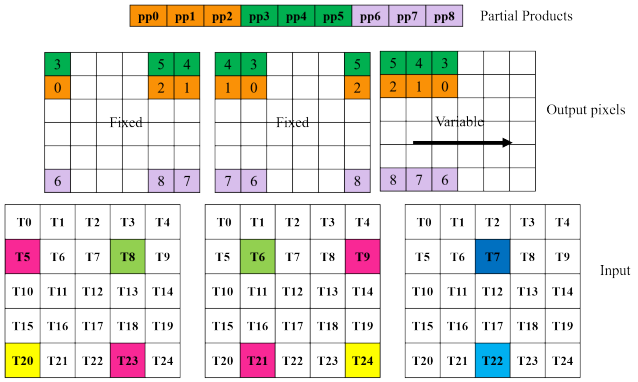


FIGURE 17. Sort for compute stages 4,5,6 (Rows 1 and N-1).

The LeNet-5 architecture consists of: two pairs of convolutional layers, average pooling layers, followed by two fully-connected layers (Figure 22). The design is implemented at a frequency of 200MHz. The design has 25 DSP48s owing to the size of the kernel ($k \times k$) which is 25. The other resources used in the design were 1901 LUTs, 3073 FFs, and 8 BRAM blocks.

The VGGNet-16 architecture consists of 13 convolutional layers and 3 fully connected layers. This is illustrated in Figure 23. The VGGNet-16 design is implemented at a frequency of 100MHz. We implement the design with a parallelism factor of1 (Figure 24) and 9 (Figure 25). The design with a parallelism factor of 9 can process 9 kernels of size 3×3 simultaneously as it had 9 parallel SPP2D engines compared to the design with parallelism factor 1.

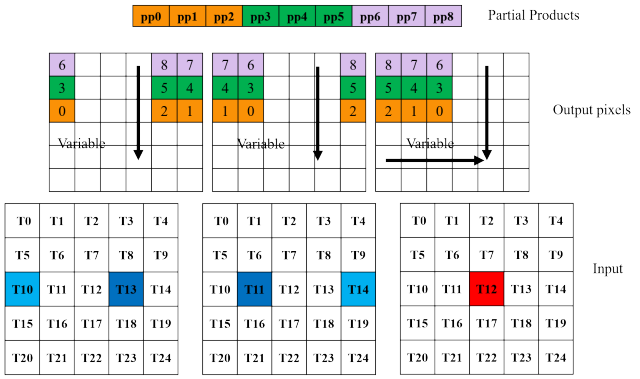


FIGURE 18. Sort for compute stages 7,8,9 (Rows 2 to N-3).

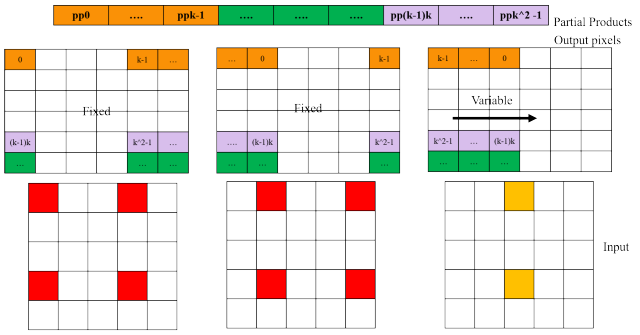


FIGURE 19. Sort for rows 0 and N-2.

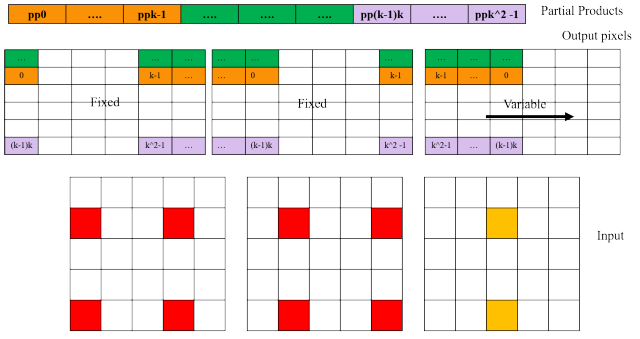


FIGURE 20. Sort for rows 1 and N-1.

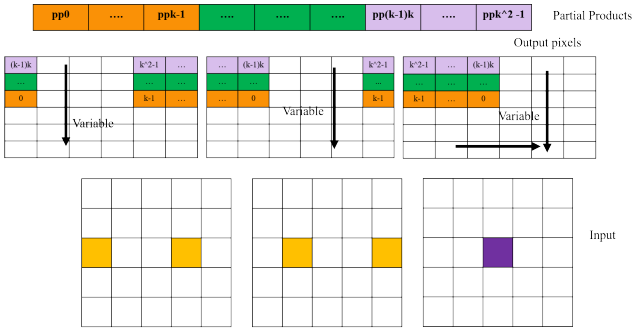


FIGURE 21. Sort for rows 2 to N-3.

VII. RESULTS

The throughput and performance of LeNet-5 and VGGNet-16 are described in Table 6 and Table 7. In Table 8 and Table 9 we compare the results for LeNet-5 and VGGNet-16 with other contemporary designs. In Table 6, we report the following -

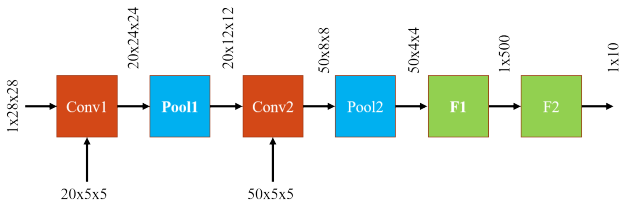


FIGURE 22. LeNet-5 CNN architecture.

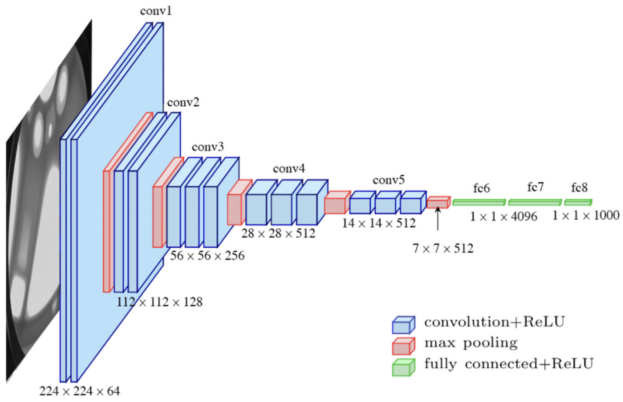


FIGURE 23. VGGNet-16 network structure.

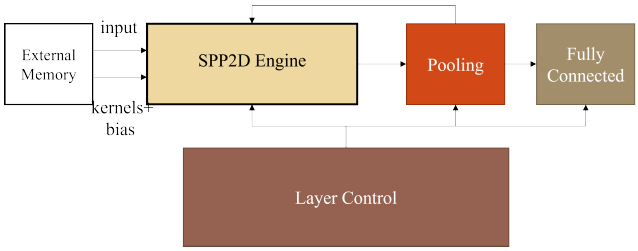


FIGURE 24. VGGNet-16 architecture with parallelism factor of 1.

TABLE 8. Comparison of the SPP2D basedLeNet-5 architecture with others.

	[48]	[47]	SPP2D
Frequency		150MHz	200MHz
DSP	5	83	25
LUT	221	303600	1901
FF	314	607200	3073
BRAM	16	2060	8
Power	-	-	0.337W
GOP/s	-	16.6	5
GOP/s/W	-	-	14.8
GOP/s/DSP	-	0.2	0.208

parameters required for each layers, multiply accumulate operations (MACs) operation for each layer and combinations, execution time for each layer and combination, performance (which is the reciprocal of execution time), GOP/s for each layer and combination. Finally, we report GOP/s/W for the entire system and GOP/s/W for the SPP2D IP. If a machine has an execution time of 1 sec, the performance metric indicates how much faster our design is compared to it. Table 6 breaks the results down to individual layers and as well as the following combinations - both convolution

TABLE 9. Comparison of the SPP2D based VGGNet-16 architectures with others.

	Arch Accel [16]		Eyeriss [18]	IDLA [11]	FIFO Based [15]	Fast Gen [12]	Mem Access [13]	High Utilization [19]	SPP2D	
Publication Year	**2018**		**2016**	**2020**	**2021**	**2017**	**2020**	**2021**		
	Area Eff	**MA Eff**							**Parallelism=1**	**Parallelism=9**
ASIC /FPGA	ASIC	ASIC	ASIC	FPGA	FPGA	FPGA	FPGA	ASIC	FPGA	FPGA
Data precision				16 bit	Int 8	16 bit	8 bit fixed	8 bit fixed	8bit fixed	8 bit fixed
Freq	200 MHz	400 MHz		167 MHz	100 MHz	200 Mhz	160 Mhz	500 MHz	100 Mhz	100 Mhz
Power	254 mW	260mW	236 mW		304 mW for IP			554.5682 mW	291 mW	298 mW
Power Gain #	1.15	1.12	1.23		0.96			0.52		
Power Gain *	1.17	1.15	1.26		0.98			0.54		
Total Latency	436.4 ms	453.3. ms	4309.5 ms	110.7 ms	2151.5 ms		46.9 ms	13.35	91.3 ms	91.3 ms
Total Latency Saving	79.08	79.86	97.88	17.52	95.76		-94.67	-583.90		
Throughput	70.3 GOP/s 2.29 fps	67.7 GOP/s 2.21 fps	0.7 fps				660 GOP/s	67.2 GOP/s	0.449 GOP/s 0.0289 fps	3.96 GOP/s 0.255 fps
Total Operations/frame				227.63 GOP		660.9 GOP		1152 GOP	15.5 GOP	15.5 GOP
Total Execution time/frame		13422.6 ms			5214.5 ms				34.6 s	3.93 s
Total Execution time/frame savings #		-158.21			-565.38					
Total Execution time/frame savings *		70.67			24.42					
Energy Efficiency	276.8 GOP/s/W	260.4 GOP/s/W							1.54 GOP/s/W	13.3 GOP/s/W
Performance Density ops/cycle									4.49 ops/cycle	39.6 ops/cycle

Parallelism Factor = 1,* Parallelism Factor = 9
Power Gain = Power of SPP2D architecture/Power of other design
Total Latency savings = ((Total Latency of other designs - Total Latency of SPP2D) *100) / Total Latency of other designs
Exec time savings = ((Exec times of other designs - Exec times of SPP2D) *100) / Exec times of other designs

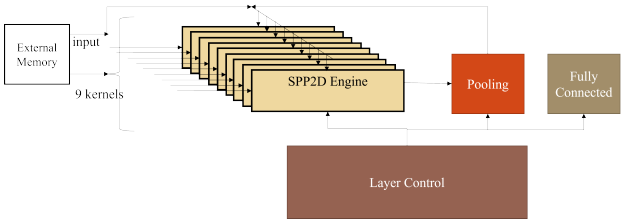


FIGURE 25. VGGNet-16 architecture with parallelism factor of 9.

layers (Conv1, Conv2), both fully connected layers (F1,F2) and all the layers (Conv1, Conv2, F1, F2). The design gives a throughput of 14.8 GOP/s/W (given onchip power) or 42.7 GOP/s/W if we consider the power consumption of the SPP2D IP without the peripherals for the both convolution layers. We considered the throughput for the convolution layers since the SPP2D architecture applies to the convolutional layers. The power consumption of the system is 0.337W and that of the SPP2D IP is 0.117 W. The SPP2D implementation of the LeNet-5 architecture is compared with two designs described in [47], [48] in Table 8. The design in [48] is based on reducing the parameters in the CNN design which decreases the footprint of the design while also increasing the throughput. The design in [47] is implemented at 150MHz and has a throughput of approximately 14.8 GOP/s. We observed that in our design we used a lower number of DSPs than [47] with a comparable workload and throughput (GOP/s). Compared to [48] we observed a better GOP/DSP. We definitely use more DSP48s than [48] but it is a small concession to make given the availability and abundance of the DSP48 blocks. The on-chip power consumed by the SPP2D architecture is 0.337 W which is very competitive with current designs.

We compare our VGGNet-16 implementation with many contemporary designs. The results of this implementation are described in Table 7 and Table 9 respectively. Table 7, describes the throughput and performance of each layer of the VGGNet-16 network as well as the complete network. The total latency of the network is 91.3 ms. The number of operations per frame the network has to perform is 15.5 GOP. These metrics remain the same for the network with a parallelism factor 1 and 9. The parallelism helps simultaneously process multiple kernels with an input. The number of operations per second (GOP/s) for parallelism of 1 is 0.448 GOP/s with an execution time of 34.6secs. The number of operations per second for a parallelism factor of 9 is 3.96 GOP/s with an execution time of 3.93 secs. Finally, we compare the performance of the SPP2D based implementation with other relevant designs [11]–[13], [15], [16], [18], [19] in Table 9. SPP2D based VGGNet-16 design has a low latency of 91.3 ms which is **79**%, **97**%, **17**% and **95**% less than that of conventional designs [11], [15], [16], [18] while the power consumption is 291 mW and 298 mW for parallelism of 1and 9 respectively. The power consumption is low and comparable to or better than that of other designs. Our design has a lower overall operation per frame of 15.5 GOP since our design aims to reduce re-fetching of data and also limiting the number or calculations. Thus the throughput of our network is lower compared to other designs (0.448 GOP/s, 0.0289 frames/s for parallelism of 1 and 3.96 GOPs/s, 0.225 frames/s for parallelism of 9). The total processing time of our design with a parallelism factor of 9 is **3.93** secs and it is **70**% less than [16] and **24**% less than [15]. The advantage of our design is that we avoid the re-reading of input pixels, thus reducing power consumption due to huge memory traffic. However, as a result, we need to wait for all the partial products to be calculated for the output to be available. This puts a lower limit on the latency of our design for each input size. In addition to this, we need a buffer space which is the size of the output feature map size, which depends on the biggest output feature map size. This can cause a strain on the SRAM budget of a design. SPP2D based architectures are suitable for an input and kernel of any size and we have demonstrated it for a stride of 1. We plan to update the SPP2D architecture to incorporate variable strides in a future iteration of the architecture. In summary, we demonstrated that an SPP2D based hardware accelerator can deliver low latency with low power and can be a competitive choice to implement any deep neural network.

VIII. CONCLUSION AND FUTURE WORK

We have presented the SPP2D convolution algorithm which is a fast and efficient method of computing 2-D convolutions. We have demonstrated that it can process the input and kernel of any size. SPP2D based architectures have delivered a more generalized design to implement any 2-D convolution. SPP2D based architectures have provided tremendous savings in re-fetching input pixels for computing partial products compared to the reuse calculation described in [11], [15], [16], [18], [47]. as well as a low latency and high throughput solution for calculating convolutions. The SPP2D based LeNet-5 and VGGNet-16 validate the concept introduced in [46] and presented in detail in this paper. Furthermore, we plan to extend this research to other networks such as MobileNet and ResNet-50 thus proving that the SPP2D architecture is truly network agnostic and adaptable to any design.

ACKNOWLEDGMENT

Xilinx's University Support Program donated the board Xilinx Kintex 7 KC705 Evaluation Kit that was used in this research.

REFERENCES

[1] O. Matan, H. S. Baird, J. Bromley, C. J. C. Burges, J. S. Denker, L. D. Jackel, Y. L. Cun, E. P. D. Pednault, W. D. Satterfield, C. E. Stenard, and T. J. Thompson, "Reading handwritten digits: A ZIP code recognition system," *Computer*, vol. 25, no. 7, pp. 59–63, Jul. 1992.

[2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.

[3] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "CuDNN: Efficient primitives for deep learning," 2014, *arXiv:1410.0759*.

[4] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on CPUs," in *Proc. Deep Learn. Unsupervised Feature Learn. NIPS Workshop*, vol. 1, 2011, p. 4.

[5] Z. Li, J. Eichel, A. Mishra, A. Achkar, and K. Naik, "A CPU-based algorithm for traffic optimization based on sparse convolutional neural networks," in *Proc. IEEE 30th Can. Conf. Electr. Comput. Eng. (CCECE)*, Apr. 2017, pp. 1–5.

[6] F. Abuzaid, "Optimizing CPU performance for convolutional neural networks," Stanford Univ., Stanford, CA, USA, Tech. Rep., 2015. [Online]. Available: http://cs231n.stanford.edu/reports/2015/pdfs/fabuzaidfinalreport.pdf

[7] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 247–257, 2010.

[8] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 243–254, doi: 10.1109/ISCA.2016.30.

[9] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, pp. 1–12.

[10] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, Jun. 2017, pp. 535–547.

[11] P. Gao, Z. Huang, H. Ye, and G. Chen, "IDLA: An instruction-based adaptive CNN accelerator," in *Proc. IEEE 15th Int. Conf. Solid-State Integr. Circuit Technol. (ICSICT)*, Nov. 2020, pp. 1–3.

[12] H. Zeng, C. Zhang, and V. Prasanna, "Fast generation of high throughput customized deep learning accelerators on FPGAs," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs (ReConFig)*, Dec. 2017, pp. 1–8.

[13] T. Tian, X. Jin, L. Zhao, X. Wang, J. Wang, and W. Wu, "Exploration of memory access optimization for FPGA-based 3D CNN accelerator," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2020, pp. 1650–1655.

[14] V. Panchbhaiyye and T. Ogunfunmi, "A FIFO based accelerator for convolutional neural networks," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2020, pp. 1758–1762.

[15] V. Panchbhaiyye and T. Ogunfunmi, "An efficient FIFO based accelerator for convolutional neural networks," *J. Signal Process. Syst.*, vol. 93, no. 10, pp. 1117–1129, Oct. 2021.

[16] A. Ardakani, C. Condo, M. Ahmadi, and W. J. Gross, "An architecture to accelerate convolution in deep neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 4, pp. 1349–1362, Apr. 2018.

[17] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2015, pp. 161–170.

[18] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[19] K.-T. Lin, C.-T. Chiu, J.-Y. Chang, and S.-C. Hsiao, "High utilization energy-aware real-time inference deep convolutional neural network accelerator," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2021, pp. 1–5.

[20] G. Lacey, G. W. Taylor, and S. Areibi, "Deep learning on FPGAs: Past, present, and future," 2016, *arXiv:1602.04283*.

[21] J. Fowers, G. Brown, P. Cooke, and G. Stitt, "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications," in *Proc. ACM/SIGDA Int. Symp. Field Program. Gate Arrays (FPGA)*, 2012, pp. 47–56.

[22] A. Ansari, K. Gunnam, and T. Ogunfunmi, "An efficient reconfigurable hardware accelerator for convolutional neural networks," in *Proc. 51st Asilomar Conf. Signals, Syst., Comput.*, Oct. 2017, pp. 1337–1341.

[23] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*.

[24] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4820–4828.

[25] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.

[26] S. Han, "Efficient methods and hardware for deep learning," Ph.D. dissertation, Stanford Univ., Stanford, CA, USA, 2017.

[27] A. Ansari and T. Ogunfunmi, "Empirical analysis of fixed point precision quantization of CNNs," in *Proc. IEEE 62nd Int. Midwest Symp. Circuits Syst. (MWSCAS)*, Aug. 2019, pp. 243–246.

[28] 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. Red Hook, NY, USA: Curran Associates, 2012, pp. 1097–1105.

[29] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*.

[30] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. ACM/IEEE 44th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2017, pp. 27–40, doi: 10.1145/3079856.3080254.

[31] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," 2018, *arXiv:1802.06367*.

[32] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," 2017, *arXiv:1705.08922*.

[33] S. Han, J. Pool, S. Narang, H. Mao, S. Tang, E. Elsen, B. Catanzaro, J. Tran, and W. J. Dally, "DSD: Regularizing deep neural networks with dense-sparse-dense training flow," 2016, *arXiv:1607.04381*.

[34] L. Jiang, M. Kim, W. Wen, and D. Wang, "XNOR-POP: A processing-in-memory architecture for binary convolutional neural networks in wide-IO2 DRAMs," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Design (ISLPED)*, Jul. 2017, pp. 1–6.

[35] T. Tang, L. Xia, B. Li, Y. Wang, and H. Yang, "Binary convolutional neural network on RRAM," in *Proc. 22nd Asia South Pacific Design Automat. Conf. (ASP-DAC)*, Jan. 2017, pp. 782–787.

[36] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *Proc. 33rd Int. Conf. Mach. Learn.*, New York, NY, USA, vol. 48, Jun. 2016, pp. 2849–2858. [Online]. Available: https://proceedings.mlr.press/v48/linb16.html

[37] N. Doi, T. Horiyama, M. Nakanishi, and S. Kimura, "Minimization of fractional wordlength on fixed-point conversion for high-level synthesis," in *Proc. Asia South Pacific Design Automat. Conf. (ASP-DAC)*, Jan. 2004, pp. 80–85.
[38] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. 32nd Int. Conf. Int. Conf. Mach. Learn.*, vol. 37, 2015, pp. 1737–1746.
[39] S. Lee and A. Gerstlauer, "Fine grain word length optimization for dynamic precision scaling in DSP systems," in *Proc. IFIP/IEEE 21st Int. Conf. Very Large Scale Integr. (VLSI-SoC)*, Oct. 2013, pp. 266–271.
[40] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 11, pp. 5784–5789, Nov. 2018, doi: 10.1109/TNNLS.2018.2808319.
[41] K. Kum and W. Sung, "Word-length optimization for high-level synthesis of digital signal processing systems," in *Proc. IEEE Workshop Signal Process. Syst. (SIPS), Design Implement.*, Oct. 1998, pp. 569–578.
[42] S. Winograd, *Arithmetic Complexity of Computations*, vol. 33. Philadelphia, PA, USA: SIAM, 1980.
[43] T. S. Kim, J. Bae, and M. H. Sunwoo, "Fast convolution algorithm for convolutional neural networks," in *Proc. IEEE Int. Conf. Artif. Intell. Circuits Syst. (AICAS)*, 2019, pp. 258–261.
[44] C. Cheng and K. K. Parhi, "Fast 2D convolution algorithms for convolutional neural networks," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 5, pp. 1678–1691, May 2020.
[45] A. Ansari and T. Ogunfunmi, "Selective data transfer from DRAMS for CNNs," in *Proc. IEEE Int. Workshop Signal Process. Syst. (SiPS)*, Oct. 2018, pp. 1–6.
[46] A. Ansari and T. Ogunfunmi, "A fast 2-D convolution technique for deep neural networks," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.
[47] Y. Zhou and J. Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," in *Proc. 4th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, vol. 1, 2015, pp. 829–832.
[48] M. Hailesellasie, S. R. Hasan, F. Khalid, F. A. Wad, and M. Shafique, "FPGA-based convolutional neural network architecture with reduced parameter requirements," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.
[49] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

ANAAM ANSARI (Graduate Student Member, IEEE) received the bachelor's degree in electronics engineering from the University of Mumbai. She is currently pursuing the Ph.D. degree with the Electrical Engineering Department, Santa Clara University. Her work involves researching new and efficient hardware architectures for convolutional neural networks. Immediately followed by her undergraduate career, she enrolled at San José State University (SJSU) into a graduate program in electrical engineering. She spent her graduate career focusing on wireless communications theory. She has participated in several projects ranging from robotics to wireless communication using software-defined radios. Before becoming a full-time Ph.D. student, she also worked at LSI Corporation as a Systems Engineer with the SerDes Architecture Team. She has also interned at autonomous driving companies, such as Velodyne and Waymo. Her current research interest includes new architecture paradigms for performing convolutional neural networks.

TOKUNBO OGUNFUNMI (Senior Member, IEEE) received the B.S. degree (Hons.) from Obafemi Awolowo University (formerly the University of Ife), Ile-Ife, Nigeria, and the M.S. and Ph.D. degrees from Stanford University, Stanford, CA, all in electrical engineering. From 2010 to 2014, he served as the Associate Dean for Research and Faculty Development for the SCU School of Engineering. He is currently a Professor of electrical and computer engineering and the Director of the Signal Processing Research Laboratory, Santa Clara University (SCU), Santa Clara, CA, USA. At SCU, he teaches a variety of courses in circuits, systems, signal processing, and related areas. His current research interests include machine learning, deep learning, speech and multimedia (audio and video) compression, digital and adaptive signal processing and applications, and nonlinear signal processing. He has published over 200 refereed journal and conference papers in these areas. He has been involved with several IEEE conference committees as a member of the organizing and technical committees. He served as the General Chair for the 2018 IEEE Workshop on Signal Processing Systems (SiPS 2018) and the Technical Program Co-Chair for the 2019 IEEE International Symposium on Circuits and Systems (ISCAS 2019). He served the IEEE as a Distinguished Lecturer, from 2013 to 2014, for the Circuits and Systems Society. He currently serves on the Editorial Board for the IEEE TRANSACTIONS ON SIGNAL PROCESSING and the *Circuits, Systems, and Signal Processing* (CSSP) journal. He also served as a Lead Guest Editor for *CSSP* Special Issue on "Algorithms and Architectures for Machine Learning Based Speech Processing" published in August 2019 and the *Journal of Signal Processing Systems* (JSPS) Special Issue on 2018 IEEE Workshop on SiPS.