LACS: A High-Computational-Efficiency Accelerator for CNNs

JIANGWEI SHANG1, LEI QIAN2, ZHAN ZHANG1, LIXING XUE1, AND HONGWEI LIU1

1School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150006, China
2State Key Laboratory of Science and Engineering Computing, Wuxi 214000, China

Corresponding author: Hongwei Liu (liuhw@hit.edu.cn)

ABSTRACT Convolutional neural networks (CNNs) have become continually deeper. With the increasing depth of CNNs, the invalid calculations caused by padding-zero operations, filling-zero operations and stride length (stride length > 1) represent an increasing proportion of all calculations. To adapt to different CNNs and to eliminate the influences of padding-zero operations, filling-zero operations and stride length on the computational efficiency of the accelerator, we draw upon the computation pattern of CPUs to design an efficient and versatile CNN accelerator, LACS (Loading-Addressing-Computing-Storing). We reduce the amount of data movements between registers and the on-chip buffer from $O(k \times k)$ to $O(k)$ by a bypass buffer mechanism. Finally, we deploy LACS on a field-programmable gate array (FPGA) chip and analyze the factors that affect the computational efficiency of LACS. We also run popular CNNs on LACS. The results show that LACS achieves an extremely high computational efficiency, 98.51% when executing AlexNet and 99.66% when executing VGG-16, significantly exceeding state-of-the-art accelerators.

INDEX TERMS Accelerator, convolutional neural networks (CNNs), field-programmable gate array (FPGA), buffer mechanism.

I. INTRODUCTION

Convolutional neural networks (CNNs) are widely used in many domains, such as object recognition [1], [2] and detection [3]–[6]. CNNs have become continually deeper for high inference accuracy. Google [7] demonstrates how to train a 10000-layer CNN. Modern CNNs require hundreds of megabytes to store parameters and contain a huge amount of operations during inference. Traditional embedded platforms cannot meet their computational demands.

In the past few years, accelerating CNNs’ inferences on a field-programmable gate array (FPGA) has received extensive attention. References [8]–[10] can reduce the memory bandwidth requirement by their architectures. [11], [12] study how to reduce power consumption. References [13]–[15] compute convolution layers in the frequency domain or use the Winograd algorithm to increase the throughput. In the embedded environment, computing resources are very limited. Reference [16] uses computational efficiency as a metric to evaluate CNN accelerators and obtains the metric by computing the utilization of multiply-accumulate (MAC) units. In [17], the authors design three logic block architectures to increase the density of MAC units to improve the throughput of MAC operations. When we study exiting accelerators and CNN algorithms, we find there are many invalid calculations that occupy computing resources, reducing the computational efficiency. We found three sources of invalid calculations.

1) Padding-zero Operations. The goal of convolution layers is to extract the feature from the feature map. To avoid losing the information of the bounds of the feature map, we need to pad zeroes around the feature map, as shown in Figure 1(a), which is called padding-zero operations. The zeroes do not affect the accuracy of the CNNs, but they occupy computing resources. From Figure 2, we can see that as the feature map becomes smaller, the proportion of invalid calculations produced by padding-zero operations increases, and the waste of computing resources is aggravated. Because of these padding-zero operations, the DRAM burst length is reduced when the data are loaded into the on-chip buffer [18].

2) Stride Length. When the convolution kernels extract features, they need to slide on the input feature maps. In some convolution layers, they may slide over two or more pixels one time, as shown in Figure 1(b).

The associate editor coordinating the review of this manuscript and approving it for publication was Songwen Pei1.
FIGURE 1. Padding-zero operation, stride length and filling-zero operation.

(a) When extracting a feature around “1” with a 3 × 3 convolution kernel, we need padding-zero operations.

Stride Length = 3

(b) Stride Length = 3

(c) Fill the data block with zeroes. When partitioning the 3×3 data block into 2×2 data blocks, we need to fill zeroes around the data block.

(d) Fill the convolution kernel with zeroes, kernel size=1, compute unit size=3×3.

The number of the pixels slide over is called the stride length. If the accelerator cannot slide over these pixels, computing resources will be wasted.

3) Filling-zero Operations. There are two types of filling-zero operations: one fills data blocks with zeroes to make sure all data blocks are at the same size, when the accelerator partitions the data, as shown in Figure 1(c); the other one fills the convolution kernels when the kernel size does not match the compute unit size, as shown in Figure 1(d). Both operations will waste computing resources.

We can improve the computational efficiency by eliminating these invalid calculations. The goal of this work is to design a computational-efficiency and universal CNN accelerator, which can be applied to execute CNNs with any convolution kernel size, feature map size and stride length, by designing a novel architecture and eliminating padding-zero operations and filling-zero operations. Our contributions are as follows.

1) We design the LACS architecture for convolution layers. LACS eliminates the influences of padding-zero operations and stride length by establishing the coordinate relationships between the input feature maps, convolution kernels, output feature maps and stride length. We also design a set of instructions for LACS.

2) A simple and practical data partitioning method is proposed to eliminate filling-zero operations and to increase the DRAM burst length.

3) A bypass buffer is designed to reduce the amount of data movements between registers and the output buffer from k × k to k.

4) We show how to extend LACS by adding a POOL module.

5) The factors affecting the computational efficiency of LACS are analyzed. According to the results, we propose a strategy to optimize LACS. We also test the computational efficiency using popular CNNS, AlexNet and VGG-16, and compare it to the latest accelerators. The results show that LACS is an efficient and versatile accelerator.

The remainder of this article is organized as follows. Section II describes related work on reducing invalid calculations. Section III introduces the background of LACS. Section IV deduces the coordinate relationships between the input feature maps, convolution kernels, output feature maps and stride length. We also propose a data partitioning method in this section. We show the architecture and compositions of LACS in Section V. The instruction set, memory access optimizations and steps for extending LACS are introduced in Section VI. The experimental results are presented in Section VII. Section VIII briefly summarizes this article and highlights our future work.

II. RELATED WORK

In this section, we introduce the existing CNN accelerators and how to eliminate the invalid calculations.

A. PADDING-ZERO OPERATIONS

Reference [19] shows a compiler for mapping CNNs onto FPGA, in which the padding-zero operations are needed. Cnvlutin [20] eliminates the padding-zero operations and...
zero-near data by pruning technique, but it cannot efficiently solve the stride length problem. Reference [21] needs padding-zero operations, although it computes the convolution layer in the frequency domain. Reference [13] assumes that the padding-zero operations are performed before running. Reference [14] tries to eliminate padding-zero operations by CAP, but the zeroes are still reserved between data tiles. Reference [18] shows a method to insert padding-zero operations on the fly, which reduces the DRAM burst length when loading the data to the on-chip buffer. References [22]–[24] consider the padding-zero operations when designing an accelerator, but they do not eliminate the operations. Reference [25] reshapes the input data to skip the zero-data, but it is customized for the GAN used in its work; it may need the filling-zero operations when it computes conventional CNNs with different kernel sizes.

### B. STRIDE LENGTH

References [29], [11] solve the stride length problem by using the algorithm in Figure 3(a), but they all need padding-zero operations, and in [29], the accelerator needs filling-zero operations to make sure all data blocks are the same size. References [8]–[10] cannot follow the convolution kernels to slide over those pixels, so they perform a group of valid calculations for each S-cycle (S represents stride length). The computational efficiency is $1/S$. When $S > 1$, the computational efficiency is only 50% at most. Reference [13] also cannot slide over these pixels, and it selects valid results according to the stride length after finishing computations, which not only reduces the computational efficiency but also increases extra overhead. In [30], the authors propose a data partitioning method based on stride length; however, that method requires a reshape operation on the input feature maps. LerGAN [27] utilizes a complex data reshaping method to solve the zero-data and stride length problems for the GAN model. As a result, LerGAN needs more memory to store the weight data. GANAX [26] eliminates the zero-data and stride length problems by reorganizing the output data and filters. As a result, the architecture of GANAX is very complex. Zara [28] is different from GANAX, as it optimizes the zero-data and stride length problems by decomposing the original kernels into multiple sub-kernels according to the computation patterns of input data. However, Zara does not provide a certain method for how to choose the sub-kernels for one instance of input feature maps, and it needs to reshape the output data when applied to execute the traditional CNNs. 

### C. FILLING-ZERO OPERATIONS

[29] uses a unified unroll factor for the entire CNN, but the factor is not suitable for each convolutional layer, and some layers need filling-zero operations to match the unroll factor. In [13], the authors employ a two-dimensional data segmentation in which it is necessary to fill the input feature maps with zeroes to ensure that all data blocks are equal in size. The size of the computing units was set to $3 \times 3$ in [10], [15], and when the convolution kernel size is $5 \times 5$ and $1 \times 1$, filling-zero operations introduce invalid calculations at rates of 44% and 800%, respectively.

### III. BACKGROUND

#### A. CNN MODEL

A typical CNN model contains convolution layers, pooling layers and fully connected layers, as shown in Figure 4(a–c). The parameters of convolution layers are called “convolution kernel”. The feature maps read by convolution layers are called input feature maps, and the feature maps generated by convolution layers are called output feature maps. The image is the input of the first layer of the CNN model. The following layers read the feature maps generated by previous layers and output new feature maps. Figure 4(d) shows the AlexNet CNN [31].

1) **CONVOLUTION LAYER**

In convolution layers, as shown in Figure 4(a), input feature maps convolve with convolution kernels to generate output feature maps. The existing CNN accelerators [11], [29], [32] usually use the algorithm shown in Figure 3(a) to calculate the convolution layers. To avoid the padding-zero operations, we modify the algorithm, as shown in Figure 3(b).

2) **POOLING LAYER**

In pooling layers, the maximum or average value of each subarea in each input feature map will be output, as shown in Figure 4(b).
3) FULLY CONNECTED LAYER
In fully connected layers, the input feature vector is subject to a linear transformation, as shown in Figure 4(c).

B. COMPUTATION PATTERN OF LACS
The computation pattern of CPUs can be divided into four stages.

1) Loading. Load data from memory to cache.
2) Addressing. Compute the addresses of data.
3) Computing. The data read from the cache are sent to the computing unit, and the results are written back to the cache.
4) Storing. Store the results from the cache to memory.

We can obtain the following conditions under which computing units are efficiently utilized.

1) Condition 1: The Addressing stage generates valid addresses without interruption.
2) Condition 2: Data have been loaded to the cache before the Computing stage reads them.
3) Condition 3: The output cache is always available to make sure the computing units are not blocked.
4) Condition 4: The Storing stage does not affect the Loading stage.

Based on these four conditions, we designed LACS (Loading-Addressing-Computing-Storing). To meet Condition 1, we designed a module to generate addresses. For Conditions 2 and 3, we designed the LOAD and STORE modules and used a double-buffer mechanism [13] to overlap the computation with the memory access. We set the priority of each memory port to solve the memory access conflicts between the Storing and Loading stages, i.e., Condition 4.

IV. ADDRESSING STRATEGY
In this section, we detail the addressing strategy of LACS. Address and coordinate have the same meaning in this paper.

A. ADDRESSING MODE
Taking a simple convolution layer with a $3 \times 3$ convolution kernel size, one input feature map and one output feature map as an example, we introduce the addressing modes and select the appropriate addressing mode. There are two addressing modes, as shown in Figure 5.

1) FOTI, from output to input, shown in Figure 5(a).
   To obtain a data instance of the output feature map, the data of the convolution kernel are sequentially traversed. In this mode, we can obtain a set of coordinates of the input feature map based on the coordinates of the output feature map and convolution kernel, corresponding to the algorithm shown in Figure 3(a).

2) FITO, from input to output, shown in Figure 5(b).
   Each data instance of the input feature map is convoluted with all data of the convolution kernel sequentially and generates a set of output data. In this mode, we can obtain a set of coordinates of the output feature map based on the coordinates of the input feature map and the convolution kernel, corresponding to the algorithm shown in Figure 3(b).

In FOTI mode, to obtain one data instance of the output feature map, the results of 9 successive multiplications are...
FIGURE 6. The results of traversing the convolution kernel with different S. The direction of the arrow shows the order in which the convolution kernel is traversed. The coordinates with gray background indicate that valid addresses are generated when traversing these coordinates, and the white background indicates that invalid addresses are generated. When S=1, each coordinate can generate a valid coordinate in (b). When S=2, the rate of coordinates generating a valid coordinate is 36% at the maximum and 16% at the minimum in (c) ~ (f).

acumulated to the same coordinate of the output feature map so that there are data dependencies between the 9 multiplications. There are two solutions for solving the dependencies between them.

1) **Solution 1:** In this solution, the 9 multiplications are performed simultaneously, after which the accumulations are performed by an addition tree. Finally, the result is written back to the output buffer. With this solution, we should choose the 2D architecture to design the computing unit. When the convolution kernel size is not equal to 3 × 3, the filling-zero operations will be introduced [10]. As a result, invalid calculations will be generated.

2) **Solution 2:** In this solution, we should design extra control circuitry. We can sequentially accumulate the results of the 9 multiplications by using a register and then write the result back to the output buffer. This solution will not introduce invalid calculations but will increase the complexity of the addressing circuitry.

There is no data dependency in the FITO mode because the results of 9 multiplications will be accumulated to different coordinates of the output feature map. The FITO mode also does not need padding-zero operations. The disadvantage of the FITO mode is that the number of reads and writes to the output buffer is k × k. For this problem, we can use a bypass buffer (detailed in Section IV-D) to reduce the number of reads from k × k to k. In this work, we choose the FITO mode as the addressing mode of LACS.

**B. ADDRESS MAPPING**

The method of address mapping is similar to Zara [28]. The differences are as follows: 1) we provide a group of formulas to choose the data of convolution kernels, 2) we do not need to separate the convolution kernels and reshape the output data, and 3) the architecture of the accelerator is also different from Zara. Now we provide the deriving process of the formulas.

According to the algorithm in Figure 3(b), we calculate the coordinates of the output feature map according to the coordinates of the input feature map and convolution kernel and stride length, as shown in (1).

\[
\begin{align*}
or &= (ir + kr)/S \\
oc &= (ic + kc)/S \\
kr &\in [-Nkx, Nkx], \quad Nkx = (\text{kernel}_\text{row}_\text{num} - 1)/2 \\
kc &\in [-Nky, Nky], \quad Nky = (\text{kernel}_\text{col}_\text{num} - 1)/2
\end{align*}
\]

or and oc represent the row and column coordinates of the output feature map, respectively. kr and kc represent the row and column coordinates of the convolution kernel, respectively. Nkx and Nky represent the numbers of the convolution
kernel rows and columns, respectively. \( S \) represents the stride length.

Line 7 of Figure 3(b) indicates that before generating a coordinate of the output feature map, it is necessary to judge whether it is divisible by \( S \). If it is divisible, the coordinate is valid; otherwise, it is not valid. We use a convolution layer with a \( 5 \times 5 \) convolution kernel size and \( S = 1 \) and 2 as an example to show the results of traversing the convolution kernel based on the algorithm in Figure 3(b), as shown in Figure 6. The gray background coordinates of the convolution kernel represent generating valid coordinates, and the white background coordinates represent generating invalid coordinates when traversing the convolution kernel. There are no invalid coordinates with \( S = 1 \) (except for the situation shown in Section V-C), as shown in Figure 6(b). Figure 6(c) \( \sim \) Figure 6(f) show that when \( S = 2 \), there are four different paths of generating valid coordinates for the data of the input feature map, and each path will generate many invalid coordinates. To meet the \textbf{Condition.1} proposed in Section III-B, we should skip those coordinates that generate invalid coordinates. The rest of this section will show how LACS skips these coordinates.

1) \textbf{COORDINATE RELATIONSHIPS BETWEEN POINTS IN THE PATH}

When \( S = 2 \), it is found that when the data of the input feature map traverse the convolution kernel, there are four paths, as shown in Figure 6(c) \( \sim \) Figure 6(f). The adjacent points in the path have the following relationship.

\[
(\text{kr}\_next, \text{kc}\_next) = \begin{cases} (kr, kc + 2), & kc + 2 \leq Nky \\ (kr + 2, 0), & kc + 2 > Nky \end{cases}
\]

(2)

Eq. (3) generalizes Eq. (2), and the convolution kernel size is \( k \times k \) and \( S = n \). \( k \) is an odd number, and \( S \) is an even number or 1.

\[
(\text{kr}\_next, \text{kc}\_next) = \begin{cases} (kr, kc + S), & kc + S \leq Nky \\ (kr + S, 0), & kc + S > Nky \end{cases}
\]

(3)

2) \textbf{THE FIRST COORDINATE OF THE PATH}

For a data instance of the input feature map, as long as the first coordinate of the path is determined, the data can then traverse the path. Figure 7(a) \( \sim \) Figure 7(d) show the data of the input feature map that traverse the same path with an \( 8 \times 8 \) input feature map size as an example.

According to Figure 7, we can obtain the relationships between the coordinates of the data of the input feature map and the first coordinate of the path.

\[
(\text{kr}\_begin, \text{kc}\_begin) = (ir\%2 - 2, ic\%2 - 2)
\]

(4)

We generalize Eq. (4) to the case where the convolution kernel size is \( k \times k \) and \( S = n \). \( k \) is an odd number, and \( S \) is an even number or 1.

\[
\begin{align*}
\text{kr}\_begin & \geq ir\%S - m \times S \\
\text{kc}\_begin & \geq ic\%S - m \times S \\
\end{align*}
\]

(5)

According to (1), (3), and (5), we can obtain the following addressing steps for a convolution layer with arbitrary stride length, convolution kernel size and input feature map size.

\textbf{Step 1:} Calculate the first coordinate of the path for the data of the input feature map based on (5).

\textbf{Step 2:} According to (3), traverse the entire convolution kernel and calculate the output addresses according to (1).

C. DATA PARTITIONING

It is impossible to store all the input feature maps, convolution kernel parameters, and output feature maps on-chip; therefore, we need to partition these data into blocks. There are four principles for data partitioning.

1) \textbf{Principle 1:} The method for data partitioning should be as simple as possible to eliminate filling-zero operations and enhance versatility.

2) \textbf{Principle 2:} For the input feature maps and convolution kernel parameters, we should increase the DRAM burst length as much as possible when loading them from external memory to the on-chip buffer.

3) \textbf{Principle 3:} For intermediate data, we should minimize the amount of data movements between the on-chip buffer and the external memory.

4) \textbf{Principle 4:} For output feature maps, the data should be used directly by the next layer without reshaping the feature maps [32], and we should also increase the DRAM burst length as much as possible when the output
feature maps are stored from the on-chip buffer to external memory.

In FITO mode, to obtain one output feature map, we only need to convolve each data instance of the input feature map with each data instance of the corresponding convolution kernel. According to this pattern, we propose a very simple coarse-grained partitioning method called row partitioning, as shown in Figure 8. In this method, the row is the basic data block. A data block contains at least one row of the feature map. We can determine the number of rows contained in a data block according to the size of the on-chip buffer. LACS naturally avoids filling-zero operations by using this uneven method. The number of input feature maps contained in a data block, shown in Figure 8(c), is equal to the number of input feature maps ($P_{in}$) that the PE units can process in one cycle. To ensure the output feature maps are read directly by the next layer, the output feature maps are also divided by rows. To eliminate the movements of intermediate data between the on-chip buffer and external memory, LACS stores the data block of output feature maps to external storage after it has traversed all data blocks needed for the output data block. Row partitioning naturally guarantees that the data are arranged sequentially in external memory, which increases the burst length.

V. LACS ACCELERATOR

A. SYSTEM OVERVIEW

Figure 9 shows the architecture of LACS. The architecture of the LACS is similar to VTA [18]. The LACS consists of three parts. One part loads instructions, including the FETCH module. The second part moves data between the on-chip buffer and external memory, including the LOAD and STORE modules. The third part performs computations, including the ADDRGEN, COMPUTE, and POOL modules (POOL module is described in Section VI-C). LACS is driven by an instruction stream and employs dependency signals between the modules as the control signals. In LACS, the dependency signals are stored in dependency queues, as shown in Figure 9. The dependencies between modules are as follows.

1) **Dependency 1:** $Dep_{L2A}$. The ADDRGEN module does not generate addresses until the LOAD module finishes loading the necessary data.

2) **Dependency 2:** $Address$. The COMPUTE module requires valid addresses generated by the ADDRGEN module before it performs computations.

3) **Dependency 3:** $Dep_{C2S}$. The STORE module does not store the results to external memory until the COMPUTE module completes all calculations.

4) **Dependency 4:** $Dep_{C2L}$. Before the LOAD module loads data, it is necessary to judge whether the COMPUTE module has released the input and weight buffers.

5) **Dependency 5:** $Clear$. Before the ADDRGEN module generates addresses, the module needs to judge whether the Address queue is cleared by the COMPUTE module.

6) **Dependency 6:** $Dep_{S2A}$. Before executing a new ADDRGEN instruction, the ADDRGEN module needs to judge whether the STORE module cleared the output buffer.

B. LOAD MODULE

The LOAD module uses the AXI4-Master interface to access the external memory via an AXI Interconnect IP [33]. Because of the handshake mechanism of the AXI4-Master interface, extra time overhead is introduced before the interface transmits data. When the burst length is long enough, the extra overhead can be shared. Therefore, when the LOAD Module loads data, we should make sure that there is enough data to be loaded, which can be guaranteed by the data partitioning method proposed in Section IV-C.

C. ADDRGEN MODULE

According to the addressing strategy shown in Section IV, we design the ADDRGEN module to generate addresses for the COMPUTE module and synthesize the ADDRGEN
FIGURE 10. Three areas of the input feature map; the convolution kernel size is $3 \times 3$.

FIGURE 11. ADDRGEN module.

module by Vivado HLS 2018.2. In Section IV-B, we rebuilt the address mapping strategy, therein reducing the number of invalid addresses generated by the ADDRGEN module; however, we found that the COMPUTE module would still stall when we tested the LACS. Through further analysis, we found the reasons and propose a solution.

1) REASON
As shown in Figure. 10, one data block is divided into three areas: corner area, edge area and center area. The data in the edge and corner areas generate some valid addresses that can be divisible by $S$ but beyond the bounds of the output feature map. When $S = 1$ and the convolution kernel size is $3 \times 3$, the data in the corner area only generate 4 addresses, the data in the edge area only generates 6 addresses, and the data in the center area can generate 9 addresses. Due to the existence of corner and edge areas, the COMPUTE module will be stalled. Another reason is that when $S > 1$, $kr_{begin}$ and $kc_{begin}$ obtained according to (5) may not be in the interval shown in (1), which may cause the COMPUTE module to pause for a maximum of $k/S$ cycles.

2) SOLUTION
We can add some judgments to ignore those addresses that are beyond the bounds, but this measure reduces the throughput of the ADDRGEN module, which cannot solve the problem caused by the corner and edge areas. We refer to the double-buffer strategy [13], and we use two cores to generate addresses in one ADDRGEN module. We use row partitioning to divide the data block used by the ADDRGEN module into two smaller blocks, block A and block B. The two cores traverse block A and block B respectively and simultaneously, as shown in Figure. 11(b). The generated addresses are stored in two address queues, QA and QB. When QA is empty, the COMPUTE module reads QB. To eliminate the overhead of switching between QA and QB, we design a FIFO Read CrossBar for the COMPUTE module to read addresses from two queues. With two-level partitioning (the first level partitions the feature maps into data blocks before loading them into the on-chip buffer, and the second level is performed by the ADDRGEN module), the COMPUTE module is almost always working. The data blocks A and B should satisfy the following two conditions:

1) Condition 1: The number of valid addresses generated from block B is greater than the number of invalid addresses generated from block A.
2) Condition 2: The number of addresses in QB is greater than or equal to the number of invalid addresses remaining in block B when block A is finished.

D. COMPUTE MODULE
1) ARCHITECTURE OF THE COMPUTE MODULE
The COMPUTE module is fully pipelined, as shown in Figure. 12. As long as the address queues are not empty, the COMPUTE module is always busy, and all computing resources are fully occupied. The COMPUTE module consists of four sub-modules:

- **Address resolution module.** This module resolves the addresses read from the address queues.
- **Data Reading module.** This module reads data from the on-chip buffers according to the addresses.
- **PE Array.** The PE array adopts two-level parallelism. Different PE units convolute the same input feature maps (inp vector) with different convolution kernels (wgt vector) to generate different output feature maps. Therefore, the parallelism of the COMPUTE module is $P_{in} \times P_{out}$, where $P_{in}$ is equal to the number of input feature maps contained in a data block of input feature maps, and $P_{out}$ is equal to the number of output feature maps contained in a data block of output feature maps and is also equal to the number of PE units. The PE units adopt the vector $\times$ vector architecture [16]. There are two reasons for adopting this architecture: this architecture is simple, i.e., the PE units have only one dimension, suitable for the addressing-computing pattern; additionally, it is easy to extend the architecture. We can simply increase the length of the vector or the number of PE units to increase the parallelism of the accelerator.
- **Data Writing Module.** This module writes data from the registers to the output buffer.

2) BYPASS BUFFER MECHANISM
The disadvantage of FITO mode is that the output buffer is read and written frequently. To generate the output feature maps, each data instance in the output buffer will be read and written $k \times k$ times ($S = 1$) according to Figure. 3(b). This is similar to the write-through policy in CPUs. To solve this problem, we refer to the write-back policy.
LACS provides a bypass buffer to implement the write-back policy, which reduces the number of read and write operations from $k \times k$ to $k$. The number of write operations is reduced by more than 40% for AlexNet and VGG-16, as obtained by simulation.

**a: THEORY**

We find that when traversing the input feature map to generate the output feature map in FITO mode, the valid addresses generated by two adjacent data $a$ and $b$ in the same row of the input feature map have an overlapped area, as shown in the green area of Figure 13(b). We use this phenomenon to design the bypass buffer. When we read and write data in the overlapped area, we directly read and write the bypass buffer instead of the output buffer.

**b: ADDRESS MAPPING**

To reduce the computational overhead of address mapping, the size of the bypass buffer is set to $b_{size} \times b_{size}$, where $b_{size}$ is $\beta \in \mathbb{Z}$, such that LACS can use the last $\beta$-bit of row and column addresses of the output buffer as the addresses of the bypass buffer.

**c: PROFIT**

When the number of columns of an output feature map is larger than the bypass buffer, the latter data generated by the data of the input feature map will replace the previous data, as shown in Figure 13(e) ~ Figure 13(g), meaning that we cannot utilize the overlapped area in Figure 13(c), i.e., Figure 13(e) and Figure 13(h). Therefore, the bypass buffer mechanism can reduce the number of reads and writes from $k \times k$ to $k$.

**d: WORKFLOW**

After adding a bypass buffer, the COMPUTE module is shown in Figure 14. The workflow of the COMPUTE module is as follows.

1) **Step 1**: To prevent read misses, the Data Reading module simultaneously reads data from both the output and bypass buffers. Therefore, the number of read operations
FIGURE 14. Workflow mechanism of the bypass buffer.

is $k \times k$. If the bypass buffer hits, the data from the bypass buffer are calculated by the PE Array; if the bypass buffer is missed, the data from the output buffer are calculated, and the data from the bypass buffer are written to the output buffer.

2) **Step 2:** After the calculations are completed, the results are written to the bypass buffer.

3) **Step 3:** After all calculations are completed, the Data Writing module writes the data from the bypass buffer to the output buffer.

e: OUT OF ORDER

The optimization used for the ADDRGEN module in Section V-C will cause an out-of-order problem: When the address queue QA is empty, the COMPUTE module reads addresses from QB. The data in the bypass buffer may be replaced with new data. Thus, when we use the two-level partitioning method, we should attempt to keep the addresses generated by the two sub-blocks A and B from being mapped to the same area of the bypass buffer.

E. STORE MODULE

The STORE module has two functions: moving the results from the output buffer to the external memory and clearing the output buffer. The STORE module uses the AXI4-Master interface to access the external memory, which is similar to the LOAD Module. Because LACS does not need to reshape the output feature maps and uses the row partitioning method, the burst length is long enough to share the extra overhead.

VI. INSTRUCTION SET AND OPTIMIZATION

A. INSTRUCTION SET

Our instruction set consists of three types of instructions: one for loading data, one for generating addresses, and one for storing data. The instruction length is 128 bits. The instructions include two types of information. One type is dependency information, describing the dependencies between the modules. The other type is function information, describing how the module works, such as how much data the LOAD module needs to load.

B. OPTIMIZATION

1) DOUBLE BUFFER MECHANISM

To overlap memory access with computations, the input, weight, and output buffers are all subject to a double-buffer mechanism [13]. When the COMPUTE module reads and writes one buffer, the LOAD module can load data to the other buffer. The STORE module stores the results from one output buffer to external memory when the COMPUTE module is accessing the other output buffer.

2) MEMORY ACCESS CONFLICT

The LOAD, STORE and FETCH modules may access external memory simultaneously, but there is only one memory port of DRAM. To not affect the COMPUTE module, we set the priority of memory access. We set the LOAD module to have the highest priority because the LOAD module accesses memory more frequently than the STORE and
FIGURE 16. Computational efficiencies of LACS with different parallelisms. For different input feature map sizes, the numbers of input feature maps and output feature maps are 128, 128, respectively, and the stride length is $S=1$. The size of the data blocks is equal to or less than the size of the input feature map.

FETCH modules. The STORE module has priority over the FETCH module because the STORE module needs to clear the output buffer for the COMPUTE module, and one instruction costs much time to execute, while the FETCH module has enough time to fetch new instructions for other modules.

C. POOL MODULE
To adapt to the development of CNNs, we consider the scalability of LACS when designing it. Next, we use the POOL module as an example to show how to add a new module to LACS. When inserting a new module, LACS needs to consider the following issues:

1) **Issue 1**: Are there dependencies between the new module and other modules, and will the new module affect the dependencies between other modules?
2) **Issue 2**: Do we need to add a cache block for the new module?
3) **Issue 3**: Do we need to design a new instruction for the new module?

1) **DEPENDENCY**
The input data of the POOL module are from the COMPUTE module, so there are dependencies between the COMPUTE and POOL modules. The results of the POOL module need to be written to external memory, so there are dependencies between the STORE and POOL modules. After adding the POOL module, the execution path is changed from COMPUTE-STORE to COMPUTE-POOL-STORE, so the POOL module affects the dependencies between the COMPUTE and STORE modules.

2) **CACHE BLOCK**
To increase the burst length when the STORE module stores data to the external memory, we need to design a cache block for the POOL module to store the results of the POOL module.

3) **INSTRUCTION**
Because the POOL module cannot be driven by other modules, we need to provide a new instruction for the POOL module.

Thus far, we have finished adding the POOL module to LACS, as shown in Figure. 15. From this process, we can see that it is easy to extend LACS. In some CNNs, such as VGG, the layer following the convolution layers may not be a pooling layer; therefore, we design an interface for the POOL module to access external memory. The memory access priority of the POOL module is the same as that of the STORE module.

VII. EXPERIMENTAL RESULTS
A. PREPARATION
The target platform is Xilinx xc7z030fbg484, the CPU is a dual-core ARM Cortex-A9, the external memory is 1 GB DDR3 DRAM with a peak bandwidth of 3.2 GB/s, and the data type is 8-bit fixed-point. The project is synthesized by
FIGURE 17. Computational efficiencies of LACS with different convolution kernel sizes.

TABLE 1. Computational efficiency of AlexNet and VGG (%).

| Layer | AlexNet | VGG-16 |
|-------|---------|--------|
|       | Baseline | LACS | Baseline | LACS |
| Conv1 | 92.12* | 93.49 | 99.13 | 99.52 |
| Conv2 | 91.31 | 99.87 | 98.81 | 99.86 |
| Conv3 | 90.01 | 98.49 | 97.63 | 99.84 |
| Conv4 | 90.01 | 98.49 | 95.30 | 99.69 |
| Conv5 | 90.01 | 98.49 | 90.70 | 98.71 |
| Conv_ave | 90.78 | 98.51 | 96.76 | 99.66 |

* the first layer is optimized by eliminating the influence of stride length. The result of first layer is less than 60% without eliminating the influence of stride length, and the Conv_ave is about 81.2%.

Vivado 2018.2. When compared with the latest accelerators, the parallelism of the COMPUTE module is simply set to $P_{in} = 16$ and $P_{out} = 16$. We do not pay more attention to exploring the design space, as it is not the work of this article.

B. ANALYSIS

In this section, we explore the factors that may affect the computational efficiency of LACS. We first analyze the influence of parallelism of the COMPUTE module, and then we analyze the influence of convolution kernel size. Finally, we analyze the influence of stride length. The formula of computational efficiency is the same as [16].

1) PARALLELISM

In Section V-D, we explained that the COMPUTE module in LACS adopts two levels of parallelism. Figure 16 shows the computational efficiency of LACS with $P_{in} = 4, 8, 16, 32$, when the parallelism of the output feature maps, $P_{out}$, is $4, 8, 16, 32$. The experimental results show that the computational efficiency of LACS is independent of the parallelism of the COMPUTE module when the size of the input feature maps is fixed.

2) CONVOLUTION KERNEL SIZE

The experimental results are shown in Figure. 17(a). When the convolution kernel size is equal to $3 \times 3$, the computational efficiency of LACS is lower than that with a convolution kernel size of $5 \times 5$. When the feature map size is equal to $7 \times 7$, it is especially obvious that the computational efficiency of LACS is reduced by approximately 3%. In addition, from Figure. 16, we also find that as the input feature map becomes smaller, the computational efficiency of LACS first increases but then decreases. The reasons are as follows.

1) Reason 1: The computational efficiency of LACS is related to the number of valid addresses generated by one ADDRGEN instruction. There are $t$ idle cycles between two ADDRGEN instructions executed. Thus, the real peak computational efficiency is (6). $Num_{valid\_max}$ is the maximum number of valid addresses included in the ADDRGEN instructions.

$$ Computational\_Efficiency_{max} = \frac{Num_{valid\_max}}{Num_{valid\_max} + t} $$

1) Reason 2: The number of valid addresses generated by one ADDRGEN instruction is related to the size of the data blocks of input feature maps obtained by the first-level partitioning and the convolution kernel sizes. Because of the coarse-grained partitioning, there will be a residual page in the buffer (When the input feature map becomes small, we can load more rows to the on-chip buffer. The size of the data block is equal or less than the size of the input feature map.), which is why the computational efficiency of LACS first increases but then decreases as the input feature map becomes smaller. In FITO mode, each data instance in the data block of the input feature maps needs to traverse the corresponding convolution kernel, where a larger kernel means more addresses, which is why the computational efficiency of LACS increases as the convolution kernel size increases.

According to the above analysis results, we can obtain a heuristic method to improve the computational efficiency. When the input feature maps become smaller, we can improve the computational efficiency by combining the data blocks to increase the number of valid addresses generated by one ADDRGEN instruction. According to this method, we optimize the convolution layer in which the size of the input feature map is $7 \times 7$. The results are shown in Figure. 17(b). The computational efficiency is improved significantly.
TABLE 2. The results of comparison.

| Accelerator     | Platform       | CNN     | Total GOPs | Clock (MHz) | Format | LUTs | MAC Units | BRAMs | Throughput (GOP/s) | Image/s | Computational Eff(%) |
|-----------------|----------------|---------|------------|-------------|--------|------|-----------|-------|-------------------|---------|----------------------|
| Snowflake[16]   | ZYNQ xc7z045   | AlexNet | 1.2        | 250         | fixed-16 | -    | 256       | -     | 120.3             | 100.3   | 94                   |
| Angel-eye[10]   | ZYNQ xz7e020   | VGG     | 31.2       | 214         | fixed-8  | 29867 | 190*      | 86    | 84.3              | 2.7     | -                    |
| Ours            | Xc7z030fbg484  | AlexNet | 2.15       | 250         | fixed-8  | 21329 | 256       | 115   | 126.1             | 64.1    | 98.51                |
|                 |                | VGG     | 30.7       |             |          |       |           |       | 127.5             | 4.27    | 99.66                |

* It's the number of DSPs used in Angel-eye.

**FIGURE 18.** Computational efficiencies of LACS with different stride lengths.

**Stride Length:** In Figure 18, we compute the computational efficiencies of LACS with $S = 1$ and $S = 2$. When $S = 2$, the changing trend of computational efficiency is the same as that with $S = 1$. When $S = 2$, the computational efficiency of LACS is lower than that when $S = 1$. The reason for this difference is that when $S = 2$, the number of valid addresses generated by one ADDRGEN instruction is less than the number of valid addresses generated by one ADDRGEN instruction when $S = 1$. When the feature map sizes are 14 × 14 and 7 × 7, we can find that the changes of computational efficiency are different from that of other feature map sizes. The reason is that we combine multiple data blocks into a big data block (When stride is 2, we combine more data blocks than that of stride is 1).

**C. COMPARISON**

In this section, we use AlexNet [31] and VGG-16 [1] to test LACS and compare LACS to the latest accelerators. Because the number of the input feature maps of the first layer of AlexNet and VGG-16 is three, we use batch mode to process the first layer, and the batch size is five.

The computational efficiency of each layer is shown in Table 1, and we assume that the theoretical peak computational efficiency of LACS is 100%. The baseline contains the invalid calculations. From Table 1, we find that the computational efficiency is improved significantly, as LACS eliminates the invalid calculations.

Table 2 shows the results of comparing LACS to other accelerators. When comparing LACS to other accelerators, we choose MAC units rather than DSPs as the computing units [16]. From Table 2, we can see that LACS’s throughput is 1.048 times that of Snowflake, and the total amount of computations of AlexNet in LACS is 1.79 times that of Snowflake; therefore, in theory, the LACS’s speed of processing images is 0.585(1.048 ÷ 1.79) times that of Snowflake. However, the Image/s of LACS is 0.639 times that of Snowflake. The same phenomenon appears when comparing LACS with Angel-eye. The total amount of computations of VGG in Angel-eye is 1.016 times that of LACS, and the throughput of LACS is 1.512 times that of Angel-eye; therefore, in theory, The LACS’s speed of processing images is 1.536 (1.016 × 1.512) times that of Angel-eye. However, the speed of processing images is 1.58 times that of Angel-eye. There are two reasons for this phenomenon: one reason is that LACS’s architecture is computational efficiency; the other reason is that LACS eliminates invalid calculations without blocking the COMPUTE module, while Snowflake and Angel-eye do not.

**VIII. CONCLUSION**

LACS is an efficient and scalable accelerator with an average computational efficiency of over 98%, which is substantially higher than that of other accelerators. In addition, LACS has good versatility and can compute a convolution layer with any stride length and convolution kernels and input feature maps of arbitrary size. The goal of the work in this article is to verify that LACS is a high-computational-efficiency architecture; we do not focus on the design space exploration or inference accuracy. Therefore, future work will emphasize the design space exploration and the inference accuracy of LACS.

**REFERENCES**

[1] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in Proc. Int. Conf. Learn. Represent. (ICLR), 2015.

[2] A. Caglayan and A. B. Can, “Volumetric object recognition using 3-D CNNs on depth data,” IEEE Access, vol. 6, pp. 20058–20066, 2018.

[3] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” in Proc. Adv. Neural Inf. Process. Syst., 2015, pp. 91–99.

[4] W. Cao, J. Yuan, Z. He, Z. Zhang, and Z. He, “Fast deep neural networks with knowledge guided training and predicted regions of interests for real-time video object detection,” IEEE Access, vol. 6, pp. 8990–8999, 2018.
[5] H. Wang, P. Wang, and X. Qian, “MPNET: An end-to-end deep neural network for object detection in surveillance video,” IEEE Access, vol. 6, pp. 30296–30308, 2018.

[6] Q. Lu, C. Liu, Z. Jiang, A. Men, and B. Yang, “G–CNN: Object detection via grid convolutional neural network,” IEEE Access, vol. 5, pp. 24023–24031, 2017.

[7] L. Xiao, Y. Bahri, J. Sohl-Dickstein, S. S. Schoenholz, and J. Pennington, “Dynamical isometry and a mean field theory of CNNs: How to train 10,000-layer vanilla convolutional neural networks,” 2018, arXiv:1806.05393. [Online]. Available: https://arxiv.org/abs/1806.05393

[8] J. Qiu, S. Song, Y. Wang, H. Yang, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, and N. Xu, “Going deeper with embedded FPGA platform for convolutional neural network,” in Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays—FPGA, 2016, pp. 26–35.

[9] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, “Optimizing the convolution operation to accelerate deep neural networks on FPGA,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 26, no. 7, pp. 1354–1367, Jul. 2018.

[10] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, “Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA,” IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst., vol. 37, no. 1, pp. 35–47, Jan. 2018.

[11] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in Proc. ACM/IEEE Int. Symp. Comput. Archit. (ISCA), Jun. 2016, pp. 367–379.

[12] Y. Wang, J. Lin, and Z. Wang, “An energy-efficient architecture for binary weight convolutional neural networks,” IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol. 26, no. 2, pp. 280–293, Feb. 2018.

[13] C. Zhang and V. Prasanna, “Frequency domain acceleration of convolutional neural networks on CPU–FPGA shared memory system,” in Proc. SIGDA Int. Symp. Field-Program-Gate Arrays (FPGA), Feb. 2017, pp. 35–44.

[14] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, “A framework for generating high throughput CNN implementations on FPGAs,” in Proc. SIGDA Int. Symp. Field-Program-Gate Arrays (FPGA), Feb. 2018, pp. 117–126.

[15] Y. Liang, L. Lu, Q. Xiao, and S. Yan, “Evaluating fast algorithms for convolutional neural networks on FPGAs,” in Proc. IEEE Int. Symp. Field-Program. Custom Comput Mach. (FCCM), Apr. 2017, pp. 101–108.

[16] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, “Snowflake: A model agnostic accelerator for deep convolutional neural networks,” 2017, arXiv:1708.02579. [Online]. Available: https://arxiv.org/abs/1708.02579

[17] A. Boutros, M. Eldafrawy, S. Yazdanshenas, and V. Betz, “Math doesn’t have to be hard: Logic block architectures to enhance low-precision multiply-accumulate on FPGAs,” in Proc. ACM/SIGDA Int. Symp. Field-Program-Gate Arrays (FPGA), Feb. 2019, pp. 94–103.

[18] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, “VTA: An open hardware-software stack for deep learning,” 2018, arXiv:1807.04188. [Online]. Available: https://arxiv.org/abs/1807.04188

[19] Y. Ma, N. Suda, Y. Cao, J.-S. Seo, and S. Vrudhula, “Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA,” in Proc. 26th Int. Conf. Field Program. Logic Appl. (FPL), Aug. 2016, pp. 1–8.

[20] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: Ineffectual-neuron-free deep neural network computing,” in Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA), Jun. 2016, pp. 1–13.

[21] H. Zeng, R. Chen, and V. K. Prasanna, “Optimizing frequency domain implementation of CNNs on FPGAs,” Univ. Southern California, Los Angeles, CA, USA, Tech. Rep., 2017.

[22] A. Koiris and S. i. C. S. Venieris Bouganis, “Cascade CNN: Pushing the performance limits of quantisation in convolutional neural networks,” in Proc. 28th Int. Conf. Field Program. Logic Appl. (FPL), Aug. 2018, pp. 1–8.

[23] Y. Chen, J. He, X. Zhang, C. Hao, and D. Chen, “Cloud-DNN: An open framework for mapping CNN models to cloud FPGAs,” in Proc. ACM/SIGDA Int. Symp. Field-Program-Gate Arrays (FPGA), Feb. 2019, pp. 73–82.

[24] F. Chen, L. Song, and Y. Chen, “ReGAn: A pipelined ReRAM-based accelerator for generative adversarial networks,” in Proc. 23rd Asia South Pacific Design Automat. Conf. (ASP-DAC), Jan. 2018, pp. 178–183.

[25] M. Song, J. Zhang, H. Chen, and T. Li, “Towards efficient microarchitectural design for accelerating unsupervised GAN–based deep learning,” in Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA), Feb. 2018, pp. 66–77.

[26] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeizadeh, “GANAX: A unified MIMD–SIMD architecture for generative adversarial networks,” in Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA), Jun. 2018, pp. 650–661.

[27] H. Mao, M. Song, T. Li, Y. Dai, and J. Shu, “LeGAn: A zero–free, low data movement and PIM–based GAN architecture,” in Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO), Oct. 2018, pp. 669–681.

[28] F. Chen, L. Song, H. H. Li, and Y. Chen, “Zara: A novel zero-free dataflow accelerator for generative adversarial networks in 3D ReRAM,” in Proc. 56th Annu. Design Automat. Conf., 2019, p. 133.

[29] C. Zhang, F. 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—FPGA, 2015, pp. 161–170.

[30] L. Jiao, C. Luo, W. Cao, X. Zhou, and L. Wang, “Accelerating low bit-width convolutional neural networks with embedded FPGA,” in Proc. 27th Int. Conf. Field Program. Logic Appl. (FPL), Sep. 2017, pp. 1–4.

[31] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in Proc. NIPS, 2012, pp. 1106–1114.

[32] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, “Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks,” in Proc. ACM/SIGDA Int. Symp. Field-Program-Gate Arrays—FPGA, 2017, pp. 45–54.

[33] Xilinx. (Dec. 2017). AXI Interconnect v2.1. [Online]. Available: https://china.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/axi-interconnect.pdf

Jiangwei Shang received the B.S. degree in computer science and technology from Yanshan University. He is currently pursuing the M.S. and Ph.D. degrees in computer science and technology with the Harbin Institute of Technology. His research interests include computer architecture, high-performance computing, and FPGA-based accelerators for AI algorithms.

Lei Qian received the M.S. degree from the Department of Computer Science and Engineering, Southeast University. His research interests include domain specific computing and artificial intelligence.

Zhan Zhang received the Ph.D. degree in computer science and technology from the Harbin Institute of Technology, in 2008. Since 2014, he has been an Associate Professor with the School of Computer Science and Technology, Harbin Institute of Technology. His research interests include fault tolerant computer, wearable computer, and system evaluation theory and technology.
LIXING XUE received the Ph.D. degree in computer science from the Harbin Institute of Technology, in 2017. His research interests include computer architecture, wearable computing, mobile computing, cloud computing, and fault tolerant computing.

HONGWEI LIU received the Ph.D. degree in computer science and technology from the Harbin Institute of Technology, in 2004. Since 2010, he has been a Professor with the School of Computer Science and Technology, Harbin Institute of Technology. He has published more than 90 articles. His research interests include parallel computing and architecture, fault tolerant computer, resource allocation and optimization in cloud computing system, evaluation theory and technology in cloud computing system, mobile computing, and software reliability modeling. He is a special member of the Fault-Tolerance Committee and a member of the Standing Committee of Computer System Architecture Committee in China. His awards include the Third Prize of scientific and technological advance in national defense (China).