A Survey of Design and Optimization for Systolic Array-based DNN Accelerators

RUI XU, Academy of Military Sciences, China
SHENG MA, YANG GUO, and DONGSHENG LI, National University of Defense Technology, China

In recent years, it has been witnessed that the systolic array is a successful architecture for DNN hardware accelerators. However, the design of systolic arrays also encountered many challenges. As DNN structures and applications become more complex, a DNN hardware accelerator based on the typical systolic array architecture suffers severe performance and efficiency penalties. So, it has motivated a significant amount of research on the redesign and optimization of the systolic array architecture. In this article, we survey these works on analyzing, redesigning, and improving the performance and efficiency of the systolic array architecture. These works are critical to the design flow of DNN accelerators based on systolic arrays. We also provide a technique classification of these works on the basis of their main research idea. Further, we attempt to compare the advantages and disadvantages of different designs and different technologies and provide quantitative results for reference. The aim of this survey is to provide researchers with knowledge of the state-of-the-art in the systolic array architecture and motivate them to design highly efficient DNN accelerators of tomorrow.

CCS Concepts: • Hardware → Hardware accelerators; • General and reference → Surveys and overviews;
Additional Key Words and Phrases: Systolic array, hardware accelerator, hardware-aware software design, architectural techniques, design automation, evaluation strategy

ACM Reference Format:

1 INTRODUCTION
The development of deep neural networks (DNNs) [78] is accompanied by an exponential increase in the requirement of data processing and computing capabilities. With the end of Moore’s law and Dennard scaling [58], there are various challenges for computing platforms, such as...
“power wall” [140] and “memory wall” [127]. Unfortunately, traditional serial execution platforms cannot solve these problems and meet the computing capability needs of deep learning [35, 58]. So, many studies choose high-performance computing platforms rather than serial execution platforms to run the DNN applications, such as GPU, FPGA, and ASIC platforms [26, 27, 59, 143]. Especially the accelerators of FPGA and ASIC platforms have become the hot research in recent years, because they are customized hardware designs for the DNN applications and can achieve higher performance and efficiency than the CPU or GPU platform. But the quality of the hardware design directly determines the performance and efficiency of the FPGA or ASIC platform for DNN applications [35]. So, it is important for designers to choose a good hardware architecture for DNN accelerators.

In 2014, to cope with the increasing demand for deep learning applications, Google proposed a custom application-specific ASIC accelerator, Tensor Processing Unit (TPU) [60], which provides acceleration services for DNNs applications and becomes the first choice for many data centers. Meanwhile, the success of the TPU makes the systolic array architecture, which is adopted by the TPU, become a hot spot of current research. After the TPU, more and more researches and studies propose systolic array-based design to accelerate DNNs applications [30, 37, 67, 96, 150, 151]. Even in GPU platforms, some designers try to use the systolic array architecture to improve the performance and efficiency of DNN applications [106].

However, the technology of DNNs is always developing, and the traditional systolic array architecture can not be compatible with these new features and new optimization techniques in DNNs. Some researchers propose optimization techniques to reduce the memory footprint and calculation requirements of DNNs, such as network pruning and sparsity [21, 137]. These methods can effectively reduce the size of the DNN model and reduce the latency of DNN inference or training. But the traditional systolic array architecture can not support the sparsity of DNNs. If these pruned DNN models are directly mapped to the systolic array, then the calculation of zero elements in weights and activations cannot be skipped, which leads to a large number of inefficient calculations.

The traditional systolic array struggles to efficiently handle new algorithms and functions in modern DNN models [39, 42, 49, 114, 118, 119, 134–136]. For instance, many CNN models adopt depthwise separable convolutional layers [42, 49, 124, 136], which consist of depthwise convolution (DWCONV) and pointwise convolution (PWCONV), as replacements for traditional convolutional layers. Depthwise separable convolution offers lower computational complexity, reducing CNN latency and enhancing hardware accelerator performance. However, we observed that the traditional systolic array accelerator is inefficient when handling depthwise separable convolution. In Figure 1, we evaluated the number of floating-point operations (FLOPs) and latency breakdown in a 16 × 16 systolic array for three state-of-the-art CNN models (EfficientNet-lite [134], MixNet [135], and MobileNetV3 [48]). DWCONV, which accounts for approximately 9%–16% of the total FLOPs, contributes to around 60% of the latency. To address these challenges, the systolic array architecture requires redesign to accommodate these new DNN features and optimization techniques.

It is also difficult to design the optimal systolic array hardware architecture for a specific application scenario. From the servers/cloud to the mobile/edge terminals, there are various performance requirements and cost constraints for different DNN applications. Meanwhile, the designers have to consider various factors in the systolic array design, such as dataflow, array size, buffer, bandwidth, and so on, all of which determine the final performance, efficiency, energy consumption, area of the accelerator [18, 20, 93, 141]. For these reasons, architects should explore the design space and try to compare different solutions with different factors to find an efficient and optimal systolic array design under various constraints.
In summary, it is important for researchers to understand the state-of-the-art of the systolic array designs and their technologies and solutions used to address the problems and challenges. Meanwhile, understanding the design techniques and using automatic tools to design systolic arrays while evaluating different designs through a rational and efficient strategy is also very important. It can bring the optimal solution for the specific application scenario and extremely reduce the design cycle time and cost [37, 56]. These studies and techniques can help designers propose more efficient systolic array architecture and design the best systolic array-based DNN accelerators of tomorrow.

In this article, we first present a survey of research work aimed at optimizing and redesigning the systolic array architecture to support DNN sparsity and diversity. These studies can improve the performance and efficiency and expand the application scenarios of systolic array-based DNN accelerators. We show these systolic array studies and designs and classify them based on several characteristics, including their application scenarios, design goals, and optimization techniques or methods. Then, we review the research work that focused on the automatic systolic array design and emulating/analyzing the systolic array performance of processing DNN applications. These works provide design automation methods or tools for designers to synthesize systolic arrays on FPGA/ASIC platforms and can get various evaluation results of different systolic array designs agilily. These works are essential for designers and architects to explore the design space of the systolic array architecture and can effectively reduce the cost of the design flow.

This article is organized as follows: Section 2 reviews the systolic array and highlights the challenges encountered in the design process. Section 4 reviews the studies on optimizing the systolic array architecture and solving the challenge of DNN sparsity. Section 3 reviews the studies on optimizing and adapting systolic arrays to improve flexibility. Section 5 discusses some methods and tools for automatic synthesis and performance emulation in detail. In Sections 4, 3, and 5, we provide an overview and classification of these studies and then discuss some of the work and techniques in detail. We finally provide concluding remarks and future research trends in Section 6.

2 BACKGROUND

2.1 The History of Systolic Array

In the late 1970s, computer technology developed at a rapid pace, especially, the number of transistors in computer processors was increasing rapidly due to Moore’s Law. At the same time, the limitations of existing computer architectures for parallel computing, such as the von Neumann architecture, were becoming increasingly apparent [65]. Therefore, computer scientists were seeking a new computer architecture that could use these available resources to improve computing efficiency.
performance. In addition, applications such as image processing and computer vision were rapidly developing and required high-performance computer architectures to support them [14, 70, 126].

For these reasons, many researches started to try to use different novel architecture solutions to improve the computing performance. In the process, systolic arrays have gradually come to the fore. In 1978, H. T. Kung summarized the previous work in detail and formally introduced the concept of systolic arrays [65], which are highly parallel and pipelined arrays of simple processing elements that operate in a lockstep, wave-like fashion. Due to the special structure and algorithm design, systolic array can achieve a high degree of parallel computing and have been successfully applied to a variety of problems in scientific computing, signal processing, image processing, and computer vision [5, 13, 14, 34, 50, 75, 83, 107, 113, 115, 126, 157].

However, after the initial excitement around systolic arrays in the 1980s and 1990s, research on this topic has become less prominent. While some researchers continue to explore new algorithms and applications for systolic arrays, the overall trend has shifted towards more flexible and programmable architectures such as GPUs, FPGAs, and multi-core processors [143].

With the emergence of deep learning, there has been renewed interest in systolic arrays [60]. As a potential architecture with high throughput and low power consumption, the systolic array is well-suited for deep learning applications. Systolic arrays have a high degree of parallelism compared to traditional architectures due to their large number of parallelizable computational cells, which is essential for deep learning applications involving a significant number of parallel computations [122]. In addition, systolic arrays are computationally efficient, thanks to the simple structure of the computational cells and the computation process, which does not require complex control logic. By caching and reusing data in the array, systolic arrays eliminate the need for repeated access operations, leading to significant energy savings. These advantages make systolic arrays a hot topic in current research on deep learning accelerator architectures [122].

### 2.2 Systolic Array

In 1978, Kung et al. proposed the systolic array architecture [65]. Figure 2 shows a typical **2-dimensional (2D)** systolic array design. It consists of a set of homogeneous and interconnected **processing elements (PEs)**, a controller module, and the on-chip memory/buffer. The PE is composed of basic arithmetic and register units, which can support a simple multiply-accumulate operation [55]. As shown in Figure 2, the data flows from the buffer and passes through multiple processing elements in the array in a rhythmic manner, or flows from the processing elements and
returns to the buffer. This process is like the cardiovascular system, so the PE array is called the systolic array.

The systolic array has many advantages. First, the interconnection of the 2D PE array and the register units in PEs naturally realize the data reuse when processing matrix/tensor algebra operation, especially matrix multiplication [122]. Through the interchange of data via PEs, it can avoid repeated operations such as reading, writing, or synchronizing data between the systolic array and on-chip/off-chip memory. Since the workload of DNNs is based on matrix and tensor operations, systolic arrays can reduce the latency of DNN accelerator operation and can also save a lot of bandwidth and energy consumption [37].

Second, it can solve the I/O and computation imbalance. The data in the systolic array flows between PEs in a pipeline model, and communication with the outside buffer occurs only at the boundary PEs [65]. It means that a large number of PEs in the array do not need to communicate with external memory so n I/O ports can at most drive n^2 PE units, which greatly reduces the memory bandwidth requirements.

Third, the simple and regular structure of the systolic array can reduce the design cost. The systolic array is composed of a few types of simple blocks, which are used repetitively with simple interfaces and interconnection network [37, 65]. These can greatly shorten the design time of the accelerator, so the systolic array is more attractive when the hardware design cost is strictly limited.

Fourth, the systolic array has good scalability. The PE array is modular and can be adjustable to various performance goals, that is, system cost can be made proportional to the performance required [65]. So, it can easily yield a cost-effective hardware accelerator design for specific applications.

Finally, the systolic array design is more efficient than other designs. If there are a large number of PEs working simultaneously in other designs, then the communication and coordination overhead will be very significant, especially the routing mechanism that will dominate the power consumption, time and area required to implement a computation operation [24].

For example, the crossbar network occupies more than 20% of the area in SCNN [109]. In NVDLA [105], the area sequencer module that prepares data layout for the multiply-and-accumulate (MAC) units accounts for 14% of the convolution pipeline area [24]. However, the data communication in the systolic arrays is actually completed by the pipeline of the registers in the PEs. It is spontaneous without any control assistance and supports high degrees of concurrency [65, 96]. Therefore, the systolic array design is a more effective and efficient choice for designers. Table 1 compares the systolic array designs with other accelerators.

Undoubtedly, other studies pertaining to the architecture of PE arrays have yielded remarkable outcomes, as evidenced by Eyeriss [19], Eyeriss v2 [20], and MAERI [73]. However, it is noteworthy that the PE arrays in these architectures predominantly employ broadcast, multicast, or other data

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td># PEs/MAC Units</td>
<td>168</td>
<td>256</td>
<td>64</td>
<td>64</td>
<td>1,024</td>
<td>256</td>
<td>1,024</td>
</tr>
<tr>
<td>Frequency (MHz)</td>
<td>250</td>
<td>1,000</td>
<td>1,000</td>
<td>1,000</td>
<td>1,000</td>
<td>1,000</td>
<td>800</td>
</tr>
<tr>
<td>Area (mm2)</td>
<td>12.3</td>
<td>3.0</td>
<td>4.9</td>
<td>7.9</td>
<td>2.4</td>
<td>1.2</td>
<td>1.7</td>
</tr>
<tr>
<td>Power (mW)</td>
<td>278</td>
<td>485</td>
<td>320</td>
<td>1,000</td>
<td>348</td>
<td>766.1</td>
<td>586</td>
</tr>
<tr>
<td>Performance (GOPS)</td>
<td>23.1</td>
<td>452</td>
<td>194</td>
<td>-</td>
<td>1,774.8</td>
<td>-</td>
<td>6,210</td>
</tr>
<tr>
<td>Efficiency (GOPS/W)</td>
<td>83.1</td>
<td>932.0</td>
<td>606.3</td>
<td>-</td>
<td>5,100</td>
<td>-</td>
<td>10,597.3</td>
</tr>
</tbody>
</table>

* Systolic array design.
transmission modes instead of pulsation transmission [18–20, 72]. Furthermore, some of these studies have altered the composition of the array by using tree or loop structures [72, 73], thereby deviating from the concept of the systolic array. As such, these studies lie beyond the scope of this article’s discussion.

2.3 The Terminology of Systolic Array Design

2.3.1 Workloads. DNNs consist of different neural network layers, the most important of which are fully connected (FC) layers and convolutional (CONV) layers [78].

In the FC layer, all inputs from one layer are connected to every activation unit of the next layer [78]. This implies that to compute a single output in the FC layer, all inputs need to be multiplied by their corresponding weights. Figure 3 showcases the structural representation of the FC layer, demonstrating its dense connectivity. Additionally, Figure 3 illustrates the equivalent matrix computation, which simplifies the multiplication process by utilizing matrix operations. Algorithm 1 further describes the calculation process involved, which typically involves a 3-nested loop [18]. In Algorithm 1, n denotes the mini-batch size, m represents the mth output, and c signifies the cth input.

**ALGORITHM 1: The 3-nested loop of FC**

**Input:** \( I[N, C] \) and \( W[M, C] \)

**Output:** \( O[N, M] \)

1. for \( (n = 0; n < N; n + +) \)
2. for \( (m = 0; m < M; m + +) \)
3. for \( (c = 0; c < C; c + +) \)
4. \( O[n, m] += W[m, c] * I[n, c] \)

Unlike the FC layer, there are multiple feature maps (FMAPs) and convolutional filters in the CONV layer [79], as shown in Figure 4. One input channel of FMAPs is convolved with the corresponding kernel in the filter. Then the results of all input channels are accumulated to obtain the output feature map corresponding to the filter. The operation of CONV can be described as a 7-nested loop [18], as seen in Algorithm 2, where \( n \) is the mini-batch, \( m \) is the mth channel of output FMAPs (OFMAPs), \( m \) is also called the number of filters [57]), \( c \) is the cth channel of input FMAPs (IFMAPs), \( r \) or \( s \) is the height or width of OFMAPs, respectively, and \( k \) or \( q \) is the height or width of kernels, respectively.

2.3.2 Loop Unrolling. To improve the performance of the systolic array, it is necessary to consider how to explore the parallelism of the workload to drive multiple PEs to work simultaneously when designing the systolic array accelerator. A simple way is loop unrolling [65]. Taking Algorithm 1 as an example, we can flatten loop1 and loop2 to the spatial hardware and compute...
ALGORITHM 2: The 7-nested loop of CONV

Output: \( O[N, M, R, S] \)

1: for \( (n = 0; n < N; n + +) \)
2: for \( (m = 0; m < M; m + +) \)
3: for \( (c = 0; c < C; c + +) \)
4: for \( (r = 0; r < R; r + +) \)
5: for \( (s = 0; s < S; s + +) \)
6: for \( (k = 0; k < K; k + +) \)
7: for \( (q = 0; q < Q; q + +) \)
8: \( O[n, m, r, s] += W[m, c, k, q] \times I[n, c, r + k, s + q] \)

these loop instances in parallel. This process is also called spatial loop unrolling [65], which is equivalent to exploiting the replication of the hardware in physical space instead of the loop unrolling in the software. Loop unrolling can explore the parallelism of the workload, but the designer has to decide which loops in the workload are worth unrolling.

2.3.3 Dataflow. To better describe the process of loop unrolling, we can use dataflow to represent which loops in the workload are unrolling. There are three typical dataflows in the systolic array design: Output Stationary (OS), Weight Stationary (WS), and Input Stationary (IS) [37, 122]. The “stationarity” means the elements of this tensor are not moved for the maximum duration of time throughout the computation [122], and it can describe the reuse situation of data in the systolic array and represent a strategy of loop unrolling. Figure 5 shows how the dataflows unroll the loops in Algorithm 1 and map data to the systolic array.

The OS dataflow depicted in Figure 5(a) [122] refers to mapping output elements to the corresponding PEs, where each PE needs to complete all the computations required for an output element. Figure 6(a) shows the PE structure of OS dataflow, and the partial sums are generated and reduced within the MAC unit in the PE [151]. Once a PE in the array completes the computation and generates the output data, the peer-to-peer links are used to transfer the data out of the array. For systolic array synchronization, the data and result matrices are properly skewed [122].
Figure 5(a) also shows that using the OS dataflow is equivalent to spatial unrolling loop1 and loop2 in Algorithm 1, while loop3 runs in the time dimension.

The WS dataflow uses a different strategy, as shown in Figure 5(b) [122]. The elements of the weight matrix need to be pre-filled and stored into each PE before the computation. Then the elements of the input data flow through the left edge of the array, and each PE generates one partial sum every cycle. The generated partial sums are then reduced along each column in parallel to generate one output element (or reduced sum) per column. The PE structure with WS dataflow is different from the PE with OS dataflow [151], as shown in Figure 6(b). Figure 5(b) also shows that using the WS dataflow is equivalent to spatial unrolling loop2 and loop3 in Algorithm 1, while loop1 runs in the time dimension.

The IS dataflow is similar to the WS dataflow, with the difference being in the order of mapping. Instead of pre-filling the array with elements of the weight matrix, the input data are stored in each PE [122], as shown in Figure 5(c). The systolic array design using the IS dataflow is equivalent to spatial unrolling loop1 and loop3 in Algorithm 1, while loop2 runs in the time dimension.

Note that when designing a systolic array, the designer must carefully choose the dataflow, because it is related to the strategy of loop unrolling, which in turn affects the data reuse and performance of the accelerator.

2.3.4 Array Size. In addition to the dataflow, the size of the systolic array can also affect the mapping of the entire workload on the systolic array. Generally, the workload is much larger than the size of the systolic array, so the systolic array is not enough to carry an entire loop unrolling. The solution is to divide the workload into multiple tiles, and then the systolic array executes these tiles in series [45], as shown in Figure 7. It needs to divide the OFMAPs of the FC layers into several \( L \times J \) tiles for a systolic array with the OS dataflow to work normally, where \( L \) is the width and \( J \) is the height of the systolic array. The usage of tiles changes the original process of the systolic array and adds more loop iterations to the workload [18]. So, it further increases the design complexity of the systolic array.
Fig. 7. The workload needs to be divided into multiple tiles for the systolic array with OS dataflow.

The size of a systolic array also has a direct impact on the data filling and emptying processes [53, 122]. Given that the system typically provides data to one or a few sides of the array, the majority of PEs within the array need to acquire new data from either the edge or adjacent PEs. Moreover, the systolic array allows data to drain exclusively from one side, requiring the PEs to accumulate or transfer data to the edge of the array one-by-one before outputting it. The systolic array is thus akin to an assembly line, which can be highly efficient but requires filling and emptying at the outset and conclusion of the computation. As with an assembly line, the time taken for filling and emptying increases with the number of stages, resulting in additional time consumed for larger array sizes. Therefore, it is essential to consider this factor while designing and deploying systolic arrays.

2.3.5 On-chip Buffer and Bandwidth. In the systolic array design, the on-chip buffer is used to temporarily store the input and weight data required for the systolic array calculation and cache the computing results like partial sums and output data. It can prevent the systolic array accelerator from frequently reading from and writing to off-chip memory to reduce the latency and energy consumption [37]. However, if the on-chip buffer is too large, then it will take up a lot of area and power consumption, and if the buffer is too small, then it will not be able to accommodate the data required for calculation [132], which will result in frequent off-chip memory accesses.

The bandwidth affects the data communication rate between off-chip memory and on-chip memory and is also limited by the hardware resources of the systolic array [37, 132]. If the bandwidth is too small, which results in the data of the on-chip memory not being updated in time, then the systolic array can only stall and wait for the arrival of the required data.

2.3.6 Design Space. As we introduce in Sections 2.3.1–2.3.5, there are various features and factors that need to be considered in the systolic array design. But if we regard these factors as parameters, then the process of designing a systolic array becomes a problem about exploring and searching in the design space, which is composed of these parameters [63]. A good set of design parameters can naturally improve the performance of the systolic array to accelerate DNN applications, however, it is not an easy job to explore the design space and find the optimal design parameters [63].

2.4 Challenges of Systolic Array Design

2.4.1 Inflexibility. With its simple and regular structure, the systolic array has become a widely used approach for designing hardware accelerators for neural network computations. However, this design choice comes with certain limitations. Specifically, due to the strict requirements it
Fig. 8. The im2col algorithm can transform the CONV to matrix multiplication, where the IFMAPs become a \((R \times S \times N) \times (C \times K \times Q)\) matrix, and the filters become a \((C \times K \times Q) \times M\) matrix.

imposes on the data mapping process, the systolic array suffers from inflexibility, which restricts the range of applications or workloads it can handle.

The FC layer can be directly mapped to the systolic array to speed up the calculation, because the main purpose of the 2D systolic array is to accelerate the matrix multiplication, and an FC layer operation is essentially a matrix multiplication operation [65]. But the CONV layer cannot be directly mapped to the systolic array [116, 125]. Fortunately, the CONV layer can be transformed from convolution to matrix multiplication through the im2col algorithm and then mapped to the systolic array for acceleration [57], as shown in Figure 8.

As DNNs continue to evolve, new convolutional layers and algorithms, such as depthwise convolution (DWCONV), have emerged. However, the systolic array is unsuitable for processing DWCONV due to its inherent inflexibility.

Unlike the CONV, where entire channels in a kernel are convolved with all IFMAPs and produce one OFMAP, DWConv only allows one filter-channel to convolve with only one IFMAP to produce one OFMAP [25, 49]. This means the number of filters is only one, and IFMAP data cannot be reused between the filters. Figure 9 shows the DWCONV structure, and Algorithm 3 describes the DWCONV process [150], where the OFMAP channel is correlated to \(c\) and the \(m\) in CONV loops is disappeared (equivalent to \(M = 1\)).

The DWCONV also cannot be directly mapped to a systolic array, and after the conversion of im2col, the filters become a vector rather than a matrix [24]. So, the DWCONV operation becomes a matrix-vector multiplication instead of the matrix-matrix multiplication. It can cause a large number of PEs to be idle and seriously affects the performance of the systolic array accelerator, as shown in Figure 10.

In addition, there are other convolutional operations such as deconvolution and dilated convolution, which are widely used in GANs [39, 114] and object detection tasks (RCNN [119], YOLO [118],...
Fig. 9. The DWCONV layer structure in DNN models.

Fig. 10. (a) Systolic arrays are good for computing matrix-matrix multiplications, but (b) for matrix-vector multiplications, there are a large number of PEs that cannot be activated and are left idle.

**ALGORITHM 3:** The 6-nested loop of DWCONV

**Input:** \( I[N, C, R + K - 1, S + Q - 1] \) and \( W[C, K, Q] \)

**Output:** \( O[N, C, R, S] \)

1: for \((n = 0; n < N; n++)\)
2: for \((c = 0; c < C; c++)\)
3: for \((r = 0; r < R; r++)\)
4: for \((s = 0; s < S; s++)\)
5: for \((k = 0; k < K; k++)\)
6: for \((q = 0; q < Q; q++)\)
7: \( O[n, c, r, s] += W[c, k, q] * I[n, c, r + k, s + q] \)
etc.). The filters of these convolutions contain a large number of zero elements and would produce a large number of inefficient computations when calculated by the systolic array directly. DNN models also contain other special calculations or operations, such as batch-normalization [52], softmax [1, 57], activation functions [1, 97, 104], and so on, which are not supported by the systolic array [37, 45].

One solution is that the systolic array, as a slave processor, is only responsible for the calculation of the matrix multiplication (like FC and CONV layers in DNN models), and the rest of the DNNs workloads are given to the host (such as Arm/RSIC-V CPU) [37]. But general processors are also inefficient to process these special operations. According to Amdahl’s law [47], this limits the upper performance of the systolic array accelerator. Moreover, the systolic array is also idle at this time and can only wait for the host to complete the calculation, which wastes a large number of computing units in the array.

2.4.2 DNN Sparsity. DNN sparsity has been a research hotspot in recent years [21, 44, 132, 137]. It is observed that contemporary DNN models tend to be over-parameterized, and a large portion of elements in the weight/activation matrix or tensors can be pruned to reduce the model complexity and the heavy cost of training/inference [17]. So, a pruned DNN model using sparse matrix or tensors can be deployed on more platforms with low computing and storage resources.

Unfortunately, the systolic array can only compute dense matrix/tensors of the DNNs due to its simple and regular structure. The weight and FAMPs in sparse DNNs have an irregular data structure. If we want to map the workload of sparse DNNs to the systolic array, then we have no choice but to refill the sparse matrix and tensors with zero elements, and it will cause a large number of inefficient calculations [91]. It means that traditional systolic array designs cannot enjoy the advantages of DNN sparsity, and the systolic array architecture must be redesigned or optimized.

2.4.3 Complex Design Space. Although the systolic array has a simple structure, it is not a simple job to design an efficient systolic array. As mentioned in Section 2.3, the design space of the systolic array involves many parameters and factors. Meanwhile, with the development of deep learning, accelerator designs also require rapid iteration to adapt to new application requirements [20, 37, 56, 59, 60]. Generally, the systolic array design is based on the designer’s experience, but some studies prove that these designs are flawed and have room for improvement [37, 59, 122, 145]. If only relying on the experience of the architect to design the systolic array, then the long cycle and the high cost of the systolic array design flow are also unbearable.

In addition, the application scenarios of the systolic array accelerators are different. Some designs may be applied to the training of DNNs in pursuit of higher throughput and PE utilization rate [60], some are used in inference and require low latency [59], and some are deployed on mobile/embedded platforms, which have strict limits on area and power consumption [37]. Therefore, it is necessary to design an automated design aid/tool to help designers automatically synthesize systolic array designs for different applications on various platforms. It also needs a tool or strategy to help designers understand the performance, energy consumption, area, and other indicators of various systolic array designs.

3 OPTIMIZATION FOR FLEXIBILITY

3.1 Overview

In Section 2.4, we discuss the importance of flexibility for DNN accelerators. Therefore, many studies propose their solutions, which can be divided into three categories. First, the hardware-aware software design method can be also applied to optimizations for flexibility [54, 125]. Specifically, the designers first optimize the algorithms to make it more friendly to the systolic array architecture and then modify the systolic array architecture to support the optimized algorithm.
A Survey of Design and Optimization for Systolic Array-based DNN Accelerators

Table 2. The Classification of Optimization for Flexibility

<table>
<thead>
<tr>
<th>Type</th>
<th>Technology</th>
<th>Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware-aware software design</td>
<td>Group convolution</td>
<td>[54]</td>
</tr>
<tr>
<td></td>
<td>Fully separable convolution</td>
<td>[125]</td>
</tr>
<tr>
<td>Architectural technique</td>
<td>Multi-tasking</td>
<td>[9]</td>
</tr>
<tr>
<td></td>
<td>Modifying architecture</td>
<td>[24, 86, 123, 150, 151]</td>
</tr>
<tr>
<td>Platform integration</td>
<td>Heterogeneous architecture</td>
<td>[40, 53, 111]</td>
</tr>
<tr>
<td></td>
<td>Software systolic arrays</td>
<td>[120]</td>
</tr>
</tbody>
</table>

Fig. 11. (a) The GCONV layer structure. (b) DWCONV is a special case of GCONV.

The second category is the architectural techniques [9]. In this scheme, the designer only modifies the systolic array architecture to solve the inflexibility issue. As introduced in Section 4.3, its advantage is that there is no need to modify applications, workloads, or algorithms. Some related work uses multi-threading to hide the latency during executing multi-neural network tasks. Others make the systolic array to support more calculation types, such as DWCONV, deconvolution, and dilated convolution.

The last category is the platform integration techniques [40]. Since the systolic array is not flexible, some related work directly transplants its architecture to other general-purpose platforms and integrates their structures. So, these designs can not only enjoy the acceleration of the systolic array, but also can switch back to a general-purpose platform to solve the problem of inflexibility.

To discuss these researches more clearly, we categorize the related work. Table 2 shows the classification of optimizations for flexibility.

3.2 Hardware-aware Software Design

As introduced in Section 2.4, DWCONV layers suffer from PE underutilization and fail to achieve peak performance and energy efficiency of the systolic array. An effective solution is to redesign the DNN models and optimize or replace DWCONV layers to make them suitable for the systolic array again.

Jha et al. propose the DRACO [54]. It replaces the DWCONV with the group convolution (GCONV). Figure 11 shows the structure of the GCONV. In the GCONV layer, the convolutional kernel is divided into several groups, and each group is only convolved with the corresponding IFMAP, which reduces the calculated amount compared with the traditional convolution. In fact, the DWCONV is a special case of GCONV with Group = 1. The DNN models using the GCONV increase the opportunity of data reuse for systolic arrays. Although it increases the amount of calculation, experimental results show it can reduce the latency of the systolic array. Compared with the state-of-the-art row stationary dataflow, it achieves 41.8% and 42.6% improvement in average PE utilization rate and inference latency (respectively) with negligible loss in the predictive accuracy in MobileNetV1 on a 64 × 64 systolic array.
Selvam et al. follow a similar approach and propose Fully Separable Convolution (FuSeC-conv) as a drop-in replacement for DWCONV [125]. FuSeConv generalizes the decomposition of convolutions fully to separable 1D convolutions along spatial and depth dimensions. The resultant computation is systolic and efficiently utilizes the systolic array with a slightly modified dataflow, where 1D convolutions of FuSeConv can be mapped into individual rows of a 2D systolic array. With FuSeConv, their design achieves a significant speedup of $3 \times$–$7 \times$ with the MobileNet family of networks on a systolic array of size $64 \times 64$, with comparable accuracy on the ImageNet dataset.

### 3.3 Architectural Techniques

As introduced in Section 2.4, the systolic array is inefficient when calculating certain workloads or CNN layers or even unable to calculate some operators and can only forward them to the host for completion. This significantly increases the application latency.

One method is to use the multi-threading technology to hide the latency. Baek et al. propose AI-Multitasking (AI-MT) [9], an optimized systolic array accelerator that enables a cost-effective, high-performance multi-neural network execution. The key idea of AI-MT is to fully utilize the accelerator’s computation resources and memory bandwidth by matching efficient and inefficient tasks from different networks and executing them in parallel. AI-MT also creates fine-grain tasks at compile time by dividing each layer into multiple identical sub-layers. During runtime, AI-MT dynamically applies the sub-layer scheduling methods based on sub-layer scheduling tables (similar to scoreboard module in CPU) to switch threads and tasks. The evaluation results show that AI-MT achieves up to $1.57 \times$ speedup over the baseline scheduling method [9].

Another method is to add the required calculation functions to the systolic array, which is achieved by modifying the systolic array architecture and optimizing its dataflow. For optimizing DWCONV processing, Cho proposes the Reinforced Systolic Array (RiSA) [24]. The RiSA is still a normal systolic array when calculating ordinary convolution. When calculating DWCONV, it changes part of the PE array to multiple 1D PE chains, and DWCONV can be mapped to these PE chains and completed in a pipeline form. To hide the bubbles in the pipeline, the 1D PE chains receive two sets of input data simultaneously. Compared to Eyeriss v2, RiSA improves the area and energy efficiency for the inference on MobileNet-V1 by $1.91 \times$ and $1.31 \times$, respectively [24].

Samajdar et al. design a flexible systolic array by augmenting a base monolithic systolic array called SARA (Self-Adaptive Reconfigurable Arrays) [123], which has additional bypass paths along the row and columns. This design achieves high mapping efficiency and data reuse simultaneously and has mapping flexibility and reconfigurability.

Meanwhile, Samajdar et al. find that as the array size increased, the number of SARA configurations increased dramatically. Therefore, they also design the ADAPTNET, which uses deep learning to complete the automatic configuration of the systolic array. The experimental results demonstrate 95% test accuracy compared to an exhaustively searched optimal configuration, beating state-of-the-art classification techniques such as SVMs, XGBoost, and MLPs [123]. The results also present a 32.768 TOPS instance called SAGAR that is capable of providing the same mapping flexibility as a compute equivalent distributed system while achieving $3.5 \times$ more power efficiency and $3.2 \times$ higher compute density demonstrated via architectural and post-layout simulation [123].

Putic et al. propose a novel DNN accelerator architecture called DyHard-DNN [111], which utilizes dynamic hardware reconfiguration and online reconfiguration techniques to achieve higher acceleration performance and energy efficiency. The architecture can adaptively adjust hardware resources at runtime to accommodate different network structures and operating loads and supports various types of DNN layers. The article presents a case study using the ResNet-50 network and demonstrates that DyHard-DNN can achieve up to $3.3 \times$ speedup and up to $2.2 \times$ energy efficiency improvement compared to state-of-the-art DNN accelerators [111].
Xu et al. propose the HeSA, a heterogeneous systolic array, to enhance DWCONV calculations [151]. They introduce the OS-S (single OFMAP channel) dataflow, optimized from the original OS dataflow, which suits the systolic array architecture. Unlike the original OS dataflow (Figure 12(a)), the OS-S dataflow reconfigures data reuse, focusing on single-channel and spatial dimension (Figure 12(b)). This approach maximizes DWCONV layer’s data reuse opportunities. Additionally, they design a new PE in the HeSA that facilitates IFMAP data reuse across rows and columns of PEs (horizontally and vertically) to support the OS-S dataflow [151] (Figure 12(d)). Evaluation shows that HeSA with OS-S achieves 4.5×–11.2× improvement in DWCONV layer’s PE utilization and over 20% energy savings compared to the state-of-the-art design. The HeSA maintains the simple structure of the naïve systolic array, resulting in minimal changes to its area requirements [151].

For deconvolution and dilated convolution, Liu et al. propose a unified systolic convolution array (USCA) [86], which solves the invalid/zero calculation problems. They find that GANs and semantic segmentation tasks add a large number of zero elements in the convolution kernel to form deconvolution and dilation convolution [86]. Fortunately, the distribution of these zeros is regular. So, they designed the USCA, with multiple flexible one-dimensional PE chains that can realize four systolic transmission modes. Through effective configuration, the USCA can skip zeros to achieve effective calculation and data reuse [86].

### 3.4 Platform Integration

Since the systolic array is inflexible, related research also considers integrating it with a general-purpose platform. In fact, the types of platform integration can be redesigning the platform structure to fuse the systolic array architecture [40] or realizing the systolic dataflow in software in the new platform.

Guo et al. propose Simultaneous Multi-mode Architecture (SMA) [40], a novel architecture and execution model that offers general-purpose programmability on the DNN systolic array accelerators. The key to SMA is the temporal integration of the systolic execution model with the GPU-like SIMD execution model. The SMA exploits the common components shared between the systolic-array accelerator and the GPU and provides lightweight reconfiguration capability to switch between the two modes in situ. The SMA achieves up to 63% performance improvement while consuming 23% less energy than the baseline Volta architecture with TensorCore [40].

As AI-based applications become pervasive, CPU vendors try to incorporate matrix engines within the datapath to boost efficiency. While systolic arrays have been the premier architectural
choice as matrix engines in offload accelerators, using them as matrix engines within CPUs can lead to reduced efficiency and increased latency due to limited register storage space. Therefore, Jeong et al. propose an efficient register-aware systolic array matrix engine called RASA (Efficient Register-Aware Systolic Array Matrix Engine for CPU) [53] to address this issue and improve the efficiency of matrix engines within CPUs. RASA achieves efficient operation by dividing the execution phase into multiple sub-phases and overlapping instructions appropriately. Experimental results show that RASA can significantly improve performance compared to other existing methods under the same hardware conditions and can be easily integrated into existing CPUs without increasing hardware costs or power consumption [53].

Rong et al. propose a language and compiler to productively build high-performance software systolic arrays that run on GPUs [120]. Based on a rigorous mathematical foundation with uniform recurrence equations and space-time transform, their language has a high abstraction level and covers a wide range of applications. Although the systolic arrays are purely software realized on generic SIMD hardware, compared with the GPU’s specialized, hardware samplers that perform the same convolutions, some of the best designs are up to 59% faster [120].

4 OPTIMIZATION FOR SPARSITY

4.1 Overview

The technical routes of related studies can be divided into two categories. One is the hardware-aware software design [17, 67, 85, 156], which modifies the pruning algorithms of DNN sparsity to make the pruned weight matrix dense or regular again [17] and optimizes the systolic array architecture to support these new algorithms, including structured sparsity, density-bound block, column packing, and conflict pruning, and so on.

Another category is architectural techniques. It only modifies the hardware structure of systolic array designs and relies on the optimization of the architecture to support DNN sparsity [45, 128]. Although the efficiency is not as good as the hardware-aware software design, the advantage of architectural techniques is that there is no need to modify or retrain the pruned DNNs.

To discuss these researches more clearly, we also categorize related work. Table 3 shows the classification of optimization for sparsity.

4.2 Hardware-aware Software Design

4.2.1 Structured Sparsity. The unstructured network pruning can effectively remove zeros in the weight matrix/tensors, but it also causes the distribution of weights to become irregular and discontinuous [44]. At the same time, due to the rigid structure of the systolic array, most of the zero weight elements still need to be mapped to the PE array [17]. Therefore, for systolic arrays, modifying the distribution of nonzero values is more influential than minimizing the number of operations or the memory footprint [7]. This is the original intention of structured sparsity.
Table 4. Comparison of Different Granularity Types Structure Sparsity and Their Impact on Systolic Array Performance [7]

<table>
<thead>
<tr>
<th>Type</th>
<th>Element-wise</th>
<th>Shape-wise</th>
<th>Kernel-wise</th>
<th>Filter-wise</th>
<th>Channel-wise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Studies</td>
<td>[44, 98, 154, 158]</td>
<td>[22, 100]</td>
<td>[6, 22, 154]</td>
<td>[80, 100, 146, 154]</td>
<td>[6, 89, 100]</td>
</tr>
<tr>
<td>Granularity</td>
<td>$W(m, c, k, q)$</td>
<td>$W(m, c, k, q)$</td>
<td>$W(m, c, c, :)$</td>
<td>$W(m, c, c, :)$</td>
<td>$W(c, c, c, :)$</td>
</tr>
<tr>
<td>Pruning opportunity</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Index complexity</td>
<td>High</td>
<td>Medium</td>
<td>Medium</td>
<td>Low</td>
<td>Low</td>
</tr>
</tbody>
</table>

Compared with the fine granularity of unstructured pruning, the granularity of structured pruning is coarser [146]. In this way, it can ensure that the data distribution is regular and the calculation process of pruned DNNs is more friendly to the systolic array architecture. However, there are many granularity types of structured pruning, and not all of them can harness the full benefits (i.e., throughput and energy efficiency) of the systolic array [7]. Table 4 compares the common structured pruning methods in detail from various aspects and evaluates their effectiveness to be used for systolic arrays.

The element-wise pruning is the fine granularity of unstructured pruning, which chooses to remove all the zeros in the weight matrix/tensors without following any rules or shapes. It can take full advantage of the compression opportunities of the CNN network model, but it will also introduce a complex and huge indexing mechanism in the calculation process. It is difficult for systolic arrays to take advantage of the element-wise pruning.

Unlike the element-wise pruning, the shape-wise pruning selects a tile of elements in weight tensor to be rejected. Since the elements in the tiles are selected across FMAP channels, the pruned weight tensor can still maintain the matrix style after im2col transformation, which is beneficial for systolic array calculation. However, the shape-wise pruning will retain some 0 values in the weight, and the calculation process still needs to be guided by the index.

The kernel-wise pruning selects unimportant kernels from filters to prune, such as kernels with too many 0 values or too small average values. Although it is structured pruning, the pruned weight data cannot form a continuous matrix space, so it is still unfriendly to systolic arrays.

The filter-wise pruning and channel-wise pruning choose to delete the weight elements in the unimportant FMAP channels. These methods may result in many zeros being retained but are efficient for the systolic array, because a regular weight matrix can be maintained while the computation process does not even require index guidance [95].

Lym et al. propose the Prune Training algorithm, which belongs to the “Channel-wise” pruning scheme and prunes unnecessary channels of weight during the training process to shorten training time [95]. To make a systolic array efficient for pruning and training, Lym et al. propose FlexSA, a flexible systolic array architecture [96]. FlexSA dynamically reconfigures the structure of the systolic array and provides various sub-systolic operating modes. These modes are specifically designed to enable efficient processing of tensors with different sizes and shapes, optimizing energy.
consumption and memory bandwidth utilization. Based on the evaluation, FlexSA improves the PE utilization rate of pruning and training modern CNN models by 37% compared with a conventional training accelerator with a large systolic array.

Some solutions can further optimize these structured pruning algorithms for systolic array designs. Soltaniyeh et al. propose the “group-wise” pruning scheme [129], a fine-grained version of “shape-wise” in Table 4. Specifically, they choose to zero the weights that are below the threshold in some but not all elements of a “shape.” This generates zero blocks of a certain size (i.e., the number of filters in the group). Further, they optimize the systolic array structure and proposed the SPOTS accelerator to support the new algorithm, which improves performance by effectively mapping the sparse data to the PE array by utilizing sparsity in both input feature maps and weights.

Guo et al. propose the “tile-wise” sparsity pattern [144], which maintains a regular pattern at the tile level for efficient execution but allows for irregular, arbitrary pruning at the global scale to maintain the high accuracy. Specifically, the weight matrix is divided into several tiles along with the columns, and then entire rows and columns of the tile can be pruned. In this way, the pruned weight matrix is still dense and can be directly sent to the systolic array for calculation without introducing complicated control and multiplexer (MUX) units [144].

4.2.2 Column Packing and Conflict Pruning. Although the DNN models with a structured sparsity can be mapped to the systolic array efficiently, the number of nonzero weights after the structured pruning is usually larger than the fine-grained pruning if the same level of accuracy is maintained [17, 36, 67]. For example, 4.1M nonzero weights are preserved after the unstructured pruning [67] on the ImageNet dataset [121], whereas 23.2M nonzero weights are kept after the structured pruning [89].

Column packing and conflict pruning can maintain the advantages of structure pruning while improving the sparsity of the CNN model [67]. The first step is to perform the unstructured pruning. Then, different weight columns are grouped, and one group can be mapped to a single column of the systolic array [17]. So, the columns in the group should be merged, which means that only one non-zero data in each row will be preserved [67]. When the systolic array is running, the input data is also grouped and sent to the corresponding columns of the systolic array. Each PE selects the input data according to the index and multiplies it with the stored weight [17]. Since the partial results are accumulated along the rows, column packing will not cause any error or hardware overhead to the accumulation process.

This strategy turns the unstructured pruned weight matrix into a dense form, but the second step will cause conflict pruning [17, 46]. Due to the large scale of CNN’s weight data, there are usually hundreds of rows in the weight matrix. When combining the columns, there will be a large number of non-zero values in the same row. These non-zero values are called conflicts between the columns. To conduct the column packing, a certain number of conflicts will be allowed. For each group of conflicts, only the data with the largest magnitude is saved, and the rest are pruned [17]. This compression method is illustrated in Figure 13. Three conflicts exist in the column group, and the non-zero weights −3, 7, and −1 will be pruned during conflict pruning. After mapping the column group to the systolic array, the appropriate input will be selected at each node to compute with the stored weight.

Kung et al. first propose the column packing and conflict pruning [67]. By combining multiple sparse columns of the filter matrix into a single dense column stored in the systolic array, the utilization efficiency of the systolic array can be greatly improved. Besides, their design uses 8-bit quantization to represent the data in the systolic array and replaces the parallel MAC unit of the PE with the bitwise multiplier to achieve a higher density of computing unit. Compared with the traditional systolic array design, it can reach up to 8× efficiency due to the increased density of nonzero weights in the resulting packed filter matrix [67].
Fig. 13. The process procedure of the Column Packing and Conflict Pruning method. There are three conflicts in the column packing of this weight matrix, and the one with the largest magnitude is kept and the rest are pruned.

Fig. 14. (a) There are two pairs of conflicts in the original column packing process. (b) But partition-wise column packing divides the weight matrix into two partitions and reduces the conflicts to one pair. (c) And after row permutation (swap the data of the fourth row of the first partition and the second row of the second partition), there is no conflict in the column packing process.

But experimental results show that the compression rate of the column packing and conflict pruning algorithm is still not enough [17, 46]. For example, 0.3M nonzero weights are preserved after conflict pruning with an accuracy of 92.9% on the CIFAR-10 dataset [64]. However, the efficient unstructured pruning can reduce the number of nonzero weights to 0.13M in the same neural network with an accuracy of 93.10% [158].

He et al. propose the Sparse-TPU based on the work of Kung et al. [46]. Their design uses an optimized column combining algorithm that divides the workload into several partitions. Then it performs partition-wise column packing to reduce data conflicts between columns, as shown in Figure 14(b). In addition, He et al. design the collision-aware algorithm to accommodate more data in the column when conflicts occur. The MAC unit in the PE is also designed to support integer and floating-point calculations to support more DNN applications [46].

Chen et al. further improve the column combining algorithm and propose Tight Compression [17]. In addition to partitioning the weight matrix, row permutation is also performed. In this step, it uses the simulated annealing algorithm to adjust the order of the rows in the weight partitions to achieve a column combining scheme with the minimal data conflict [17], as shown in Figure 14(c). At the same time, it is more flexible and allows the partitions to have a different number of columns. It also performs the data quantization in their design, and the column can contain two types of data, 8-bit and 4-bit, so the compression rate is further improved [17].
4.2.3 Density-bound Block. Another way to optimize DNN sparsity is the Density-bound block (DBB). Kang et al. first propose the DBB sparsity strategy [61]. It essentially divides the input/weight tensor into blocks and sets the upper bound of the number of non-zero (NNZ) elements in each block. Figure 15 shows the difference between the DBB sparsity and other types of sparsity. Similar to structured sparsity, DBB performs block-wise pruning, but performs unstructured pruning on the elements within the block. In addition, DBB sparsity constrains the maximum number of NNZs in a block such that the maximum workload is known at the design time. Then these blocks are compressed, and only NNZs and the index mask are saved [91].

DBB offers two advantages: First, it resolves load imbalance and eliminates distributed accumulator issues associated with unstructured sparsity, thus obviating the need for energy- and area-intensive buffers [91]. Second, DBB enables a straightforward hardware design by leveraging the movement of compact DBB data blocks. As a result, the DBB algorithm finds extensive application in GPUs and other accelerator architectures, including systolic arrays [92].

In short, all of the work discussed above are hardware-aware software design schemes, that is, the DNN applications need to be modified, such as modifying the pruning algorithm and retraining the neural network, and the systolic array architecture also needs to be optimized at the hardware level. The experimental results prove that these researches and designs can achieve high performance/efficiency, but it is difficult to deploy and not suitable for cross-application scenarios due to the need of application modifications.

4.3 Architectural Technique

4.3.1 Simultaneous Multithread. There is a lot of related research on architectural techniques for DNN sparsity. An effective approach to exploit zero operands in the PE is to clock gate the input operand registers to reduce datapath toggling and therefore the dynamic power consumption [19, 91]. However, this method will stall the PE to wait for the non-zero elements and effective calculations to arrive.

An alternative is the traditional sparse linear algebra, which involves storing and computing with explicit indexes that accompany the non-zero data. For example, Han et al. implement a sparse matrix-vector accelerator for fully connected layers [43], and Parashar et al. implement index-based sparse CNN layers [109]. However, index-based approaches have significant overheads for storing and computing on the indexes themselves [91] (e.g., a 10-bit index for each 16-bit element in Reference [43]). At the same time, this method still faces the problem of irregular data distribution and discontinuous memory addresses, which is not suitable for the systolic array architecture.

To solve these problems, Shomrom et al. propose the synchronous multithreaded systolic array (SMT-SA) to process multiple sparse matrices simultaneously [128], which is similar to the simultaneous multithreading on CPUs. Figure 16 shows the structure of the SMT-SA [128]. The PE in the systolic array receives more than one input pair (in this context, a thread), allocates one
non-zero-valued thread for its MAC unit in each cycle, and bypasses all of its zero-valued threads. Non-zero-valued threads are kept in the PE (in a finite-sized queue) to be chosen in one of the subsequent cycles. The evaluation results show that compared with conventional systolic arrays, FP16-based 4-thread SMT-SA achieves up to 3.6× speedup with 1.7× area overhead [128].

4.3.2 Data Quantification. In addition to exploring the sparsity of workloads (weights, activation) in DNNs, another research field of DNN sparsity is data quantification [41], which is used to explore the bit-wise sparsity of the value. Gupta et al. propose the limited numerical precision scheme [41]. They replace the floating-point computations with the low-precision fixed-point computations in DNN training. In addition, they design the stochastic rounding scheme [41], which plays a crucial role in determining the network’s behavior during training.

This optimization can also be used in the design of systolic array to realize the architecture optimization. For example, the PE of the systolic array can adopt the multiplier and accumulator with lower precision [41]. Generally, to ensure sufficient accuracy, the weight value and input characteristic value of the incoming pulsating array are specified as 8-bit, which means that the accuracy of the result calculated by the computing unit is 16-bit. Therefore, each PE consumes 8-bit of data bandwidth when transmitting weight or input feature data and 16-bit of bandwidth is required to calculate the inter-transmission of intermediate values or the calculation results.

Gupta et al. also design a novel systolic array accelerator, which is more efficient than CPU and GPU platforms when training DNN networks. Nowadays, their method is widely used, and the data quantization has changed from 16-bit to 8-bit [59, 60, 142], 4-bit [112], or even 1-bit [29, 117] low-precision data representation.

In summary, the architectural techniques scheme has no requirements for application scenarios and no need to modify the application. However, compared with the hardware-aware software design, the performance improvement is limited, and the hardware overhead of the systolic array with the architectural techniques is also larger.

5 DESIGN AUTOMATION AND EVALUATION STRATEGY

5.1 Overview

As introduced in Section 2.4, designing high-performance systolic arrays is never an easy task. It requires the expert knowledge for both the target application and the hardware [141]. To reduce the design difficulty, shorten the design cycle, and improve the design productivity, many solutions have been proposed [28, 37, 141, 145]. To show these research works more clearly, Table 5 shows their classification.

5.2 Design Automation Tools

5.2.1 Code Templates. The design automation tools of the systolic array can be realized based on HLS tools and code templates. Wei et al. propose an automated systolic array architecture
Table 5. The Classification of Design Automation and Modeling and Simulation Strategies

<table>
<thead>
<tr>
<th>Type</th>
<th>Technology</th>
<th>Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Automation</td>
<td>Code templates</td>
<td>[31, 37, 55, 102, 145]</td>
</tr>
<tr>
<td></td>
<td>Space-Time Transformation</td>
<td>[10, 55, 56, 71, 74, 76, 101, 116, 131]</td>
</tr>
<tr>
<td></td>
<td>Polyhedral model</td>
<td>[11, 28, 82, 84, 110, 139, 141]</td>
</tr>
<tr>
<td>Modeling and Simulation</td>
<td>Roofline model</td>
<td>[18, 20, 60, 133, 147, 155]</td>
</tr>
<tr>
<td>Strategies</td>
<td>Analytical models</td>
<td>[122, 150]</td>
</tr>
<tr>
<td></td>
<td>Software simulator</td>
<td>[3, 72, 93, 108, 122, 149]</td>
</tr>
</tbody>
</table>

synthesis method for FPGA platforms [145], which generates a systolic array design with the help of HLS tools by configuring OpenCL code templates. It first explores the design space according to the input CNN load (in C language). Then it automatically configures the OpenCL code template according to the exploration results and generates kernel-code and SDK. Finally, it implements the kernel-code or SDK to the FPGA platforms to complete the process of automated systolic array architecture synthesis [145].

Genc et al. use agile hardware design and generator methodologies to propose the Gemmini, which reconfigures the code template by manually/automatically specifying parameters to generate a systolic array design [37]. It is written in the Chisel [8] hardware description language, enabling parameterization and configurability through high-level meta-programming and functional programming abstractions. Gemmini produces instances of systolic architectures that can be integrated with the Rocket Chip SoC generator [37].

These methods construct systolic arrays by configuring code templates. However, the internal structure of the systolic array used by the template is fixed, and it is difficult for users or designers to apply some optimization methods for the systolic array design. Moreover, these designs lack an effective strategy of design space exploration and cannot explore the entire design space [93].

5.2.2 Space-Time Transformation. An alternative is the STT-based systolic array generator. The STT (Space-Time Transformation) [10, 71, 76, 101, 116] serves as the foundation for automatic systolic array synthesis, as well as a description method of systolic array dataflow. For tensor applications, the dataflow is represented in two aspects: (1) space-mapping—the PE where a loop instance is executed; and (2) time-mapping—the execution sequence of loop instances in the PEs [141]. The STT applies loop transformations to the target workloads and assigns new semantics (space and time) to the generated loops. Specifically, the STT transforms a loop instance into a space-time vector for hardware execution using a transformation matrix [56]. The space vector represents the distribution of loop instances across the PE array at a specific moment, while the time vector specifies the order in which each PE performs computations on the loop instances [74, 141].

For example, Figure 17 shows the loop iterations perform matrix multiplication: \( O[n, m]^+ = I[n, c] \times W[c, m] \). The systolic array, resembling a hypercube, executes the computation in a spatial vector \( \bar{p} \) and a temporal scalar \( t \) [56].

The STT provides designers with a powerful tool to map and schedule the computation of systolic arrays and to describe their dataflow. The relationship between the STT and systolic array dataflows can be obtained through \( [\Delta \bar{p}, \Delta t]^T \) [56], as shown in Table 6. In this notation, a loop instance (or data element) is represented as a blue dot with coordinates \( [x, y, t] \). The space vector \( [x, y] \) indicates the location of the processing element (PE) where the instance is computed, while the time vector \( [t] \) indicates the time when the computation occurs.

A stationary dataflow occurs when the space vector \( [x, y] \) remains constant and only the time vector \( [t] \) increases. Here, the instance or element stays within a fixed PE and participates in
A Survey of Design and Optimization for Systolic Array-based DNN Accelerators

Fig. 17. The STT transforms the 3-nested loop to the time- and space-loops and maps these loop instances to the PE array.

Table 6. Dataflow Analysis with STT [7]

<table>
<thead>
<tr>
<th>Dataflow Type</th>
<th>Stationary</th>
<th>Systolic</th>
<th>Multicast</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[\Delta \vec{p}, \Delta t]$</td>
<td>$[\Delta \vec{p} = 0, \Delta t \neq 0]$</td>
<td>$[\Delta \vec{p} \neq 0, \Delta t \neq 0]$</td>
<td>$[\Delta \vec{p} \neq 0, \Delta t = 0]$</td>
</tr>
</tbody>
</table>

multiply-accumulate operations without transferring to other PEs. In a broadcast or multicast operation, the time vector $[t]$ of an instance or element remains constant, but there are multiple values of $[x, y]$. This allows the instance or element to be simultaneously transmitted to multiple PEs in the array. In a systolic dataflow, the space vector of an instance or element changes to the adjacent coordinate point ($[x+1, y]$ or $[x, y+1]$) as time increases ($[t+1]$). This enables PEs to reuse data with neighboring PEs during computation. A systolic array, particularly a two-dimensional design, combines stationary and systolic dataflows. Exploring the dataflow of systolic arrays can be accomplished using different transformation matrices $T$ in the STT [141].

Based on the SST, Lai et al. propose the SuSy, a programming framework that enables programmers to productively build high-performance systolic arrays on the FPGA platform [74]. With SuSy, programmers can apply the STT and several other memory and I/O optimizations to build a highly efficient systolic architecture productively. In addition, the input program is specified in the SuSy DSL, which is composed of (1) an algorithm (or temporal definition) expressed in uniform recurrence equations (UREs) and (2) decoupled spatial optimization. Then, the compiler produces the HLS code (in OpenCL) as output, which is eventually compiled to the bitstream for FPGA execution [74]. Experimental results show that SuSy can describe various algorithms with UREs and generate high-performance systolic arrays by spatial optimizations [74].

However, the SuSy requires programmers to analyze the systolic array execution pattern from the algorithms and specify the necessary transformation required to generate the systolic array [28, 74, 141].

5.2.3 Polyhedral Model. Cong et al. propose PolySA, a polyhedral-based systolic array compilation framework for FPGA platforms [28]. It enables end-to-end compilation by accepting high-level language workloads. The framework utilizes the polyhedral model to optimize performance.
through intricate loop nest restructuring. It employs a parametric polyhedron as the internal representation (IR) and conducts space-time transformation to map the IR to a systolic array. Ultimately, PolySA generates systolic array designs in high-level synthesis (HLS) codes [28].

PolySA is the first fully automated compilation framework for generating high-performance systolic array architectures on the FPGA leveraging recent advances in high-level synthesis [28, 141]. However, PolySA is limited in its generality, as it only supports the program with a single statement in perfectly nested loops. Besides, PolySA is limited in its implementation, which is built upon Matlab and is unscalable in handling complex designs [141]. As mentioned in its paper, it takes up to 23 minutes to perform the polyhedral transformation on a single layer of CNN [141].

Wang et al. propose AutoSA, an upgraded version of polySA, which is an end-to-end compilation framework for generating FPGA-based systolic arrays [141]. AutoSA utilizes the polyhedral framework and incorporates optimizations such as SIMD vectorization, double buffering, and others to enhance performance. It employs STT and conducts efficient design space exploration to discover high-performance designs. Experimental results demonstrate that AutoSA achieves impressive performance in a short time. For instance, on Xilinx Alveo U250, AutoSA achieves 934 GFLOPs, 3.41 TOPs, and 6.95 TOPs in floating point, 16-bit, and 8-bit integer data types, respectively [141].

Jia et al. propose TensorLib, a versatile framework for generating spatial hardware accelerators, including systolic array designs [56]. TensorLib employs tensor algebra expressed through nested loops and the transformation matrix $T$ as input. It utilizes the Space-Time Transformation (STT) technique to explore various dataflows, compactly representing the hardware dataflow with a transformation matrix. Common structures across different dataflows are identified, and parameterized hardware module templates are constructed using Chisel [55, 56]. Subsequently, Tensorlib selects the required hardware modules for each dataflow, connects them using a specified interconnection pattern, and automatically generates a complete hardware accelerator design. Experimental results demonstrate that TensorLib can generate hardware designs with different dataflows and achieve a 21% performance improvement on FPGA compared to state-of-the-art approaches [56].

5.3 Modeling and Simulation Strategies

At the end of the design flow, every new systolic array design needs to be evaluated, which is a very important reference for the designers. The designer must adjust and modify the systolic array design based on the evaluation results. However, the cost of using traditional evaluation methods such as using EDA tools or FPGA platforms to simulate the design is too high. So, many studies propose new modeling and simulation strategies and tools.

5.3.1 Roofline Model. To quantify the impact of insufficient bandwidth on performance, some research work adapts the well-known roofline model [147] for the analysis of systolic array designs. Figure 18 shows the roofline model, which is a tool that visualizes the performance of the architecture with different operational intensity [18]. It assumes that the PE array does not have enough on-chip memory to store the entire workload, and therefore its performance can be limited by insufficient bandwidth between the core and the memory [20].

In Figure 18, the Y-axis represents performance measured in operations per second, illustrating the peak computation rate as the “flat” part of the roofline. However, the X-axis represents operational intensity, which is quantified as operations per DRAM byte accessed [60]. The memory bandwidth is measured in bytes per second, thus forming the “slanted” part of the roofline, since \( \text{OPS/sec} / \text{OPSBYTE} = \text{Bytes/sec} \). When the operational intensity is below the inflection point, the performance is limited by memory bandwidth, as exemplified by workload 2 in Figure 18. Conversely, when the operational intensity surpasses the inflection point, the performance is limited.
by computation, as depicted by workload 1 in Figure 18. The roofline serves as an indicator of the upper bound of performance, with actual workloads typically falling within the area beneath the roofline [20, 60].

This simple visual model is not perfect, yet it offers insights on the causes of performance bottlenecks. Jouppi et al. first adapt the roofline performance model to analyze the TPU and compare their design with other platforms [60]. Chen et al. and Xu et al. also use the roofline performance model to assist their design processes [20, 151].

5.3.2 Analytical Models. Due to the simple structure and fixed dataflow of the systolic array, some researchers establish simple models to analyze the performance of the systolic array. For example, Xu et al. propose a mathematical model to analyze the PE utilization rate [150]. The PE utilization rate of the systolic array refers to the ratio of the number of activated PEs to the total number of PEs in the calculation process. This ratio can reflect the current computing efficiency and directly affect the performance and energy consumption of the systolic array.

Samajdar et al. also propose Systolic Array Simulator (SCALE-SIM) [122], which is based on these analytical models. In addition to the runtime and PE utilization rate, it can also provide other performance information such as bandwidth and power consumption. SCALE-SIM also exposes various micro-architectural features as well as system integration parameters to the designer to enable comprehensive design space exploration. This is the first systolic array simulator tuned for running DNNs, to the best of our knowledge [122].

However, SCALE-SIM is based on an ideal memory model and still lacks accurate simulation of data access, which seriously affects the accuracy of the evaluation results such as bandwidth and accelerator delay.

5.3.3 Simulators. There are also other simulators that can give the evaluation results of the systolic array design. These simulators are based on the technology of compiler optimization and simulate and analyze the process of mapping the workload to the spatial PE array, like the systolic array. The evaluation results generated by these simulators are more accurate, but there are still many differences between the simulators.

Muñoz-Martinez et al. present STONNE (Simulation Tool of Neural Network Engines), a cycle-level microarchitectural simulator for state-of-the-art rigid DNN inference accelerators, including the systolic array design [103]. STONNE uses a simulation engine to implement all the major components required to build both rigid and flexible DNN accelerators. Users can plug STONNE into any high-level DNN framework as an accelerator device and perform a full-model evaluation of both dense and sparse real, unmodified DNN models.

Parashar et al. propose Timeloop, an infrastructure for evaluating and exploring the architecture design space of DNN accelerators, including the systolic array design [108]. Timeloop
uses the compute-centric directive, a concise and unified notation, to represent the different architecture and implementation attributes of DNN accelerators to describe a broad space of hardware topologies. The compute-centric notation provides great flexibility for describing how the computation is performed in an imperative programming style [56, 108, 153]. It can then emulate those topologies to generate an accurate projection of performance and energy efficiency for a DNN workload through a mapper that finds the optimal dataflow for the specified architecture. This enables fair comparisons across different architectures and makes DNN accelerator design more systematic [108].

Kwon et al. also introduce a set of data-centric directives to concisely specify the DNN dataflow space in a compiler-friendly form, including the systolic dataflow [72]. They also propose the **Modeling Accelerator Efficiency via Spatio-Temporal Reuse and Occupancy (MAESTRO)**, which can analyze these directives to infer various forms of dataflow and exploit them using hardware design, including the systolic array [72]. It estimates various cost-benefit tradeoffs of a dataflow including execution time and energy efficiency for a DNN model and hardware configuration.

The compute-centric notation uses the loop transformation directives, such as reorder, blocking, and parallel, to describe the dataflow [108, 153]. Meanwhile, the loop order in the notation determines the loop instance execution sequence. For example, in Figure 19(a), the parallelism is specified using the parallel directive. However, the workload assignment requires extra directives [93].

The data-centric notation uses data mapping directives including the spatial and temporal map, data movement order [72, 93]. It explicitly allocates data to PEs with two key primitives, including Spatial Map and Temporal Map. The spatial map assigns the data along a certain dimension to the PE array, and the temporal map specifies the data movement across different timestamps. For example, in Figure 19(b), spatial map (1,1) i means assigning one element of the data in dimension i to each PE, and the offset along the spatial dimension is one. Temporal map (1,1) j distributes the data in dimension j across different timestamps within the same PE, and the offset is one [93].

However, to use this notation, the users must manually write the directives, which is not straightforward for complex dataflows. What is more, both compute- and data-centric notations are fundamentally limited in their expressiveness and performance-modeling capability. First, both notations fail to support some features of the systolic array dataflow, such as skewed data access. For example, in Figure 19, these two notations can only describe dataflows using rectangle-like data access, lacking the support for complex dataflows with skewed data access [93].

Second, compute-centric notation-based models only analyze data reuse opportunities in a coarse-grained manner, and the simulator with data-centric notations uses simple polynomials to estimate data reuse, which is less precise, as it calculates the data movements using a simple polynomial. Neither of them can get the performance metrics of the systolic array design correctly [93].
for\(n=0;\ n<2;\ n++\)
for\(m=0;\ m<2;\ m++\)
for\(c=0;\ c<4;\ c++\)
\(S:\ O[n,m]+=[n,c]^tW[c,m];\)

(a) Workload

\[
\begin{align*}
\text{space-stamp} & : \{S[n,m,c]\rightarrow PE[n,m]\} \\
\text{time-stamp} & : \{PE[n,m]\rightarrow O[n,m]\}
\end{align*}
\]

Interconnect
\[
\begin{align*}
\text{space assignment of O} & : \{PE[n,m]\rightarrow PE[n,m+1]\} \\
\text{Time assignment of O} & : \{PE[n,m]\rightarrow PE[n+1,m]\}
\end{align*}
\]

(b) Relation-centric notation

(c) Mapping

Fig. 20. (a) Workload. (b) Relation-centric notation transforms the loop iteration to the space/time-stamp and can describe the data assignment and interconnection. (c) Using relation-centric notation, TENET can accurately simulate the data mapping of the OS dataflow in the systolic array design.

So, to solve these problems, Lu et al. introduce a relation-centric notation, which formally describes the hardware dataflow for tensor computation [93]. The relation-centric notation specifies the hardware dataflow, PE interconnection, and data assignment in a uniform manner using relations.

Figure 20 shows an example of mapping GEMM on a systolic array with relation-centric notation [93]. In this example, \(S[n, m, c]\) is a loop instance of the workload in Figure 20(a). It is executed on \(PE[n, m]\) (space-stamp) and is assigned a one-dimensional timestamp \(t(i + j + k)\). So, the space/time-stamp in the relation-centric notation is an affine transformation of loop iterators.

The relation-centric notation can also describe the data assignment and PE interconnection [93]. Figure 20(b) also shows the data assignment of tensor \(O\) in relation-centric notation. It means that each PE always calculates the same output tensor \((Y)\) at different timestamps, which is the data assignment of the OS dataflow in the systolic array design.

With the relation-centric notation and space and timestamp constraints, the TENET can accurately simulate the data mapping process of arbitrary dataflows and can track the calculation of loop instances in the systolic array design, as shown in Figure 20(c).

Lu et al. also propose the TENET, a framework that analyzes the relation-centric notation and models hardware dataflow of tensor applications [93]. The relation-centric notation is more expressive than the compute-centric and data-centric notations by using more sophisticated affine transformations. Specifically, it uses space and timestamps to describe and analyze the dataflow and each dimension of the space- or time-stamp can be a linear transformation of multiple loop dimensions [93].

Of course, if the evaluation tool is combined with the design automation tool, then it will have a better effect and can help designers to find the optimal systolic array design. By feeding the dataflow found by TENET to TensorLib, it can automatically generate the dataflow hardware written in Chisel [8, 55]. Overall, TENET with TensorLib achieves 37.4% and 51.4% latency reduction for CONV and GEMM kernels compared with the state-of-the-art [56, 93].
6 CONCLUSION AND FUTURE DIRECTIONS

In this review, we have presented the related work of the systolic array design and classified them according to their purpose and technical characteristics. According to the design goals, the study field of the systolic array-based DNNs accelerator can be divided into four categories: supporting sparsity, improving flexibility, design automation, and evaluation strategy. Then, every category is further divided into some subcategories, depending on their design methods and technical characteristics. In particular, supporting sparsity and improving flexibility are further divided into the hardware-aware software design method and architectural techniques. We also compare these designs in the same category and give some deep analysis. Of course, the technique of the systolic array design is not limited to the methods we presented, and a panoramic view of this fast-expanding field is rather challenging, thereby resulting in possible omissions. Therefore, this review serves as a pedagogical tool, providing researchers with insights into typical methods of the systolic array design. In practice, researchers also could use these guidelines to adopt the most suitable technique for their specific designs.

However, there are still many challenges, but we believe that these will be breakthroughs in the future design of systolic arrays. In the following, we simply provide an outlook on the problems to be solved and trends to expect in the future:

1) **Flexibility.** At present, most systolic array designs only focus on convolution or matrix multiplication in DNN workloads and ignore other operations [35]. The flexibility introduced in Section 3 only solves some problems related to special computing operations. In fact, other types of operations are needed in DNNs, such as transposition [81, 152], im2col [24, 57], encoding [51, 62, 148], and so on. If the DNNs accelerator based on the systolic array cannot handle these operations properly during the design process, then it is likely that these workloads will become the bottleneck of the accelerator performance. In fact, the systolic array itself can solve many operations, such as matrix transposition [50, 107] and encoding [65, 83, 126]. If these operations are incorporated into the systolic array-based DNN accelerators, then we believe that the performance of the accelerators can be further improved.

2) **Fast algorithms.** In the research field of the DNNs acceleration, another way to reduce the computational density of DNNs workload is the fast algorithms [77]. For example, on the GPU platform, users can use different fast algorithms (cuDNN) [21] including the FFT algorithm [99, 138] or the Winograd algorithm [77, 88] to accelerate the training and inferring DNNs. Experimental results also prove that these algorithms can effectively accelerate the convolution of DNNs. On FPGA and ASIC platforms, we can also see the DNN accelerators based on the fast algorithms [2, 88, 94], but few designs are related to the systolic array architecture. In fact, the systolic array can support discrete Fourier transformation [13, 14, 65], which means that it has the potential to be combined with these fast algorithms to design more optimized accelerator designs.

3) **Processing-in-memory.** A bottleneck or challenge usually encountered in accelerator design is memory bandwidth. As introduced in Section 5.3, due to insufficient bandwidth, the accelerator performance of many workloads is bandwidth-limited. Processing-in-memory [4, 38], which directly processes the data in the memory, can solve this problem. If the DNN accelerator can be designed and implemented with the processing-in-memory technology, then it will avoid a large number of data read and write operations between the accelerator and external memory and greatly reduce the energy consumption of data transmission. There are many research works on processing-in-memory [23, 33, 130], but the systolic arrays are rarely used for design. We believe this technology is beneficial for the systolic array design [15].
(4) **3D circuit.** 3D stacking technology is the use of stacking technology and through-chip interconnection to form a three-dimensional integration in the Z-axis direction of the chip [12]. Relevant studies have shown that by mapping systolic arrays onto 3D circuit structures instead of conventional 2D structures, it can accelerate CNN inference and reduce power consumption [66]. Kung et al. also propose a building block design using **through-silicon vias (TSVs)** for the 3D realization of systolic subarrays [69]. However, there are still too few 3D IC designs related to the systolic array architecture, and we believe that there is still a lot of research room for exploration.

(5) **Unified architecture.** As introduced in Section 3, to be more versatile and flexible, some work combines the systolic array structure with the GPU to form a unified architecture. But not all computing platforms can have GPUs, such as embedded terminals. Indeed, there are already many systolic array accelerators designed for the embedded platform, but they are only used as slave processors and require a lot of data communication with the host. If a unified CPU-systolic array architecture [51] for the embedded platform can also be realized, then communication problems can be solved, and a more versatile and flexible DNNs accelerator can be designed.

**REFERENCES**


A Survey of Design and Optimization for Systolic Array-based DNN Accelerators


A Survey of Design and Optimization for Systolic Array-based DNN Accelerators


A Survey of Design and Optimization for Systolic Array-based DNN Accelerators


Received 30 May 2022; revised 16 March 2023; accepted 31 May 2023