This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

An Efficient Hardware Design for Accelerating Sparse CNNs with NAS-based Models

Yun Liang, *Senior Member, IEEE*, Liqiang Lu, Yicheng Jin, Jiaming Xie, Ruirui Huang, *Member, IEEE*, Jiansong Zhang, *Member, IEEE*, Wei Lin

***Abstract*—Deep convolutional neural networks (CNNs) have achieved remarkable performance at the cost of huge computation. As the CNN models become more complex and deeper, compressing CNNs to sparse by pruning the redundant connection in the networks has emerged as an attractive approach to reduce the amount of computation and memory requirement. On the other hand, FPGAs have been demonstrated to be an effective hardware platform to accelerate CNN inference. However, most existing FPGA accelerators focus on dense CNN models which are inefficient when executing sparse models as most of the arithmetic operations involve addition and multiplication with zero operands.**

In this work, we propose an accelerator with software-hardware co-design for sparse CNNs on FPGAs. To efficiently deal with the irregular connections in the sparse convolutional layers, we propose a weight-oriented dataflow that exploits element-matrix multiplication as the key operation. Each weight is processed individually which yields low decoding overhead. Then we design an FPGA accelerator that features a tile look-up table (TLUT) and a channel multiplexer (CMUX). The tile look-up table is designed to match the index between sparse weights and input pixels. Using TLUT, the runtime decoding overhead is mitigated by using an efficient indexing operation. Moreover, we propose a weight layout to enable efficient on-chip memory access without conflicts. To cooperate with the weight layout, a channel multiplexer is inserted to locate the address. Last, we build a Neural Architecture Search (NAS) engine that leverages the reconfigurability of FPGAs to generate an efficient CNN model and choose the optimal hardware design parameters. Experiments demonstrate that our accelerator can achieve 223.4-309.0 GOP/s for the modern CNNs on Xilinx ZCU102, which provides a 2.4X-12.9X speedup over previous dense CNN accelerators on FPGAs. Our FPGA-aware NAS approach shows 2X speedup over MobileNetV2 with 1.5% accuracy loss.

***Index Terms*—FPGA, CNN, sparse, accelerator, NAS.**

I. Introduction

Inspired by the biological nervous system, deep learning has recently achieved remarkable accuracy improvement. Convolutional neural networks (CNNs), the most commonly used model in deep learning, have been adopted in various domains, including image and speech recognition [1–4]. The significant accuracy improvement of CNNs comes at the cost of huge computational complexity as it requires a comprehensive assessment of all the regions across the feature maps. Towards such overwhelming computation pressure, FPGAs have emerged as a promising solution due to their high performance, energy-efficiency, and programability [5–8].

In a typical CNN model, each neuron is regarded as a node in the network while the weight represents connections between nodes in two adjacent layers. Pruning the connections in the deep neural networks has been proved as an effective solution to compress the overall computation and memory requirements of these models while maintaining high accuracy. In general, compression techniques can be divided into two categories: unstructured compression and structured compression. The unstructured compression techniques prune the weights with irregularity in a fine-grain manner of pixels [9–11]. For example, Han et al. [9] have shown that there is significant redundancy (up to 90%) for certain DNNs, which can be pruned without sacrificing accuracy. The structured compression aims at pruning the networks with a certain shape in the weight [12–15]. However, the structured pruning often leads to a lower compression rate as it shows a strict mathematical formalization.

In this paper, we mainly focused on accelerating CNNs with unstructured compression on FPGAs. Our approach can also be applied to structure compression. Though pruning techniques theoretically reduce the number of operations in the convolution algorithm and potentially provide the opportunity for faster inference process, existing accelerators on FPGAs for dense models are not suitable for sparse CNN models. Most of these works optimize their dataflows based on loop operations like loop interchange and loop unrolling [16–20]. The dense accelerator can result in high hardware inefficiency since most multiplication operations involve zero operands [5, 6, 16, 21–25]. Implementation of sparse DNNs has been studied in recent years on FPGAs[26]. These accelerators mainly focus on the fully-connected (FC) layers, which use matrix-vector multiplication operations and are used for RNNs and LSTMs. However, the major operators of the modern CNN's computation are convolution operations. For example, the convolution operations occupy 90% of the total computation in GoogLeNet. Although the spatial convolution can be mapped to matrix-vector multiplications, this will increase the local memory requirement since the pixels in the input feature maps have to be copied multiple times when being flattened to a vector.

The challenges to design an efficient FPGA accelerator can be summarized as follows,

- The convolutional layers involve complex connections be-

Yun Liang, Liqiang Lu, Yicheng Jin and Jiaming Xie are with Center for Energy-efficient Computing and Applications, Peking University, Beijing, 100871 China. Yun Liang is also with Peng Cheng Laboratory, Shenzhen, China. Yun Liang is the Corresponding Author. (e-mail: {ericlyun,liqianglu,yicheng.jin,jmxie}@pku.edu.cn).

Ruirui Huang, Jiansong Zhang and Wei Lin are with Alibaba Group, Hangzhou, China (e-mail: {ruirui.huang, muduan.zjs, weilin.lw}@alibaba-inc.com).

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems


```
Weight: N×M×R×S
Output feature maps: N×H×W

for  h = 0 to H {
   for w = 0 to W {
      for n = 0 to N {
         for m = 0 to M {
            for r = 0 to R {
               for s = 0 to S {
                  Output(n,i,j) +=
                  Weight(n, m, r, s) ×
                  Input(m, i*stride+r, j*stride+s);
}}}}}}
```

Fig. 1: A typical convolutional layer

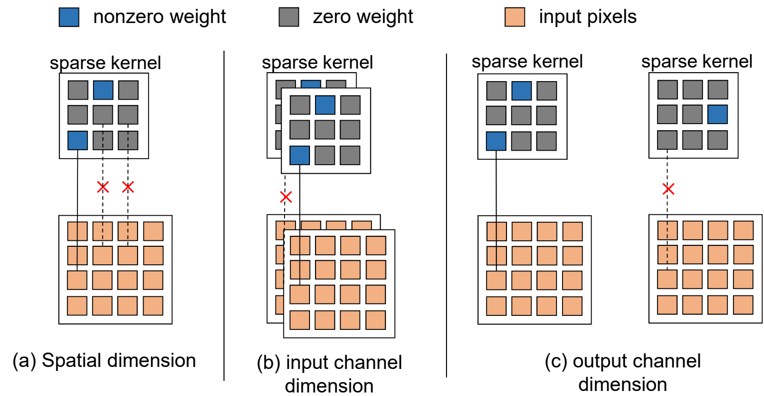


Fig. 2: Invalid computation caused by redundant connections in sparse CNNs

tween input feature maps and output feature maps for sparse CNNs. Clearly, each output pixel is connected with part of the input pixels through the sliding kernels. The connection becomes irregular when the network becomes sparse. It is difficult to design a dataflow to deal with the irregularity but can leverage the high parallelism of FPGA and maintain FPGA efficiency.

- The sparse weights are encoded in sparse format, which requires extra coordinate computation to locate the weights. However, the distribution of the sparse weights (non-zeros) is irregular, which leads to inefficient memory access and low on-chip bandwidth utilization.
- A CNN model generally consists of different types of convolutional layers. Given a specific architecture design, the performance can be different when the layer parameter changes. Therefore, it is challenging to design a hardware-friendly CNN model that can maximize the performance.

To address the first challenge, we propose a weight-oriented dataflow where each PE performs element-matrix multiplication instead of spatial convolution. Here, the element refers to the sparse weight and the matrix refers to the input tile. In this dataflow, the sparse weights are processed separately. By doing this, we successfully avoid the design issues related to sparsity such as irregular connections and load imbalance, etc. For the second challenge, we propose a weight layout, which can enable efficient on-chip memory access of the weights. In this layout, the weights processed in parallel are stored continuously, and the results are accumulated from different BRAM banks to avoid access conflicts. Moreover, we design an efficient accelerator for sparse CNNs that features a tile look-up table (TLUT) and a channel multiplexer (CMUX). TLUT can reduce the overhead of runtime index matching and CMUX helps to locate the output address easily when updating the results. Finally, we build a Neural Architecture Search (NAS) engine based on analytical models that are used to predict the latency and resource utilization. For a specific deep learning task, we use the engine to explore the design space and identify the optimal CNN model architecture with hardware design parameters.

A preliminary version of this paper was reported in [27]. In [27], we propose an architecture design for accelerating sparse CNNs on FPGAs. In this article, we extend previous work with software-hardware co-design to further improvement the performance. In particular, we propose a FPGA-aware NAS framework to search for the optimal hardware design parameters and network architectures simultaneously. We perform architecture search on ImageNet and draw comparisons with several state-of-the-art hand-crafted and auto-designed models.

In conclusion, this work makes the following contributions,

- We propose a dataflow with element-matrix multiplication as the key operation, where the element and the matrix refer to the sparse weight and input tile, respectively.
- We propose an architecture design for the dataflow with a set of optimization techniques. We use a look-up table to match the sparse weight with the corresponding input pixels. We also design the weight layout and compression format which can enable efficient on-chip memory access.
- We develop an analytical model to estimate the latency. This model considers different types of operators in modern CNNs, e.g., point-wise convolution, depth-wise convolution.
- We develop a NAS engine to automatically generate CNN model that can match our hardware design. This engine searches both hardware design parameters and possible CNN models under resource constraints, and outputs the CNN model with high accuracy and low latency.

Experiments demonstrate that our accelerator can achieve 309.0, 223.4, 291.4 and 257.4 GOP/s for VGG, Alexnet, Resnet-152 and GoogLeNet on Xilinx ZCU102, respectively. Our accelerator achieves a 2.4X-12.9X speedup over the previous dense CNN FPGA accelerators. Compared to TitanX GPU platform, our accelerator shows 7.56X energy-efficiency. Our FPGA-aware NAS approach shows 2X speedup over MobileNetV2 with 1.5% accuracy loss.

II. BACKGROUND

A. Sparse CNN model

CNNs are a class of deep, feed-forward artificial neural networks, which are composed of a series of layers including convolutional layers, pooling layers and fully-connected layers (FC layer). The convolutional layer is the most important layer in which the kernels extract features from the input feature map. Figure 1 shows the typical convolution operation. The convolution operation uses a small $R \times S$ kernel to slide through the input feature map. And the pixels inside the sliding window conduct a multiply-and-add operation with the

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

weights in the kernel to compute a pixel value in the $H \times W$ output feature map. There are usually many input feature maps (aka input channels) and output feature maps (output channels) in a single convolutional layer, and the numbers of input feature maps and output feature maps are M and N as shown in Figure 1, respectively. Note that the convolution results in the different input channels are accumulated to obtain the output channel results.

CNNs usually have a large number of weights, which could introduce the problem of over-fitting. The weights pruning techniques [9, 28] have been proven to be an effective method to reduce the computation and memory size while maintaining the overall model accuracy. For example, Deep Compression [9, 28] can reduce the number of weights in AlexNet [29] and VGG-16 [3] by 9X and 13X, respectively. These are known as unstructured pruning techniques. There are other pruning techniques that prune the weights with structured patterns [15, 30, 31]. The advantage of structured pruning techniques is they are hardware friendly. However, they often yield a low compression rate due to the strict mathematical formalization. The sparse CNN accelerators we propose can be used for both structured and unstructured pruning techniques.

B. Neural architecture search

Neural architecture search (NAS) aims at automating NN architecture design, in analogy to deep learning automating feature engineering. Generally, a NAS program consists of search space, cost functions and search algorithm. First, the search space generates a concrete NN architecture by combining different types of convolutional layers. Then, the NN architecture is evaluated by the cost functions which consider the accuracy, the network size and the execution latency on a target platform. As for the algorithm, the earliest NAS algorithms train it from scratch on the whole dataset using a controller recurrent neural network (RNN) [32, 33]. However, these approaches mainly focused on the model accuracy without the consideration of the execution latency. Besides, they are prohibitively computation-intensive and are limited to small datasets and cell-level search spaces. Recently, there are NAS works introducing the hardware latency in the cost function [34–36]. However, these approaches only target a fixed hardware architecture like GPU platforms and mobile phones. And the search algorithm does not take the hardware reconfigurability into account, which cannot be applied to FPGA platform.

Different from previous NAS approaches, we take full advantage of the flexibility of FPGA design where we incorporate hardware parameters into our neural architecture search framework. More concretely, we build a resource model to estimate the FPGA resource utilization with architectural parameters and a latency model with both convolution parameters and architectural parameters. In this manner, FPGA architecture design can be taken into account in the search space. Section VI will provide the details of our hardware-aware NAS approach.

TABLE I: Analysis of recent sparse CNN dataflow

dataflow	type	inner computation	Decoding overhead
SCNN[37]	pixel-oriented	Cartesian product	high
CambriconX[38]	kernel-oriented	vector dot product	medium
Cnvlutin[39]	pixel-oriented	vector multiplication & reduction	medium
SparTen[40]	pixel-oriented	vector dot product with inner join	high
Bit-Pragmatic[41]	bit-oriented	vector dot product with shifters	high
Ours	weight-oriented	element-matrix multiplication	low

III. FPGA DATAFLOW DESIGN

A. Dataflows for sparse CNNs on ASIC

There have been prior efforts on designing dataflows for sparse CNNs on ASIC platforms. However, these dataflows will be inefficient for FPGA platforms due to the distinct architectures. In Table I, we classify prior ASIC designs based on the inner computation of the dataflow. SCNN architecture [37] applies the pixel-oriented dataflow where the innermost computation is a Cartesian product. Using Cartesian product, this dataflow multiplies input pixels with weights and returns multiple partial sums. This method requires significant coordinates computation to locate the sparse weights. Besides, the partial sums are connected with different output pixels, which bring great challenges for pipelining on FPGAs due to complex data dependency. Cambricon-X [38] design applies direct and step indexing technique to select the input pixels by detecting the nonzeros. Cambricon-X performs the vector dot product across channels by gathering the weights into a vector, which needs to dynamically select the input vector. This dataflow only performs parallel computation in channel dimensions, which also leads to poor parallelism on FPGAs. Cvnlutin design [39] leverages the sparsity in the input feature maps by using zero-skip computation. However, this dataflow requires runtime control to identify the nonzeros in the input pixels. Besides, the vector multiplication results are dynamically reduced to the output via an adder tree, resulting in high decoding overhead. SparTen [40] shares the similar inner computation to Cvnlutin. The difference is that SparTen applies an inner join scheme to gather the partial sums where the nonzeros are represented using a bit-mask. since the input sparsity depends on the results of the previous layer, such bit-mask requires to dynamically encode the nonzeros into bit-mask representation. Bit-Pragmatic [41] focuses on the sparsity in bit level. Based-on the bit-serial unit of Stripes [42], Bit-Pragmatic performs parallelized bit-serial multiplications, and gathers the partial sums via a reduction tree. However, the zero bit of input pixels cannot be determined off-line, which increases the logical overhead to detect nonzero bits in the input vector.

B. Dataflows for dense CNNs on FPGAs

There have been dense CNNs dataflows on FPGAs [6, 16, 21, 43, 44]. However, these dataflows will lead to invalid multiplications caused by the redundant connections between weights and input/ouput channels for sparse CNNs. As shown in Figure 2, the invalid multiplications can be from spatial

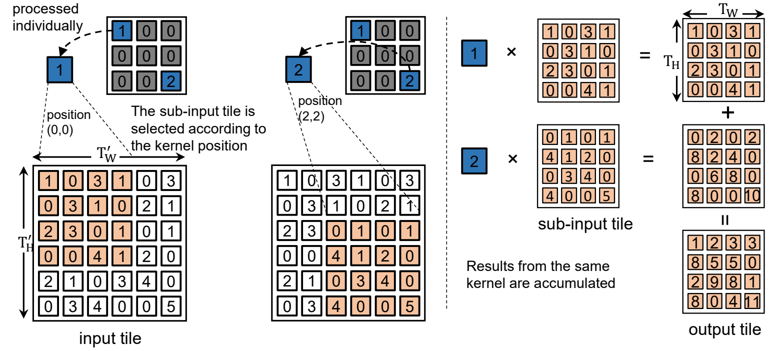
This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems


```
Sparse weight: SPw[M][N×R×S]
nonzero # in each input channel: NZ[M]

{L_M}for m = 0 to M {
{L_H}  for  h = 0 to H, h+=T_H{
{L_W}    for w = 0 to W, w+=T_W{            } step ❶
               get(in_tile, h, w, m);
{L_K}      for k = 0 to NZ(m), k+=T_N{pipelined
do in parallel:
{L_I}        for i = 0 to  T_N {
                 weight* = SPw(m, k)
                   n = weight*. n                } step ❷
                   out_tile = weight*⊙in_tile
{L_X}            for x = 0 to T_H{
{L_Y}            for y = 0 to T_W{               } step ❸
                       out(n, h + x, w + y) +=out_tile(x,y)
}}}}}}}
```

(a) Pseudo code of the dataflow



(b) Inner computation of the dataflow

Fig. 3: Weight-oriented dataflow

kernel, input channel, and output channel dimensions, respectively. The input feature maps share the same index with the weight in the spatial kernel dimension and in the input channel dimension. In other words, the input pixel whose index matches the weight is needed when convolving the input with the kernel. Besides, different kernels are connected to different output feature maps, and the zero weight will not contribute to the corresponding output feature map.

C. Our weight-oriented dataflow

We propose to transform the convolution computation to element-matrix multiplication by processing each weight as a single element. We compress the sparse weights into two arrays: (1) $SPw\ array$, where the nonzero weights in the same input channel are compressed into a vector. (2) $NZ\ array$, which records the number of non-zero weights in each input channel. One input channel is processed at a time. Figure 3 (a) shows the pseudo code of our dataflow which consists of three steps. In the **step 1**, we gather the necessary input pixels into an input tile according the position (h, w, m). A $T_H \times T_W$ tile in the output feature map is connected with $T_H \times T_W$ pixels in the input feature map through a specific weight. Given a specific kernel size and the sliding stride, a $T_H \times T_W$ tile corresponds to a $T_{H'} \times T_{W'}$ tile in the input feature map as follows,

$$T_{H'} = R + stride \times (T_H - 1), \quad T_{W'} = S + stride \times (T_W - 1) \tag{1}$$

where the kernel size is $R \times S$. Then, the input tile slides with a vertical stride T_H and a horizontal stride T_W as shown in Figure 3 (a). **Step 2** is the inner computation of our dataflow where T_N weights are multiplied with the input tile in parallel.

Figure 3 (b) presents the details of the inner computation in the weight-oriented dataflow. Based on the position of the weight, we select a tile of input pixels that are connected with the weight. More clearly, given an output tile, each weight corresponds to a certain sub-input tile determined by the position of the weight in the kernel. For example, the value '1' in the top-left corner of the sparse weight multiplies with all the 4×4 top-left tiles of input feature maps. The weights are from different output channels. Finally, the multiplication results will be accumulated the output pixels according to the index (n, h, w) in **Step 3**.

Our dataflow and its element-matrix multiplication inner computation has the following advantages. First, our dataflow processes the sparse weights one by one separately. By doing this, we can effectively exploit the sparsity and meanwhile reduce the sparsity decoding overhead. Second, our dataflow provides sufficient parallelism on FPGAs. More clearly, the output pixels in the spatial kernel and output channel dimensions are computed in parallel. Third, our dataflow has low data dependency overhead. The results from **Step 2** in Figure 3 are accumulated to different output pixels which have no read-and-write conflicts.

IV. Architecture Optimization

In Section III, we transform the convolution operation to element-matrix multiplication. However, implementing this dataflow on FPGA arises two challenges. The first challenge is to select the necessary pixels for a specific weight. A single weight is connected to only part of the pixels in the input feature maps, and the weight in the different position of the kernel is connected to different input pixels, as shown in **step 1** of Figure 3. Second, to ensure multiple results can be accumulated to the output buffer in parallel in **step 2**, a dedicated data layout is required under the hardware constraints of FPGA memory structure (e.g., dual-port BRAM). Furthermore, the PEs should be pipelined to increase the throughput.

A. Architecture overview

As shown in Figure 4, The input buffer contains four rows of feature maps. The output buffer size is set to store all pixels in one row of feature maps. our FPGA accelerator consists of T_N PEs with each PE has $T_H \times T_W$ multipliers. Each PE is connected with a tile look-up table (TLUT) to match the weight and the required input pixels (Section IV-B). In Section IV-C, we propose a novel weight layout where the parallel weights are stored continuously. Besides, the layout can ensure the results from the PE array are accumulated to different output banks without data access conflicts in the pipeline. To cooperate with the layout, in Section IV-E, we propose a channel multiplexer (CMUX) to locate the channel

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

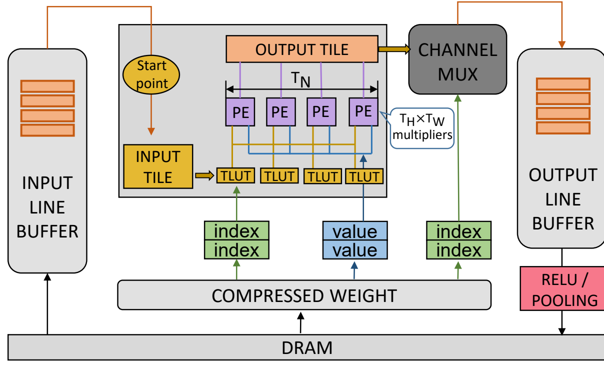


Fig. 4: Architecture overview

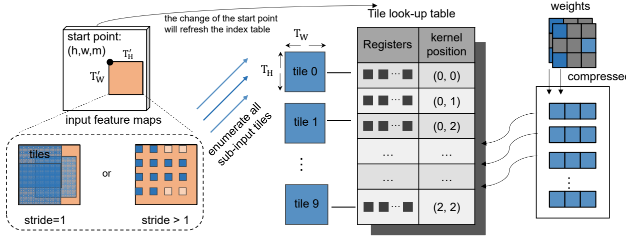


Fig. 5: Tile look-up table to locate the sub-input tile

address. The channel multiplexer receives the weight index in the sparse format and outputs which bank the results should be accumulated to. Since the weight distribution across output channels might be unbalanced, we analyze the load balancing problem in Section IV-F.

B. TLUT module

As aforementioned, the weights represent the connections between the input feature map and the output feature map. However, when the weight is sparse, the connection loses its structured topology. To bridge the gap between irregular connections to input pixels and the regular PE array, we insert a tile look-up table between the input tile and PEs. Figure 5 depicts how the weight and the input pixels are paired. When the kernel is sliding in the input tile, the weight in a $R \times S$ kernel is connected to a set of input pixels in the input tile. These pixels are batched together into a new tile. For example, in Figure 2, the position of the weight with value '1' is $(0, 0)$ which corresponds the top-left tile, and we can directly fetch the pixels from the TLUT module which has been pre-fetched when the start point (h, w, m) is determined.

There are $R \times S$ sub-tiles in total with $R \times S$ positions in the kernel. These tiles are stored in local registers in the TLUT module. As the PE array processes multiple weights in parallel, each PE has its own TLUT module. As shown in Figure 4, the input tile is reused by duplicating the pixel into multiple TLUT modules. And the weight is reused by multiple pixels in the input tile. The TLUT module replaces runtime index matching with a simple array indexing operation by introducing additional local registers. This helps to save the logic resources significantly since the runtime index matching requires a large number of multiplexers.

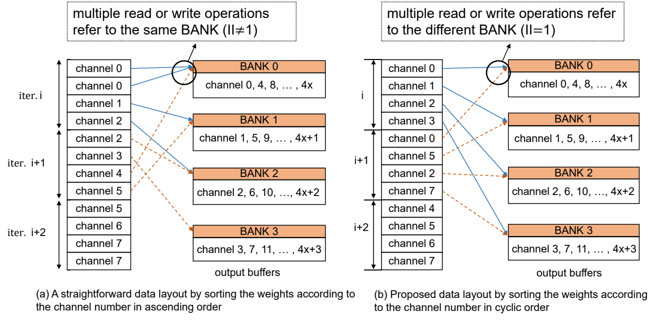


Fig. 6: Weight layout in the output channel dimension. II is the loop iteration interval of pipeline.

C. PE design and weight layout

The PE receives the decoded weight and the selected tile from the tile look-up table. We initiate a PE array with each PE conducting an element-matrix multiplication operation. In the **step 2** of our dataflow, we compute multiple output pixels from different output channels in parallel. There are T_N homogeneous PEs process multiple weights and input tiles in parallel. Furthermore, we apply pipelining technique to our PE design. Pipelining allows multiple operations in **step 2** to process concurrently to increase throughput, and the pipeling efficiency is determined by the iteration interval (II). According to Figure 3, the iteration interval is bounded by the weight access bandwidth and output access bandwidth.

To enable simultaneous update of multiple output channels, the output buffer is partitioned to $T_N \times T_H \times T_W$ banks where each bank i in the channel dimension stores the weights from the $n = (T_N \times x + i)$ output channel as shown in Figure 8. Traditionally, the weights are sorted in the ascending order of channels. If more than one weight need to be read from the same bank, this will lead to a long read latency. To address this problem, we rearrange the weight layout according to its remainder Re by dividing the output channel n with T_N ($Re = n \ mod \ T_N$), as shown in Figure 6, so that the results from the PE array are accumulated to the output buffers. For example, in Figure 6, 4 weights are processed in parallel. In our weight layout, the results from the PE array need to be accumulated to the output channel $(0, 5, 2, 7)$ in iteration $i+1$, whose remainder are $(0, 1, 2, 3)$. In this manner, multiple write operations refer to different banks, resulting in an improved iteration interval.

D. Sparse Format

To cooperate with the weight layout, we propose an *interleaved output channel coordinate list* (IOCOO) format to store the sparse weights. More clearly, weights in one input channel is stored in a vector. Each element is 5-tuple $(n, r, s, value, valid)$ which represents the indices and the value of the weight. These tuples are stored with different bit width as follows. Using this format, the compressed weight can be directly sent to PEs without decoding overhead, leading to a high PE bandwidth utilization.

tuples	r	s	n	valid	value
bit width	4	4	10	1	16

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

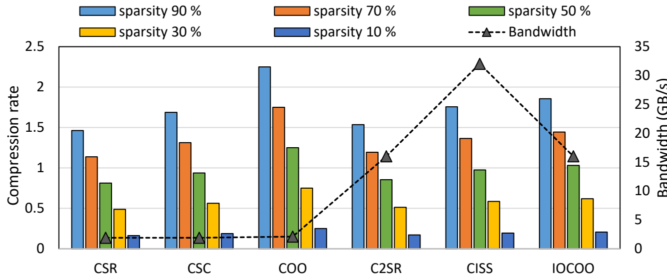


Fig. 7: Compression rate of different format. We assume the filter size is $1024 \times 1024 \times 3 \times 3$.

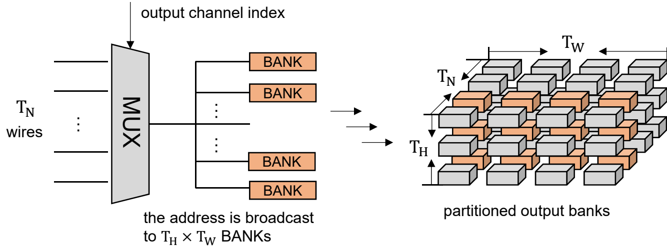


Fig. 8: Channel multiplexer to locate the output channel address

Figure 7 shows the compression rate of a few well-known formats such as CSR, CSC, COO, and recently proposed formats including C^{2}SR of MatRaptor [45] and CISS of Tensaurus [46]. For CSR, CSC, C^{2}SR formats, we flatten the weight into a $(1024 \times 3 \times 3) \times 1024$ matrix. The compression rate is given by

$$compression\ rate = \frac{the\ compressed\ data\ size}{dense\ data\ size}$$

In Figure 7, CSR has a lower compression rate because of the matrix shape. COO format shows a higher memory requirement as indices are stored individually. The compression rate of our approach is similar to CSC because of the same number of pointers. In CISS format, extra information is required to store pointers of higher dimensions as the weight is a 4D-tensor. C^{2}SR shares a similar idea to our design where each row is assigned a fixed channel in a cyclic manner to avoid memory conflict. Though our format requires a little higher memory, less logic resource is needed for decoding. For example, using CSR format, the spatial coordinate of the weight needs to be calculated according to the row pointer.

We also analyze the available bandwidth of different formats, as shown in Figure 7. The bandwidth is calculated with the assumption that the PE number is 8, and the frequency is 1GHz. Traditional formats, like CSR, CSC and COO, only have a single entry for compressed weights resulting low bandwidth. The bandwidth of CISS and IOCOO is similar since both of these two formats partition the weight according to the PE numbers. C^{2}SR has higher bandwidth. This is because each PE in MatRaptor[45] is responsible for multiple weights in one iteration. To enable enough bandwidth, C^{2}SR partitions the weight matrix into more pieces leading to higher bandwidth.

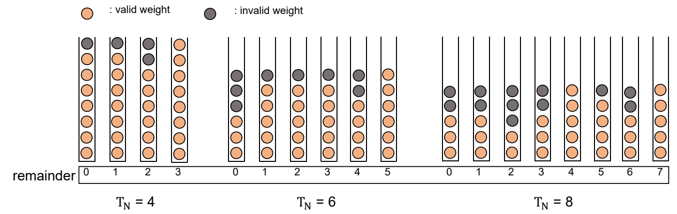


Fig. 9: Unnecessary computation under proposed weight layout

E. CMUX module

In the PE array, each PE generates a tile of results that belong to a distinct output channel. The address that the results need to be accumulated to is determined by the index in the format. A channel multiplexer is inserted between the PE array and the output buffer to locate the address as shown in Figure 8. The channel multiplexer consists of T_N input wires which represent the number of banks in the output channel dimension. The CMUX module will first compute the output address according to the remainder, e.g., the 1st input wire means its remainder is 1. And then the channel multiplexer will output which bank the results need to be accumulated.

F. Load balancing analysis

In our architecture, PEs strictly process T_N weights with different remainders together. However, the weights with different remainders cannot be evenly distributed. So we align the weights with unnecessary data among all the remainders so that the number of weights across different remainders is equal. There is a $valid$ signal in IOCOO format to indicate whether the weight is valid. As a result, the latency is always bounded by the remainder with the maximum nonzeros. The computation efficiency can be computed as follows,

$$Compute_{eff} = \frac{\#\ of\ valid}{\#\ of\ valid + \#\ of\ unnecessary} \quad (2)$$

In the example of Figure 9, the parallelism factor T_N is set to 4, 6, 8 with a fixed number of nonzeros 28. For example, when $T_N = 8$, the computation efficiency is $\frac{28}{28+12} = 70\%$. In the experimental section, we will analyze the computation efficiency using real networks.

V. Implementation details

A. Memory system

The on-chip memory of FPGAs is not large enough to hold all the channels of feature maps. Besides, there exist data reuse opportunities both horizontally and vertically since there is overlapping when the kernel slides across the input feature maps. Line buffer design is widely used in previous accelerators and can effectively reuse the input data [6, 21, 24, 47]. Hence, we apply line buffer design to load and calculate feature maps. We implement line buffer design using loop tiling techniques where the required data in tiled loops are stored in the on-chip BRAM. Different tiling strategies can lead to different data reuse opportunities. In our design, we choose to tile the loop in the channel dimension with factor K_M and K_N, as shown in Figure 3 (a). Because when the kernel sliding across feature maps, the relationship between

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

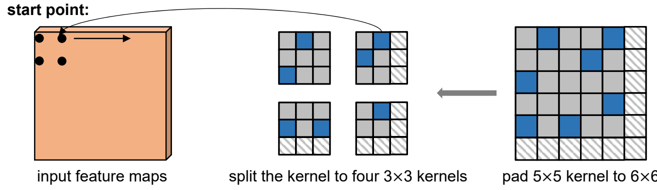


Fig. 10: An uniform design different kernel size

the data of different channels is irrelevant or independent. Assuming the sliding stride of convolution kernel is one, each input line buffer contains $K_M \times W$ elements, and each output line buffer contains $K_N \times W$ elements. To sustain sufficient on-chip bandwidth for PE computation, we partition each buffer according to the parallelization degrees. For example, each output line buffer is partitioned with factor T_W in width dimension and factor T_N in channel dimension.

The modules in the PE are also pipelined to increase throughput. Clearly, there are two input tiles working in a ping-pong manner to overlap the latency of the tile look-up table and the latency of the PE array. The latency of generating the tile look-up table can be regarded as a constant. The latency of the PE array depends on the loop count of L_K and the pipeline depth, as shown in Figure 3. In general, the latency of loop L_K is much larger than the latency of the tile look-up table, therefore our PE design can achieve high efficiency.

B. Implementation of other layers

In general, the modern CNN networks contain different kernel sizes. For example, Resnet has 1×1 and 3×3 kernels in the residual block, and GoogLeNet has 1×1, 3×3 and 5×5 kernels in the inception module. Since each weight is processed independently in our dataflow, our architecture can flexibly handle different kernel sizes. To unify the structure of the tile look-up table, we transform all the kernels to the 3×3 kernel. Figure 10 shows an example that transforms the 5×5 kernel to the 3×3 kernel. The 5×5 kernel is padded to 6×6 kernel with zeros then split into four 3×3 kernels. Apart from the convolutional layers, there are other layers in CNN models. In our architecture, we implement two widely-used layers: pooling layer and Rectified Linear Unit (ReLU) layer. Pooling layer outputs the maximum values in sub-regions of input feature maps. ReLU layer sets any input value less than zero to zero. These two layers are implemented by introducing comparison operators when writing the results to off-chip memory.

To accelerate an end-to-end CNN model, our design also supports Rectified Linear Unit (ReLU) layer, Pooling Layer and fully-connected (FC) layer. FC layers connect all the neurons in the previous layer to every single neuron in the weight matrix. We treat FC layer as a convolution layer with 1×1 kernel. Max Pooling layers are widely used in CNNs, which output the maximum values in subregions of input feature maps. ReLU layers set any input value less than zero to zero. As shown in Figure 4, pooling and ReLU logic are set before storing the output line buffer to the off-chip memory. ReLU is a pixel-wise operation, which is implemented by introducing comparison operators for each pixel. Pooling layer is implemented by gating the non-maximum value in the pooling region when storing the output pixels.

TABLE II: Hardware design parameters

Parameters	Description
Memory design parameters	
T_H	Tiling factor of output image height
K_M	Tiling factor of input channel
K_N	Tiling factor of output channel
Computation design parameters	
T_H	Parallelization factor of output image height
T_W	Parallelization factor of output image width
T_N	Parallelization factor of output channel

VI. FPGA-AWARE NAS

In previous sections, we provide an efficient architecture design for sparse CNN acceleration. Our hardware implementation involves several design parameters as shown in Table II. These parameters will affect both FPGA resource utilization and performance. More importantly, the execution latency of a CNN model is determined by both hardware parameters and convolution parameters, which makes it hard to find the optimal hardware design and an FPGA-friendly CNN model. Here, we propose a NAS framework that searches both architecture parameters and CNN model parameters. Specifically, we first formulate a resource model as the search constraint, and a latency model that is incorporated into the loss function. Then, we perform architecture search on a sparse supernet, which minimizes the total latency meanwhile maintains the accuracy.

A. Resource model

We use the number of memory banks to estimate the BRAM usage which is mainly used for input buffer, output buffer and encoded weight buffer. In our design, the PE array generates multiple $T_H \times T_W$ output tiles in T_N different output channels. Considering double buffer design, the number of output buffer banks is $2T_H \times T_W \times T_N$. In line buffer design, the input line buffers are rotated to reuse the overlapped area during the kernel sliding. According to equation 1, the number of input buffer banks is $(T_H + T_{H'}) \times T_{W'}$. As for the compressed weight in IOCOO format, the five tuples are represented with different bitwidth. The value and output channel number are stored individually. While the rest tuples are packed together and stored as a 9-bits element. Each part is partitioned with factor T_N. Therefore, the bank number of sparse weight is $3 \times T_N$. In summary, the total number of banks can be written as follows.

$$Banks^1 = 2T_H \times T_W \times T_N + (T_H + T_{H'}) \times T_{W'} + 3 \times T_N \quad (3)$$

Most DSP resource is consumed to perform multiplications between the input element and the sparse weight. Here, we assume a single DSP can be implemented as one multiplier[2].

[1] In Xilinx FPGA, one BRAM can store 1024 words of 18bits. The required BRAMs is determined by the data size in one bank.

[2] In most Xilinx FPGA Platform, a single DSP(DSP48E1) slice can be implemented as one 18×25 fixed-point multiplier. In most Intel FPGA Platform, a single DSP slice can be implemented as two 18×18 fixed-point multipliers

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

In CMUX module, each output tile needs to locate the output channel number, which takes 3 DSPs to calculate the address (3 comes from Vivado HLS). In this manner, the DSP utilization can be estimated as follows.

$$DSPs = T_H \times T_W \times T_N + 3 \times T_N \quad (4)$$

Modeling the LUT consumption on FPGA is more complex. For simplicity, we only model the LUT consumption for TLUT modules and CMUX modules. In our dataflow, there are T_N tile look-up tables working in parallel. The LUT consumption depends on the input tile size $T_H \times T_W$. Besides, there are T_N CMUX modules, which is a crossbar with T_N input wires. In summary, the LUT consumption is formulated as

$$LUTs = T_N \times (\alpha log(T_H \times T_W) + \beta log(T_N)) \quad (5)$$

where $\alpha log(T_H \times T_W)$ is the LUT consumption to store a single input tile, and $\beta log(T_N)$ is the LUT consumption for one CMUX module. α and β can be obtained on different platforms in advance. We first get the LUT consumption for a set of (T_H, T_W) based on Vivado High level Synthesis Tool. Then, we get α and β based on linear regression.

B. Latency model

The overall latency is either bounded by the PE computation time or the data transfer time in the line buffer design. First, we model the latency of loading/storing T_H rows of feature maps to/from the line buffer. Assuming each data is stored in 16 bits fixed point and the kernel sliding stride is 1, the data transfer time can be formulated as follows.

$$T_{tra} = \frac{T_H \times W \times max(K_M, K_N) \times 16bits}{Bandwidth} \quad (6)$$

where K_M, K_N are tiling factors in the channel dimension in Table II.

To simplify the estimation of the computation time, we use eff_{ave} to represent the average computation efficiency as discussed in Section IV-F. We define the computation time as the time to generate $T_H \times K_N$ output elements.

$$T_{com} = (\frac{sparsity}{eff_{ave}} \times \lceil \frac{K_N}{T_N} \rceil \times K_M \times \lceil \frac{W_{out}}{T_W} \rceil \times II + P_{depth}) \times \frac{1}{Freq} \quad (7)$$

where $Freq$ is the operating frequency of the FPGAs. II denotes the iteration interval of the pipeline. In our implementation, the loop L_K in Figure 3 (a) are perfectly pipelined, so the $II = 1$. P_{depth} is the pipeline depth, which can be ignored when the loop trip count is large enough.

Involving sparsity can result in low computation latency. Therefore, we also consider the initial time to load the first $T_{H'}$ rows of input feature map and the sparse weight.

$$T_{init} = \frac{K_M \times K_N \times T_{H'} + 3 \times R \times S \times K_M \times K_N \times sparsity}{Bandwidth/16bits} \quad (8)$$

Putting it all together, the total latency is determined by the initial time and maximum between data transfer time and computation time.

$$T_{total} = \lceil \frac{M}{K_M} \rceil \times \lceil \frac{N}{K_N} \rceil \times (\lceil \frac{H}{T_H} \rceil \times max(T_{tra}, T_{com}) + T_{init}) \quad (9)$$

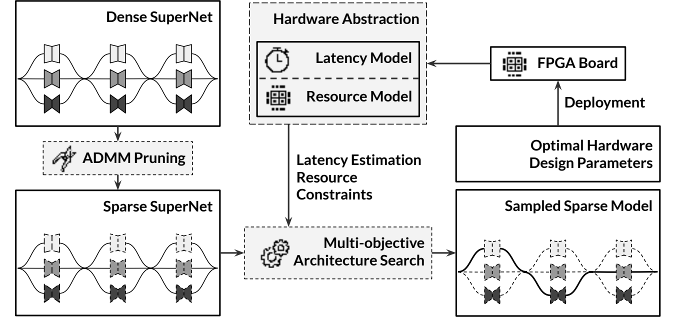


Fig. 11: FPGA-aware neural architecture search flow

TABLE III: Configurations of candidate operations.

Operation Type	Expansion Ratio	Kernel Size
3×3_MBConv3	3	3
3×3_MBConv6	6	3
5×5_MBConv3	3	5
5×5_MBConv6	6	5
7×7_MBConv3	3	7
7×7_MBConv6	6	7
skip layer	-	-

C. Search algorithm

Our NAS algorithm models the FPGA architecture. As aforementioned, our architecture is parameterized with parallelization factor T_H, T_W, T_N and buffering factor K_M, K_N, which affect the resource utilization and execution latency. Therefore, our problem setting specifies each layer o with three groups of parameters, including model weights w, and architecture parameters p and hardware design parameters h. More specifically, when deriving child networks from the supernet, each layer is sampled from a multinomial distribution parameterized by p over a pre-defined candidate set $\{o^j\}$. The hardware design parameters are also taken into account in two folds. First, the resource model ensures that the resource requirement of the search result can be met for a given FPGA device. Second, the latency model is integrated into the loss function for software-hardware co-design.

We use the same search space and supernet architecture as shown in Table III and Table IV. We adopt mobilenet inverted residual block as the basic building block of the supernet, whose kernel size can be chosen from {3, 5, 7} and expansion ratio from {3, 6}. Furthermore, to permit flexibility in network depth, a special skip layer is added to the candidate set if the input and output of a layer are of the same size.

Figure 11 shows our search algorithm which comprises three stages, i.e. warm-up, searching and retraining. In the warm-up stage, we adopt ADMM pruning to obtain a sparse supernet [10]. The supernet is trained normally for several epochs before we introduce the ADMM regularizer to promote sparsity. After convergence, we zero out the least significant connections according to the magnitude. A binary mask is then applied over each pruned parameter to prevent back-propagation from tampering with the weights of removed connections. In the second stage, we perform neural architecture search over the sparse supernet. Our loss function integrates the above-mentioned resource constraints and latency model,

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE IV: Supernet specification. Here MixOp denotes the mixed operation which can be chosen from 7 candidate operations, 3×3_Conv denotes a normal 3×3 convolution, and 3×3_MBConv1 represents a 3×3 mobilenet inverted residual block of expansion ratio 1.

Input Size	Operator	Output	s	n
224×224x3	3×3_Conv	32	2	1
112×112×32	3×3_MBConv1	16	1	1
112×112×16	MixOp	24	2	4
56×56×24	MixOp	40	2	4
28×28×40	MixOp	80	2	4
14×14×80	MixOp	96	1	4
14×14×96	MixOp	192	2	4
7×7×192	MixOp	320	1	1
7×7×320	GAP	-	-	1
320	FC	1000	-	1

which is given by

$$\begin{aligned} \underset{\{w_i\},\{p_i\},\{h_i\}}{\text{minimize}} \quad & \text{Loss}_{CE} + \lambda \mathbb{E}[\text{latency}] \\ \text{subject to} \quad & LUTs(h_i) \le \text{LUT}_{max}, \\ & DSPs(h_i) \le \text{DSP}_{max}, \\ & h_i \in \mathcal{H},\ i = 1, \dots, N, \end{aligned} \tag{10}$$

where λ is a scaling factor, $Loss_{CE}$ is the accuracy loss, LUT_{max} and DSP_{max} denoting respectively the limitation of LUT and DSP resources. The weight decay term is omitted here for simplicity. The second term of the loss function stands for the expected latency of the whole network, which can be calculated as follows:

$$\mathbb{E}[\text{latency}] = \sum_i \sum_j p_i^j \times T_{total}(o_i^j, h_i) \tag{11}$$

Here o_i^j and p_i^j are the j-th candidate in the i-th layer and its assigned possibility. The estimated latency T_{total} is derived by assigning a specific candidate and hardware configuration as shown in equation 9.

To simplify the problem, we optimize hardware parameter h in a separate subproblem:

$$\begin{aligned} \underset{h}{\text{minimize}} \quad & T_{total}(o, h) \\ \text{subject to} \quad & LUTs(h) \le \text{LUT}_{max}, \\ & DSPs(h) \le \text{DSP}_{max}, \\ & h \in \mathcal{H}, \end{aligned} \tag{12}$$

Since the search space $\mathcal{H}$ is a finite discrete in our case, a simple parameter sweep will find the globally optimal solution h^*. In our case, the latency overhead of exhaustive search is negligible compared to neural network training. Then the original loss function can be rewritten as

$$\underset{\{w_i\},\{p_i\}}{\text{minimize}} \quad \text{Loss}_{CE} + \lambda \sum_i \sum_j p_i^j \times T^*_{total}(o_i^j) \tag{13}$$

where $T^*_{total}(o_i^j)$ is a shorthand for $T_{total}(o_i^j, h_i^*)$.

In this manner, the simplified objective resembles a regular hardware-aware NAS problem that does not involve hardware parameters, we can solve the remaining part according to the technique in [35]. After the search algorithm has sufficiently converged, we obtain the optimal subnetwork by only keeping the most promising candidate at each layer. Since we have pruned the supernet in the warm-up stage, the compact network retains its sparsity. At last, the best architecture is retrained from scratch for the final evaluation.

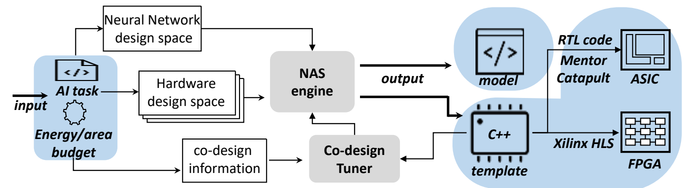


Fig. 12: Automatic hardware generation for ASIC and FPGA.

Note that our framework is able to support more complex cases where $\mathcal{H}$ can be a large discrete space or even a continuous one, and where a more sophisticated performance model is adopted. For example, given a learned performance model T_{total} parameterized by θ, we could back-propagate gradients through it and optimize hardware parameters h with gradient descent by fixing θ.

D. Automatic Hardware Generation

In this section, we integrated our architecture design with our NAS engine into an automatic flow for hardware generation. The intermediate passes are summarized in Figure 12. The inputs consist of a specific task and budget constraints. For ASIC design, the budget involves energy and area constraints under certain technology. For FPGAs, the budget is the on-chip resources. Then, our flow explores the space of the hardware-software co-design using the proposed NAS engine. The co-design information includes the target execution latency and the accuracy of a specific task. The co-design tuner helps to find the optimal architecture parameters and model parameters in the search space. Last, our automatic flow generates a CNN model and hardware implementation written in high-level template. The template can be synthesized using Xilinx Vivado HLS to get the FPGA bitstream, or using Mentor Catapult HLS tool [50] to generate Verilog RTL.

VII. Experiment

In this section, we first introduce the experiments setting. In Section VII-B, we show the performance of our accelerator for the state-of-the-art CNNs and compare it with previous dense CNN FPGA accelerators. Then, we measure the resource utilization and analyze the utilization breakdown. In Section VII-D, we examine the hardware efficiency of different configurations using four state-of-the-art networks. Last, we evaluate our NAS approach on FPGAs and compare to other NAS work.

A. Experiments Setup

We evaluate our design on Xilinx ZCU102 platform. ZCU102 consists of an UltraScale FPGA, quad ARM Cortex-A53 processors, 500 MB DDR3. Our FPGA implementation is operated at 200MHz frequency on this platform. To measure the runtime power, we plugged in a power meter in the FPGA platform. In this work, we first use Xilinx Vivado HLS

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE V: Performance comparison with previous implementation

	[21]	[16]	[48]	[49]	[43]	Ours	Ours	Ours	Ours
CNN type	VGG	VGG	VGG	VGG	Alexnet	VGG	Alexnet	Resnet-152	GoogLeNet
Device	Arria-10 GX1150	Arria-10 GX1150	Xilinx Virtex-7	Zynq ZC706	Zynq ZC706	Zynq ZCU102	Zynq ZCU102	Zynq ZCU102	Zynq ZCU102
Frequency (MHz)	150	370	200	100	-	200	200	200	200
Precision	16bit fixed	32bit float	16bit fixed	-	-	16bit fixed	16bit fixed	16bit fixed	16bit fixed
DSP Utilization	1518 (100%)	1320 (87%)	3200 (89%)	-	-	1144 (45%)	1144 (45%)	1144 (45%)	1144 (45%)
Logic Utilization	161K (38%)	182K (43%)	237K (55%)	-	-	552K (92%)	552K (92%)	552K (92%)	552K (92%)
BRAM	1900 (70%)	1250 (46%)	1244 (85%)	-	-	912 (48%)	912 (48%)	912 (48%)	912 (48%)
Performance (GOP/s)	64.5 (eff.)[1]	128.5 (eff.)	281.6[2]	297	71.2	309.0	223.4	291.4	257.4
DSP efficiency (GOP/s/DSP)	0.042	0.097	0.088	-	-	0.27	0.19	0.25	0.23
Logic efficiency (GOP/s/logic(K))	0.40	0.71	1.12	-	-	0.56	0.41	0.53	0.47
Power (W)	45.0	41.7	-	9.6	9.6	23.6	23.6	23.6	23.6

[1] eff. is the effective performance on sparse networks.
[2] 281.6 is estimated according to the number of frame per second in the paper.

TABLE VI: Training configurations.

Stage		Epoch	LR	Sched	bs
warmup	pretrain	40	0.05	cosine	256
	prune	60	0.01	step	256
	retrain	30	0.01	cosine	256
search		60	0.025	cosine	256
retrain		180	0.05	cosine	256

(v2017.4) tool chain to transform C code into RTL implementation. Then, we employ Xilinx SDSoC (v2017.4) to compile the source code into bitstream. We apply [9, 28] methods to train the CNN model with sparsity using the Caffe framework [51]. Specifically, we set the expected sparsity of the network by setting the value that is less than a threshold to zero, followed by retraining the network to regain any lost accuracy. In our experiment, we use the state-of-the-art CNNs including Alexnet, VGG-16, Resnet-152 and GoogLeNet. We achieve 10.8%, 11.7%, 23.5%, 34.2% sparsity[3] of Alexnet, VGG-16, Resnet-152, GoogLeNet without accuracy loss, respectively.

We perform neural architecture search on the full ImageNet. The validation set contains 50000 images randomly sampled from the original training set. As for ADMM pruning, we set rho[4] to 1e-3, 1e-2 and 1e-1 respectively in multi-rho training. Both the ADMM pruning and the NAS framework are implemented in PyTorch[52]. We use $\lambda = 0.1$ in our experiments, and summarize other training configurations in Table VI. Note that we spend more GPU hours (130 epochs vs 40) in the warm-up stage because of pruning. All training stages are run on 4 NVIDIA V100 GPUs.

B. Performance Analysis

In this section, we show the performance of our accelerator using modern CNNs. We set the accelerator configuration as $< T_H, T_W, T_N > = < 8, 8, 16 >$, which involves 1024 multipliers. In this configuration, the peak available performance can be computed as

$$peak\ performance = \#\ of\ multipliers \times frequency \times 2$$

Here 2 means multiplication and addition operations. The peak performance of our design is $409.6\ GOP/s$.

[3]Sparsity is defined as the percentage of nonzeros.
[4]Rho is the scale factor of the penalty term in ADMM pruning

We also compare our design with previous FPGA accelerators in Table V. [16, 21] are dense CNN accelerators and [43, 48, 49] are sparse CNN accelerators. The performance in Table V represents the effective performance. For the dense CNN accelerators, the effective performance is computed by multiplying the performance of dense CNNs with sparsity. According to Table V, our implementation achieves 223.4 GOP/s effective performance on sparse Alexnet which shows 2.4X speedup compared with [43][5]. [49] shows similar performance to our design, but it applies low bit precision which requires less resources. The performance on VGG network is 309.0 GOP/s which is 3.6X-4.8X higher than [16, 21]. [48] shows higher performance because they pruned the network in the frequency domain which results in element-wise multiplication pattern. This computation pattern shows less complexity compared with the convolution operator. For Resnet-152 and GoogLeNet, our design achieves 291.4 GOP/s and 257.4 GOP/s, respectively.

To make a fair comparison across different platforms, we also present the DSP-efficiency and logic-efficiency on each platform. On average, our design exhibits 0.24 GOP/s/DSP DSP-efficiency, which shows 2.5X-5.7X improvement compared with prior works [16, 21, 48]. On the other hand, our design shows lower logic-efficiency. This mainly comes from the TLUT module and CMUX module for the data decoding. Most previous designs target dense CNNs that exhibit regular data access. Therefore, they have higher logic-efficiency.

The speedup of our design is because the weight-oriented dataflow can effectively eliminate the useless multiplications. In addition, the dataflow maintains a high utilization of on-chip resources. Previous implementations cannot efficiently exploit the zeros in the computation, which results in a waste of on-chip resources. On the other hand, previous dense CNN accelerators are highly optimized and DSPs are fully utilized to conduct multiplications. In our implementation, only half of DSPs are utilized, and the performance is bounded by the logic resource.

The inefficiency of our implementation mainly comes from three aspects. First, there exist some invalid weights in our weight layout, which leads to imbalanced workload among PEs. Section VII-D presents the details of load imbalance problem. Second, the feature map size in the CNN layers

[5]This paper [43] only reported the performance and the platform

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

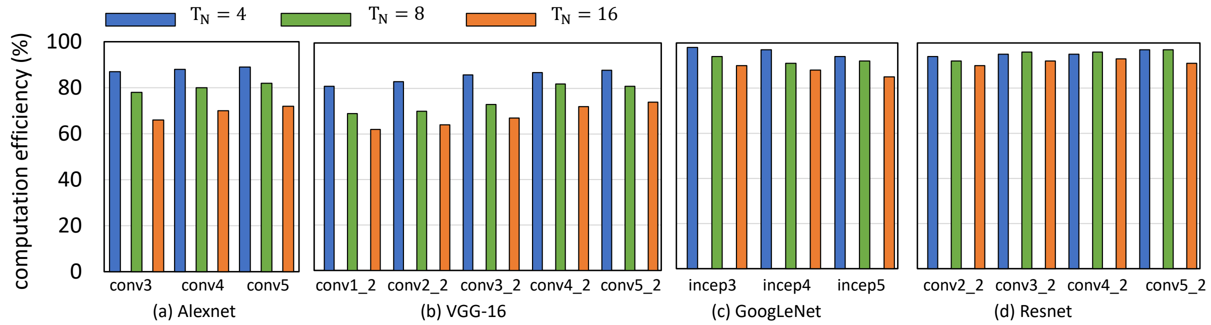


Fig. 13: Computation efficiency

TABLE VII: Resource utilization and latency.

	BRAM	DSP[6]	FF	LUT
Buffers	609	0	0	0
TLUT	0	0	48894	59150
PEs	0	288	1088	32
CMUX	0	24	394	43976
Others	8	52	18408	12758
Total	617	364	68784	132344
Predict(error)	556(9.9%)	312(14.3%)	-	103158(9.6%)
Available	1824	2520	548160	274780
Actual latency	191011 cycles			
Predicted latency	182523 cycles (4.4% error)			

cannot divide T_H and T_W evenly. We choose 8×8 as the output tile size. Take the last convolutional layer of VGG as an example, the feature map size is 14×14 leading to a 12.5% waste of computation. Third, as mentioned in Section IV-C, we apply pipeline technique in PEs. When the workload is small after pruning, the latency of PE can be bounded by the depth of pipeline. In our implementation, the pipeline depth is 8 cycles. Compared with VGG network, Resnet-152s and GoogLeNet consist of many convolutional layers with 1×1 kernels, leading to low performance. The speedup of VGG-16 is higher than that of Alexnet, because VGG-16 is a structured and regular network. The kernel size of all layers is 3×3, however, Alexnet contains many layers in different types.

C. Resource Utilization Characteristics

Table VII shows the resource utilization breakdown with the configuration ($T_H = T_W = 6, T_N = 8$). In Table VII, we also present the prediction accuracy of our resource model and latency model. The cycle number of latency is tested on the convolutional layer with 112×112 feature map size and $64 \times 64 \times 3 \times 3$ kernel size. The inaccuracy sources of DSP utilization mainly come from the padding operation and FIFOs in the line buffer design. A few extra BRAMs are used for FIFOs and pipeline buffers. The actual latency is obtained using Xilinx HLS simulation tools. The prediction error of latency model results from the padding operation, which only occupied a few cycles.

Figure 14 shows the resource utilization of different configurations obtained from Xilinx Vivado tool (v2017.4). In Figure 14, the LUT utilization increases as the parallelism factor T_N increases because of CMUX. When T_N is large, the utilization of BRAMs is mainly determined by the parallelization degree of feature maps (T_H, T_W). When T_N is small, BRAM utilization is similar. This indicates that the consumption of BRAM is determined by the input and output data size, instead of the

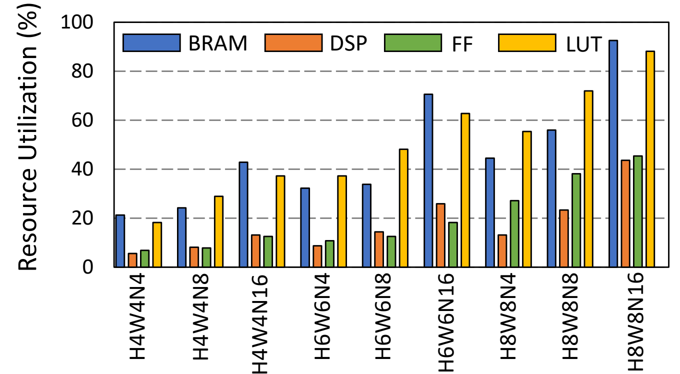


Fig. 14: Resource utilization (HaWbNc means $T_H = a, T_W = b, T_N = c$)

partition factors. For example, when $T_N = 4$ in Equation 3, each bank is constructed using two BRAMs. While each bank only needs one BRAM when $T_N = 8$.

D. Computation efficiency

In our implementation, the PE array processes P_N weights with different remainders at the same time. However, the remainder distribution is irregular which can result in load imbalance problem as discussed in Section IV-F. Figure 13 shows the efficiency across different layers with different parallelism factors. We find that the format efficiency increases when the modular factor T_N becomes larger. Because a large T_N will bridge the gap between the maximum number of valid values and the average number of valid values among different remainders. We find that the efficiency increases as the network goes deeper. This because the number of channels increases as the network goes deeper, which makes the total number of nonzeros larger. A large number of nonzeros can compensate for the gap between the maximal and minimal number of remainders. Also, we observe that the computation efficiency of GoogLeNet and Resnet-152 is much higher. This is because the sparsity of these two networks is relatively large which leads to a large number of nonzeros. Besides, the computation efficiency of some layers in GoogLeNet is low. Because the channel number cannot be divided evenly by T_N. For example, the output channel number of inception_4b layer in GoogLeNet is 24 which is not a multiple of $T_N = 16$. In conclusion, our dataflow can maintain high computation efficiency for different configurations and networks.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

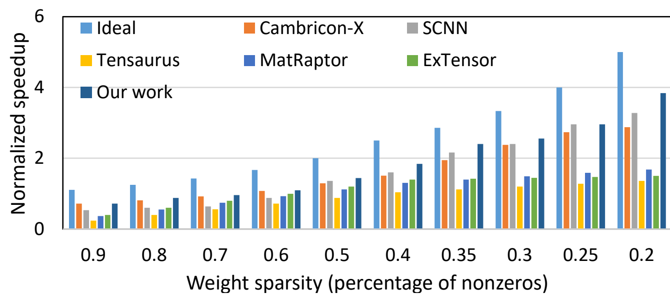


Fig. 15: Comparison with other ASIC dataflows.

E. Comparison with ASIC dataflows

In this section, we build several cycle-accurate models for the comparison with ASIC sparse tensor accelerators. Cambricon-X [38] and SCNN [37] are CNN-specific ASIC accelerators. We only model the behavior of the PE array of these designs. To make a fair comparison, we scale the number of multipliers to 1024. Tensaurus [46] and ExTensor [53] are general sparse tensor accelerators. MatRaptor [45] focuses on the sparse matrix-matrix multiplication (GEMM). When comparing with them, we transform the convolutional layer to GEMM using the image-to-column (im2col) operation. The input image is regarded as a dense matrix.

Figure 15 gives the comparison results. We also draw the line of theoretical speed up calculated by $(1/sparsity)$. When the sparsity is higher than 70%, Cambricon-X and our design show near-ideal speedup. However, SCNN shows a lower speedup because of a large amount of fanout of the PE array when most weights are non-zero. On the other hand, our design outperforms Cambricon-X when the sparsity decreases. This is because Cambricon-X requires index comparison to conduct sparse vector dot-product which is inefficient for low sparsity. Our weight-oriented dataflow minimizes the indexing overhead by introducing a static TLUT module.

The speedup of general sparse tensor accelerators is always lower than the CNN-specific accelerators. Such limitation mainly comes from the im2col operation where the input images are duplicated multiple times and flattened into a matrix. On the other hand, CNN accelerators naturally leverage the convolution properties to avoid data rearrange overhead. To be specific, Tensaurus unifies the sparse computation as an operation between a scalar and a fiber, where each PE performs the multiplication between one weight and one input vector. MatRaptor applies the row-wise matrix multiplication dataflow where one weight is multiplied with the corresponding row of the input matrix. Though these two dataflows share a similar idea to our design that each PE is responsible for the multiplication between one weight and multiple inputs, their speedup is limited by the sparsity. PEs in Tensaurus and MatRaptor is parallelized with inputs from different rows, which requires a synchronization operation between different PEs when accumulating partial sums. Such synchronization overhead is small when the sparsity is extremely low. However, in sparse CNN models, the sparsity is usually around 0.1 - 0.2, which can lead to high synchronization overhead. ExTensor is parallelized using multiple dot-product, which requires to compare the index between two vectors. When the sparsity is high, most indices can be matched. Therefore, ExTensor outperforms Tensaurus and MatRaptor when the sparsity is higher than 0.4.

TABLE VIII: Comparison with GPU platform using Resnet-152

Device	TitanX[1]	TitanX[2]	ZC706	ZCU102
Technology	28 nm	28 nm	28 nm	16 nm
Frequency (MHz)	1075	1075	166	200
Precision	32bits float	32bits float	16bits fixed	16bits fixed
conv average (GOP/s)	212(eff.)	119	134	291
Power (W)	130	134	9.4	23.6
Energy efficiency (GOP/s/W)	1.63	0.88	12.66	12.33

[1] The sparse network is considered as the dense network and accelerated using CuDNN.
[2] The sparse network is accelerated using CuSparse.

TABLE IX: Layer types for latency profiling. S-CONV: spatial convolution. DW-CONV: depthwise convolution.

Layer ID	Filter	Type	Channel	Resolution
C1	1×1	S-Conv	256×256	112×112
C2	1×1	S-Conv	256×256	56×56
C3	5×5	DW-Conv	32×32	14×14
C4	7×7	DW-Conv	3×64	14×14
C5	3×3	S-Conv	32×32	112×112
C6	3×3	S-Conv	64×64	112×112
C7	3×3	DW-Conv	128×128	112×112
C8	3×3	S-Conv	256×256	14×14

F. Scalability and comparison with GPU

We also test our design on ZC706 platform to demonstrate the scalability. Our implementation is operated at 166MHz frequency on this platform. ZC706 has 900 DSPs, 1090 BRAMs and 305K logic cells. We set the configuration parameter as $(T_H = T_W = 8, T_N = 8)$ and achieve 134.2 GOP/s on Resnet-152 which means our design can be scaled to different platforms. Besides, we conduct a comparison between GPU and FPGA platforms. We measure the performance of dense Resnet-152 using the latest CuDNN on NVIDIA TitanX platform. To make a fair comparison, we also apply CuSparse library to accelerate sparse Resnet-152. The sparse version shows a lower performance because of the memory uncoalesing problem. In conclusion, our design shows 1.37X speedup and 7.56X energy-efficiency compared with TitanX platform.

G. Latency profiling

We first profile the latency of different layer types, which helps to find the FPGA-friendly neural network architecture. In Table IX, we list 8 representative layer types including spatial convolution and depth-wise convolution. Figure 16 shows the detailed profiling results involving initial time, compute time and data transfer time. According to Figure 16, the compute time is the bottleneck for most layers (C4-C8). However, the transfer time of the layer that shows less data reuse is higher than the compute time (C1-C2). These layers are often spatial convolution with 1×1 filter and depth-wise convolution. Besides, we also observe that the initial time can also affect the performance when the feature map size is small (C3-C4). For example, for the C3 layer, the feature map only contains three input channels, which is less computation-intensive. This

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE X: Performance comparison with other handcrafted models (MobileNetV2 and ShuffleNetV2) and NAS techniques (ProxylessNAS and FBNet). MobileNetV2-1.0 and ShuffleNetV2-1.0 is the original network

Model	Search method	HW	Search space	#Params (M)	#OPs (M)	FPGA latency (ms)	Top-1 (%)
MobileNetV2-0.75	-	-	-	2.7	290	34.471	67.9
MobileNetV2-1.0	-	-	-	3.4	600	40.791	72.0
MobileNetV2-1.4	-	-	-	6.9	770	54.486	74.7
ProxylessNAS-CPU	gradient	✗	cell-level	4.4	954	61.889	75.3
ProxylessNAS-GPU	gradient	✗	cell-level	7.1	930	109.421	75.1
ProxylessNAS-mobile	gradient	✗	cell-level	4.1	640	99.770	74.6
ShuffleNetV2-0.5	-	-	-	1.4	82	10.867	60.3
ShuffleNetV2-1.0	-	-	-	2.3	292	16.786	69.1
ShuffleNetV2-1.5	-	-	-	3.5	598	22.019	72.6
ShuffleNetV2-2.0	-	-	-	7.4	1182	28.019	74.9
FBNet-A	gradient	✗	cell-level	4.3	498	56.328	73.0
FBNet-B	gradient	✗	cell-level	4.2	590	64.532	74.1
FBNet-C	gradient	✗	cell-level	5.0	750	87.766	74.9
Ours	gradient	✓	cell-level	3.9	642	27.3	73.3

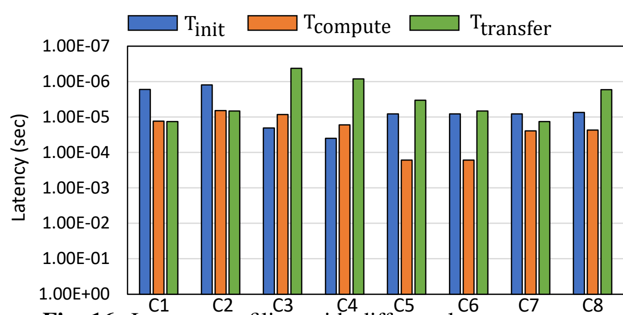


Fig. 16: Latency profiling with different layer parameters.

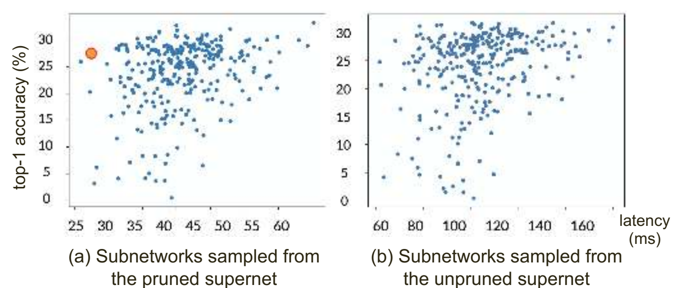


Fig. 17: Accuracy v.s. latency of sampled subnetworks.

phenomenon can also be found in C4 layer whose channel number is 32.

H. NAS results

As shown in Table X, compared to other compact models that are with similar accuracy, our method consistently improves upon inference latency. Specifically, when confronted with MobileNetV2-1.0 [54], a manually designed architecture that targets no specific platform, our model achieves 1.49X speedup. Note that ProxylessNAS and FBNet are NAS networks, but target a fixed hardware, e.g., CPU, GPU, mobile-phones, which did not search architecture parameters. While maintaining accuracy on par with FBNet-B [34] and ProxylessNAS-mobile [35], our architecture is 2.36X and 3.65X faster respectively. Nevertheless, our method takes longer time than [35] in the warmup stage, because we conduct ADMM pruning on the supernet additionally. Yet since we can perform architecture search multiple times on the same supernet, the pruning cost will only occur once.

In our experiments, we choose a conservative sparsity of 30% to avoid significant accuracy loss. Mobilenet-like compact networks on ImageNet have been observed to be less tolerant to network pruning, which can be attributed to less redundancy in parameters and a more challenging task. Figure 17 shows the accuracy v.s. latency of 300 subnetworks randomly sampled from the dense and sparse supernet respectively, with the highlighted point denoting the resultant architecture. This graph indicates (a) the ranking of the child networks still correlates with their actual performance since the distribution characteristics are basically retained after pruning; and (b) the search algorithm succeeds in finding out an architecture near the Pareto-front with acceptable latency. We also provide the resultant network architecture and hardware parameters for ZCU102 in Table XI.

TABLE XI: The searched network architecture.

Input	Operator	Ksize	Output	Expansion
$224^2 \times 3$	Conv	3×3	32	-
$112^2 \times 32$	MBConv	3×3	16	1
$112^2 \times 16$	MBConv	5×5	24	3
$56^2 \times 24$	MBConv	3×3	24	6
$56^2 \times 24$	MBConv	7×7	40	3
$28^2 \times 40$	MBConv	3×3	40	3
$28^2 \times 40$	MBConv	5×5	40	3
$28^2 \times 40$	MBConv	3×3	80	6
$14^2 \times 80$	MBConv	3×3	80	6
$14^2 \times 80$	MBConv	7×7	80	3
$14^2 \times 80$	MBConv	5×5	80	3
$14^2 \times 80$	MBConv	3×3	96	6
$14^2 \times 96$	MBConv	3×3	96	3
$14^2 \times 96$	MBConv	7×7	96	6
$14^2 \times 96$	MBConv	3×3	192	3
$7^2 \times 192$	MBConv	3×3	192	6
$7^2 \times 192$	MBConv	3×3	192	3
$7^2 \times 192$	MBConv	5×5	192	3
$7^2 \times 192$	MBConv	3×3	320	6
$7^2 \times 320$	Conv	1×1	1280	-
1280	FC	-	1000	-
Searched hardware parameters: $T_W = T_H = 3, T_N = 22$				

To evaluate the scalability of our NAS method, we conduct the search algorithm on ZC706 FPGA and ZCU102 FPGA, which show different hardware resources. Table XII reports the results on two devices with different resource constraints and hardware search space. Based on the size of the available

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

TABLE XII: NAS on different device

Device	Resource constraints				Hardware parameter search range			Latency	Top-1 acc (%)
	DSP	Logic	BRAM	Frequency	P_N range	P_H range	P_W range		
ZC706	900	350K	1090×18 Kb	166M	[4,16]	[3,8]	[3,8]	33.7	73.0
ZCU102	2520	600K	1824×18 Kb	200M	[4,32]	[3,8]	[3,8]	27.3	73.3

resource, we set the hardware parameter search range which differs in the output channel number that computed in parallel. The search algorithm returns models with almost the same top-1 accuracy under these two settings. As we can see, our method can adapt to various hardware platforms and reflect hardware constraints in the search process.

VIII. Related work

Architecture for dense CNNs on FPGAs. Prior efforts to accelerate CNNs have shown substantial successes on FPGAs. Ma [21] et al. make an in-depth analysis of loop optimization techniques in spatial convolution, which includes loop tiling, loop unrolling and loop interchange. Zhang [16] et al. focus on reducing the on-chip memory bandwidth requirement. Wei et al. [23] implemented CNN on an FPGA using a systolic array architecture, which can achieve high clock frequency under high resource utilization. Zhang et al. [55] proposed AccDNN tool which included high-quality RTL network layer IPs, a fine-grained layer-based pipeline architecture and an automatic design space exploration tool. Besides, there are some works that implement fast algorithms to further accelerate CNNs [47, 56–59].

Architecture for sparse CNNs on ASICs. Recently, some works explore the dataflow and architecture to accelerate sparse CNNs on ASICs. Han et al. [60] proposed EIE CNN accelerator which exploits sparsity both in input feature maps and filters but only focused on the fully-connected layer. The fully-connected layer is computed using matrix multiplication, in EIE design, the matrix is stored in CSC format and multiple columns are computed in parallel. Parashar et al. proposed SCNN accelerator with a dataflow named PT-IS-CP (planar-tiled input-stationary Cartesian-product) [37]. Zhang et al. [38] presented Cambricon-X accelerator which applies step indexing techniques. In Cambricon-X design, the nonzeros in the same row are divided into multiple segments with the same size in subsequent addresses. And the row that contains nonzeros less than the size will be aligned to the size. In recent years, some ASIC accelerators apply hardware-software design that prunes the weight with structured pattern [15, 31].

Neural Architecture Search. Early NAS algorithms [33] are inefficient in terms of search time and hardware-friendliness during inference. There are two trends related to our work in the subsequent works that tackle these issues. *One-shot NAS* [35] constructs a supernet and defines candidate architectures as its subgraphs. Rather than training from scratch each time, the weights of a sampled architectures are generated by the pretrained supernet. DARTS [61] relaxes the discrete search space into a concrete distribution by assigning a real-valued weight to each candidate path. Instead of optimizing all paths jointly, ProxylessNAS[35] samples a few paths in each training step to reduce GPU memory consumption. *Device-aware multi-objective NAS*[34, 35] explicitly incorporates resource efficiency into the objective function, either device-related (such as latency, energy consumption) or device-agnostic (such as FLOPs and model size). These works often adopt a compact search space that's inspired by hand-crafted networks. Depending on the nature of the target hardware, the efficiency is either measured through runtime measuring [36], a pre-measured lookup table [34, 35]. In parallel to our work, there is a recent trend of incorporating NAS into SW-HW co-design frameworks. This line of works fuses NN architecture parameters and hardware implementation parameters into a single search space, thereby optimizing them simultaneously via stochastic coordinate descent[62], or gradient-based methods[63]. Our method is to some extent similar to the one mentioned in [63], in that both adopt a mobilenet-like search space and a gradient-based approach to NAS. Nevertheless, to the best of our knowledge, this is the first work in this area targeting sparsity in both NN and accelerator design.

IX. Conlusion

In this work, we propose an FPGA accelerator for sparse CNNs. We first propose a weight-oriented dataflow that exploits element-matrix multiplication. Based on this dataflow, we design an FPGA architecture mainly composed of a tile look-up table and a channel multiplexer. Besides, we propose a weight layout where the weights calculated in parallel are stored continuously. To cooperate with the weight layout, a channel multiplexer is inserted to locate the address which can ensure no data access conflict. Finally, we develop an FPGA-aware NAS approach to find the hardware-friendly network structure. Experiments demonstrate that our accelerator can achieve 223.4-309.0 GOP/s for the modern CNNs on Xilinx ZCU102, which provides a 2.4X-12.9X speedup over previous dense CNN FPGA accelerators. Our FPGA-aware NAS approach shows 2X speedup over MobileNetV2 with 1.5% accuracy loss.

Acknowledgement

This work was supported in part by the Beijing Natural Science Foundation (No. JQ19014) and in part by the Beijing Academy of Artificial Intelligence (BAAI). This work was also supported by Key-Area Research and Development Program of Guangdong Province (No. 2019B010155002)

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015.
[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR*, 2014.
[3] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *arXiv preprint arXiv:1409.1556*, 2014.
[4] J. Redmon *et al.*, "You only look once: Unified, real-time object detection," in *CVPR*, 2016.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[5] N. Suda *et al.*, “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,” in *FPGA*, 2016.

[6] J. Qiu *et al.*, “Going deeper with embedded FPGA platform for convolutional neural network,” in *FPGA*, 2016.

[7] X. Zhang, J. Zou, X. Ming, K. He, and J. Sun, “Efficient and accurate approximations of nonlinear convolutional networks,” *CoRR*, vol. abs/1411.4229, 2014. [Online]. Available: http://arxiv.org/abs/1411.4229

[8] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[9] S. Han, J. Pool, J. Tran, and W. Dally, “Learning both weights and connections for efficient neural network,” in *NIPS*, 2015.

[10] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, “A systematic DNN weight pruning framework using alternating direction method of multipliers,” *CoRR*, vol. abs/1804.03294, 2018. [Online]. Available: http://arxiv.org/abs/1804.03294

[11] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, “Scalpel: Customizing dnn pruning to the underlying hardware parallelism,” in *ACM SIGARCH Computer Architecture News*, 2017.

[12] J. Luo, J. Wu, and W. Lin, “Thinet: A filter level pruning method for deep neural network compression,” *CoRR*, 2017.

[13] S. Lin, R. Ji, Y. Li, Y. Wu, F. Huang, and B. Zhang, “Accelerating convolutional networks via global & dynamic filter pruning,” 07 2018, pp. 2425–2432.

[14] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[15] C. Ding *et al.*, “CIRCNN: accelerating and compressing deep neural networks using block-circulant weight matrices,” in *MICRO*, 2017.

[16] J. Zhang and J. Li, “Improving the performance of OpenCL-based fpga accelerator for convolutional neural network,” in *FPGA*, 2017.

[17] Q. Xiao, L. Lu, J. Xie, and Y. Liang, “Fcnnlib: an efficient and flexible convolution algorithm library on fpgas,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[18] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong, “Tgpa: tile-grained pipeline architecture for low latency cnn inference,” in *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[19] L. Jia, L. Lu, X. Wei, and Y. Liang, “Generating systolic array accelerators with reusable blocks,” *IEEE Micro*, vol. 40, no. 4, pp. 85–92, 2020.

[20] Q. Xiao and Y. Liang, “Zac: Towards automatic optimization and deployment of quantized deep neural networks on embedded devices,” in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2019, pp. 1–6.

[21] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo, “Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks,” in *FPGA*, 2017.

[22] Y. Guan *et al.*, “FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates,” in *FCCM*, 2017.

[23] X. Wei *et al.*, “Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs,” in *DAC*, 2017.

[24] Q. Xiao *et al.*, “Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs,” in *DAC*, 2017.

[25] Y. Liang, L. Lu, and J. Xie, “Omni: A framework for integrating hardware and software optimizations for sparse cnns,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[26] S. Han *et al.*, “ESE: Efficient speech recognition engine with sparse lstm on FPGA,” in *FPGA*, 2017.

[27] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, “An efficient hardware accelerator for sparse convolutional neural networks on fpgas,” in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 17–25.

[28] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding,” in *ICLR*, 2015.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *NIPS*, 2012.

[30] D. Chunhua *et al.*, “PERMDNN: Efficient Compressed DNN Architecture with Permuted Diagonal Matrices,” in *MICRO*, 2018.

[31] Z. Xuda *et al.*, “Cambricon-S: Addressing Irregularity in Sparse Neural Networks through a Cooperative Software-Hardware Approach,” in *MICRO*, 2018.

[32] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” *ArXiv*, vol. abs/1611.01578, 2016.

[33] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8697–8710, 2017.

[34] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, “Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search,” in *CVPR*, 2018.

[35] H. Cai, L. Zhu, and S. Han, “Proxylessnas: Direct neural architecture search on target task and hardware,” *ArXiv*, vol. abs/1812.00332, 2018.

[36] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, “Mnasnet: Platform-aware neural architecture search for mobile,” in *CVPR*, 2018.

[37] A. Parashar *et al.*, “SCNN: An accelerator for compressed-sparse convolutional neural networks,” in *ISCA*, 2017.

[38] S. Zhang *et al.*, “Cambricon-X: An accelerator for sparse neural networks,” in *MICRO*, 2016.

[39] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, “Cnvlutin: ineffectual-neuron-free deep neural network computing,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 1–13.

[40] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. Vijaykumar, “Sparten: A sparse tensor accelerator for convolutional neural networks,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 151–165.

[41] J. Albericio, A. Delmás, P. Judd, S. Sharify, G. O’Leary, R. Genov, and A. Moshovos, “Bit-pragmatic deep neural network computing,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 382–394.

[42] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, “Stripes: Bit-serial deep neural network computing,” in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[43] S. Li *et al.*, “An FPGA Design Framework for CNN Sparsification and Acceleration,” in *FCCM*, 2017.

[44] J. H. Ko *et al.*, “Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks using Frequency-Domain Computation,” in *DAC*, 2017.

[45] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, “Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 766–780.

[46] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, “Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations,” 2020.

[47] L. Lu, Y. Liang, Q. Xiao, and S. Yan, “Evaluating fast algorithms for convolutional neural networks on FPGAs,” in *FCCM*, 2017.

[48] Y. Niu, H. Zeng, A. Srivastava, K. Lakhotia, R. Kannan, Y. Wang, and V. Prasanna, “Spec2: Spectral sparse cnn accelerator on fpgas,” in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019, pp. 195–204.

[49] W. You and C. Wu, “A reconfigurable accelerator for sparse convolutional neural networks,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019, pp. 119–119.

[50] Mentor, “Catapult high-level synthesis,” https://www.mentor.com/hls-lp/, 2018.

[51] Y. Jia *et al.*, “Caffe: Convolutional architecture for fast feature embedding,” *arXiv preprint arXiv:1408.5093*, 2014.

[52] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” *ArXiv*, vol. abs/1912.01703, 2019.

[53] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “Extensor: An accelerator for sparse tensor algebra,” in *Proceedings of the International Symposium on Microarchitecture*, 2019.

[54] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

[55] X. Zhang *et al.*, “AccDNN: An IP-Based DNN Generator for FPGAs,” in *FCCM*, 2018.

[56] U. Aydonat, S. O’Connell, D. Capalija, A. C. Ling, and G. R. Chiu, “An OpenCL™ deep learning accelerator on arria 10,” in *FPGA*, 2017.

[57] C. Zhang and V. Prasanna, “Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System,” in *FPGA*, 2017.

[58] L. Lu and Y. Liang, “SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs,” in *DAC*, 2018.

[59] Y. Liang, L. Lu, Q. Xiao, and S. Yan, “Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs,” *TCAD*, 2019.

[60] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: efficient inference engine on compressed deep neural network,” in *ISCA*, 2016.

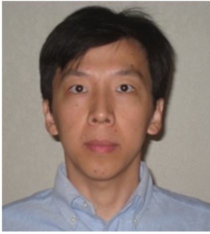
[61] H. Liu, K. Simonyan, and Y. Yang, “Darts: Differentiable architecture search,” *ArXiv*, vol. abs/1806.09055, 2018.

[62] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-M. Hwu, and D. Chen, “Fpga/dnn co-design: An efficient design methodology for 1ot intelligence on the edge,” *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[63] Y. Li, C. Hao, X. Zhang, X. Liu, Y. Chen, J. Xiong, W. Hwu, and D. Chen, “Edd: Efficient differentiable dnn architecture and implementation co-search for embedded ai solutions,” *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2020.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2021.3066563, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems



Yun (Eric) Liang is an associate professor (with tenure) in the School of EECS, Peking University, China. His research interests include computer architecture, compiler, electronic design automation, and embedded system. He has authored over 90 scientific publications in premier international journals and conferences in related domains. His research has been recognized by best paper awards at FCCM 2011 and ICCAD 2017 and best paper nominations at PPoPP 2019, DAC 2017, ASPDAC 2016, DAC 2012, FPT 2011, and CODES+ISSS 2008. He serves as Associate Editor for ACM Transactions in Embedded Computing Systems (TECS), ACM Transactions on Reconfigurable Technology and Systems (TRETS), and Embedded System Letters (ESL). He also serves in the program committees in the premier conferences in the related domain including MICRO, DAC, HPCA, FPGA, ICCAD, FCCM, ICS, etc.

Liqiang Lu obtained his B.S. degree from Institute of Microelectronics, Peking University, Beijing, China in 2017. He is now a Ph.D. candidate in School of EECS, Peking University. His research focuses on algorithm-level and architecture-level optimizations of FPGA for machine learning applications.

Yicheng Jin Yicheng Jin is an undergraduate student at Yuanpei College, Peking University and he is graduating in 2020 with a BS in Computer Science. He holds interest in machine learning systems and software-hardware co-design for machine learning applications.

Jiansong Zhang Dr. Jiansong Zhang is a staff engineer working on heterogeneous computing for machine learning and big data processing in Alibaba Group. Before joining Alibaba Group, he was a researcher working on heterogeneous computing for wireless communication & networking, IoT systems and datacenter acceleration in Microsoft Research Asia. He got PhD from the Hong Kong University of Science and Technology, and Master/Bachelor from Tsinghua University. His research interests are building innovative software and hardware systems for various applications. He has published tens of papers in Sigcomm, NSDI, Mobicom, Ubicomp, Mobisys, HotNets, HotChips, FCCM, Infocom, etc.

Ruirui Huang Dr. Ruirui (Raymond) Huang is the Director of Cloud Architecture and is responsible for the overall cloud architecture design of the cloud platform and products in Alibaba Cloud. Before joining Alibaba Cloud, he was an architect for the Xeon server SoC in Intel. He graduated from the Cornell University with a Ph.D. degree specializing in Computer Architecture research with a focus on highly reliable, secure, and available computing design. He has published papers in ASPLOS, ISCA, HPCA, DAC, FSE, FPGA, etc.

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