A Survey on Neural Network Inference Acceleration with FPGA

Qinyang Bao
dept. Electrical and Computer Engineering of University of Toronto
Toronto, Canada
qinyang.bao@mail.utoronto.ca

***Abstract*—World count: 3376. In recent years, deep learning has achieved superior performance over traditional machine learning methods for a variety of tasks. However, their high computation cost grows exponentially and calls for acceleration. FPGA has emerged as a suitable platform for such acceleration. A discussion on existing accelerator designs would be helpful for guiding future research.**

***Index Terms*—FPGA architecture, neural network, deep learning**

I. Introduction

In recent years, the field of deep learning has seen tremendous progress, and has achieved state-of-the-art performance in a variety of tasks including vision tasks such as image classification and object recognition; natural language tasks such as speech recognition and translation; and much more [1]. Deep learning is characterized by layers of neural networks that learn representations from low level to high level for a set of data. Different architectures of neural networks have been proposed to tackle tasks from different domains. For example, Convolutional Neural Networks (CNNs) work well for vision tasks, while Recurrent Neural Networks (RNNs) are naturally suited for language tasks or any sequence-based tasks. Lately, the transformer architecture with the self-attention mechanism has been applied to a variety of domains, all achieving state-of-the-art performance [2], [3], suggesting a unified architecture.

While the network architectures are likely to keep evolving, one common trend is that the size of deep learning models is growing at an accelerated pace, and so does the amount of resources (compute, power, etc.) required for both inference and training [4]; hence the need for accelerating neural network computation rises. Traditionally GPUs are employed for accelerating the training of neural networks, yet they are rarely good candidates for inference workloads, especially real-time ones due to their high power consumption, and low performance at small or single batch sizes. FPGA-based accelerators has emerged as an attractive alternative for high energy efficiency, abundant resources for massive parallelism, and great reconfigurability to customize for different network architectures without incurring the cost of an ASIC design. Correspondingly, there has been much research on using FPGA to accelerate various neural networks. [5] proposes to accelerate the matrix multiplication and activation functions via fully pipelined processing units. [6] employs a line buffer design to accelerate convolutions layers, while it uses SVD to reduce weight storage for fully connected (FC) layers. [7] simplifies the convolution operation using Winograd transform and introduces parallelism in four dimensions. [8] enhances the work of [7] by adding a sparse matrix packing module to support pruned neural networks. [9] adopts a layer-based pipeline architecture, and implements an end-to-end automation tool to generate optimized parallelism guidelines. [10] has a similar design while enhancing it to support sparsity and also improving the CAD tool. [11] builds a SIMD soft processor with vector-matrix multiplication primitives, aiming to support single-batch inference. [12] builds an accelerator specifically for pruned Long-Short-Term-Memory (LSTM) networks, supporting sparse vector-matrix multiplications. [13] and [14] builds an accelerator for transformers, leveraging different model compression techniques. The remainder of this paper will analyze these existing works, discuss the various optimization techniques they employ, compare their performance, and offer some suggestions for future research.

II. Compute Bound Optimization

As with any other computation task, the execution of a neural network is either compute-bound or memory-bound. This section discusses optimizations for compute-bound challenges, which are handled by increasing the peak throughput.

The building block of any neural network computation is the dot product between two vectors, which is the key optimization target for almost all accelerators. The simplest way is to build a processing element (PE) that computes in parallel for all the 2N operations in a dot product for vector length of N. This can be implemented with N multipliers, usually as DSPs, followed by a binary adder tree, as shown in Fig. 1. Often, the multipliers and adders are fully pipelined to maximize the throughput. Some works such as [5] stops at this point and reuses a single dot product PE for all the computations, relying on scheduling and memory transfers to feed the PE weights and activation from different layers. This is a simple approach and it can increase the peak throughput for 2N times comparing to a single ALU used in simple CPUs. Nevertheless, it has two down sides. First, it can only exploit parallelization in one dimension while even the simplest FC layers can have two parallelization dimensions (input activations and each output channel). Second, it is not flexible as the optimum vector length N varies across network layers.

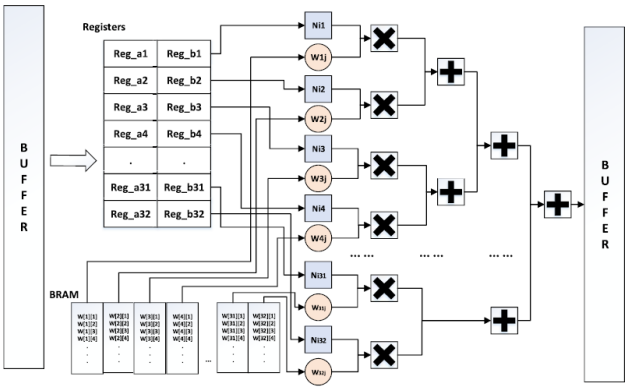


Fig. 1. Basic Dot Product PE [5]

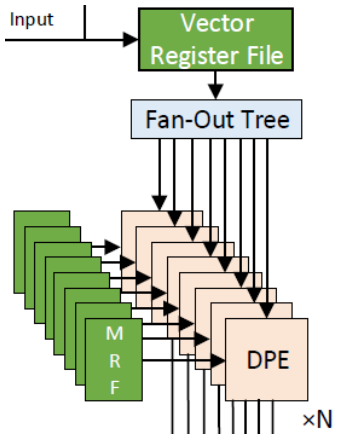


Fig. 2. Multiple MVM PEs [11]

To address the first disadvantage, one could simply duplicate the dot product PEs. For example, a vector matrix multiplication (MVM) PE can be made with several dot product PEs where one vector is multiplied with the different vectors forming a matrix. This is usually referred as a tile and multiple tiles can be created to expand the parallelism, where each tile may operate on different vectors, while the matrices can be varied or kept the same. [11] is an example of this design, and Fig. 2 shows an visualization. The MVM PE can accommodate a variety of network layers. For FC layers (which are integral for all RNN, CNN, and transformer architectures), the activations and weights corresponds to the vector and the matrix respectively. Multiple tiles could map to different partitions of the activations / weights and some extra steps are needed to accumulate the partial sums.

Similarly, for self attention layers, one can represent the matrix matrix multiplications (MMMs) as several MVMs and map them to different tiles. In addition, multiple MMMs can be created to parallelize each head in a multi-headed self attention (MSA), as done in [13] and [14].

Meanwhile, convolutional layers are less straightforward to implement with MVM PEs. The naive way is to perform a *im2col* operation that rearranges a 2D feature map into vectors

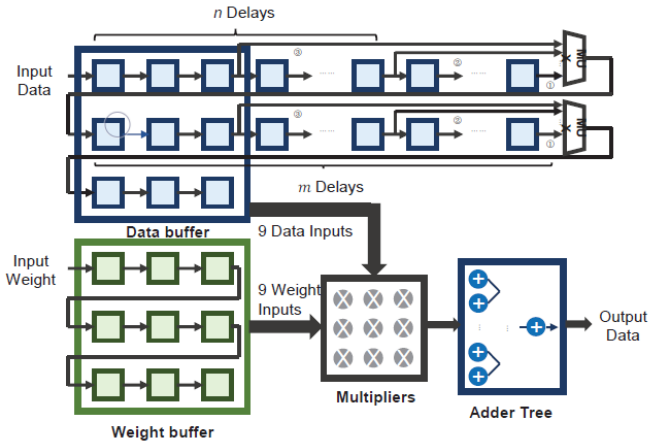


Fig. 3. Line Buffer Convolution [6]

of the convolution windows; hence, the filter weights is the vector and the activations is the matrix. This is seldomly done in FPGA accelerators as it requires duplicating and storing a lot of the input feature maps, which is wasteful. The alternative is to use a line buffer design where rows of input activations are buffered and the filter window is selected from the buffer, as done in [6] and visualized in Fig. 3. This avoids the unnecessary replication of input activations, and is very compatible with the streaming compute pattern to maximize throughput. The line buffer convolver can be replicated to have the same activations convolve with different filters to parallelize along the output channel dimension. The same is usually not done to parallelize the input channel dimension because that requires duplicating both the activation and filter weights buffers, which is too costly.

To overcome this limit, one could increase the granularity of the line buffer by breaking one convolution window into smaller ones and accumulate the partial sums. For example, [7] proposes to only buffer one row of the input feature map, and trades of the space to buffer the same row from multiple input feature maps, hence parallelizing the input channel dimension, as shown in Fig. 4. The benefit of parallelism along the input channel dimension is the ability to reuse the filter weights, in addition to the activations.

Orthogonal to these optimizations that replicate the dot product PE in various ways, the second disadvantage of limited flexibility could be solved with the layer-pipeline architecture. Instead of using a PE of uniform size for all layers in a network, the layer-pipeline architecture maps each layer as a pipeline stage on the FPGA, and builds dedicated PEs for each of them that accounts for the specific compute and memory demands [9]. The implication is that each PE has to be made smaller, given the sane FPGA resources. For FC or self-attention layers, this simply means reducing the tile size, but for convolution layers that adopt a line buffer-based design, it would mean to further split up the convolution window. Specifically, in [9], instead of buffering one row, it buffers a few columns (which are usually shorter than row for images),

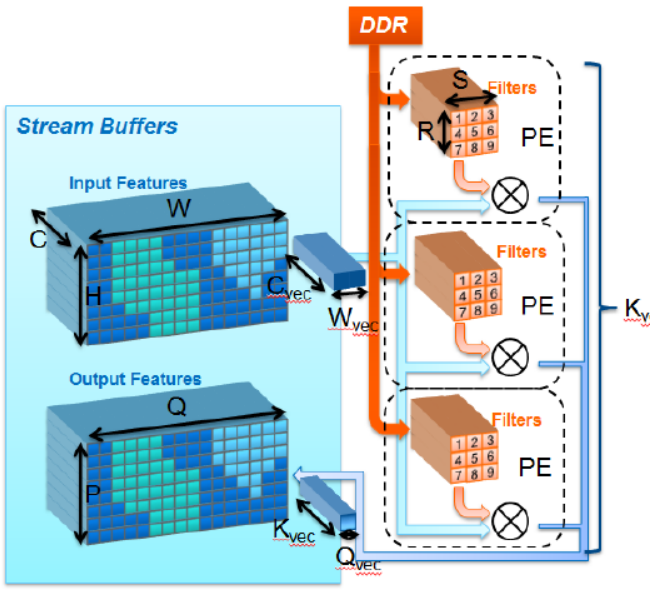


Fig. 4. One Row of Line Buffer [7]

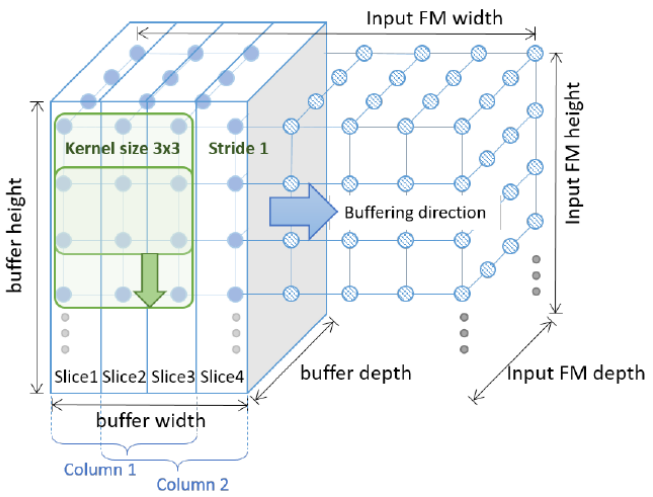


Fig. 5. Column buffer Convolution [9]

and computes the dot product between a feature map and a filter one multiplication at a time (though still parallelized in the input and output feature dimension), as illustrated in Fig. 5 and Fig 6. Another implication is that the output of an inference has to be generated part by part since it is usually not feasible to store all the intermediate activations generated from each layer on chip. This makes the layer-pipeline architecture particular suitable for object detection tasks that may start producing valid output after processing only a portion of the image, as shown in Fig. 7.

The optimizations discussed so far all tried to increase the amount of processing units to achieve a higher peak throughput, while there are another class of techniques that aim to reduce the amount of operations needed in the first place. The

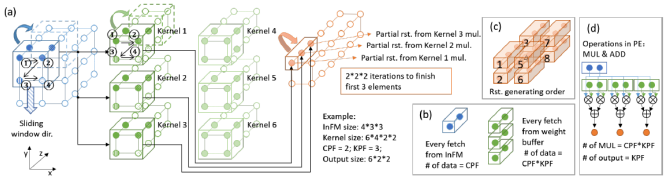


Fig. 6. One Pixel Column buffer Convolution [9]

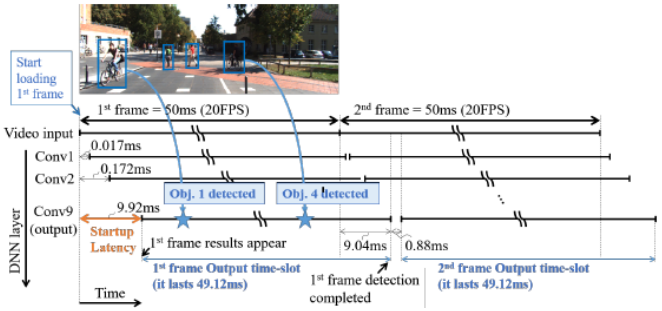


Fig. 7. Early Detection [9]

best example of this is [7], where the convolution of one line of input feature map and the filter weights is transformed to a dot product using the Winograd's minimal filter algorithm, which significantly reduce the number of multiplications and additions needed, especially for small filters. Fig. 8 illustrates the Winograd workflow. Note that as filter weights are cached and input features are reused in [7], cost of the transformation is amortized.

Alternatively, model compression techniques also reduces the amount of operation required by representing the model with fewer and/or smaller weights. These methods be discussed in details in Section IV.

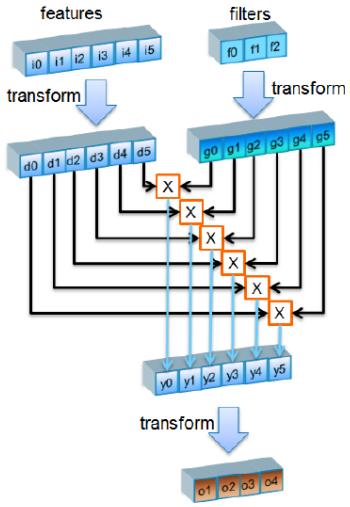
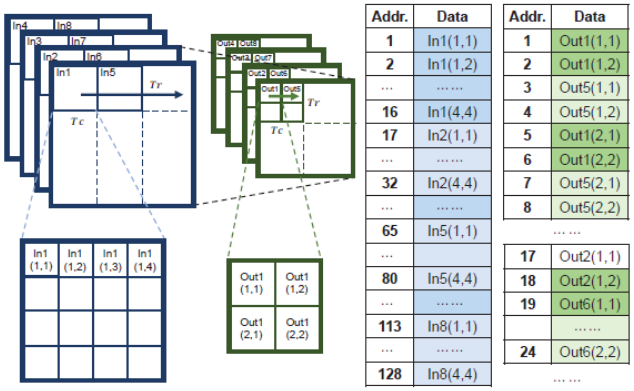


Fig. 8. Winograd Workflow [7]



Addr.	Data
1	In1(1,1)
2	In1(1,2)
...	.....
16	In1(4,4)
17	In2(1,1)
...	.....
32	In2(4,4)
...	.....
65	In5(1,1)
...	
80	In5(4,4)
...	.....
113	In8(1,1)
...	.....
128	In8(4,4)

Addr.	Data
1	Out1(1,1)
2	Out1(1,2)
3	Out5(1,1)
4	Out5(1,2)
5	Out1(2,1)
6	Out1(2,2)
7	Out5(2,1)
8	Out5(2,2)
... ...	
17	Out2(1,1)
18	Out2(1,2)
19	Out6(1,1)
	.....
24	Out6(2,2)
... ...	

Fig. 9. Convolution Layer Memory Storage Pattern [6]

III. Memory Bound Optimization

While compute bound optimizations improves the peak throughput, memory bound optimization ensures that the utilization of these parallel compute units are high, by feeding them data on time. This effectively means to reduce the memory bandwidth demand, by fetching less data from the memory (or simply use memory with higher bandwidth as in [8], but that is not always possible, and there is an upper limit for how much bandwidth even the top memory chip can provide).

The obvious way to reduce memory bandwidth is to reuse the data brought from memory as much as possible. Earlier works like [5] has no reuse strategy, fetching activations and weights from the memory each time while storing intermediate activations back to memory. [6] improves on this by storing weights and input activations used by a layer in on-chip buffers, and accumulate the intermediate results generated by each tile in another buffer. However, after all output for a layer is generated, they are stored to external memory and then loaded back for next layer, incurring unnecessary memory movement. Though, it does optimizes the memory storage pattern of the input and output feature maps such that the activations used by all tiles of PEs can be loaded continuously, as shown in Fig. 9. b7 further optimizes this by caching all intermediate feature maps on chip, reusing the outputs of a previous layer as inputs for the next layer.

Nevertheless, for certain layers, namely the FC layers, there is no reuse opportunity to begin with. This is made worse as prior studies have point out that while other layers are compute bound (before acceleration), FC layers are memory bound by nature [6], [7], [9]. To address this issue, [7] computes the FC layer in batches so the weights can be reused. This method would not be suitable for latency sensitive inference tasks as a batch needs to be accumulated before the final results can be computed. Alternatively, [9] proposes to use more aggressive compression techniques for FC layer weights, while also adjusting the memory bandwidth resources allocated to each layer in the layer-pipeline architecture.

Aside from data reusing, scheduling may also alleviate

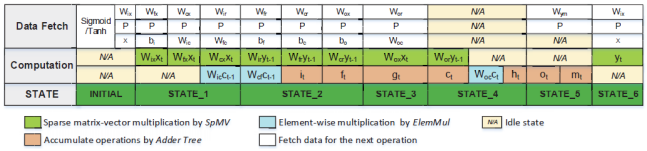


Fig. 10. ESE Schedule, operations in the horizontal direction and vertical direction are executed sequentially and concurrently respectively [12]

the memory bound by overlapping computation with memory transfer. A common form of this is double buffering where two buffers alternatively stores new data fetched from memory and old data used in the current computation. [12] proposes a more sophisticated scheduling for LSTM networks that consists of several FC and element-wise operation layers in each recurrent iteration, as shown in Fig. 10. In contrast, other network architectures would require more granular scheduling within each layer as they don't have this many independent sub-layers. Such scheduling highly depends on the data reuse strategy and mainly involves sequencing the computation performed by each tile or PE.

Lastly, one aggressive approach is to store all the weights and intermediate activations on chip and make no request to the external memory, as done in [10] and [11]. This works particularly well for FPGAs because the on chip SRAM (and LUTRAM) is highly distributed, meaning that the data can usually be placed very close to the compute units. When the limited SRAM buffers on chip cannot hold all these data, one would use multiple FPGAs each computing a portion of the whole network. Still, even with multiple FPGAs, for any recent neural networks, this would only be feasible after model compression, which will be discussed in details in Section IV. Note that even for approaches that does not keep everything on chip, model compression still helps with overcoming the memory bound, as the model is smaller, requiring less data transfer.

IV. Model Compression

An interesting property of neural networks is that they are amenable to approximate computing/model compression: these highly over-parameterized models can achieve almost the same accuracy when the parameters are represented with less precision (quantization) or when some less significant parameters are removed (pruning).

A. Quantization

Quantization is well suited for FPGA accelerators as they mostly contain DSPs that operate on fixed point, and low precision data types. The simplest form of quantization is to use 16 bit floating point instead of 32 bit floating point, which was shown to incur virtually no accuracy lose. [7] optimizes this by transforming the floating values to a shared exponent before computation, so that the expensive floating point operations become cheaper fixed point ones that map efficiently to DSPs. [11] shows success with even lower precision floating point called BFP, which has 5 bit shared

Algorithm 1: FPGA-aware mixed-scheme ViT quantization.

input : 32-bit floating-point pre-trained ViT model $\mathcal{M}$ with weights $\mathbf{W}$; bit-width for fixed-point b; bit-width for PoT b'; ratio of PoT quantized rows in each layer k_{PoT};

output: Quantized model $\hat{\mathcal{M}}$.

foreach *batch* **do**
  // forward propagation
  foreach *layer* i in $\mathcal{M}$ **do**
    // calculate variance for each row
    foreach row $\mathbf{W}_{ij}$ in *layer* i **do**
      $\text{var}_{ij} \leftarrow variance(\mathbf{W}_{ij})$;
    // assign weight quantization scheme for rows
    foreach row $\mathbf{W}_{ij}$ *in layer* i **do**
      if var_{ij} *belongs to the bottom* k_{PoT} *group* **then**
        $\hat{\mathbf{W}}_{ij} \leftarrow \prod_{b'}^{\text{PoT}}(\mathbf{W}_{ij})$;
      else
        $\hat{\mathbf{W}}_{ij} \leftarrow \prod_{b}^{\text{Fixed}}(\mathbf{W}_{ij})$;
    $\mathbf{A}_i \leftarrow \hat{\mathbf{W}}_i \cdot \hat{\mathbf{A}}_{i-1}$;
    // quantize activations
    $\hat{\mathbf{A}}_i \leftarrow \prod_{b}^{\text{Fixed}}(\mathbf{A}_i)$;
  // backward propagation
  foreach *layer* i *(reverse order)* **do**
    $\frac{\partial Loss}{\partial \mathbf{W}_i} \leftarrow \frac{\partial Loss}{\partial \hat{\mathbf{W}}_i}$;
    $\frac{\partial Loss}{\partial input_i} \leftarrow \frac{\partial Loss}{\partial \hat{\mathbf{A}}_{i-1}}$;
Return $\hat{\mathcal{M}} \leftarrow \mathcal{M}\{\hat{\mathbf{W}}\}$.

Fig. 11. [15]

exponent and only 2-5 bit mantissa. Aside from low precision floating point quantization, it is also common to use fixed point quantization. 16 bit appears to be the standard for balancing accuracy and computation/storage reduction, as adopted in [10]. Furthermore, some works propose to use lower precision by using different dynamic range for each layers, as some layers can tolerate even 2 or 4 bit quantization as in [6] and [9]. [15] further increases the quantization granularity and uses different quantization strategy for each row in a weight matrix. Specifically, each row may be quantized in fixed intervals or in powers of two (PoT), whose computations maps to DSPs and LUTs respectively, as shown in Algorithm 1. This uses the heterogeneous resources on FPGA more efficiently, but also incurs more accuracy drop and requires non-trivial fine-tuning/retraining of the network.

Besides reducing storage requirement (hence memory bandwidth), one key advantage of low precision quantization is that they allow DSP packing techniques so that with the same number of DSPs, the amount of operations performed increases. Though this technique is less relevant for modern FPGAs such as Stratix 10 NX that features DSPs that directly operate on a lot of low precision values [15]. In addition to DSPs, such low precision operations may also be implemented with look-up tables or soft logic [11], allowing even more processing units to be instantiated on the chip.

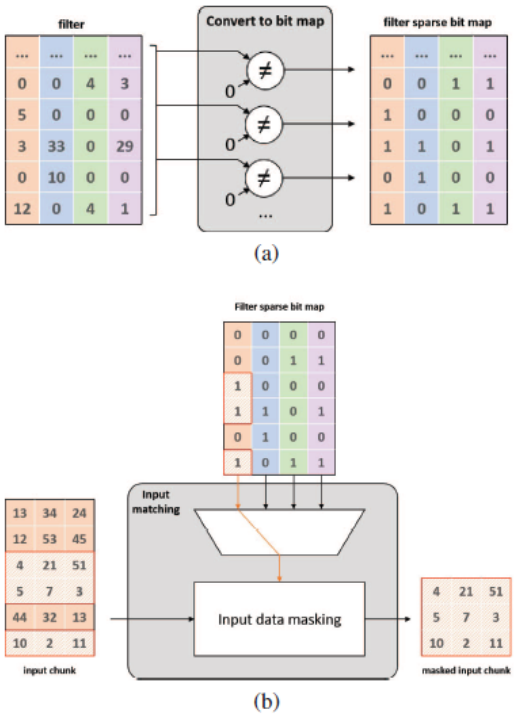


Fig. 12. (a) Representing positions of zero and non-zero numbers with sparse bitmap. (b) Compress input feature maps based on the filter sparse bitmap. [8]

B. Pruning

Pruning removes weights with values lower than a threshold, and results in sparse matrices. The amount of sparsity depends on the threshold, and the sparsity pattern depends on pruning granularity.

At the finest granularity, each weight is considered for pruning, which results in irregular sparsity. [8] proposes a sparse matrix packing module that dynamically extracts the corresponding activations given sparse weight matrices as shown in Fig. 12. It then accumulate the matching activations in a buffer, and feeds those to the PEs when the buffer is full as shown in Fig. 13; hence fully utilizing the PEs regardless of the sparsity. This introduces no change to the PE design nor the tiling/scheduling, though it only reduces the amount of compute, not the memory bandwidth as the weights are still stored in dense format, and the sparsity is extracted at run time.

In contrast, [12] works on weights stored in compressed format. It broadcasts each activation to PEs each holding a subset of the weight matrix and performs MAC on the non-zero weights, as shown in Fig. 14. It can be seen that the work in each PE is not balanced due to the irregular sparsity pattern, resulting in wait time. Thereby, authors of [12] present load balance-aware pruning that keeps the level of sparsity in each submatrix uniform, as shown in Fig. 15.

Pruning may also be done at coarser granularity to produce regular sparsity which makes it easier to parallelize the com-

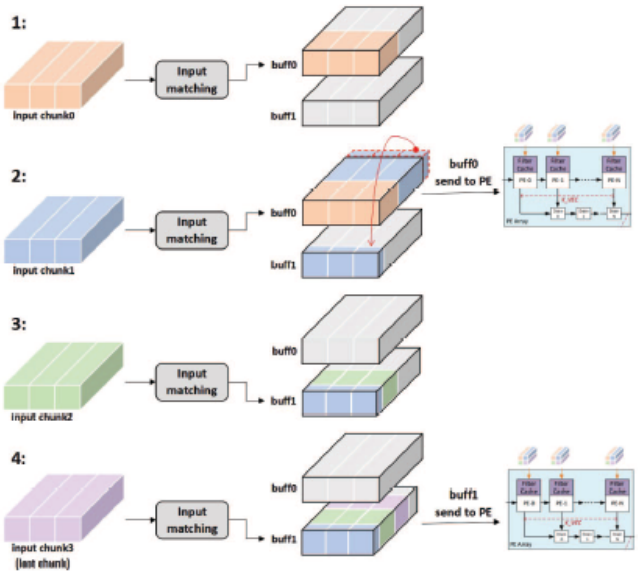


Fig. 13. Accumulating input feature maps after sparsity matching [8]

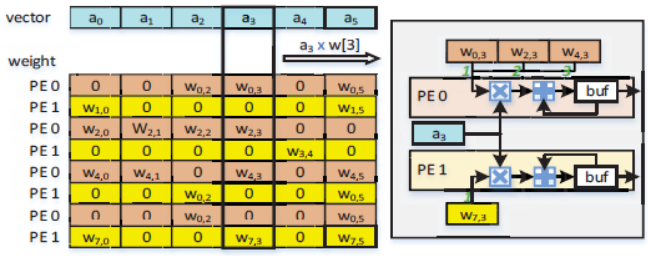


Fig. 14. The computation pattern: non-zero weights in a column are assigned to 2 PEs, and every PE multiply-add their weights with the same element from the shared vector. [12]

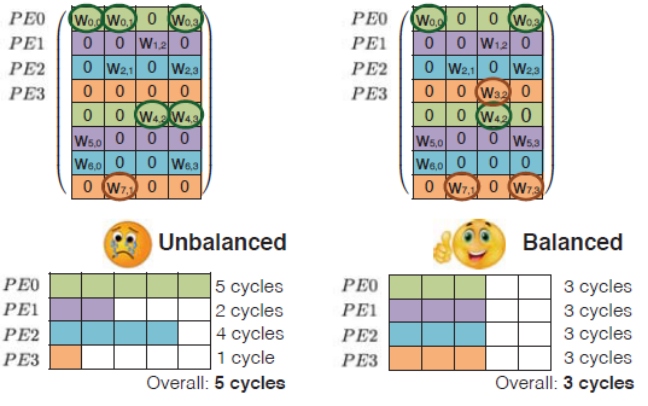


Fig. 15. Load Balance Aware Pruning and its Benefit for Parallel Processing. [12]

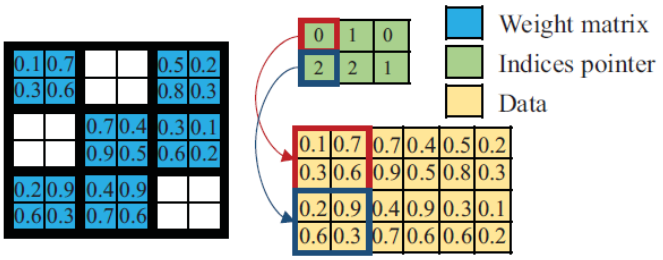


Fig. 16. Example of column balanced block-wise sparse matrix and its storage format, block size 2x2. [14]

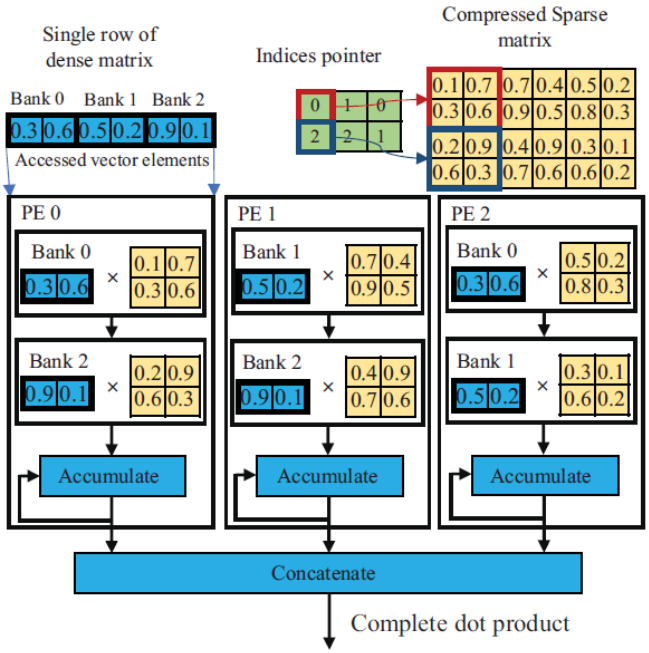


Fig. 17. PE for column balanced block-wise sparse matrix vector multiplication. [14]

putation. For example, b14 extends the idea of load balance-aware pruning with column-balanced block pruning, where weights are pruned away in blocks and the number of pruned blocks in each column is constant, as illustrated in Fig. 16. Not only does this leads to simple PE design, but also balanced workload for each PE, as depicted in Fig. 17.

C. Other Techniques

[6] compresses the FC layer weights by applying singular value decomposition (SVD) and keeping only the first d singular values. It reduces the number of weights by 7.04 times while incurring only a 0.04% accuracy loss.

V. Automated Framework

Given the vast design space of FPGA accelerators, many works also include some form of automated framework to explore the design space. For uniform PE architecture, this involves finding the best configuration for the PE and the

tiles, which translates to the amount of parallelizing in each dimension (input and output channel, height/width of input activation, etc.). On the other hand, for layer-pipeline architecture ([9] and [10]), it entails finding the optimum combination of PE and tile configurations for each layer.

Either way, they all require establishing a performance model based on the amount of FPGA resource (DSP, LUT, BRAM, etc.) used and the network architecture which determines the compute and memory demand. The alternative would be to let the CAD tool compile each design point and measure the performance on an actual FPGA, which is more accurate, but often forbiddingly slow given the sheer number of configurations.

VI. Evaluation

Table 1, Table 2 and Table 3 respectively summarizes the approach used by each design, their FPGA utilization, and their performance.

VII. Suggestions

It is probable that different neural network architectures will arise as deep learning continues to evolve. New architectures would likely require different accelerator design, as the speed up from FPGA largely comes from its ability to make custom circuit for a specific network architecture, or even a specific model. Nevertheless, most of the accelerator design today is still manual, which is not sustainable with the fast evolving pace of deep learning. Future works should aim to improve automated framework for mapping neural networks to FPGAs, including more accurate performance modelling, faster run time, tighter integration with software framework, etc. The work in [10] is a good starting point as it is able to map a TensorFlow model to FPGA. Though, it is only able to handle a few types of layers, and suffers a performance loss when mapping certain layer types.

The improved automated framework should be able to map various network architectures and models, which is only possible by not limiting itself to only one specific PE/datapath design and model compression scheme. Ideally, it should consider different accelerator architectures and experiment with different quantization/pruning strategies, and find the most optimum combination.

It is notable that [10] customizes the FPGA CAD tool (improving memory mapping and fan-out reduction algorithms) which leads to a substantial increase in frequency and overall performance. This suggests that future automated framework should also include better CAD tools, in addition to HDL generation.

While the neural network architecture changes, model sizes also grow relentlessly, especially for recent transformer models. On the one hand, this requires more research into model compression techniques, though there is a limit on how much a model can be compressed without degrading its performance noticeably. Thereby, on the other hand, future works should study on how to efficiently scale up FPGA accelerators. For example, multiple boards may work together to accelerate a

TABLE I
Design Approach Summary

Design	***Applicability***	***Compute***	***Memory***	***Model Compression***
[5]	General	Single dot product unit	No weights and activation reuse, double buffering	None, FP32
[6]	Conv and FC	Line buffer, input and output channel parallelism, height and width tiling	Per layer weights and activation on chip buffering	Per layer quantization (INt4-INT16 bit), SVD
[7]	Conv and FC	Single row, multiple depth line buffer, Winograd, input and output channel parallelism, input and output width parallelism	Double buffering, Per layer weights and activation on chip buffering, inter-layer activation on chip buffering, FC batching	Shared exponent FP16
[8]	Conv and FC	Same as [7] + sparse matrix input matching	Same as [7] + high bandwidth memory HBM2	Per weight pruning
[9]	Conv and FC	Column buffer, single MAC, input and output channel parallelism, column tiling	Layer-pipeline, double buffering, adjustable per layer bandwidth resource	Per layer quantization
[10]	Conv and FC	Same as [9] + sparse gather based convolution	Same as [9], except all weights on chip	INT16 quantization
[11]	RNN and FC and Conv	Soft processor with MVM primitives	Processor style transfer (register file transfer), all weights on chip	BFP, 1bit sign, 5 bit exponent, 2-5 bit mantissa
[12]	LSTM	Single MAC on non-zero activation and weight	Activation broadcast, sub-layer scheduling	INT12 quantization and load balance-aware pruning
[13]	MSA and FC	MVM PE, tiling on weight matrix column and multiple heads	Layer pipeline, similar to [9]	Column balanced block pruning
[14]	MSA and FC	Different PE for fixed and PoT quantization, tiling on input channel, multiple heads, and output channel dimension. Exact PE design not discussed	Not discussed, presumably not layer-pipeline	Mixed scheme per matrix row quantization (Fixed W4A4 + PoT W3A4)

TABLE II
FPGA UTILIZATION SUMMARY

Design	*FPGA*	*LUT (%)*	*DSP (%)*	*FF (%)*	*BRAM (%)*	*freq (MHz)*
[5]	XC7Z020	68.4	75.9	26.6	12.5	200
[6]	ZC706	83.5	89.2	29.2	86.7	150
[7]	Arria 10 1150	58	97	-	92	303
[8]	S-10 MX	45	36	-	-	257
[9]	ZC706 (AlexNet)*	39	90	12	56	200
[9]	ZC706 (ZF)*	40	92	12	61	200
[9]	ZC706 (VGG16)*	52	76	14	99	200
[9]	ZC706 (YOLO)*	39	76	11	61	200
[9]	KU115 (AlexNet)*	40	88	13	46	220
[9]	KU115 (ZF)*	40	90	14	54	225
[9]	KU115 (VGG16)*	39	78	13	81	235
[9]	KU115 (YOLO)*	40	96	13	63	220
[10]	S10 2800 (ResNet50)*	63	87	-	96	580
[10]	S10 2800 (MobileNetV1)*	40	89	-	37	430
[10]	S10 2800 (MobileNetV2)*	38	51	-	40	530
[11]	S10 2800	91	91	-	69	250
[12]	XCKU060	88.6	54.4	68.3	87.7	200
[13]	Alveo U200	45.8	49.3	27.8	-	-
[14]	ZCU102	65	62	-	-	-

- Utilization is not given.
* Layer-pipeline utilization varies with the network.
# Value not directly reported by paper, but can be calculated based on reported ones.
! Total number of FF unkown, reporting exact number of FF used.

single model. How to schedule the workload and efficiently reuse / transfer the data and intermediate results would be the focus.

VIII. CONCLUSION

Overall, this paper has reviewed many of the state-of-the-art FPGA accelerator designs for a variety of neural network architectures. Based on current trend of deep learning development, it is suggested that future studies aim to build more capable automated framework, and devise ways to scale up the accelerator for larger models.

REFERENCES

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," Nature, vol. 521, no. 7553, pp. 436–444, May 2015, doi: 10.1038/nature14539.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," presented at the NAACL, minneapolis, Minn., USA, Jun. 2019, vol. 1, pp. 4171–4186.

[3] A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," presented at the ICLR, May 2021. [Online]. Available: https://iclr.cc/virtual/2021/poster/3013

TABLE III
OVERALL PERFORMANCE

Design	*Network*	*TP (GOP/s)*	*TP (img/s)*	*Power (W)*	*Eff (GOPS/W)*	*Eff (img/s/W)*
[5]	LeNet	12.8*	-	1.814	7.06*	-
[6]	VGG16	136.97	-	9.63	14.22	-
[7]	AlexNet	1382	1020	45	30.71	23
[8]	MobileNet	-	725.5	36.7#	-	19.77
[8]	ResNet50	-	1581.2	174.3#	-	9.07
[9]	AlexNet (ZC706)	494	340	7.2	68.6#	47.2
[9]	ZF (ZC706)	526	224.4	7.2	73.1#	31.2#
[9]	VGG16 (ZC706)	524	55.4	7.2	72.8	7.7#
[9]	YOLO (ZC706)	468	44.2	7.2	65#	6.1#
[9]	AlexNet (KU115)	3265	2252	22.9	142.6#	98.3
[9]	ZF (KU115)	3552	1518	22.9	155.1#	66.3#
[9]	VGG16 (KU115)	4022	130	22.3#	180.4	5.83#
[9]	YOLO (KU115)	4218	398	22.9	184.2#	17.4#
[10]	ResNet50	33200	4551	-	-	-
[10]	MobileNet V1	10600	5157	-	-	-
[10]	MobileNet V2	3900	6023	-	-	-
V100	ResNet50	8400 [10]	1156 [10]	300! [16]	28#	3.9#
V100	MobileNet V1	5100 [10]	4605 [10]	300! [16]	17#	15.4#
[11]	GRU	35900	503#	125	287.2#	4.0 #
[11]	LSTM	22620	13512#	125	181.0#	108.1 #
[11]	ResNet50	4080#	559	125	32.6#	4.5 #
[12]	LSTM	2515.7	12092#	41	61.4#	294.9#
Titan X	LSTM	-	4161# [12]	202 [12]	-	20.6#
[13]	Small transformer	-	3091.8	-	-	-
Jetson TX2	Small transformer	-	1485.6 [14]	-	-	-
[14]	DeiT base	1970.3	56.8	11.03	178.6#	5.15
Jetson TX2	DeiT base	-	7.87 [15]	12.28 [15]	-	0.6#

- Not reported.
*For papers that don't provide throughput, the throughput reported here is the calculated peak throughput, and the efficiency is based on this as well.
Value not directly reported by paper, but can be calculated based on reported ones.
! TDB is assumed as power usage.

[4] P. Villalobos et al., “Machine Learning Model Sizes and the Parameter Gap,” 2022, arXiv:2207.02852.
[5] C. Wang et al. “DLAU: A Scalable Deep Learning Accelerator Unit on FPGA,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 36, no. 3, pp. 513–517, Mar. 2017, doi: 10.1109/tcad.2016.2587683. [Online]. Available: https://arxiv.org/pdf/1605.06894.pdf
[6] J. Qiu et al., “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” presented at Proc. of the ACM/SIGDA ISFPGA, Monterey, CA., USA, Feb. 2016, pp. 26–35, doi: 10.1145/2847263.2847265.
[7] U. Aydonat et al., “An OpenCLTM Deep Learning Accelerator on Arria 10,” presented at Proc. of the ACM/SIGDA ISFPGA, Monterey, CA., USA, Feb. 2017, pp. 55–64, doi: 10.1145/3020078.3021738. [Online]. Available: https://arxiv.org/pdf/1701.03534.pdf
[8] C. Jiang, D. Ojika, B. Patel, and H. Lam, “Optimized FPGA-based Deep Learning Accelerator for Sparse CNN using High Bandwidth Memory,” presented at the FCCM, May 2021, doi: 10.1109/fccm51124.2021.00026.
[9] X Zhang et al., “DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs,” presented at the ICCAD, Nov. 2018, doi: 10.1145/3240765.3240801.
[10] M. Hall, V. Betz, “From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation,” presented at the ICFPT, May 2021, doi: 10.1109/ICFPT51103.2020.00017.
[11] J. Fowers et al., “A Configurable Cloud-Scale DNN Processor for Real-Time AI,” presented at the ISCA, Jun. 2018, doi: 10.1109/isca.2018.00012. [Online]. Available: https://www.microsoft.com/en-us/research/uploads/prod/2018/06/ISCA18-Brainwave-CameraReady.pdf
[12] S. Han et al., “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA,” presented at the Proc. of the ACM/SIGDA ISFPGA, Monterey, CA., USA, Feb. 2017, pp. 75–84, doi: 10.1145/3020078.3021745. [Online]. Available: https://arxiv.org/pdf/1612.00694.pdf
[13] H. Peng et al., “Accelerating Transformer-based Deep Learning Models on FPGAs using Column Balanced Block Pruning,” presented at the ISQED, Apr. 2021, doi: 10.1109/ISQED51717.2021.9424344. [Online]. Available: https://wangshusen.github.io/papers/ISQED2021.pdf
[14] Z. Li et al., ”Auto-ViT-Acc: An FPGA-Aware Automatic Acceleration Framework for Vision Transformer with Mixed-Scheme Quantization”, 2022, arXiv:2208.05163
[15] A. Boutros et al., “Beyond Peak Performance: Comparing the Real Performance of AI-Optimized FPGAs and GPUs,” presented at the ICFPT, Maui, HI, USA, Dec. 2020, doi: 10.1109/ICFPT51103.2020.00011.
[16] “NVIDIA TESLA V100 GPU ARCHITECTURE” NVIDIA, Santa Clara, CA, USA, WP-08608-001_v1.1, 2017 [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf