Evaluating Low-Memory GEMMs for Convolutional Neural Network Inference on FPGAs

Wentai Zhang
Department of Computer Science
Peking University
Email: rchardx@gmail.com

Ming Jiang
School of Mathematical Sciences
Peking University
Email: ming-jiang@pku.edu.cn

Guojie Luo
Department of Computer Science
Peking University
Email: gluo@pku.edu.cn

***Abstract*—FPGAs are becoming significant for implementing low-latency convolutional neural networks, because of performance demands and power constraints. Conventional implementations of convolutional layers are usually direct convolution, involving nested loops over channels, feature maps, and filters. Explicit general matrix multiplications (GEMMs) cost extra memory space, and the limited on-chip RAMs prevent an efficient GEMM-based implementation. In this paper, we evaluate a low-memory method of GEMMs on FPGAs based systolic arrays. We design a novel accelerator to save the bandwidth and increase the parallelism. We evaluate our design on MobileNet V1 and Inception V4. Our implementation achieves a throughput of around 3.5 TOP/s for both models. We also reduce the memory usage by 21% compared to explicit GEMM implementation for MobileNet V1 and 44% for Inception V4.**

I. Introduction

FPGAs are playing more and more significant roles in convolutional neural network (CNN) inference because of their high performance and energy efficiency. Popular CNNs such as ResNet [1], Inception [2], and VGG [3] consist of many convolutional layers. Convolutional layers involve massive floating-point multiply-accumulate operations (MAC) and require numerous parameters to infer. Therefore, CNN inference is both memory- and compute-intensive. The latest FPGAs incorporate many hardware resources, including DSPs, on-chip BRAMs, and enormous logic cells, to provide effective computational power. There are many works using FPGAs to implement low-latency and high-throughput CNN inference.

There are various implementations for convolutional layers. The first kind of implementation is element-wise direct convolution, used in many FPGA accelerators [4], [5], [6], [7], [8], [9], [10]. Optimization for direct convolution is usually based on polyhedral model [11] and tiles the feature maps/weights to utilize the on-chip memory. However, the nested loops of direct convolution are controlled by too many counters, including the number of input/output channels, the filter size, etc. The cost of exploring the design space is great, and it is difficult to optimize the performance. The second kind of implementation is general matrix multiplications (GEMMs), widely used on the platforms like CPUs, GPUs, and ASICs. GEMMs are usually based on computational libraries such as the Basic Linear Algebra Subprograms (BLAS) [12] or NVIDIA cuBLAS [13]. The third kind of implementation uses fast algorithms for convolution or matrix multiplication. There are Fast Fourier Transform-based algorithm [14] and Winograd algorithm [15]. These algorithms can outperform conventional BLAS libraries in some cases. Winograd algorithm is particularly suitable for convolutional layers with 3×3 filter size (very common in ResNet [1]). Moreover, recent studies [16], [17] can replace a large-filter convolutional layer with several small-filter convolutional layers. For example, Inception model [16] decomposes the $k \times k$ convolution into $1 \times k$ and $k \times 1$ convolutions.

Computing convolutional layers via GEMMs works efficiently on CPUs [18], GPUs [13], and ASICs [19]. However, GEMMs have one fatal drawback when implemented on FPGAs. GEMMs transform the convolutional layers into matrix multiplications through reshaping and packing, such as image to column ("im2col") [20]. The additional reshaping and packing require extra memory space, addressing operation, and hence incur a non-trivial time penalty. The on-chip memory is very limited on FPGAs, and the off-chip communication is usually the bottleneck of FPGA designs. Therefore, data redundancy makes GEMMs on FPGAs inefficient. On other platforms, data redundancy is not a serious problem. For von Neumann architecture (typically CPUs), the parallelism (the data width processed per instruction) is the bottleneck. ASICs utilize plentiful silicon area for the on-chip buffers to satisfy the bandwidth and capacity requirement (e.g. unified buffer in TPU [19] occupies one-third of the total area). Both the latest FPGAs (like Xilinx U50/U280 cards and Intel Stratix 10 MX FPGAs) and TPU V3 use faster memory modules like HBM2 [21]. However, HBM2 is far more expensive than conventional DDR memory.

In this paper, we evaluate a low-memory GEMM method for CNN inference on FPGAs to overcome the data redundancy. By rewriting GEMM calls using different data layouts, we can explore the execution order and the memory requirements. We evaluate an algorithm that requires $O(KW)$ additional space [22]. This makes GEMMs feasible on FPGAs and consumes fewer BRAMs. We also apply FPGA-specific optimization on it to further increase the data reuse rate and parallelism. We test our design on MobileNet V1 and Inception V4. We reduce 21% on-chip memory usage for MobileNet V1 and 44% for Inception V4 while achieving a throughput of 3.5 TOP/s.

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00013

II. Background

The convolutional layers dominate both the computation and storage of the network execution [20] in a CNN model. Therefore, the focus of designing a CNN inference accelerator falls on the architecture of convolutions. The representation of direct convolution contains nested loops with 6 upper bounds (M, C, H, W, K_h, K_w). M and C are the numbers of output and input channels. H and W are the sizes of the input feature map. K_h and K_w are the filter sizes of the convolution. In most cases, K_h is equal to K_w and we denote them as one variable $K = K_h = K_w$. When K_h does not equal K_w, we denote $K = \max\{K_h, K_w\}$. Optimization for direct convolution usually involves tiling, as shown in Listing 1 (S is the stride of convolution). Listing 1 is a two-level loop tiling dataflow by partitioning (H, W, M, C) and fully unrolling the inner kernel. Direct convolutions use $O(CHW)$ space for the input feature map, $O(MCK_hK_w)$ for the weights, and $O(MHW)$ for the output feature map, without extra memory space.

```
for(m=0;m<M;m+=Tm){          //output feature
 for(c=0;c<C;c+=Tc){         //input feature
  for(h=0;h<H;h+=Th){        //feature row
   for(w=0;w<W;w+=Tw){       //feature column
    for(th=h;th<min(h+Th,H);++th){
     for(tw=w;tc<min(w+Tw,W);++tw){
      for(tm=m;tm<min(m+Tm,M);++tm){
       for(tc=c;tc<min(c+Tc,C);++tc){
        //unrolling the inner loop body
        for(i=0;i<Kh;i++){  //filter size
         for(j=0;j<Kw;j++){ //filter size
          out[tm][th][tw]+=w[tm][tc][i][j]
                  *in[tc][S*th+i][S*tw+j]
}}}}}}}}}}
```

Listing 1. Direct convolution with tiling.

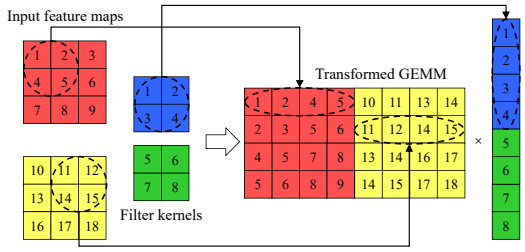

Fig. 1. The "im2col"+GEMM (explicit GEMM) method.

"im2col"+GEMM [20] (explicit GEMM) is one of the common solutions used in CPUs and GPUs. In Fig. 1, we demonstrate how "im2col" works. We have red and yellow input feature maps and two filter kernels for them. The number of input channels M is 2. The number of output channels C is 1. The shape of feature maps (H, W) is $(3, 3)$. Kernel size K_h and K_w are both 2. We reallocate the MACs as dotted circles and solid arrows indicate. In the end, we have a 4×8 matrix multiplying another 8×1 matrix. We usually denote this as a $(M' = 4, N' = 1, K' = 8)$ or (4,1,8) GEMM. "im2col" requires $O(K_hK_wCHW)$ extra memory space for the transformed feature map matrix. If $K_h = K_w = 1$, explicit GEMM does not suffer from the data replication.

Explicit GEMM requires too much memory (K_hK_w-level), and there exist other alternatives. For example, implicit GEMM used by NVIDIA Tensor Core [23] computes directly on the convolution input feature maps and converts the operations as matrix multiplications on the fly. Implicit GEMM is still a variant of direct convolution and is not as effective as explicit GEMMs. We will evaluate the effectiveness of "kernel to row" based GEMM on FPGAs in this paper.

III. Methodology

A. Low-Memory GEMM Algorithm

Toeplitz-based "im2col" method transforms a 3D input tensor into a 2D matrix. The kernel to row ("kn2row") method is a 4D extension of Toeplitz-based transformation. The "kn2row" method is based on the decomposition of convolutions and reorder the data layout. A $K_h \times K_w$ convolution can be computed using K_hK_w 1×1 convolutions. We shift and add the intermediate outputs to achieve the final results.

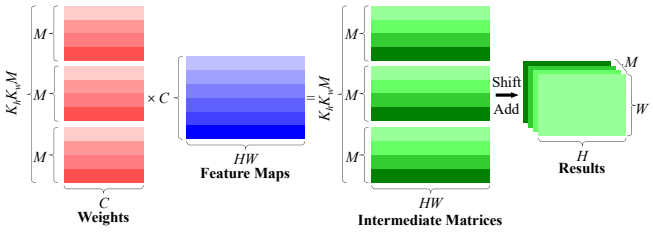

Fig. 2. The "kn2row" method.

As we have discussed in Section II, 1×1 convolution can be done without extra memory workspace. $K_h \times K_w$ convolution can be expressed as the sum of $K_h \times K_w$ separate 1×1 convolutions. Each 1×1 convolution will produce a matrix with dimension $M \times HW$, but all these matrices correspond to different kernels value back to $K_h \times K_w$ kernel. We can add these matrices by offsetting every pixel in every channel vertically and/or horizontally. For example, if $K_h = K_w = 3$ (3×3 sized filter), the central point of the kernel outputs a perfectly aligned matrix after the multiplication. This perfectly aligned matrix does not need to be shifted. For the upper-right value of the 3×3 kernel, the output matrix must be offset up and right by one pixel. After shifting, the intermediate matrices can be out of final bounds. These out-of-bounds values should be discarded. In Fig. 2, we give an illustration of "kn2row". The extra memory space is the intermediate matrices marked by the green color. Hence, "kn2row" method requires an extra workspace of $O(K_wK_hMHW)$.

Anderson et al. proposed the accumulating "kn2row" [22], a variant with only $O(KW)$ extra memory consumption. In the conventional "kn2row", the intermediate matrices are generated by $K_h \times K_w$ GEMM operations. If these GEMM operations work in an accumulating way ($O = A \times B + O$), K_hK_w intermediate matrices can be reduced to two matrices: one accumulating buffer and one intermediate buffer. We illustrate the accumulating "kn2row" in Fig. 3. The accumulating algorithm regroups the weight matrix as K_hK_w smaller $M \times C$

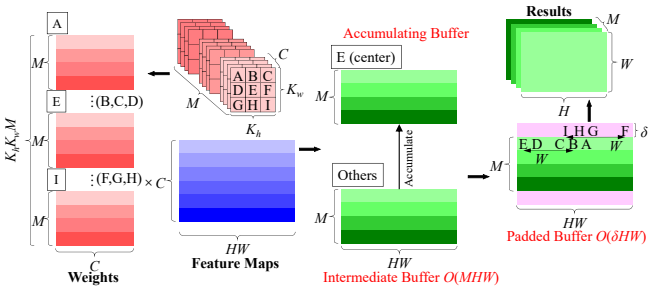

Fig. 3. The accumulating "kn2row" method.

sized matrices. In Fig. 3, these matrices are labeled from A to I (A,B,C,...,I) according to their original positions in the filter kernel. The accumulating "kn2row" algorithm uses an extra buffer to accumulate the $K_h K_w$ $M \times C$ matrices and requires an extra workspace of $O(MHW)$. However, the extra memory space can be further reduced. Since the offsets of intermediate matrix fall in a pre-determined range, the starting address of each matrix generated by (A,B,C,...,I) kernels can be pre-calculated. In Fig. 3, we mark the starting addresses on the rightmost matrix by their labels. Therefore, we need to reserve a buffer of size $(M + 2\delta) \times (HW)$, where $\delta = \lceil \frac{K}{2H} \rceil$. This reserved space is allocated just enough for the padding (after shifting). Hence, the final version of accumulating "kn2row" algorithm uses $O(\delta HW) = O(KW)$ extra space.

B. FPGA-Specifc Optimization

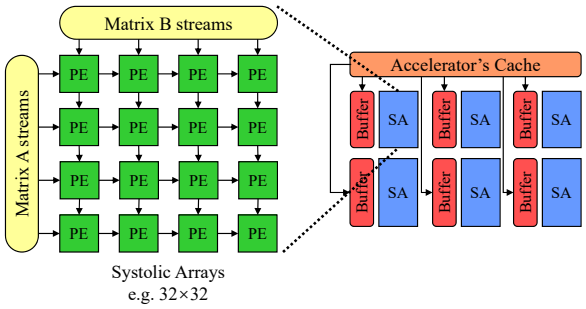

Fig. 4. The overall accelerator design.

For our FPGA-based implementation, we use a homogeneous structure, as shown in Fig. 4. We leverage systolic arrays [24], [9] to implement GEMM operations. Input matrices of systolic arrays are the representations of feature maps and weights. Systolic arrays are specialized for parallel computing with a deeply pipelined network of processing elements (PEs). For a processed tile, on-chip buffers can fully store the weights and the input/output feature maps. On the right side of Fig. 4, we demonstrate the accelerator perspective. An accelerator consists of several systolic arrays (SAs) and each SA has its own buffer to fulfill the streams of PEs. We use a post-processing unit to perform auxiliary operations such as relu and pooling. In other words, we fuse these layers into convolutions.

We use a small fixed size for all the SAs, e.g. 8×16 or 32×32, to break down the GEMMs. DSPs and BRAMs are distributed in columns throughout the physical layout. A kernel with limited size can be bounded in a small area of the physical layout. This benefits the synthesis of the design and may result in a higher frequency [25]. We replicate the small SAs to achieve the same performance of a large SA.

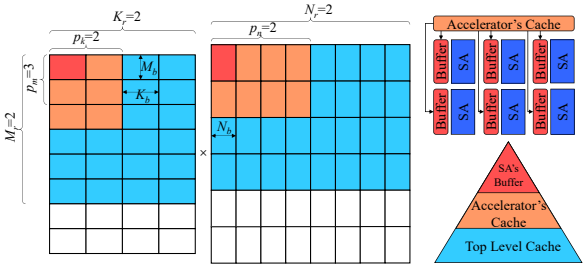

Fig. 5. Parallel SAs and three-level cache.

We propose a method to achieve both high parallelism and data reuse rate. GEMM operation (M', N', K') multiplies a (M', K') sized matrix with a (K', N') sized matrix. We define that each SA deals with (M_b, N_b, K_b) and we partition the input tensors into smaller tiles. (M_b, N_b) means that the systolic arrays will handle M_b rows of A and N_b columns of B. In Fig. 5, the smallest unit of two matrices is represented as a (M_b, N_b, K_b) tile (the red). In addition, each PE of the systolic arrays has an accumulator. This accumulator performs a vector reduction add of length K_b, which means the accumulator takes K_b elements at a time. Hence, each PE takes $(M_b + N_b) \cdot K_b$ elements from A and B as inputs each time. In order to utilize DSPs and BRAMs, we duplicate the SAs in three directions: M', N', and K'. The number of copies along three directions are (p_m, p_n, p_k), shown as the light brown tiles in Fig. 5. Therefore, there are $p = p_m p_n p_k$ parallel SAs in total.

However, simple duplication uses few BRAMs, and data reuse rate will not satisfy the off-chip bandwidth requirements, because the DSP usage reaches the limit first. We propose a three-level cache to solve this issue. We introduce new parameters (K_r, N_r, M_r) and expand the cache by $K_r \times N_r \times M_r$ times, shown as the blue regions in Fig. 5. This third-level cache consumes extra memory to prefetch data and increase the data reuse rate. The p parallel systolic arrays will iterate over the blue tiles while parallel SAs are working on the light brown tiles. We can enumerate (K_r, N_r, M_r) to find the optimal choice with maximum data reuse rate and satisfy resource limitation.

In summary, our design space has three groups of parameters: the shape of tiles (M_b, N_b, K_b), the parallelism parameters (p_m, p_n, p_k), and the reuse buffer parameters (K_r, N_r, M_r). We can explore the design space by enumerating these values to find out the optimal design with the maximum throughput.

IV. Experimental Results

We choose Xilinx Virtex UltraScale+ VU9P as our experimental platform. VU9P provides 2586 K system logic cells and

TABLE I
OVERALL COMPARISON WITH OTHER MOBILENET V1 AND INCEPTION V4 IMPLEMENTATIONS.

Implementation	Ours		[26]	[27]	[28]		
Model	MobileNet V1	Inception V4	MobileNet V1	MobileNet V1	Inception V4		
Platform	XCVU9P			Stratix V	XCVU9P		
Data type	INT8			INT16	INT8	INT16	float
Frequency (Mhz)	300		125	200	180	180	160
Logic Utilization	246K (21%)	469K (40%)	N/A	N/A	543K (46%)	673K (57%)	1016K (86%)
DSP Utilization	5149 (75%)	5254 (77%)	N/A	1664 (41%)	5130 (75%)	5130 (75%)	5130 (75%)
BRAM Utilization	1280 (59%)	1664 (77%)	N/A	N/A	562 (26%)	454 (21%)	475 (22%)
URAM Utilization	0 (0%)	0 (0%)	N/A	N/A	845 (88%)	845 (88%)	768 (80%)
Latency (ms)	0.54	5.29	0.40	0.88	6.03	6.97	28.26
Throughput (GOP/s)	3651	3448	2833	1287	1528	1319	325

TABLE II
DETAILS OF OUR DESIGNS.

Model	MobileNet V1	Inception V4
(M_b, N_b, K_b)	(8, 8, 8)	(8, 8, 16)
(p_m, p_n, p_k)	(4, 4, 8)	(4, 4, 8)
(K_r, N_r, M_r)	(8, 8, 8)	(8, 4, 4)
Memory Reduction	21.49%	44.18%

6840 DSPs. And it has about 40 megabytes on-chip memory resources including 75.9 Mb block RAMs (BRAMs) and 270 Mb UltraRAMs (URAMs). Our design is implemented and synthesized using Xilinx SDAccel 2019.1. We use quantized CNN models based on INT8. By default, we use a batch size of 1 for low-latency inference.

We choose MobileNet V1 [17] and Inception V4 [29], because they have reduced parameters for better inference performance. MobileNet V1 uses depthwise separable convolutions [16] to eliminate 3×3 filters. Inception structure breaks down 7×7 filters into 7×1 and 1×7 filters. Therefore, these models involve fewer 3×3 filters like ResNet, or 7×7 filters like AlexNet and VGG. This avoids the comparison with domain-converting methods such as Fast Fourier algorithm [14] and Winograd algorithm [15].

In Table I, we demonstrate the performance results compared with other implementations. We compare our results with three existing works [26], [27], [28]. BRAM usage is missing in two of them [26], [27]. For Inception V4, our total SRAM usage is smaller than Wei et al. [28] because they utilize URAMs, and we do not. One URAM_288K unit provides $8\times$ storage space than BRAM_36K. We achieve a latency of 0.54 ms and a throughput of 3.7 TOP/s for MobileNet V1; a latency of 5.29 ms and a throughput of 3.4 TOP/s for Inception V4.

In Table II, we show some details of our designs including the parameters (M_b, N_b, K_b), (p_m, p_n, p_k), and (K_r, N_r, M_r). The parameters for MobileNet V1 is mainly based on the $(M = 512, C = 512, H = W = 14, K_h = K_w = 1)$ layer. The difference between MobileNet V1 and Inception V4 is that Inception V4 has more GEMM operations with larger K'. Most of the layers in Inception V4 has $K_h \cdot K_w = 3$ or 7. Therefore, we increase K_b for Inception V4.

In Fig. 6, we present our normalize memory usage for MobileNet V1 by layers. The X-axis represents the shape of GEMMs of the convolutional layers. The Y-axis is the

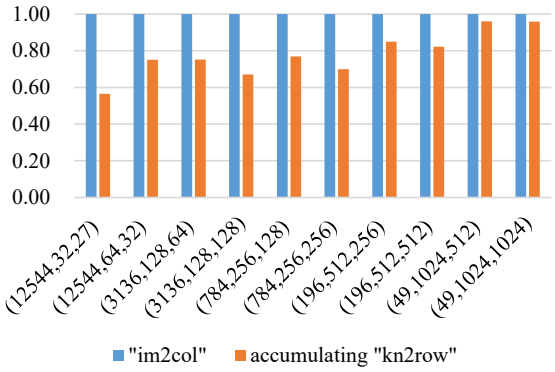

Fig. 6. Normalized memory usage of MobileNet V1.

normalized memory usage, and we use explicit GEMM's memory consumption as the baseline. We can see that the top layers benefit from the low-memory GEMM more than the bottom layers, because the top layers usually have larger feature maps. Our method uses 78.51% memory (21.49% reduction) for MobileNet V1 on average. For Inception V4, our design uses 55.82% memory (44.18% reduction) on average.

V. CONCLUSION

In this paper, we evaluate a low-memory GEMM algorithm for CNN inference on FPGAs. We design a novel accelerator based on systolic arrays and corresponding three-level cache to increase the parallelism and save the bandwidth. We evaluate our implementation on MobileNet V1 and Inception V4. Compared to other direct convolution implementations, we achieve great throughput and latency performance. We also reduce 21% to 44% memory usage compared to explicit GEMM baselines.

ACKNOWLEDGMENT

This work is partly supported by National Natural Science Foundation of China (NSFC) under Grant No. 61520106004, No. 11961141007, and No. 61631001, Beijing Academy of Artificial Intelligence (BAAI), State Grid Corporation of China under Grant No. 5500-201958484A-0-0-00 "The Research of Electric Power Artificial Intelligence Computing Components Based on Heterogeneous Instruction Set," and Beijing Natural Science Foundation under Grant No. L172004.

REFERENCES

[1] K. He, X. Zhang, S. Ren *et al.*, "Deep residual learning for image recognition," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[2] C. Szegedy, V. Vanhoucke, S. Ioffe *et al.*, "Rethinking the Inception architecture for computer vision," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *Int'l Conf. on Learning Representations (ICLR)*, 2015.

[4] C. Zhang, P. Li, G. Sun *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2015.

[5] N. Suda, V. Chandra, G. Dasika *et al.*, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2016.

[6] Y. Shen, M. Ferdman, and P. A. Milder, "Maximizing CNN accelerator efficiency through resource partitioning," in *Int'l Symp. on Computer Architecture (ISCA)*, 2016.

[7] J. Zhang and J. Li, "Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network," in *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[8] Y. Ma, Y. Cao, S. Vrudhula *et al.*, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[9] X. Wei, C. H. Yu, P. Zhang *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Design Automation Conf. (DAC)*, 2017.

[10] X. Wei, Y. Liang, X. Li *et al.*, "TGPA: Tile-grained pipeline architecture for low latency CNN inference," in *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[11] L.-N. Pouchet, C. Bastoul, A. Cohen *et al.*, "Iterative optimization in the polyhedral model: Part II, multidimensional time," *SIGPLAN Not.*, vol. 43, no. 6, pp. 90–100, 2008.

[12] L. S. Blackford, A. Petitet, R. Pozo *et al.*, "An updated set of basic linear algebra subprograms (BLAS)," *ACM Trans. on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[13] NVIDIA. cuBLAS — NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/cublas

[14] T. Highlander and A. Rodriguez, "Very efficient training of convolutional neural networks using fast fourier transform and overlap-and-add," *arXiv preprint arXiv:1601.06815*, 2015.

[15] L. Lu, Y. Liang, Q. Xiao *et al.*, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Int'l Symp. on Field-Programmable Custom Computing Machines (FCCM)*, 2017.

[16] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[17] A. G. Howard, M. Zhu, B. Chen *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[18] Intel. Intel® Math Kernel Library. [Online]. Available: https://software.intel.com/en-us/mkl

[19] N. P. Jouppi, C. Young, N. Patil *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[20] Y. Jia, E. Shelhamer, J. Donahue *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *ACM Int'l Conf. on Multimedia (MM)*, 2014.

[21] J. H. Cho, J. Kim, W. Y. Lee *et al.*, "A 1.2V 64Gb 341GB/S HBM2 stacked DRAM with spiral point-to-point TSV structure and improved bank group data control," in *Int'l Solid-State Circuits Conf. (ISSCC)*, 2018.

[22] A. Anderson, A. Vasudevan, C. Keane *et al.*, "Low-memory GEMM-based convolution algorithms for deep neural networks," *arXiv preprint arXiv:1709.03395*, 2017.

[23] S. Chetlur, C. Woolley, P. Vandermersch *et al.*, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[24] S. Kung, "VLSI array processors," *IEEE ASSP Magazine*, vol. 2, no. 3, pp. 4–22, 1985.

[25] J. Zhang, W. Zhang, G. Luo *et al.*, "Frequency improvement of systolic array-based CNNs on FPGAs," in *Int'l Symp. on Circuits and Systems (ISCAS)*, 2019.

[26] Y. Zhao, X. Gao, X. Guo *et al.*, "Automatic generation of multi-precision multi-arithmetic CNN accelerators for FPGAs," *arXiv preprint arXiv:1910.10075*, 2019.

[27] R. Zhao, H.-C. Ng, W. Luk *et al.*, "Towards efficient convolutional neural network for domain-specific applications on FPGA," *arXiv preprint arXiv:1809.03318*, 2018.

[28] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management," in *Design Automation Conf. (DAC)*, 2019.

[29] C. Szegedy, S. Ioffe, V. Vanhoucke *et al.*, "Inception-V4, Inception-ResNet and the impact of residual connections on learning," in *AAAI Conf. on Artificial Intelligence (AAAI)*, 2016.