A Survey on System-Level Design of Neural Network Accelerators

Kenshu Seto

Department of Electrical, Electronics and Communication Engineering, Tokyo City University, Tokyo, Japan
e-mail: kseto@tcu.ac.jp

Abstract— In this paper, we present a brief survey on the system-level optimizations used for convolutional neural network (CNN) inference accelerators. For the nested loop of convolutional (CONV) layers, we discuss the effects of loop optimizations such as loop interchange, tiling, unrolling and fusion on CNN accelerators. We also explain memory optimizations that are effective with the loop optimizations. In addition, we discuss streaming architectures and single computation engine architectures that are commonly used in CNN accelerators. Optimizations for CNN models are briefly explained, followed by the recent trends and future directions of the CNN accelerator design.

Index Terms— Convolutional Neural Network (CNN); Inference; Accelerator; Optimization

I. INTRODUCTION

Deep neural networks (DNNs) have been gaining more and more popularity, since they provide significant improvements in various applications such as image recognition, natural language processing, etc. Unfortunately, abundant compute and memory resources are necessary to execute DNNs. DNNs have different use cases, objectives and constraints depending on whether they are executed on edge devices (such as smart phones or sensor nodes) or data centers. Wherever DNNs are executed, low power, low energy, programmability, low cost, small footprint, low latency, high throughput and real-time execution are required. To address these requirements, many techniques related to DNN accelerator design have been already published as known from survey papers [1][2][3][4][5][6]. [3] investigates CNN-to-FPGA tools flows, [6] provides a comprehensive survey on compression techniques for DNN models. [4] surveys DNNs on edge devices. [2] shows a comprehensive survey on DNN inference accelerators. [1] includes the recent programmable CNN accelerator from Xilinx.

There are various types of DNNs depending on the target applications, and accelerators to efficiently execute these DNNs have been proposed. Convolutional neural networks (CNNs) are the first breakthrough DNNs [7] targeting mainly for image applications such as image recognition, object detection [8] and segmentation [9]. 3D CNNs are used for human action recognition [10] and the memory footprint for 3D CNNs are larger than (2D) CNNs [11][12]. For speech recognition, recurrent neural networks (RNNs) such as LSTM have been used [13]. Recently, a neural network model based on transformers called BERT [14] was proposed and made a breakthrough in natural language processing. Although the BERT model is successful, it requires large amount of memory resources. Graph Neural Networks (GNNs) can handle recognition tasks on graph structures

[15]. Spiking neural networks (SNNs) have been proposed for power-efficient implementations [16]. Generative Adversarial Networks (GANs) [17] are DNNs that generate synthetic data from limited genuine datasets.

Many accelerator design have been proposed to accelerate DNN execution exploiting traditional architectures such as systolic arrays, VLIW/SIMD/Vector (manycore/multicore) processors, CPUs, GPUs, ASICs and FPGAs. In addition, the use of novel architectures and emerging devices such as processing-in-memory (PIM) and ReRAMs [5], etc. has been actively studied for energy-efficient and/or high-performance execution of DNNs.

There are two types of accelerators for DNNs: inference accelerators and training accelerators. Training of DNNs is much more compute and memory intensive compared to inference because we must execute many iterations of forward and backward propagation. Although GPUs are currently most often used for training, the research on hardware accelerators for DNN training is an active area [18][19][20]. In both types of accelerators, the memory architectures of accelerators have significant impact on the performance, power and area of the accelerators.

In this paper, we provide a brief review on CNN inference accelerator design with traditional architectures such as ASICs, FPGAs, etc. In particular, we review the CNN accelerator design from the viewpoints of loop optimizations and memory access optimizations for nested loops of CNN layers, because these optimizations significantly affect the quality of the accelerators. We expect that these optimizations can be utilized for quickly generating efficient CNN accelerator RTL with high-level synthesis (HLS) or for compiling CNNs to programmable CNN accelerators.

The organization of this paper is as follows. In Section II., the inference by convolutional neural networks (CNNs) are briefly reviewed. Section III. shows the nested loops of the convolutional layer and discusses how the ordering of the loops affects the data reuse. In Section IV., we illustrate optimization techniques on the nested loops for designing efficient CNN accelerators. In Section V., we describe how multiple layers in CNNs can be efficiently executed. Section VI. reviews important optimizations of CNN models such as quantization and pruning. We mention some trends in recent CNN accelerators in Section VII. and discuss future directions in Section VIII..

II. CONVOLUTIONAL NEURAL NETWORKS (CNNs)

In this section, we briefly overview convolutional neural networks (CNNs) used in image recognition tasks. Fig. 1(a) shows an example CNN called VGG11. VGG11 consists of a sequence of 3 types of layers: convolution (CONV) layer, pooling layer and fully connected (FC) layer, and VGG11

Fig. 1 Example neural network (VGG11)

```

for (n=0; n<N; n++) {
  for (k=0; k<K; k++) {
    for (p=0; p<P; p++) {
      for (q=0; q<Q; q++) {
        OA[n][k][p][q] = 0;
        for (r=0; r<R; r++) {
          for (s=0; s<S; s++) {
            for (c=0; c<C; c++) {
              OA[n][k][p][q] += IA[n][c][p+r][q+s] * W[k][c][r][s];
            }
          }
        }
        OA[n][k][p][q] = Activation(OA[n][k][p][q]);
      }
    }
  }
}

```

Fig. 2 Nested loop for CONV layer

contains 11 layers that have weights, namely, 8 CONV layers and 3 FC layers. CONV layers perform image filtering and execute a large portion of computations for CNNs. Pooling layers perform image subsampling and reduce the sizes of images. FC layers compute matrix-vector multiplications, and some CNNs do not include FC layers. For more details on CNNs, please refer to the survey paper [2]. Fig. 1(b) shows the inputs and outputs of the VGG11 layers. Inputs and outputs of CNN layers are called input activations (IA) (or input feature maps (IFMs)) and output activations (OAs) (or output feature maps (OFMs)), respectively, and OAs of a layer become IAs of the following layer. These inputs and outputs have 3 dimensions of width, height and channel. For example, the first CONV layer of VGG11 accepts the IA of size 224x224x3 and generates the OA of sizes 224x224x64 as shown in Fig. 1(b).

The following expression (1) shows the computation performed in a CONV layer and Fig. 2 shows the code for the computation. The multiply-and-accumulate (MAC) operation in the innermost loop in Fig. 2 is the major computation in CONV layers.

Table I. Symbol used in the code in Fig. 2

Symbol	Meaning
N	Batch size
K	Output activation (OA) channel size
P	Output activation (OA) width
Q	Output activation (OA) height
R	Weight (filter) width
S	Weight (filter) height
C	Input activation (IA) channel size

```

for (n=0; n<N; n++) {
  for (k=0; k<K; k++) {
    for (p=0; p<P; p++) {
      for (q=0; q<Q; q++) {
        for (r=0; r<R; r++) {
          for (s=0; s<S; s++) {
            for (c=0; c<C; c++) {
              if (r==0 && s==0 && c==0)
                OA[n][k][p][q] = 0;
              OA[n][k][p][q] += IA[n][c][p+r][q+s] * W[k][c][r][s];
              if (r==R && s==S && c==C)
                OA[n][k][p][q] = Activation(OA[n][k][p][q]);
            }
          }
        }
      }
    }
  }
}

```

Fig. 3 Fully-nested loop for CONV layer

$$OA[n][k][p][q] = \sum_{i=0}^{S-1} \sum_{j=0}^{R-1} \sum_{k=0}^{C-1} IA[n][c][p+i][q+j] \times W[k][c][i][j] \quad (1)$$

The arrays OA, IA and W in Fig. 2 represent output activation, input activation and weight, respectively. The meanings of the other symbols in Fig. 2 are summarized in Table I. Batch size means the number of input images that are processed by a layer at one time. In the code shown in Fig. 2, each element of output activation $OA[n][k][p][q]$ is computed and an activation function is applied to each of the element. Usually, ReLU shown in the expression (2) is used for the activation function.

$$ReLU(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases} \quad (2)$$

In Fig. 2, we can move the code for initialization and applying the activation function into the innermost loop and change the nested loop to a fully nested loop as shown in Fig. 3. In the following discussions, we omit the code for initialization and activation for simplicity, since the omission does not affect the discussions. In addition, we also assume batch size to be 1 for simplicity. Fig. 4 shows the simplified code for a CONV layer and we use the code in the following discussions.

III. DATAFLOW OF CONVOLUTIONAL LAYERS

The code in Fig. 4 has 6 loops whose loop variables are (k, p, q, r, s, c) from the outermost loop to the innermost loop, respectively. An important property of the code in Fig. 4 is that the computation of each output activation (OA) is independent of each other. As a result, we can perform *loop*

```

for (k=0; k<K; k++) {
  for (p=0; p<P; p++) {
    for (q=0; q<Q; q++) {
      for (r=0; r<R; r++) {
        for (s=0; s<S; s++) {
          for (c=0; c<C; c++) {
            OA[k][p][q] += IA[c][p+r][q+s] * W[k][c][r][s];
          }
        }
      }
    }
  }
}

```

Fig. 4 Simplified fully-nested loop for CONV layer

```

for (k=0; k<K; k++) {
  for (p=0; p<P; p++) {
    for (q=0; q<Q; q++) {
      for (r=0; r<R; r++) {
        for (s=0; s<S; s++) {
          for (c=0; c<C; c++) {
            OA[k][p][q] += IA[c][p+r][q+s] * W[k][c][r][s];
          }
        }
      }
    }
  }
}

```

Fig. 5 Output stationary dataflow

```

for (k=0; k<K; k++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      for (c=0; c<C; c++) {
        for (p=0; p<P; p++) {
          for (q=0; q<Q; q++) {
            OA[k][p][q] += IA[c][p+r][q+s] * W[k][c][r][s];
          }
        }
      }
    }
  }
}

```

Fig. 6 Weight stationary dataflow

interchange optimization, or in other words, the order of the 6 loops can be changed in any order. Such orderings of loops are called *dataflow* in some literatures [2] [21]. If we change the ordering of loops, the array access patterns to arrays OA, IA and W are also changed, significantly impacting the performance of accelerators [22].

Fig. 5 shows the nested loop with a loop ordering called *output stationary*. Hereafter, we denote a loop ordering with a vector consisting of loop variables from the outermost loop to the innermost loop. For the output stationary dataflow, the loop ordering is (k, p, q, r, s, c). As the name suggests, the array indices k, p and q of output activation $OA[k][p][q]$ do not change while executing the inner loops (r, s, c) enclosed by the broken line in Fig. 5. Because of the temporal data reuse, the number of array accesses to OA is reduced by a factor of $R \times S \times C$ in the output stationary dataflow compared to the case without the data reuse.

On the other hand, Fig. 6 shows the nested loop with a loop ordering called *weight stationary*. For weight stationary dataflow, the loop ordering is (k, r, s, c, p, q). As

```

for (k=0; k<K; k++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      for (c=0; c<C; c++) {
        for (pt=0; pt<P/Tp; pt++) {
          for (qt=0; qt<Q/Tq; qt++) {
            } Iterations for the next tile
          }
        }
      }
    }
  }
}

```

```

(1) load inputs (IA and W) for a tile
(2) compute OA from IA and W for a tile
for (ps=0; ps<Tp; ps++) {
  for (qs=0; qs<Tq; qs++) {
    } Iterations within a tile
    p = Tp * pt + ps;
    q = Tq * qt + qs;
    OA[k][p][q] += IA[c][p+r][q+s] * W[k][c][r][s];
  }
}

```

```

(3) store outputs (OA) for a tile
}
}
}
}
}

```

Fig. 7 Weight stationary dataflow (in Fig. 6) with loop tiling

the name suggests, the array indices k, c, r and s of weight $W[k][c][r][s]$ do not change while executing the innermost loops (p, q) enclosed by the broken line in Fig. 6. Because of the temporal data reuse, the number of array accesses to W is reduced by a factor of $P \times Q$ in the weight stationary dataflow compared to the case without the data reuse.

From the cases of output stationary and weight stationary, we see that loop orderings significantly impact the number of memory accesses. As we will see later, loop orderings also affect the datapath (MAC units) implementing the loop body. There are other different loop orderings such as input stationary, row stationary [23], etc., which will lead to the different performance of accelerators. Since the parameter values K, P, Q, R, S and C largely vary per CONV layers, the optimal dataflow for improved data reuse varies per layer in general.

IV. SOURCE-LEVEL OPTIMIZATIONS FOR CNN ACCELERATOR DESIGN

In this section, we review loop and memory optimizations that are useful for designing CNN accelerators. Since these optimizations are applied to the loop nests of CONV layers, they are especially useful when designing CNN accelerators with high-level synthesis (HLS). In Section III., we have reviewed one of the loop optimizations called loop interchange. Memory optimizations are usually more effective when combined with appropriate loop optimizations. Furthermore, some loop optimizations require memory optimizations for the loop optimizations to be effective.

A. Loop tiling

One of useful loop optimizations for optimizing CNN accelerators is *loop tiling* (also called *loop blocking*) [24]. Loop tiling breaks a given loop into a set of smaller loops and optionally changes the execution ordering of the loop body by loop interchange. After tiling a loop, we have two loops: one that iterates over tiles and one that iterates within a tile. Fig. 7 shows the result of loop tiling for the code in Fig. 6, where the inner loops (p, q) in Fig. 6 are tiled with the tile sizes of T_p and T_q , respectively. Fig. 8 depicts the iterations of inner loops (p, q) before and after loop tiling for the

Fig. 8 Iterations before and after loop tiling

Fig. 9 Pipeline execution of tiled loops

code in Fig. 6. After loop tiling, the iteration space is broken into a set of smaller iterations (tile) with the tile size of (T_p, T_q) and each tile is executed one by one. For efficient use of local memories, as shown in Fig. 7 (details are omitted for simplicity), inputs IA and W for a tile are loaded from an off-chip memory ((1) in Fig. 7), tiled OA is computed from tiled IA and W ((2) in Fig. 7), and tiled OA computed from the tiled IA and W are stored in the off-chip memory ((3) in Fig. 7). As shown in Fig. 9, the loading of inputs for a tile, computation of the inner loops (ps, qs) and storing of outputs for the tile are pipelined to overlap the computation and communication using ping-pong buffers explained in Section IV.B..

By changing the tile sizes, the sizes of tiled arrays IA, W or OA referenced in the loop body can be flexibly changed in order to satisfy the constraints on on-chip memory size. Loop tiling also enables flexible partial parallelization to satisfy chip area constraints by selectively tiling loops with appropriate tile sizes and by unrolling the selected loops. For the code of the CONV layer, we can apply loop tiling at any loop, since there are no RAW (Read-After-Write) or WAR (Write-After-Read) dependence that prevents the reordering of the loop body execution. This gives us flexibility to tile the CONV loop at any loop level, so the target loops to be tiled and their tile sizes can be freely chosen to satisfy design constraints.

B. Ping-pong buffers

Fig. 10 illustrates ping-pong buffers where PE (Processing Element) array is, for example, an array of Multiply-and-accumulate (MAC) units and blue arrows represent communication between blocks. A ping-pong buffer is a pair of duplicated local memories which are read and written alternately from two modules to avoid memory access conflicts. Ping-pong buffers are often used in CNN accelerators and we can parallelize computations in the PE array and data transfers between the DRAM and local memories with ping-pong

Fig. 10 Ping-pong buffers

```

for (k=0; k<K; k++) {
  for (r=0; r<R; r++) {
    for (s=0; s<S; s++) {
      for (c=0; c<C; c++) {
        w_tmp = W[k][c][r][s];
        for (p=0; p<P; p++) {
          for (qt=0; qt<Q/4; qt++) {
            OA[k][p][qt*4+0] +=
              IA[c][p+r][qt*4+0+s] * w_tmp;
            OA[k][p][qt*4+1] +=
              IA[c][p+r][qt*4+1+s] * w_tmp;
            OA[k][p][qt*4+2] +=
              IA[c][p+r][qt*4+2+s] * w_tmp;
            OA[k][p][qt*4+3] +=
              IA[c][p+r][qt*4+3+s] * w_tmp;
          }
        }
      }
    }
  }
}

```

Fig. 11: Weight stationary dataflow with register promotion [25], loop tiling and unrolling

buffers.

C. Loop unrolling

Loop unrolling duplicates a loop body multiple times and it is a common technique to parallelize the execution of a loop body. In general, unrolling more times means more degrees of parallelism and more hardware units needed to execute the unrolled loop body. By applying loop tiling and unrolling the tiled loops completely, we can control the degree of loop unrolling by changing the tile size. Often, memory optimizations are necessary to get expected parallel performance with loop unrolling, since memory accesses conflicts usually occur after loop unrolling by the duplication of array accesses.

Fig. 11 shows the code generated from the code in Fig. 6 by tiling the innermost loop q by the tile size of 4 and unrolling the tiled innermost loop completely. In addition, the array access $W[k][c][r][s]$ is moved out of the inner loops (p, q) to reduce the array access instances, since the indices of the array access do not change during the inner loops.

Fig. 12 shows the datapath circuit that is extracted from the unrolled loop body in Fig. 11. In the loop body, four different elements of OA (output activation) are computed with four different elements of IA and one common weight value. To execute the datapath in Fig. 12 in one clock, four input data (IA) and four output data (OA) must be accessed in one clock. When we use high-level synthesis (HLS) to generate

Fig. 12 Datapath from the unrolled loop body in Fig. 11

Fig. 13 Output stationary dataflow with loop tiling and unrolling

Fig. 14 Datapath from the unrolled loop body in Fig. 11

an accelerator from the code in Fig. 11, arrays OA and IA must be partitioned into multiple local memory banks to increase the bandwidth of the local memories, since an array is mapped to a RAM which typically has at most 2 ports. Memory partitioning techniques [26][27] for resolving the issue are discussed in Section IV.D.

Fig. 13 shows the code generated from the code in Fig. 5 by tiling the innermost loop c by the tile size of 4 and unrolling the tiled innermost loop completely. From the code in Fig. 13 that performs reduction operations to OA (output activation), two types of datapath as shown in Fig. 14 (a)(b) can be generated. With loop pipelining, pipeline registers can be inserted into the datapath. As in the case of Fig. 12, memory access conflicts exist for the arrays IA and W and memory partitioning explained in Section IV.D. will be necessary to resolve the conflicts. By comparing Fig. 12 and Fig. 14, loop orderings (or dataflow) significantly change not only memory access patterns but also the datapaths of accelerators.

D. Memory partitioning

Memory partitioning (also called array partitioning or memory banking) is a technique to increase the bandwidth

Fig. 15 Example of memory partitioning

Fig. 16 Loop fusion

of a local memory by partitioning the local memory into smaller ones [26] [27] [28]. Fig. 15 shows an example of simple memory partitioning. Fig. 15 (a) is a sample loop and Fig. 15 (b) is a loop where the innermost loop j of Fig. 15 (a) is completely unrolled. Unfortunately, if the array A is mapped to a 1-port memory, it takes 4 clock cycles to execute the loop body in Fig. 15 (b). By applying the memory partitioning as shown in Fig. 15(c), the array A is partitioned into 4 memories A0, ..., A3 and the loop body can be executed in one clock cycle. The array A before and after memory partitioning is depicted in Fig. 15 (d) and (e), respectively. Memory partitioning techniques for more complex array indices are proposed in [27] [28].

E. Loop fusion

Loop fusion [29] fuses a set of loops into a single fully nested loop as shown in Fig. 16. With loop fusion, data locality is improved for shared temporal array accesses between loops (In Fig. 16, the array B is the temporal array). A memory access optimization called scalar replacement [30][31][32][33] can be used to remove the temporal arrays and generate line buffers. Typically, loop fusion is effective for fusing loops with the similar loop bounds. In CNNs, successive loops corresponding to convolutional layers can be fused and the buffers for storing the activations between the layers can be replaced with line buffers.

V. CNN ACCELERATOR ARCHITECTURES

In the previous sections, we discussed the accelerator architectures for a single CONV layer. In this section, we discuss how a set of layers in a CNN can be executed. Fig. 17 illustrates two types of accelerator architectures that execute a set of CNN layers, where compute arrays can be systolic

Fig. 17 Possible high-level architectures for CNN accelerators [34]

arrays [35], Vector/SIMD/VLIW/FPGA/CGRA-based parallel processing units [1][34][36], etc.

Fig. 17 (a) shows a streaming architecture [1][36][37][38][39][40][41][42] (also called layer-wise pipeline architecture [43], dataflow architecture [34], etc.). In the streaming architecture, each compute array corresponds to a layer and a set of compute arrays run in parallel in a pipelined manner. These compute arrays are connected by buffers (on-chip RAMs or FIFOs), and the amount of computations in compute arrays are balanced so that the buffer sizes between compute arrays are minimized. To minimize the latency of a streaming architecture by balancing the computation and communication, the roofline model [44] is utilized in [39]. To minimize the buffer sizes, the uses of line buffers were proposed in [38][40] and a proposal to reduce the line buffer is proposed in [42]. Since putting all layers on chip is usually prohibitive because of large amounts of weights and MAC units, FINN-R [34] utilizes quantization to reduce the bitwidth of weights and activations. In [38], CNN layers are partitioned, and each partition is run by the streaming architecture in turn, thus reducing the accelerator size at the cost of increased off-chip memory accesses.

Fig. 17(b) shows a single computation engine architecture [34]. In Fig. 17(b), each layer in a CNN is executed one by one in the computation array. The output of a layer is either stored in the off-chip main memory or on-chip memory. If the size of the on-chip memory is large enough, the output of the layer is stored in the on-chip memory and used by the next layer as its input to reduce the off-chip memory accesses [35]. Some of the earlier works [45][46][47] perform loop tiling to reduce the on-chip memory size and the result for each tile is written back to the off-chip memory, leading to increased off-chip accesses. Recent work reduces the number of multiplications in the compute array with the Winograd or FFT algorithm [48]. An accelerator generator of a single computation engine using high-level synthesis (HLS) is proposed in [21]. In [49][50], a technique to utilize on-chip memories for CNNs with branches such as ResNet or GoogLeNet was proposed and speedup was achieved by utilizing the limited on-chip memories for memory bounded layers. A recent single computation engine architecture [51] has a huge on-chip memory (144MB)

so that memory-hungry BERT models can be efficiently handled.

Recent work [43] proposed a combined architecture of a streaming architecture and a single computation engine architecture and also presented a design space exploration method to achieve higher performance. Recent programmable CNN accelerators [1][40][52] support both streaming architecture and a single computation engine architecture. The combination of a streaming architecture and a single computation engine architecture opens up the larger design space of CNN accelerators. The work [36] proposes an analytical model for energy and hardware cost of streaming architectures in CGRAs, and discusses that streaming architectures are more energy efficient than single computation engine architectures. A recent streaming architecture [53] has L2 SRAM to further reduce off-chip memory accesses.

VI. CNN MODEL OPTIMIZATIONS

In the previous sections, we reviewed optimization techniques for nested loops of CNN layers. Basically, these techniques do not degrade the accuracy of the CNN inference. In this section, we briefly review CNN model optimizations, or in other words, CNN algorithm optimizations, such as quantization and pruning by which significant reductions of weights are expected at the cost of accuracy loss. The required accuracies depend on applications, so we can reduce the CNN model size as long as the accuracy degradations are tolerable for target applications. One example of manually optimized compact CNN model is MobileNet [54] that greatly reduces multiply-and-accumulate (MAC) operations and memory size by the use of depth-wise and point-wise convolutions. Network Architecture Search (NAS) tries to find accurate, compact and hardware friendly CNN model automatically [1][6].

A. Quantization

Quantization is a technique to reduce bitwidth of activations and/or weights of CNN models. By quantization, memory size, compute array size and communication bandwidth demand are significantly reduced at the cost of accuracy loss. The relation between quantized bitwidth and accuracy is summarized in [4]. For some target applications, such as particle detectors, quantization significantly reduces hardware resources with enough accuracy [55]. On the other hand, for

some applications, 16-bit floating point MAC units are necessary [51] to reduce significant development cost. State-of-the-art CNN accelerators [51][53][56] have both 8bit MAC units for efficient execution of CNNs and 16bit floating-point MAC units for the accurate execution.

B. Pruning

Pruning is a technique to set redundant weight values to zero [4][6][57] while keeping the accuracy. By pruning, we expect to reduce the amount of memories and computations, since we can skip storing zero weight values and computation with zero weight values. Pruning is classified into two categories: non-structural pruning and structural pruning. In non-structural pruning, any weights can be set to zero, however, in structural pruning, weights are set to zero satisfying some forms of regularity. A training technique for structured sparsity is proposed in [58]. In principle, non-structural pruning has better compression rate compared to structural pruning, since non-structural pruning has no constraint on the positions of zero weights.

In non-structural pruning, resulting pruned weights are typically stored in compressed formats like CSC (Compressed Sparse Column) format [23], COO (Coordinate) format [1], etc. In such formats, not only data but also indices to non-zero elements must be stored in memories, which possibly cancels the effect of pruning when sparsity after pruning is not high enough. In addition, computations that use pruned weights become more complex than those that do not use pruned weights.

Fig. 18 illustrates channel pruning, which is one of the structural pruning techniques [1][4]. As shown in the figure, the whole weights corresponding to an output channel are completely removed in channel pruning so the channel of the output activation (OA) is also removed. The channel pruning can be implemented in CNN accelerators with a little modification that simply reduces the number of weights and channels. In general, structural pruning [59] are hardware friendly compared to non-structural pruning.

VII. TRENDS IN RECENT CNN ACCELERATOR

In this section, we review recent trends in CNN accelerator research which are not covered in the previous sections. In the previous sections, we mentioned the trends of CNN accelerators, for example, large on-chip memories (144MB [51], 220MB [56]), 8bit fixed point MAC units and 16bit floating point MAC units for CNN accelerators, popularity of streaming architectures [41][52][53].

A. Exploiting dynamic sparsity [60][61]

Sparsity of weights as we briefly reviewed in Section VI.B. is static sparsity in which the positions of zero weights do not change at runtime of CNN inferences. Static sparsity can be exploited to reduce the amount of computations and memories. Not only weights but also activations can be zero due to the application of ReLU or multiplications by zero weights. Differently from weights, the activations that are zero are typically known at runtime, so the sparsity related to activations is called dynamic sparsity. If a significant number of activations are zero, we can reduce both MAC operations and the memories for storing activations. Sparse

CNN accelerators utilize weight sparsity, activation sparsity or both. For example, [60] utilizes both of them by intersecting the non-zero indices of weights and activations and reducing the number of MAC operations. [60], however, increases hardware complexity in order to handle compressed forms of weights and activations and to balance workloads.

B. Exploiting inter-frame similarity for video [62]

CNNs are utilized not only for single picture applications but also video applications such as autopilot and surveillance cameras. When processing a video with CNNs, it is inefficient to process each video frame independently. In [62], the similarity of sequential video frames is exploited to reduce the per-frame energy consumption without accuracy loss.

C. Programmable accelerators [1][51]

In order to change accelerator architectures at the last minute, to update accelerator architectures according to rapidly evolving CNN models, and to reduce the costs of chip development, programmability of CNN accelerators is very important. An example recent programmable CNN accelerators is TPUv4i [51] which contains a systolic array and executes VLIW instructions. Another example [1] consists of an alternating 2D array of processing engines and local memories. The processing engines include floating-point vector instruction units, and flexibly supports both streaming architectures and single computation engine architectures.

D. Design space exploration [22][36][63]

As we saw in the previous sections, the design space of CNN accelerators is huge, since the single convolutional layer is a nested loop with 7 loops. Each of the 7 loops can be interchanged, can be tiled with any tile size, unrolled or pipelined. In addition, recent CNNs have complex topologies with many layers of significantly different loop bounds. Since it is very hard to manually explore such huge design spaces of CNN accelerators, automatic design space exploration techniques have been proposed [22][36][63]. The recent work [22] explores the design space with a mixed-integer program solver. Unfortunately, it is limited to a single computation engine architecture.

VIII. SUMMARY AND FUTURE DIRECTIONS

In this paper, we briefly reviewed the basics on convolutional neural networks (CNNs) and system-level optimizations used in CNN accelerator designs. A significant number of works have been proposed to improve efficiency and performance of CNN accelerator designs and these works are summarized in survey papers [1][3][4][5][6]. Based on the discussions in this paper, we point out the following future directions.

A. Compiler for programmable CNN accelerators

Programmable accelerators have significant advantages over fixed-functionality accelerators. Programmable accelerators [1] have SIMD/VLIW/Vector processors with distributed memories and customized interconnections. To fully exploit the programmable CNN accelerators in short design time, optimizing compilers are necessary for them. Since there are many types of CNNs, many architectural choices

Fig. 18 An example hardware-friendly structural pruning called channel pruning [4]

and many parameters to be selected, it is challenging but useful to develop optimizing compilers for CNN programmable accelerators. A compiler infrastructure MLIR [1] could be a good starting point for developing such compilers.

B. Design environment of CNN accelerators from DNN frameworks

CNN algorithms are usually implemented with frameworks such as PyTorch/TensorFlow, etc. Unfortunately, it is not yet easy to implement CNN applications written in PyTorch/TensorFlow on CNN accelerators. Tools that generate efficient accelerators from PyTorch/TensorFlow descriptions of CNN applications would be very useful to quickly implement ideas in edge devices.

C. Quick design tool of CNN accelerators

There are many architecture choices in CNN accelerator design such as single computation engine architectures, streaming architectures, combination of both architectures, loop orderings, degree of parallelism, memory hierarchy and organizations, programmability or non-programmability, static/dynamic sparsity, etc. Tools that quickly perform design space exploration, generate implementations and evaluate performance of CNN accelerators would be very useful. Unfortunately, the previous design tools can handle only limited architectures.

REFERENCES

- [1] S. Neuendorffer, A. K. Khodamoradi, K. Denolf, A. K. Jain, and S. Bayliss, "The evolution of domain-specific computing for deep learning," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 75–96, 2021.
- [2] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017.
- [3] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, Jun. 2018.
- [4] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep learning on mobile and embedded devices: State-of-the-art, challenges, and future directions," *ACM Comput. Surv.*, vol. 53, no. 4, Aug. 2020.
- [5] Y. Chen, Y. Xie, L. Song, F. Chen, and T. Tang, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, no. 3, pp. 264–274, 2020.
- [6] L. Deng, G. Li, S. Han, L. Shi, and Y. Xie, "Model compression and hardware acceleration for neural networks: A comprehensive survey," *Proceedings of the IEEE*, vol. 108, no. 4, pp. 485–532, 2020.
- [7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'12. Red Hook, NY, USA: Curran Associates Inc., 2012, pp. 1097–1105.
- [8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *arXiv:1506.02640v5*, 2015.
- [9] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, ser. LNCS, vol. 9351. Springer, 2015, pp. 234–241.
- [10] D. Tran, L. D. Bourdev, R. Fergus, L. Torresani, and M. Paluri, "C3D: generic features for video analysis," *CoRR*, vol. abs/1412.0767, 2014.
- [11] K. Hegde, R. Agrawal, Y. Yao, and C. W. Fletcher, "Morph: Flexible acceleration for 3d cnn-based video understanding," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 933–946.
- [12] M. Sun, P. Zhao, M. Gungor, M. Pedram, M. Leeser, and X. Lin, "3d cnn acceleration on fpga using hardware-aware pruning," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
- [13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.
- [14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019.
- [15] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
- [16] P.-Y. Chuang, P.-Y. Tan, C.-W. Wu, and J.-M. Lu, "A 90nm 103.14 tops/w binary-weight spiking neural network cmos asic for real-time object classification," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.
- [17] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "Ganax: A unified mimd-simd acceleration for generative adversarial networks," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 650–661.
- [18] S. K. Venkataramanaiah, H.-S. Suh, S. Yin, E. Nurvitadhi, A. Dasu, Y. Cao, and J.-s. Seo, "Fpga-based low-batch training accelerator for modern cnns featuring high bandwidth memory," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ser. ICCAD '20. New York, NY, USA: Association for Computing Machinery, 2020.
- [19] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. Baghsorkhi, and J. Torrellas, "Save: Sparsity-aware vector engine for accelerating dnn training and inference on cpus," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 796–810.

- [20] S. Kang, D. Han, J. Lee, D. Im, S. Kim, S. Kim, and H.-J. Yoo, "7.4 ganpu: A 135tflops/w multi-dnn training processor for gans with speculative dual-sparsity exploitation," in *2020 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2020, pp. 140–142.
- [21] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. Fojtik, B. Keller, A. Klinefelter, N. Pinckney, P. Raina, Y. Zhang, B. Zimmer, W. J. Dally, J. Emer, S. W. Keckler, and B. Khailany, "Magnet: A modular accelerator generator for neural networks," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.
- [22] Q. Huang, M. Kang, G. Dinh, T. Norell, A. Kalaiah, J. Demmel, J. Wawrzyniek, and Y. S. Shao, "Cosa: Scheduling by constrained optimization for spatial accelerators," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 554–566.
- [23] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 2, pp. 292–308, 2019.
- [24] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [25] J. Lu and K. D. Cooper, "Register promotion in c programs," in *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, ser. PLDI '97. New York, NY, USA: Association for Computing Machinery, 1997, pp. 308–319.
- [26] A. Sharma, D. Smith, J. Koehler, R. Barua, and M. Ferguson, "Affine loop optimization based on modulo unrolling in chapel," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: Association for Computing Machinery, 2014.
- [27] Y. Wang, P. Li, and J. Cong, "Theory and algorithm for generalized memory partitioning in high-level synthesis," ser. FPGA '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 199–208.
- [28] Y. Zhou, K. M. Al-Hawaj, and Z. Zhang, "A new approach to automatic memory banking using trace-based address mining," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 179–188.
- [29] Y. Kato and K. Seto, "Loop fusion with outer loop shifting for high-level synthesis," *IPSS Transactions on System LSI Design Methodology*, vol. 6, pp. 71–75, 2013.
- [30] K. Seto, "Scalar replacement with polyhedral model," *IPSS Transactions on System LSI Design Methodology*, vol. 11, pp. 46–56, 2018.
- [31] —, "Scalar replacement with circular buffers," *IPSS Transactions on System LSI Design Methodology*, vol. 12, pp. 13–21, 2019.
- [32] —, "Shift register initialization in scalar replacement for reducing code size," *IPSS Transactions on System LSI Design Methodology*, vol. 13, pp. 2–9, 2020.
- [33] —, "Scalar replacement in the presence of multiple write accesses for accelerator design with high-level synthesis," *Design, Automation and Test in Europe*, 2021.
- [34] M. Blott, T. B. Preußner, N. J. Fraser, G. Gambardella, K. O'Brien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018.
- [35] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellman, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–12.
- [36] C. Heidorn, F. Hannig, and J. Teich, "Design space exploration for layer-parallel execution of convolutional neural networks on cgras," in *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, ser. SCOPES '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 26–31.
- [37] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016.
- [38] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas," ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017.
- [39] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.
- [40] H. Mo, L. Liu, W. Zhu, Q. Li, H. Liu, W. Hu, Y. Wang, and S. Wei, "A 1.17 tops/w, 150fps accelerator for multi-face detection and alignment," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019.
- [41] Y. Yamada, T. Sano, Y. Tanabe, Y. Ishigaki, S. Hosoda, F. Hyuga, A. Moriya, R. Hada, A. Masuda, M. Uchiyama, T. Koizumi, T. Tamai, N. Sato, J. Tanabe, K. Kimura, R. Murakami, and T. Yoshikawa, "7.2 a 20.5tops and 217.3gops/mm2 multicore soc with dnn accelerator and image signal processor complying with iso26262 for automotive applications," in *2019 IEEE International Solid-State Circuits Conference - (ISSCC)*, 2019, pp. 132–134.
- [42] K. Seto, H. Nejatollahi, J. An, S. Kang, and N. Dutt, "Small memory footprint neural network accelerators," in *20th International Symposium on Quality Electronic Design (ISQED)*, 2019, pp. 253–258.
- [43] X. Zhang, H. Ye, J. Wang, Y. Lin, J. Xiong, W.-M. Hwu, and D. Chen, "Dnnexplorer: A framework for modeling and exploring a novel paradigm of fpga-based dnn accelerator," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.
- [44] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
- [45] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, 2013, pp. 13–19.
- [46] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Di-annao: A small-footprint high-throughput accelerator for ubiquitous

- machine-learning,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 269–284.
- [47] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 161–170.
- [48] Y. Liang, L. Lu, Q. Xiao, and S. Yan, “Evaluating fast algorithms for convolutional neural networks on fpgas,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 857–870, 2020.
- [49] X. Wei, Y. Liang, and J. Cong, “Overcoming data transfer bottlenecks in fpga-based dnn accelerators via layer conscious memory management,” in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.
- [50] 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 *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.
- [51] N. P. Jouppi, A. M. Yoon, Doe Hyun, M. Gottscho, T. B. Jablin, G. Kurian, J. Laudon, S. Li, P. Ma, X. Ma, T. Norrie, N. Patil, S. Prasad, C. Young, Z. Zhou, and D. Patterson, “Ten lessons from three generations shaped google’s tpuv4i,” in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 1–14.
- [52] Y. Jiao, L. Han, R. Jin, Y.-J. Su, C. Ho, L. Yin, Y. Li, L. Chen, Z. Chen, L. Liu, Z. He, Y. Yan, J. He, J. Mao, X. Zai, X. Wu, Y. Zhou, M. Gu, G. Zhu, R. Zhong, W. Lee, P. Chen, Y. Chen, W. Li, D. Xiao, Q. Yan, M. Zhuang, J. Chen, Y. Tian, Y. Lin, W. Wu, H. Li, and Z. Dou, “A 12nm programmable convolution-efficient neural-processing-unit chip achieving 825tops,” in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 136–140.
- [53] C.-H. Lin, C.-C. Cheng, Y.-M. Tsai, S.-J. Hung, Y.-T. Kuo, P. H. Wang, P.-K. Tsung, J.-Y. Hsu, W.-C. Lai, C.-H. Liu, S.-Y. Wang, C.-H. Kuo, C.-Y. Chang, M.-H. Lee, T.-Y. Lin, and C.-C. Chen, “A 3.4-to-13.3tops/w 3.6tops dual-core deep-learning accelerator for versatile ai applications in 7nm 5g smartphone soc,” in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 134–136.
- [54] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *CoRR*, vol. abs/1704.04861, 2017.
- [55] C. N. Coelho, A. Kuusela, S. Li, H. Zhuang, J. Ngadiuba, T. K. Aarrestad, V. Loncar, M. Pierini, A. A. Pol, and S. Summers, “Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors,” *Nature Machine Intelligence*, Jun 2021.
- [56] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, and B. Kurtz, “Think fast: A tensor streaming processor (tsp) for accelerating deep learning workloads,” in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 145–158.
- [57] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, “Exploring the regularity of sparse structure in convolutional neural networks,” 2017.
- [58] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, “Learning structured sparsity in deep neural networks,” *CoRR*, vol. abs/1608.03665, 2016. [Online]. Available: <http://arxiv.org/abs/1608.03665>
- [59] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, “An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 9, pp. 1953–1965, 2020.
- [60] J.-F. Zhang, C.-E. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, “Snap: An efficient sparse neural acceleration processor for unstructured sparse deep neural network inference,” *IEEE Journal of Solid-State Circuits*, vol. 56, no. 2, pp. 636–647, 2021.
- [61] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “Scnn: An accelerator for compressed-sparse convolutional neural networks,” in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.
- [62] Z. Yuan, Y. Yang, J. Yue, R. Liu, X. Feng, Z. Lin, X. Wu, X. Li, H. Yang, and Y. Liu, “14.2 a 65nm 24.7uj/frame 12.3mw activation-similarity-aware convolutional neural network video processor using hybrid precision, inter-frame data reuse and mixed-bit-width difference-frame data codec,” in *2020 IEEE International Solid- State Circuits Conference - (ISSCC)*, 2020, pp. 232–234.
- [63] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, “Dmazerunner: Executing perfectly nested loops on dataflow accelerators,” *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019.