Journal of
Low Power Electronics
Vol. 14, 496–507, 2018

Optimizing Memory Efficiency for Deep Convolutional Neural Network Accelerators

Xiaowei Li*, Jiajun Li, and Guihai Yan*

State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing, 100190, China

(Received: 1 April 2018; Accepted: 18 September 2018)

Convolutional Neural Network (CNN) accelerators have achieved nominal performance and energy efficiency speedup compared to traditional general purpose CPU- and GPU-based solutions. Although optimizations on computation have been intensively studied, the energy efficiency of such accelerators remains limited by off-chip memory accesses since their energy cost is magnitudes higher than other operations. Minimizing off-chip memory access volume, therefore, is the key to further improving energy efficiency. The prior state-of-the-art uses rigid data reuse patterns and is sub-optimal for some, or even all, of the individual convolutional layers. To overcome the problem, this paper proposed an adaptive layer partitioning and scheduling scheme, called SmartShuttle, to minimize off-chip memory accesses for CNN accelerators. Smartshuttle can adaptively switch among different data reuse schemes and the corresponding tiling factor settings to dynamically match different convolutional layers and fully-connected layers. Moreover, SmartShuttle thoroughly investigates the impact of data reusability and sparsity on the memory access volume. The experimental results show that SmartShuttle processes the convolutional layers at 434.8 multiply and accumulations (MACs)/DRAM access for VGG16 (batch size = 3), and 526.3 MACs/DRAM access for AlexNet (batch size = 4), which outperforms the state-of-the-art approach (Eyeriss) by 52.2% and 52.6%, respectively.

Keywords: Deep Convolutional Neural Networks, Accelerator Architecture, Memory Efficiency.

1. INTRODUCTION

Convolutional neural networks (CNNs) have achieved a state-of-the-art performance across a broad range of applications such as image classificationand speech recognition.[1–5] However, improvements in CNN accuracy are accompanied by a rapid increase in computational cost. With the networks going larger and deeper, there is a request of an unprecedented computation capacity which poses a great challenge to the computing architectures. CNNs have already grown to the point where multi-core CPUs are no longer a viable computing platform. GPUs can offer adequate performance, but it poses a challenge in the power consumption especially at data-center scale. As a result, many dedicated accelerator solutions for CNNs have been proposed in tecent years.[6–17]

CNN accelerators have achieved nominal performance and energy efficiency speedup compared to traditional general purpose CPU- and GPU-based solutions.[18–21] Although optimizations on computation have been extensively explored, the energy efficiency of such accelerators remains limited by memory accesses. It is well known that the highest energy expense in CNN accelerators is related to data movement, rather than computation.[7] State-of-the-art CNNs have millions of connections causing a large amount of data movements, including both on-chip and off-chip memory accesses, typically SRAM and DRAM accesses. The energy cost of SRAM and DRAM accesses is orders of magnitude higher than other computational operations such as ALU operations,[22] thereby dominating the system energy consumption. This phenomenon can be further illustrated by the breakdown of the energy consumed by the state-of-the-art accelerator DianNao[6] and Cambricon-X[9] in Figure 1 where DRAM accesses consumed more than 80% of the total energy. Optimizing memory access efficiency, therefore, is the key for further improvement of energy efficiency.

Maximizing the data reuse can reduce memory accesses, thereby improving memory access efficiency. The prior state-of-the-art uses rigid data reuse patterns for

*Authors to whom correspondence should be addressed.
Emails: lxw@ict.ac.cn, yan@ict.ac.cn

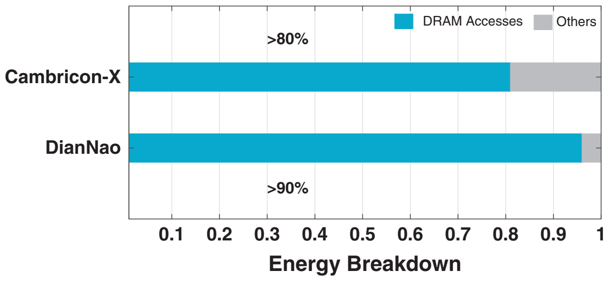


Fig. 1. Energy breakdown for state-of-the-art CNN accelerators.

convolutional layers (CVLs) iteratively, one by one. The data reuse efficiency depends on the compatibility of the reuse patterns with the configuration of CVLs. To maximize data reuse, the data reuse pattern is chosen which jointly optimized for the ensemble of the layers to achieve overall highest memory access efficiency. This one-for-all approach cannot fit the varying shapes of the convolutional layers in CNNs. Using fixed data reuse patterns are sub-optimal for some, or even all, of the individual CVLs. Furthermore, fully-connected layers (FCLs) exhibits quite different data reuse patterns from that of CVLs, thus the optimization opportunity is also quite different. Since the parameters in FCLs almost dominates the parameters in CNNs, the optimization for FCLs is quite important for maximizing memory access efficiency. In a nutshell, existing approaches are far from optimal on memory access efficiency.

In this paper, we look into the memory issues of CNN accelerators, and propose a set of methods to optimize the memory efficiency for such accelerators. The main contributions of this paper are:

- First, we present an analytical framework to quantify the off-chip memory access volume for CVLs under different layer partitioning and scheduling configurations. This framework thoroughly investigates the impact of data reusability and sparsity on off-chip memory access volume.
- Second, based on the framework, we propose an adaptive layer partitioning and scheduling scheme, named SmartShuttle, to minimize the off-chip memory accesses for CNN accelerators. SmartShuttle is orthogonal to prior on-chip accelerator architecture designs (detailed in Section 4.2) and can be easily integrated to prior accelerator designs to improve the energy efficiency.
- Third, we study the memory behavior of both CVLs and FCLs, and propose corresponding optimization techniques, respectively.
- Finally, we perform rigorous evaluation and result analysis on representative CNN models and state-of-the-art CNN accelerators, and demonstrate significant memory access reduction thereby performance. As a case study, we apply SmartShuttle on two widely used CNNs: VGG16 and AlexNet. The experimental results show that SmartShuttle processes the convolutional layers at 434.8 multiply and accumulations (MACs)/DRAM access for VGG16 (batch size = 3), and 526.3 MACs/DRAM access for AlexNet (batch size = 4), which outperforms the state-of-the-art approach (Eyeriss) by 52.2% and 52.6%, respectively.

The rest of this paper is organized as follows: Section 2 provides background and motivation. Section 3 presents the DRAM access pattern analysis for CVLs, followed by SmartShuttle in Section 4. Section 5 describes the memory access optimizations on FCLs. Section 5 presents the implementation. Section 6 shows experiments, followed by related works discussed in Section 7. Section 8 concludes this paper.

2. CNN ACCELERATORS

The computational dominance of CVLs, has sparked significant interest in the design and optimization of accelerator structures for these layers. Figure 2 illustrates a typical architecture of state-of-the-art CNN accelerators.[15] It consists of an accelerator chip and off-chip memory (usually DRAM). The accelerator chip is primarily composed of a

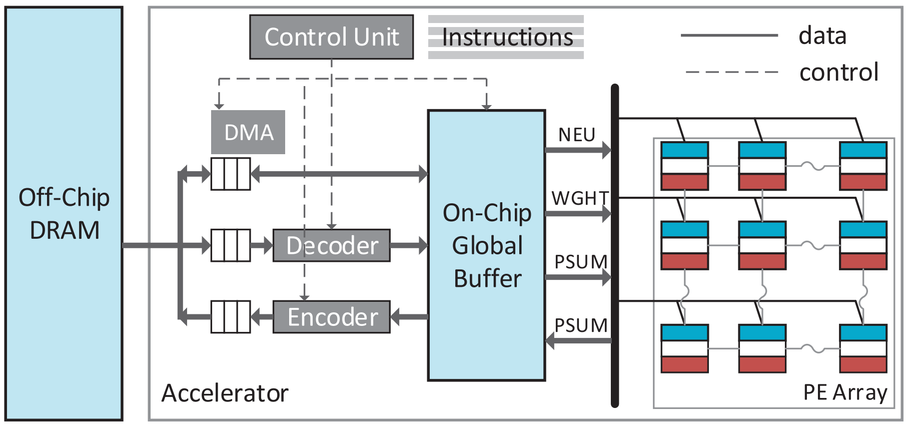


Fig. 2. A typical CNN accelerator architecture design.

```
for(d=0; d<D; d++) {          EXTERNAL DATA TRANSFER
 for(row=0; row<R; row+=Tr) {
  for(col=0; col<C; col+=Tc) {
   for(to=0; to<M; to+=Tm) {
    for(ti=0; ti<N; ti+=Tn) {
     //load ofm (psum), wght, ifm
      for(i=0; i<K; i++) {    ON-CHIP COMPUTATION
       for(j=0; j<K; j++) {
        for(trr=row; trr<min(row+Tr,R); trr++) {
         for(tcc=col; tcc<min(col+Tc,C); tcc++) {
          for(too=to; too<min(to+Tm,M); too++) {
           for(tii=ti; tii<min(ti+Tn,N); tii++) {
            ofm[d][too][trr][tcc]+=
            wght[too][tii][i][j]*
            ifm[d][tii][S*trr+i][S*tcc+j];
      } } } } } }
     //store ofm (psum)
} } } } }
```

Fig. 3. Pseudo Code of a tiled convolutional layer.

PE array and a global buffer (GLB). The PE arrays are connected with each other via a network on chip (NOC) and can support high compute parallelism to perform the massive convolution operations. GLB can be used to exploit input data reuse and hide DRAM access latency, or for the storage of intermediate data. An Encoder/Decoder unit is also used to reduce DRAM accesses. The accelerator provides four levels of storage, including DRAM, GLB, inter-PE connections and Register Files in each PE. Accessing data from a different level implies a different energy cost. In this paper we focus on the most expensive memory accesses, the one between off-chip DRAM and the on-chip GLB.

The data movement between off-chip and on-chip can be illustrated by the pseudo code in Figure 3, which has been transformed by loop tiling to fit a small portion of data into GLB. The pseudo code is partitioned into two parts, the communication part (outer loops) and the computation part (inner loops in the shaded box). A CNN dataflow defines how the inner loops are partitioned, ordered and parallelized, while the partitioning and scheduling of the outer loops determines the off-chip memory accesses. Take CVLs as an example, for each iteration of the outer loops, *ifms* and *wghts* are brought from DRAM into GLB, the convolutions are performed, and the generated *ofms* or partial sums (*psum*) are written back to DRAM. The specification for a convolutional layer is: M/N-the number of *ifm*/*ofm*, R/C-the height/width of *ofm*, K-the kernel size, S-the stride of convolution, D-the batch size. For FCLs, we denote P/Q as the number of input/output neurons.

3. DATA REUSE DILEMMA

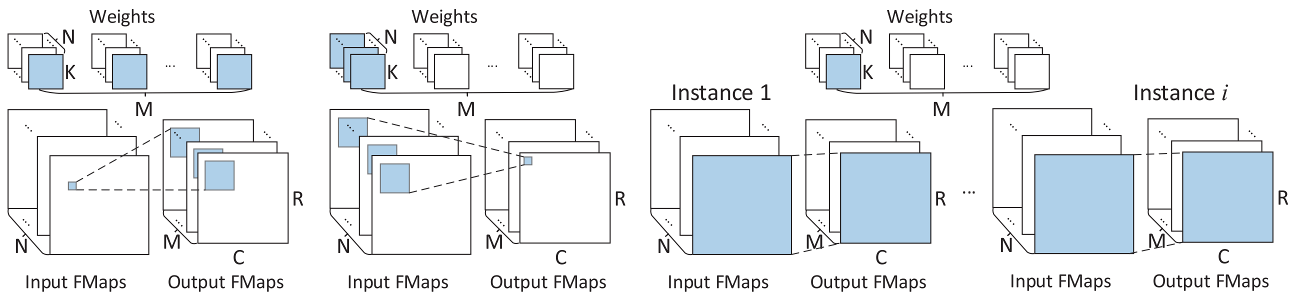
3.1. Data Reuse Analysis

Maximizing the data reuse can improve memory access efficiency. However, CNN layers, both CVLs and FCLs, entail pairwise multiplication among the input neurons and weights to generate the output neuons. Focusing on specific data reuse can only minimize the DRAM accesses of the corresponding data type, but will sacrifice the reuse of other data types. To further illustrate this dilemma, we first introduce the data reuse froms of different data types in CVLs as an example (see Fig. 4), and then present how this dilemma impacts the memory access efficiency.

The reuse forms of the different data types in CVLs are listed as follows:

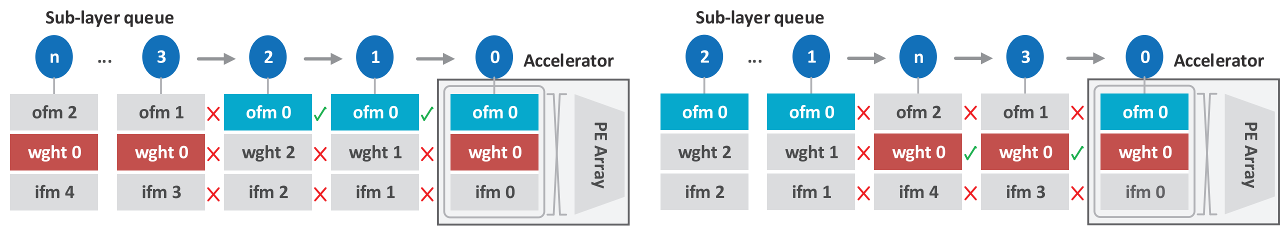
1. ifmap reuse: Each ifmap is reused across M filters and generate M ofmap channels. Each ifmap pixel is usually reused $K \times K$ times in the same filter plane.
2. ofmap reuse: Each ofmap contains N Partial Sums (psum), each generated by an ifmap using a filter. Each psum pixel is the sum of $K \times K$ multiplication results in the same filter plane.
3. *wght* reuse: Each filter weight pixel is reused $R \times C$ times in the same ifmap plane. For batch processing (Batch Size $= D$), each filter weight is reused across the batch of D ifmaps.

Figure 5 demonstrates the data reuse dilemma. Large CVLs or FCLs have millions of neurons and weights that do not fit in on-chip storage thus need to be partitioned into small sublayers. The scheduling of these sublayers implies different data reuse patterns. Figure 5(a) maximizes the reuse of *ofm0* since the sublayers containing *ofm0* (sublayer 0, 1, 2) are processed consecutively, but *wght0* will be frequently shuttled between on-chip and off-chip. By contrast, Figure 5(b) maximizes the reuse of *wght0* using a different processing order of the sub-layers,



(a) IFM reuse (b) OFM reuse (c) WGHT reuse

Fig. 4. Data reuse patterns.



(a) Maximizing OFM reuse

(b) Maximizing WGHT reuse

Fig. 5. Dilemma of maximizing reuse of which data type.

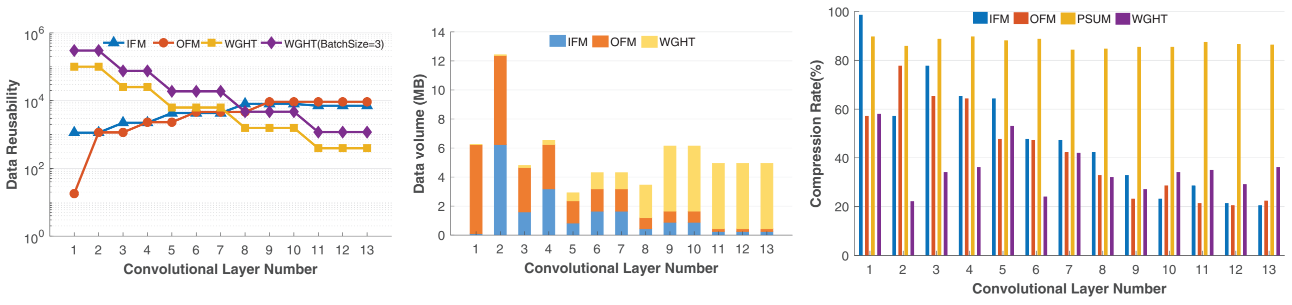
then *ofm0* cannot be reused causing intensive DRAM accesses on *ofm0*. Therefore, there exists a dilemma of maximizing the reuse of which data type, i.e., *ifm*, *ofm* or *wght*. Focusing on specific data reuse can only minimize the DRAM accesses of specific data types, but will sacrifice the reuse of other data types. One-for-all solution.

The prior state-of-the-art uses rigid data reuse patterns for convolutional layers (CVLs) iteratively, one by one. The data reuse efficiency depends on the compatibility of the reuse patterns with the configuration of CVLs. To maximize data reuse, the data reuse pattern is chosen which jointly optimized for the ensemble of the layers to achieve overall highest memory access efficiency. This one-for-all approach cannot fit the varying shapes of the convolutional layers in CNNs. Using fixed data reuse patterns are suboptimal for some, or even all, of the individual CVLs. Previous work simply maximize the reuse of one single data type for all convolutional layers, e.g., Zhang et al.[8] for *ofm*, Alwani et al.[23] for *wght*. According to our observation, for some layers, maximizing *ofm* reuse can achieve the minimal DRAM access volume, but for other layers, maximizing *wght* or *ifm* reuse may obtain a better result. Thus, sticking to one type of data reuse cannot fit the varying shapes of the convolutional layers in CNNs and is far from optimal. We observed that this diversity between CVLs can be characterized by data reusability and sparsity variance between these layers.

3.2. Data Reusability and Sparsity Variance

Data reusability is defined as the reuse times of data. The reusability of different data types exhibits large variance across the convolutional layers, as demonstrated in Figure 6(a) taking VGG16[24] as an example. In the bottom layers (the leftmost layers), *wght* reusability is much higher than *ifm/ofm*, while the opposite phenomenon is observed in the later layers (the rightmost layers). The reason lies in the fact that the data volume of *ifm/ofm* dominates the bottom layers, while the data volume of *wght* dominates the later layers, as shown in Figure 6(b). The increasing data volume of *wght* results in the decreasing of its reusability since a convolutional layer entails pairwise multiplication. The similar pattern is also observed in other CNNs such as AlexNet.[25] Notably, the batch processing will increase the reusability of *wght*. Since maximizing data reuse can reduce the off-chip memory access, the reusability variance of different data types across the layers implies that using static partitioning and scheduling schemes cannot fit all layers.

The sparsity of the three data types also varies across different layers. Since the data in CNNs are intrinsically very sparse,[16,25,26] lots of work exploit data compression to further minimize off-chip memory accesses. The compression rate can be very high due to the ReLu function[25] and weight pruning.[16,26] Figure 6(c) demonstrates the data sparsity variance across the layers in VGG16. As the layer progresses, there exists an uptrend for the compression rate of feature maps, and no obvious trends for weights. Note that *psum* is hard to be compressed, since it has not been activated by ReLu function.[25] Under the same reusability, data arrays with higher compression rate indicate lower cost when shuttled between on-chip and off-chip. The varying data sparsity again invokes the necessity



(a) Data reusability

(b) Data volume

(c) Data compression rate

Fig. 6. Data reusability and sparsity variance.

for adaptive partitioning and scheduling schemes to minimize off-chip memory accesses.

Based on the above analysis, we propose an adaptive partitioning and scheduling scheme, SmartShuttle, which has the following features: (1) configuring different tiling factors (T_m, T_n, T_r, T_c) for each layer individually; (2) switching among the scheduling schemes (IRO, ORO, WRO) to dynamically match different layers.

We propose an adaptive layer partitioning and scheduling scheme, named SmartShuttle, to minimize the off-chip memory accesses for CNN accelerators. Firstly, we present an analytical framework to quantify the off-chip memory access volume for different layer partitioning and scheduling configurations. This framework thoroughly investigates the impact of data reusability and sparsity on off-chip memory access volume. Based on the framework, SmartShuttle adaptively switches to the best partitioning and scheduling configuration which minimizes the overall off-chip memory access volume. Furthermore, SmartShuttle is orthogonal to prior on-chip accelerator architecture designs (detailed in Section 4.2) and can be easily integrated to prior accelerator designs to improve the energy efficiency.

4. SMARTSHUTTLE: CVLS

This section presents a detailed quantitative analysis on the key factors that affect DRAM access volume. Firstly, the partitioning and scheduling of convolutional layers determine the DRAM access patterns. Secondly, the data reusability and sparsity variance implies that adaptive partitioning and scheduling schemes should be applied to dynamically match the different shapes of convolutional layers.

4.1. Layer Partitioning and Scheduling

The tiling factor configuration determines the layer partitioning scheme and influences the DRAM access volume. As shown in Figure 3, the loop tiling technique partitions the large data arrays (*ifm*, *ofm*, *wght*) into smaller blocks, thus partitioning the convolutional layer into multiple sublayers, each corresponding to an invocation of external data transfer (DRAM access). The total number of the sublayers is:

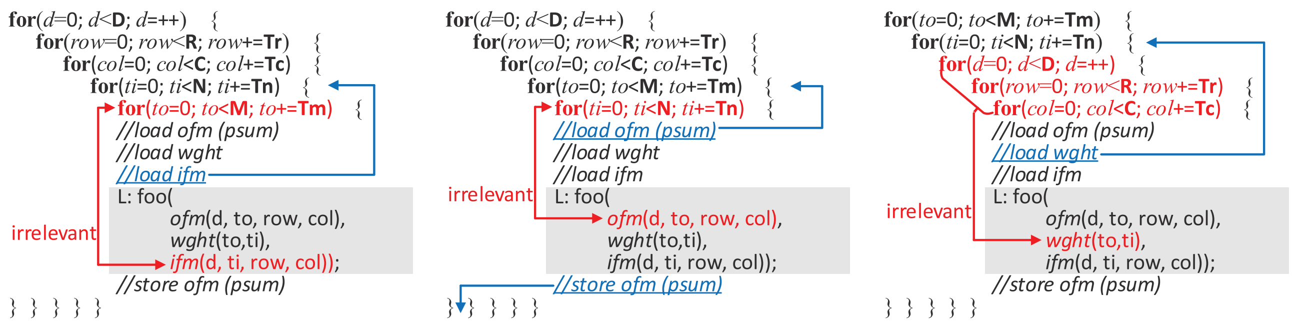
$$Q = D \cdot \left\lceil \frac{M}{T_m} \right\rceil \cdot \left\lceil \frac{N}{T_n} \right\rceil \cdot \left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil \tag{1}$$

The tiling factors determine the accessed array regions of each sublayer, thus different tiling factor configurations imply different layer partitioning schemes. We observed that different layer partitioning schemes exhibit large differences on DRAM access volume.

Meanwhile, the order of the outer loops determines the processing sequence of the sublayers and also influences the DRAM access volume. The five outer loop iterators generate $A_5^5 = 120$ possible permutations, each represents a scheduling configuration. All the permutations are feasible since they only differ in the processing orders of sublayers.

Different permutations exploit the reuse of different data types and minimize the DRAM accesses of the corresponding data type. Take the permutation in Figure 7(a) as an example, the innermost loop dimension t_o is irrelevant to array *ifm*, i.e., loop iterator t_o does not appear in the access function of array *ifm*. Hence, *ifm* can be reused in all iterations of t_o without extra DRAM accesses on *ifm*, while the accessed regions of *ofm* and *wght* change with t_o resulting in frequent shuttling of them between off-chip and on-chip. Thus, this permutation maximizes the data reuse of *ifm*, and is denoted as Ifm Reuse Oriented (IRO) scheduling scheme. Similarly, there are Ofm Reuse Oritend (ORO) and Wght Reuse Oriented (WRO) scheduling schemes as shown in Figures 7(b) and (c). It should be emphasized that d, *row*, *col* are all irrelevant to *wght*, thus *wght* reuse will be maximized only if the three loop dimensions are all in the innermost positions (the order of the three iterators does not matter).

Based on the above analysis, none permutations can maximize the reuse of all data types. Therefore, there exists a dilemma of maximizing the reuse of which data type, i.e., *ifm/ofm/wght*, corresponding to the three scheduling schemes. We found that the three scheduling schemes have their own advantages on different layers,



```
for(d=0; d<D; d=++)  {
  for(row=0; row<R; row+=Tr)  {
    for(col=0; col<C; col+=Tc)  {
      for(ti=0; ti<N; ti+=Tn)  {
        for(to=0; to<M; to+=Tm)  {
          //load ofm (psum)
          //load wght
          //load ifm
          L: foo(
              ofm(d, to, row, col),
              wght(to,ti),
              ifm(d, ti, row, col));
          //store ofm (psum)
} } } } }
```

(a) Ifm Reuse Oriented Scheme

```
for(d=0; d<D; d=++)  {
  for(row=0; row<R; row+=Tr)  {
    for(col=0; col<C; col+=Tc)  {
      for(to=0; to<M; to+=Tm)  {
        for(ti=0; ti<N; ti+=Tn)  {
          //load ofm (psum)
          //load wght
          //load ifm
          L: foo(
              ofm(d, to, row, col),
              wght(to,ti),
              ifm(d, ti, row, col));
          //store ofm (psum)
} } } } }
```

(b) Ofm Reuse Oriented Scheme

```
for(to=0; to<M; to+=Tm)  {
  for(ti=0; ti<N; ti+=Tn)  {
    for(d=0; d<D; d=++)  {
      for(row=0; row<R; row+=Tr)  {
        for(col=0; col<C; col+=Tc)  {
          //load ofm (psum)
          //load wght
          //load ifm
          L: foo(
              ofm(d, to, row, col),
              wght(to,ti),
              ifm(d, ti, row, col));
          //store ofm (psum)
} } } } }
```

(c) Wght Reuse Oriented Scheme

Fig. 7. Pseudo codes for different reuse schemes.

Table I. Tiling factor and scheduling scheme configuration for VGG16.

	Layer configure	Data statistics	IRO		ORO		WRO		SmartShuttle	
Layer no.	$\langle M, N, R, C, K, S\rangle$	$\langle cri, cro, crw\rangle$	Tiling factors	$V_i^\dagger$	Tiling factors	$V_o^\dagger$	Tiling factors	$V_w^\dagger$	RS	$V_s^\dagger$
1	$\langle 64, 1, 224, 224, 3, 1\rangle$	$\langle 0.99, 0.90, 0.58\rangle$	$\langle 8, 1, 82, 82\rangle$	16.8	$\langle 64, 1, 28, 28\rangle$	17.2	$\langle 64, 1, 28, 28\rangle$	17.1	ORO	17.2
2	$\langle 64, 64, 224, 224, 3, 1\rangle$	$\langle 0.57, 0.86, 0.22\rangle$	$\langle 8, 64, 35, 35\rangle$	59.8	$\langle 64, 9, 30, 30\rangle$	40.6	$\langle 64, 64, 22, 22\rangle$	71.0	ORO	40.6
3	$\langle 128, 64, 112, 112, 3, 1\rangle$	$\langle 0.78, 0.89, 0.34\rangle$	$\langle 10, 64, 30, 30\rangle$	29.7	$\langle 128, 14, 20, 20\rangle$	20.1	$\langle 128, 64, 13, 13\rangle$	33.5	ORO	20.1
4	$\langle 128, 128, 112, 112, 3, 1\rangle$	$\langle 0.65, 0.90, 0.36\rangle$	$\langle 12, 128, 23, 23\rangle$	36.3	$\langle 128, 13, 20, 20\rangle$	29.9	$\langle 128, 105, 8, 8\rangle$	40.0	ORO	29.9
5	$\langle 256, 128, 56, 56, 3, 1\rangle$	$\langle 0.64, 0.88, 0.53\rangle$	$\langle 10, 128, 23, 23\rangle$	17.4	$\langle 256, 8, 14, 14\rangle$	16.0	$\langle 256, 32, 8, 8\rangle$	17.1	ORO	16.0
6	$\langle 256, 256, 56, 56, 3, 1\rangle$	$\langle 0.48, 0.89, 0.24\rangle$	$\langle 12, 256, 19, 19\rangle$	22.1	$\langle 256, 16, 14, 14\rangle$	19.1	$\langle 256, 69, 8, 8\rangle$	19.3	ORO	19.1
7	$\langle 256, 256, 56, 56, 3, 1\rangle$	$\langle 0.47, 0.84, 0.42\rangle$	$\langle 9, 256, 19, 19\rangle$	26.9	$\langle 256, 12, 14, 14\rangle$	21.8	$\langle 256, 41, 8, 8\rangle$	22.2	ORO	21.8
8	$\langle 512, 256, 28, 28, 3, 1\rangle$	$\langle 0.42, 0.85, 0.32\rangle$	$\langle 11, 256, 20, 20\rangle$	15.0	$\langle 512, 13, 9, 9\rangle$	12.4	$\langle 512, 18, 8, 8\rangle$	10.3	WRO	10.3
9	$\langle 512, 512, 28, 28, 3, 1\rangle$	$\langle 0.33, 0.85, 0.27\rangle$	$\langle 8, 512, 16, 16\rangle$	21.4	$\langle 512, 9, 10, 10\rangle$	19.4	$\langle 512, 21, 8, 8\rangle$	16.5	WRO	16.5
10	$\langle 512, 512, 28, 28, 3, 1\rangle$	$\langle 0.23, 0.85, 0.34\rangle$	$\langle 9, 512, 18, 18\rangle$	24.9	$\langle 512, 12, 9, 9\rangle$	22.3	$\langle 512, 17, 8, 8\rangle$	14.9	WRO	14.9
11	$\langle 512, 512, 14, 14, 3, 1\rangle$	$\langle 0.29, 0.87, 0.35\rangle$	$\langle 14, 512, 14, 14\rangle$	20.7	$\langle 512, 11, 9, 9\rangle$	19.9	$\langle 512, 16, 8, 8\rangle$	5.6	WRO	5.6
12	$\langle 512, 512, 14, 14, 3, 1\rangle$	$\langle 0.21, 0.86, 0.29\rangle$	$\langle 22, 512, 14, 14\rangle$	17.3	$\langle 512, 8, 10, 10\rangle$	16.5	$\langle 512, 19, 8, 8\rangle$	4.3	WRO	4.3
13	$\langle 512, 512, 14, 14, 3, 1\rangle$	$\langle 0.20, 0.86, 0.36\rangle$	$\langle 19, 512, 14, 14\rangle$	21.1	$\langle 512, 11, 9, 9\rangle$	20.3	$\langle 512, 16, 8, 8\rangle$	4.9	WRO	4.9
Total	–	–	–	329.5	–	275.5	–	276.9	–	221.2

Note: †Off-chip memory access volume. V_i for IRO, V_o for ORO, V_w for 'WRO', V_s for 'SmartShuttle'.

i.e., there does not exist one that can consistently outperform the others for all layers.

Though understanding the impact of layer partitioning and scheduling on DRAM access volume, it is not easy to decide which scheduling scheme and what value of the tiling factors are optimal for a given layer. However, simply using static tiling factors and fixed scheduling scheme for all layers as many prior work did[8,15] is far from optimal due to the data reusability and sparsity variance across different layers.

4.2. An Analytical Framework

This section introduces how to decide the tiling factors and the scheduling schemes for a given convolutional layer. Firstly, we present an analytical framework to measure DRAM access volume under a given partitioning and scheduling configuration. Base on this framework, we propose an empirical rule of how to find the proper partitioning and the scheduling configuration for each layer.

Through theoretical estimation, we establish an analytical framework to calculate the DRAM access volume for a given layer as follows:

$$V_i = B^T \times A_i, \quad i = 1, 2, 3$$

$$B = \begin{bmatrix} B_i \\ B_o \\ B_w \end{bmatrix} = \begin{bmatrix} \gamma_i \cdot T_n \cdot (S \cdot T_r + K - S) \cdot (S \cdot T_c + K - S) \\ \gamma_o \cdot T_m \cdot T_r \cdot T_c \\ \gamma_w \cdot T_m \cdot T_n \cdot K^2 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} A_1^i \\ A_1^o \\ A_1^w \end{bmatrix} = \begin{bmatrix} D \cdot \frac{N}{T_n} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \left(2 \cdot \left\lceil \frac{N}{T_n} \right\rceil - 1\right) \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \frac{N}{T_n} \cdot \left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil \end{bmatrix}$$

$$A_2 = \begin{bmatrix} A_2^i \\ A_2^o \\ A_2^w \end{bmatrix} = \begin{bmatrix} D \cdot \left\lceil \frac{M}{T_m} \right\rceil \cdot \frac{N}{T_n} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \frac{N}{T_n} \cdot \left\lceil \frac{R}{T_r} \right\rceil \cdot \left\lceil \frac{C}{T_c} \right\rceil \end{bmatrix}$$

$$A_3 = \begin{bmatrix} A_3^i \\ A_3^o \\ A_3^w \end{bmatrix} = \begin{bmatrix} D \cdot \left\lceil \frac{M}{T_m} \right\rceil \cdot \frac{N}{T_n} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \left(2 \cdot \left\lceil \frac{N}{T_n} \right\rceil - 1\right) \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ \frac{M}{T_m} \cdot \frac{N}{T_n} \end{bmatrix} \tag{2}$$

where V_1, V_2, V_3 denote the total DRAM access volume under the scheduling schemes IRO, ORO, WRO, B_i, B_o, B_w denote the DRAM access volumes of *ifms/ofms/wghts* for each invocation of DRAM access, A_1, A_2, A_3 denote the invocation counts for *ifms/ofms/wghts* under the scheduling schemes IRO, ORO, WRO, γ_i, γ_o, γ_w denote the compression rate of the three data types.

It should be pointed out how A_1^o and A_3^o are calculated. Since *psum/ofm* has to be stored back to DRAM if not using *ORO* while *ifm* or *wght* can be directly replaced, the DRAM access invocation counts for *psum/ofm* under IRO/WRO (A_1^o/A_3^o) are doubled based on A_2^o. Since the initial values of *ofm* (generated from the first chunk of *ifms*) are zeros and have not to be loaded from off-chip memory, the doubled invocation counts minus 1 as $(2 \cdot \lceil N/T_n \rceil - 1)$.

We formulate the following optimization problem to minimize DRAM access volume:

$$
\begin{aligned}
\underset{\langle T_m, T_n, T_r, T_c\rangle}{\mathit{Minimize}} \quad & V = \min(V_1, V_2, V_3) \\
\text{s.t.} \quad & B_i + B_o + B_w \leq \text{GLB size} \\
& T_m \geq \min(M_{th}, M) \\
& T_n \geq \min(N_{th}, N) \\
& T_r \geq \min(R_{th}, R) \\
& T_c \geq \min(C_{th}, C)
\end{aligned} \tag{3}
$$

We set lower bounds $M_{th}/N_{th}/R_{th}/C_{th}$ for $T_m/T_n/T_r/T_c$, to make sure that further tiling and unrolling can be applied to the inner loops to fit the computing engine designs as explored by many previous work.[7,8] For example, Zhang et al.[8] tiled and unrolled loop dimensions *too* and *tii*, while Du et al.[7] tiled and unrolled *trr* and *tcc* for their computing engine designs. To ensure that the inner loops can be further tiled and unrolled, the outer loop tiling factors should not be too small. In this way, we guarantee that the adaptive partitioning scheme is orthogonal to prior computing engine designs.

4.3. An Empirical Rule

To solve the optimization problem is non-trivial since there are four individual variables. Clearly, by enumerating all the combinations of the variables we can find the optimal solution but it takes a lot of time due to the large search space. Hence, we opted for a suboptimal solution by an empirical rule shown in Table II, which is concluded from many simulation results.

SmartShuttle switches among ORO and WRO according to the conditions listed in Table II. It should be mentioned that IRO is not used in this rule. Because *ifm* usually has a comparable reusability but a higher sparsity than *psum* (except for the first layer), shuttling *ifm* between on-chip and off-chip will benefit more than shuttling *psum*. Hence, ORO is often used rather than IRO. The TF setting priority indicates which tiling factor has the priority for larger number settings under the constraint of GLB-size. For example, when ORO is used, T_m has the highest priority for larger number setting, which means T_m will be set to the maximal number while satisfying other constraints. T_r, T_c have the second highest priority. When T_m is already set as the largest number, then T_r, T_c will be set as large as possible. The priority rules of WRO should also be complied when *WRO* is used.

Table II. The empirical rule for SmartShuttle.

Conditions	RS	TF setting priority	Examples
$\gamma_o \cdot R \cdot C \geq \gamma_w \cdot D \cdot N \cdot K^2$	ORO	①T_m②T_r, T_c③T_n	VGG16-C5
$\gamma_o \cdot R \cdot C < \gamma_w \cdot D \cdot N \cdot K^2$	WRO	①T_m, T_n②T_r, T_c	VGG16-C11

To better understand the empirical rule, Table I presents the tiling factor and scheduling scheme configuration for the layers in VGG16. The platform specifications are: GLBsize = 108 KB, $M_{th} = 8$, $N_{th} = 8$, $R_{th} = 8$, $C_{th} = 8$, batch size $D = 3$.

5. SMARTSHUTTLE: FCLS

Fully-connected layer (FCLs) are usually in the top of a CNN model. In an FCL, each neuron is connected with all the neurons in the previous layer, leading to a large amount of weights. Different from CVLs, FCLs generally require a relatively small amount of computation with a huge amount of memory accesses, causing high memory transaction costs and even, significant performance degradation for CNN accelerators. Although the computation volume is much dwarfed compared to CVLs, the amount of parameters in FCLs is usually much higher than that of CVLs. Figure 8 demonstrates the layer-wise data volume comparison (including both neurons and weights) of a representative CNN model, VGGNet. The data volume in FCLs is magnitudes higher than that of CVLs, which implies FCLs occupy a large portion of memory accesses. Hence, the optimization of memory accesses in FCLs is also important.

The high memory transaction cost of FCLs can be largely reduced by network compression. Previous studies[27,28] have proven that more than 90% of the parameters in FCLs can be pruned to zero without loss of accuracy. However, network compression is only effective in inference phase and it doesn't work in training, because it might not converge when training compressed CNNs. Hence, this section focus on the memory access optimization for dense FCLs, rather than sparse FCLs. This section presents a similar analysis on FCL optimzations based on pseudo codes of FCLs.

We also use the pseudo code of FCLs to analyze the memory access patterns. Similarly, the reuse schemes in FCLs can also be classified into three types, as shown in Figure 9. The loop tiling technique partitions the large data arrays (*ifv*, *ofv*, *wght*) into smaller blocks, thus partitioning FCL into multiple sublayers, each corresponding to an

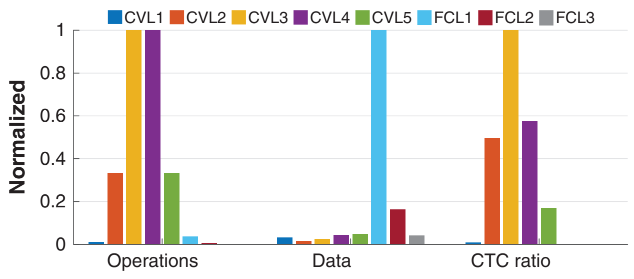
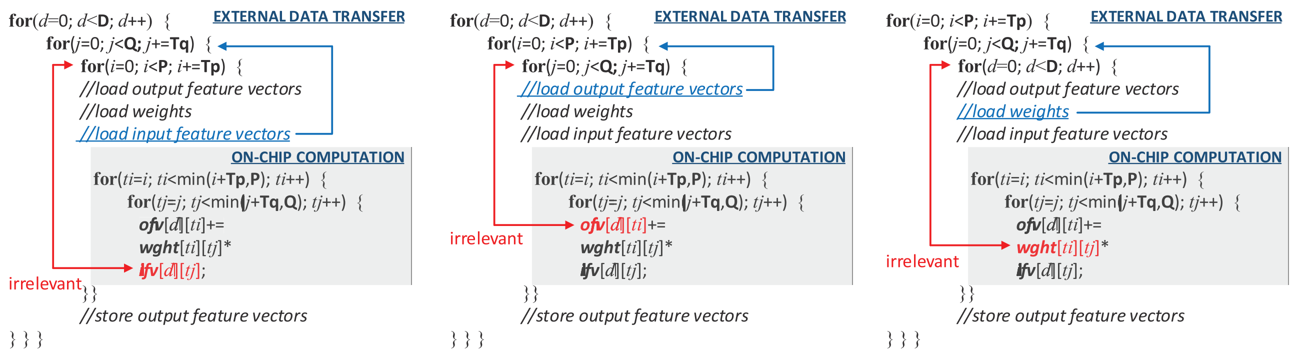


Fig. 8. The diverse resource requirements between CVLs and FCLs (normalized to the corresponding largest number).



```
for(d=0; d<D; d++) {
    for(j=0; j<Q; j+=Tq) {
        for(i=0; i<P; i+=Tp) {
        //load output feature vectors
        //load weights
        //load input feature vectors
            for(ti=i; ti<min(i+Tp,P); ti++) {
                for(tj=j; tj<min(j+Tq,Q); tj++) {
                    ofv[d][ti]+=
                    wght[ti][tj]*
                    ifv[d][tj];
        }}
        //store output feature vectors
}}}
```

(a) Ifv Reuse Oriented Scheme

```
for(d=0; d<D; d++) {
    for(i=0; i<P; i+=Tp) {
        for(j=0; j<Q; j+=Tq) {
        //load output feature vectors
        //load weights
        //load input feature vectors
            for(ti=i; ti<min(i+Tp,P); ti++) {
                for(tj=j; tj<min(j+Tq,Q); tj++) {
                    ofv[d][ti]+=
                    wght[ti][tj]*
                    ifv[d][tj];
        }}
        //store output feature vectors
}}}
```

(b) ofv Reuse Oriented Scheme

```
for(i=0; i<P; i+=Tp) {
    for(j=0; j<Q; j+=Tq) {
        for(d=0; d<D; d++) {
        //load output feature vectors
        //load weights
        //load input feature vectors
            for(ti=i; ti<min(i+Tp,P); ti++) {
                for(tj=j; tj<min(j+Tq,Q); tj++) {
                    ofv[d][ti]+=
                    wght[ti][tj]*
                    ifv[d][tj];
        }}
        //store output feature vectors
}}}
```

(c) Wght Reuse Oriented Scheme

Fig. 9. Pseudo codes for different reuse schemes of FCLs.

invocation of external data transfer (DRAM access). The total number of the sublayers is:

$$Q = D \cdot \left\lceil \frac{P}{T_p} \right\rceil \cdot \left\lceil \frac{Q}{T_q} \right\rceil \tag{4}$$

The tiling factors determine the accessed array regions of each sublayer, thus different tiling factor configurations imply different layer partitioning schemes. We observed that different layer partitioning schemes exhibit large differences on DRAM access volume.

Meanwhile, the order of the outer loops determines the processing sequence of the sublayers and also influences the DRAM access volume. The three outer loop iterators generate $A_3^3 = 6$ possible permutations, each represents a scheduling configuration. All the permutations are feasible since they only differ in the processing orders of sublayers. It is obviously that the permutations in FCLs is much less than that of CVLs. It is confirmed that using *WRO* for FCLs can achieve the lowest off-chip memory accesses.

Note that the optimization of FCLs is based on the batch processing of CNNs, which may result in increased latency. However, since the optimizations for FCLs target on CNN training, rather than real-time applciations with latency requirements, the throughput should be the preferred metric.

6. EXPERIMENTAL METHODOLOGY

Accelerator Implementation. To evaluate SmartShuttle, we implement an accelerator in Synopsys design flow

Table III. The characteristics of the baselines.

Baselines	Partitioning	Scheduling	Compression
Zhang	Static	ORO	×
Eyeriss	Static	ORO	✓
IRO-DP	Adaptive	IRO	✓
WRO-DP	Adaptive	WRO	✓
ORO-DP	Adaptive	ORO	✓
SmartShuttle-CF	Adaptive	Adaptive	×
SmartShuttle	Adaptive	Adaptive	✓

on TSMC 65 nm technology: simulating with Synopsys Verilog Compile Simulator (VCS), synthesizing with Synopsys Design Compiler (DC), and placing them with Synopsys IC Compiler (ICC). For ease of comparison with prior work, the accelerator is designed to match[15] as closely as possible. We use 16-bit fixed point numbers, 108KB GLB, and other details including the PE array and the buffer organization can refer to Ref. [15]. The lower bounds for the tiling factors are all set as 8. Furthermore, we designed a compiler to generate the macro instruction flow for loop scheduling control. The DRAM accesses are controlled by DMA requests encoded in the instructions, thus the access volume can be acquired by counting the total data volume of DMA requests.

Baselines. We compare SmartShuttle with two state-of-the-art CNN accelerators Eyeriss and Zhang.[8] Eyeriss uses ORO as the loop scheduling scheme. Data compression is also used in Eyeriss to reduce DRAM accesses. Zhang uses static loop order with static tiling factors, and without data compression. Meanwhile, the static scheduling schemes with adaptive tiling factor settings are also selected as the baselines. Table III lists the characteristics of the baselines. To make a fair comparison, our implementation has two versions, one with data compression (SmartShuttle, to compare with Eyeriss) and another without data compression (SmartShuttle-CF, compression free, to compare with Zhang).

Benchmarks. We use two representative CNN models as our benchmark: VGG16 and AlexNet. The batch size of 3 and 4 is used as the same with Eyeriss[15] for a fair comparison. VGG16 and AlexNet have 13 and 5 convolutional layers, respectively, and provide a wide range of shapes that are suitable for testing the adaptability of SmartShuttle.

7. EXPERIMENTAL RESULTS

In this section, we first evaluate SmartShuttle for the single layers in VGG16, and then evaluate the impact on all five complete networks.

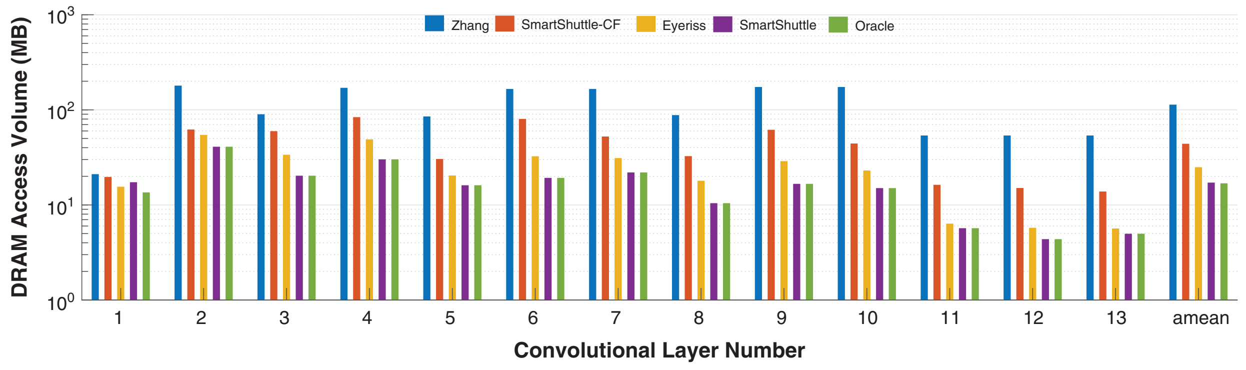


Fig. 10. Total DRAM access volume comparison on CVLs of VGG16.

7.1. Layer-Wise Off-Chip Memory Accesses

CVLs. We firstly make the comparison on off-chip memory accesses of CVLs, as shown in Figure 10. SmartShuttle provides a tremendous DRAM access advantage over the baselines. Specifically, SmartShuttle achieves a reduction of 31.2% on DRAM access volume compared to Eyeriss. Measured by "multiply and accumulations (MACs)/DRAM access" which can be viewed as DRAM access efficiency, Smarshuttle achieves up to 434.8 MACs/DRAM access, outperforming Eyeriss (285.7 MACs/DRAM access for VGG16) by 52.2%. The high DRAM access efficiency of SmartShuttle mainly stems from: (1) adaptive loop scheduling scheme. Eyeriss used a static loop order (ORO), which maximized the reuse of ofms for all layers, sacrificing *ifm* and *wght* reuse. SmartShuttle uses adaptive loop scheduling scheme and can switch to WRO to maximize the *wght* reuse in the later layers; (2) the adaptive tiling factor setting of Smarshuttle can make the best use of GLB capacity. On the contrary, Eyeriss uses static tiling factors, causing the inefficient use of GLB capacity. Meanwhile, SmartShuttle-CF achieves a 64.4% DRAM access volume reduction compared with Zhang. Note that the 'oracle' case is derived by enumerating all the feasible solutions to find the optimal one.

To gain more insights of the above benefit, we further compare the performance of IRO-DP/ORO-DP/WRO-DP to evaluate the impact of different scheduling schemes on each layer as shown in Figure 11. Firstly, ORO-DP beats IRO-DP in most layers due to the higher transfer cost of *psums* since they are at a low compression rate. Secondly, ORO-DP beats WRO-DP in the bottom layers but performed poorer in the later layers. As for specific data types, IRO-DP, ORO-DP, WRO-DP minimize the off-chip memory accesses of *ifm*, *ofm* and *wght*, respectively.

FCLs. Figure 12 compares the off-chip memory accesses of FCLs to evaluate the impact of different scheduling schemes. Obviously, WRO consistently outperforms the other scheduling schemes. The reason is that WGHT dominates the data amount in any FCLs. Hence, the off-chip memory accesses of WGHT also dominate the whole off-chip memory accesses in FCLs.

7.2. Adaptivity to Other CNN Models

To demonstrate the generality of SmartShuttle for various models, we ran similar experiments on commonly used network model, AlexNet. Figure 13 shows that SmartShuttle-CF reduced 61.3% DRAM accesses compared with Zhang. SmartShuttle processes AlexNet at 526.3 MACs/DRAM access and outperforms Eyeriss (344.8 MACs/DRAM access on AlexNet) by 52.6%.

7.3. The Impact of Global Buffer Size

The analytical framework in Section 4.2 indicates that the GLBsize is an important specification for DRAM access volume. We tested the DRAM access volume of these

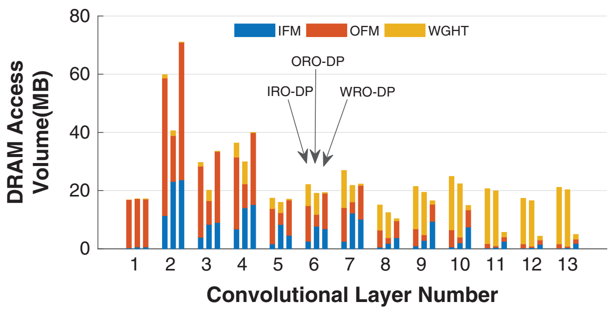


Fig. 11. DRAM access volume of *ifm/ofm/wght* under different reuse schemes (the stacks of each group from *left* to *right*: IRO-DP, ORO-DP, WRO-DP).

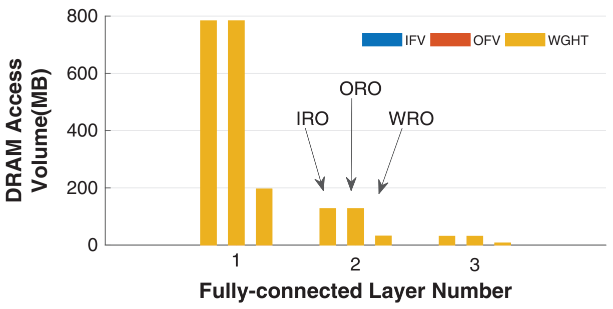


Fig. 12. DRAM access volume of *ifv/ofv/wght* under different reuse schemes (the stacks of each group from *left* to *right*: IRO, ORO, WRO).

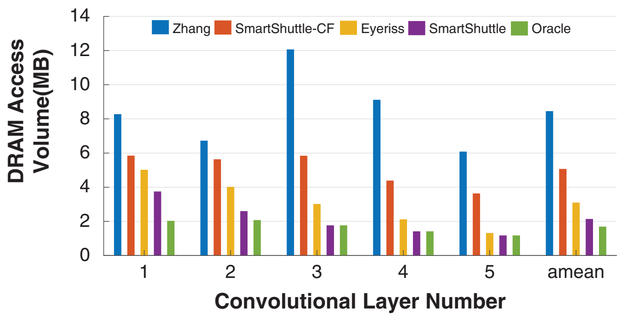


Fig. 13. Comparison with prior approaches on AlexNet.

schemes under different GLBsize configurations through simulation, as shown in Figure 14. The y axis denotes the total DRAM access volume of all convolutional layers in VGG16. All schemes achieve DRAM access reduction with larger GLB provisioning, especially when GLBsize is small (GLBsize $<$ 192 KB). With the GLBsize going larger, the benefit gains slower. For specific schemes, ORO-DP outperforms IRO-DP and WRO-DP in all GLB settings due to the high transfer cost of *psums*, while IRO-DP always performs the poorest. SmartShuttle is quite close to the oracle case, which forms a frontier between DRAM access volume and GLBsize. Note that larger GLB can decrease the DRAM access volume at the expense of larger static power of GLB. Hence, there exists a trade-off between the energy consumption on DRAM accesses and GLB static power, which is beyond the scope of this paper.

7.4. Integration of SmartShuttle to Prior Accelerators

SmartShuttle can minimize DRAM accesses for most of CNN accelerators and can be easily integrated into them, since the techniques used in SmartShuttle are orthogonal to those used in computing engine designs. As a case study, we integrate SmartShuttle into a state-of-the-art CNN accelerator, FlexFlow.[10] The results show that SmartShuttle can help reduce up to 45.1% and 47.6% DRAM access volume for FlexFlow on VGG16 and AlexNet, while the energy savings reach up to 30% and 36%, respectively.

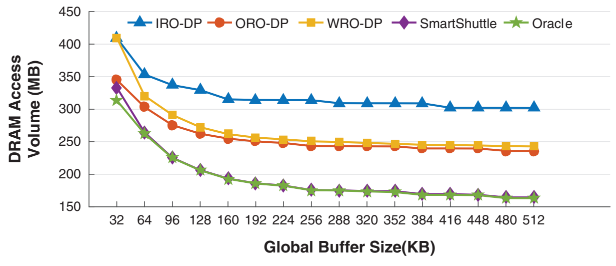


Fig. 14. GLB size impact on off-chip memory access for VGG16.

8. RELATED WORK

In recent years, accelerating deep learning applications has been one of the hottest research tops in computer architecture. This section reviews some of these recent efforts.

Many neural network accelerators focused on optimizing CVLs from computing perspective or memory perspective. DianNao[6] designed a general neural network accelerator which focuses on memory bandwidth utilization. Work[8] focuses on the balance between computing resource and memory bandwidth for CVLs. Work[13] proposed a methodology to find the optimal accelerator design for any CNN model implementation. Eyeriss[15] is a recent ASIC CNN accelerator that couples a compute grid with a NoC, enabling flexibility in scheduling CNN computation. FlexFlow[10] proposed a flexible dataflow architecture which exploits the different parallelism schemes in CVLs. These solutions only optimize for CVLs, and offer little insight on the optimization of FCLs. They inevitably incur performance degradation when accommodating CNN training applications.

Sparse CNNs have emerged as an effective solution to reduce the amount of computation and memory accesses while maintaining the high accuracy. Cambricon-X[9] and Cnvlutin[14] removes the ineffective computations from zero weights and activations, respectively. SCNN[11] leverages the sparsity in both weights and activations. EIE[16] is an energy efficient inference engine that performs inference on the compressed FCLs and accelerates the resulting sparse matrix-vector multiplication. However, the pruning techniques can only be used after training CNNs, hence these solutions cannot address the challenges in CNN training.

9. CONCLUSION

Motivated by the observation that over 80% of the energy are consumed by DRAM accesses for state-of-the-art CNN accelerators, this work proposed an adaptive layer partitioning and scheduling scheme, called SmartShuttle, to minimize the DRAM access volume for such accelerators. SmartShuttle can adaptively switch among different data reuse schemes and the corresponding tiling factor settings to dynamically match the different shapes of convolutional layers and fully-connected layers. Since the optimizations on DRAM accesses are orthogonal to those on computation, the key ideas of SmartShuttle have a broad applicability for many CNN accelerators. Compared with the state-of-the-art approaches, SmartShuttle improves the DRAM access efficiency (MACs per DRAM access) by 52.2% and 52.6% for VGG16 and AlexNet, respectively.

Acknowledgments: This work is supported by the National Natural Science Foundation of China under Grant Nos. 61872336, 61532017, 61572470, 61432017, 61521092, 61376043, and in part by Youth Innovation

Promotion Association, CAS under grant No.Y404441000. The corresponding authors are Xiaowei Li and Guihai Yan.

References

1. G. E. Dahl, T. N. Sainath, and G. E. Hinton, Improving deep neural networks for lvcsr using rectified linear units and dropout, *International Conference on Acoustics, Speech and Signal Processing* (**2013**).
2. C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, Cnp: An fpga-based processor for convolutional networks, *2009 International Conference on Field Programmable Logic and Applications* (**2009**).
3. V. Mnih and G. E. Hinton, Learning to label aerial images from noisy data, *Proceedings of the 29th ICML (ICML-12)* (**2012**).
4. P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, Learning deep structured semantic models for web search using clickthrough data, *Proceedings of the 22nd ACM International Conference on Conference on Information and Knowledge Management* (**2013**).
5. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, and M. Lanctot, Mastering the game of go with deep neural networks and tree search. *Nature* 529, 484 (**2016**).
6. T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning, architectural support for programming languages and operating systems, *Proceedings of the 19th ASPLOS* 49, 269 (**2014**).
7. Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, Shidiannao: Shifting vision processing closer to the sensor, *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)* (**2015**), pp. 92–104.
8. C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, Optimizing fpga-based accelerator design for deep convolutional neural networks, *Proceedings of the 2015 FPGA* (**2015**).
9. S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, Cambricon-x: An accelerator for sparse neural networks, *Proc. of MICRO* (**2016**).
10. W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks, *2017 IEEE 23th HPCA* (**2017**).
11. 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, *Proceedings of 44th ISCA*, ACM (**2017**).
12. J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, Going deeper with embedded fpga platform for convolutional neural network, *Proc. of FPGA* (**2016**).
13. N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-S. Seo, and Y. Cao, Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks, *Proceedings of FPGA* (**2016**).
14. J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, Cnvlutin: Ineffectual-neuron-free deep neural network computing, *Proceedings of the 43rd ISCA* (**2016**).
15. Y. Chen, T. Krishna, J. S. Emer, and V. Sze, Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 127 (**2017**).
16. S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, Eie: Efficient inference engine on compressed deep neural network, *Proceedings of the 43rd ISCA* (**2016**).
17. J. Li, G. Yan, W. Lu, S. Jiang, S. Gong, J. Wu, and X. Li, Ccr: A concise convolution rule for sparse neural network accelerators, *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March (**2018**), pp. 189–194.
18. Y. Jia, E. Shelhamer, and E. A. Donahue, Caffe: Convolutional architecture for fast feature embedding, *Proc. of MM* (**2014**).
19. V. Vanhoucke, A. Senior, and M. Z. Mao, Improving the speed of neural networks on cpus, *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011* (**2011**).
20. J. Gu, M. Zhu, Z. Zhou, F. Zhang, Z. Lin, Q. Zhang, and M. Breternitz, Implementation and evaluation of deep neural networks (dnn) on mainstream heterogeneous systems, *Proc. of Asia-Pacific Workshop on Systems* (**2014**).
21. C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, Optimizing memory efficiency for deep convolutional neural networks on gpus, *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*, November (**2016**), pp. 633–644.
22. M. Horowitz, Energy table for 45 nm process, Stanford VLSI wiki, Available: https://sites.google.com/site/seecproject.
23. M. Alwani, H. Chen, M. Ferdman, and P. Milder, Fused-layer cnn accelerators, *Proc. of MICRO* (**2016**).
24. K. Simonyan and A. Zisserman, Very deep convolutional networks for large-scale image recognition, *International Conference on Learning Representations* (**2015**).
25. A. Krizhevsky, I. Sutskever, and G. E. Hinton, Imagenet classification with deep convolutional neural networks, *Neural Information Processing Systems* 1097 (**2012**).
26. S. Han, H. Mao, and W. J. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, *International Conference on Learning Representations* (**2015**).
27. S. Han, J. Pool, J. Tran, and W. Dally, Learning both weights and connections for efficient neural network. *Advances in Neural Information Processing Systems* 1135 (**2015**).
28. N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15, 1929 (**2014**).

Xiaowei Li

Xiaowei Li (SM'04) received the B.Eng. and M.Eng. degrees in Computer Science from the Hefei University of Technology, Hefei, China, in 1985 and 1988, respectively, and the Ph.D. degree in Computer Science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1991. He was an Associate Professor with the Department of Computer Science and Technology, Peking University, Beijing, from 1991 to 2000. In 2000, he joined ICT, CAS, as a Professor, where he is currently the Deputy Director of the State Key Laboratory of Computer Architecture. He has co-authored over 280 papers in journals and international conferences, and he holds 60 patents and 30 software copyrights. His current research interests include VLSI testing, design for testability, design verification, dependable computing, and wireless sensor networks. Dr. Li has been the Vice Chair of the IEEE Asia and Pacific Regional Test Technology Technical Council (TTTC) since 2004. He is currently Vice Chair of IEEE TTTC. He was the Chair of the Technical Committee on Fault Tolerant Computing, China Computer Federation (2008–2012), and Steering Committee Chair of IEEE Asian Test Symposium (2011–2013). He was Steering Committee Chair of IEEE Workshop on RTL and High Level Testing (2007–2010). He services as Associate Editor of JCST, JOLPE, JETTA and TCAD.

Jiajun Li

Jiajun Li *received the B.Eng. degree from the Department of Automation, Tsinghua University, Beijing, China, in 2013. He is currently pursuing the Ph.D. degree with the Institute of Computing Technology, Chinese Academy of Sciences, China. His current research interests include machine learning and heterogeneous computer architecture.*

Guihai Yan

Guihai Yan *(M'10) received the B.Sc. degree in Electronics and Software Engineering (dual-degree) from Peking University, Beijing, China, in 2005, and the Ph.D. degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, China, in 2011. He is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences. He was awarded CCF (China Computer Federation) Outstanding Doctoral Dissertation, and Chinese Academy of Sciences Outstanding Dotoral Dissertation. His research interests include computer architecture, domain-specific microsystems, and energy-efficient computing.*