SmartShuttle: Optimizing Off-Chip Memory Accesses for Deep Learning Accelerators

Jiajun Li, Guihai Yan, Wenyan Lu, Shuhao Jiang, Shijun Gong, Jingya Wu, Xiaowei Li
State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences
University of Chinese Academy of Sciences
{lijiajun, yan, luwenyan, jiangshuhao, gongshijun, wujingya, lxw}@ict.ac.cn

***Abstract*— Convolutional Neural Network (CNN) accelerators are rapidly growing in popularity as a promising solution for deep learning based applications. Though 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. However, we observed that sticking to minimizing the accesses of one data type as many prior work did cannot fit the varying shapes of convolutional layers in CNNs. Hence, there exists a dilemma of minimizing the accesses of which data type. 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. 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.**

I. Introduction

Convolutional Neural Network (CNN) accelerators [1]–[7] have achieved nominal performance and energy efficiency speedup compared to traditional general purpose CPU-based solutions [8]. Though optimizations on computation have been extensively explored, the energy efficiency of such accelerators remains limited by off-chip memory accesses.

State-of-the-art CNNs have millions of connections that do not fit in on-chip storage and hence require a large amount of off-chip memory accesses, usually DRAM accesses. The energy cost of DRAM accesses is orders of magnitude higher than other operations such as ALU operations [9], 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 [1] and Cambricon-X [4] in Fig. 1 where DRAM accesses consumed more than 80% of the total energy. Minimizing DRAM access volume, therefore, is the key for further improvement of energy efficiency.

Maximizing the data reuse can reduce the DRAM accesses. However, since a convolutional layer entails pairwise multiplication among the input feature maps (*ifm*) and filter weights (*wght*) to generate the output feature maps (*ofm*), focusing on specific data reuse can only minimize the DRAM accesses

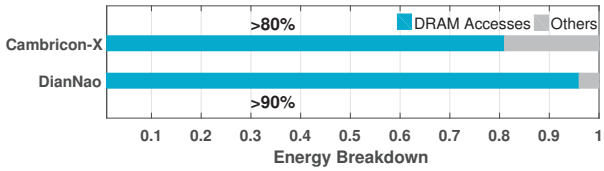


Fig. 1. Energy breakdown for state-of-the-art CNN accelerators.

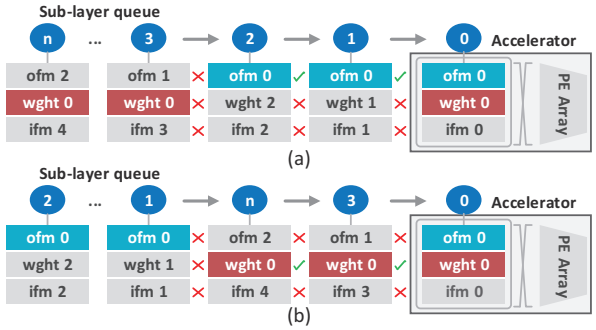


Fig. 2. Dilemma of maximizing reuse of which data type.

of the corresponding data type, but will sacrifice the reuse of other data types. For example, Fig. 2(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, Fig. 2(b) maximizes the reuse of *wght0* using a different processing order of the sub-layers, 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*.

Previous work simply maximize the reuse of one single data type for all convolutional layers, e.g. Zhang *et al.* [3] for *ofm*, Alwani *et al.* [10] 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.

In this paper, 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

978-3-9819263-0-9/DATE18/©2018 EDAA

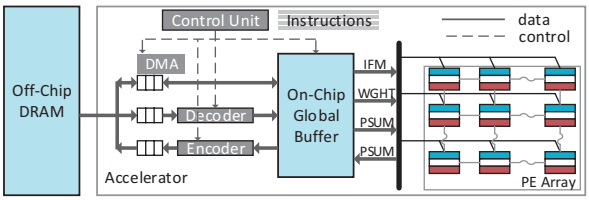


Fig. 3. A typical CNN accelerator architecture design.

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 IV) and can be easily integrated to prior accelerator designs to improve the energy efficiency. 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.

II. Background

The computational dominance of the convolutional layers, has sparked significant interest in the design and optimization of accelerator structures for these layers. Fig. 3 illustrates a typical architecture of state-of-the-art CNN accelerators [11]. It consists of an accelerator chip and off-chip memory (usually DRAM). The accelerator chip is primarily composed of a 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 Fig. 4, 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. 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-

```
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. 4. Pseudo Code of a tiled convolutional layer.

the height/width of ofm, K-the kernel size, S-the stride of convolution, D-the batch size.

III. DRAM Access Pattern Analysis

This section presents a detailed 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.

A. Layer Partitioning and Scheduling

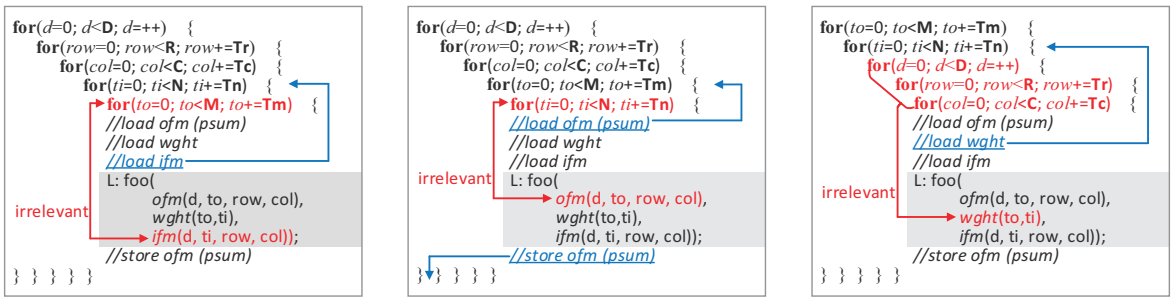
The tiling factor configuration determines the layer partitioning scheme and influences the DRAM access volume. As shown in Fig. 4, the loop tiling technique partitions the large data arrays (*ifm,ofm,wght*) into smaller blocks, thus partitioning the convolutional layer into multiple sublayers, each corresponding to an invocation of external data transfer (DRAM access). The total number of the sublayers is:

$$Q = \lceil \frac{M}{T_m} \rceil \cdot \lceil \frac{N}{T_n} \rceil \cdot \lceil \frac{R}{T_r} \rceil \cdot \lceil \frac{C}{T_c} \rceil \quad (1)$$

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 Fig. 5(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



(a) Ifm Reuse Oriented Scheme (b) Ofm Reuse Oriented Scheme (c) Wght Reuse Oriented Scheme

Fig. 5. Pseudo codes for different reuse schemes.

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 Fig. 5(b) and 5(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, 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 [3], [11] is far from optimal due to the data reusability and sparsity variance across different layers.

B. 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 Fig. 6(a) taking VGG16 [12] 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 Fig. 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 [13]. 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 [13]–[15], lots of work exploit data compression to further minimize off-chip memory accesses. Fig. 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 [13]. 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 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.

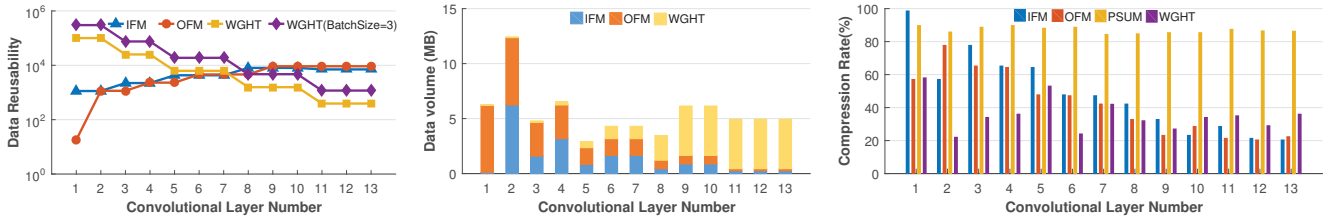
IV. SmartShuttle

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.

A. An Analytical Framework on DRAM Access Volume

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

$$
\begin{aligned}
&V_i = B^{\mathrm{T}} \times A_i,\ i = 1, 2, 3 \\
&B = \begin{bmatrix} B_i \\ B_o \\ B_w \end{bmatrix} = \begin{bmatrix} \gamma_i \cdot T_n \cdot (S \cdot T_r + K - S) \cdot (S \cdot T_c + K - S) \\ \gamma_o \cdot T_m \cdot T_r \cdot T_c \\ \gamma_w \cdot T_m \cdot T_n \cdot K^2 \end{bmatrix} \\
&A_1 = \begin{bmatrix} A_1^i \\ A_1^o \\ A_1^w \end{bmatrix} = \begin{bmatrix} D \cdot \frac{N}{T_n} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot (2 \cdot \lceil \frac{N}{T_n} \rceil - 1) \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \frac{N}{T_n} \cdot \lceil \frac{R}{T_r} \rceil \cdot \lceil \frac{C}{T_c} \rceil \end{bmatrix} \\
&A_2 = \begin{bmatrix} A_2^i \\ A_2^o \\ A_2^w \end{bmatrix} = \begin{bmatrix} D \cdot \lceil \frac{M}{T_m} \rceil \cdot \frac{N}{T_n} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \frac{N}{T_n} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot \frac{N}{T_n} \cdot \lceil \frac{R}{T_r} \rceil \cdot \lceil \frac{C}{T_c} \rceil \end{bmatrix} \\
&A_3 = \begin{bmatrix} A_3^i \\ A_3^o \\ A_3^w \end{bmatrix} = \begin{bmatrix} D \cdot \lceil \frac{M}{T_m} \rceil \cdot \frac{N}{T_n} \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ D \cdot \frac{M}{T_m} \cdot (2 \cdot \lceil \frac{N}{T_n} \rceil - 1) \cdot \frac{R}{T_r} \cdot \frac{C}{T_c} \\ \frac{M}{T_m} \cdot \frac{N}{T_n} \end{bmatrix}
\end{aligned}
\tag{2}
$$



(a) Data reusability (b) Data volume (c) Data compression rate

Fig. 6. Data reusability and sparsity variance.

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*, $\gamma_i, \gamma_o, \gamma_w$ 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 $(2 \cdot \lceil \frac{N}{T_n} \rceil - 1)$.

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

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

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 [2], [3]. For example, Zhang *et al.* [3] tiled and unrolled loop dimensions *too* and *tii*, while Du *et al.* [2] 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.

B. An Empirical Rule for SmartShuttle

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 I, which is concluded from many simulation results.

SmartShuttle switches among ORO and WRO according to the conditions listed in Table I. 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 GLBsize. 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 I
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_m ②T_r, T_c ③T_n	VGG16-C5
$\gamma_o \cdot R \cdot C < \gamma_w \cdot D \cdot N \cdot K^2$	WRO	①T_m, T_n ②T_r, T_c	VGG16-C11

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

V. EVALUATION

A. Experimental Setup

Accelerator Implementation. To evaluate SmartShuttle, we implement an accelerator in Synopsys design flow on TSMC 65nm 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 [11] 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 [11]. 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 [3]. 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

TABLE II
TILING FACTOR ANS SCHEDULING SCHEME CONFIGURATION FOR VGG16.

Layer No.	Layer Config.	Data Statistics	IRO		ORO		WRO		SmartShuttle	
	$\langle M, N, R, C, K, S\rangle$	$\langle cri, cro, crw\rangle$	Tiling Factors	V_i†	Tiling Factors	V_o†	Tiling Factors	V_w†	RS	V_s†
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

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

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	√

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 [11] 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.

B. Experimental Results

1) DRAM Access Volume on VGG16: We firstly make the comparison on total DRAM access volume, as shown in Fig. 7. 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.

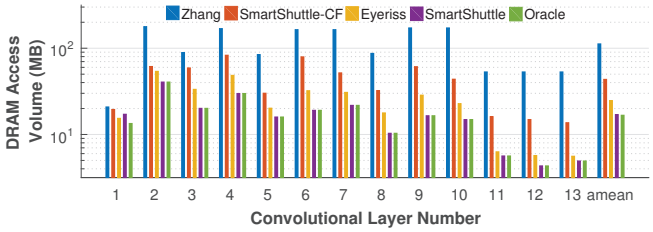


Fig. 7. Total DRAM access volume comparison on VGG16.

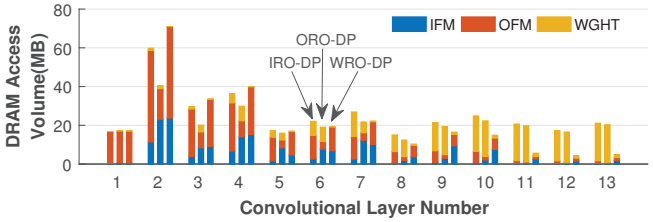


Fig. 8. DRAM access volume of *ifm/ofm/wght* under different reuse schemes (the stacks of each group from *left* to *right*:IRO-DP, ORO-DP, WRO-DP).

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 Fig. 8. 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.

2) DRAM Access Volume on AlexNet: To demonstrate the generality of SmartShuttle for various models, we ran similar experiments on another commonly used network model,

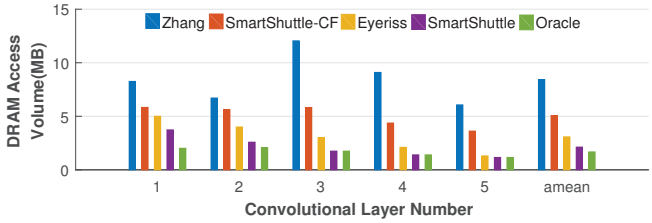


Fig. 9. Comparison with prior work on AlexNet.

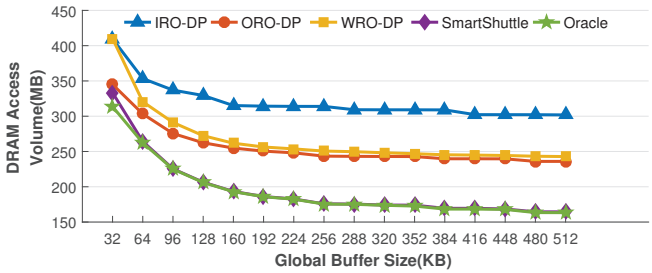


Fig. 10. GLB size impact on off-chip memory access for VGG16.

AlexNet. Fig. 9 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%.

3) The Impact of Global Buffer Size: The analytical framework in Section IV indicates that the GLBsize is an important specification for DRAM access volume. We tested the DRAM access volume of these schemes under different GLBsize configurations through simulation, as shown in Fig. 10. 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 ($GLBsize < 192KB$). 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 tradeoff between the energy consumption on DRAM accesses and GLB static power, which is beyond the scope of this paper.

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 [5]. 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.

VI. 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. 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.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant Nos. 61532017, 61572470, 61432017, 61521092, 61376043, and in part by Youth Innovation Promotion Association, CAS under grant No.Y404441000. The corresponding authors are Guihai Yan and Xiaowei Li.

References

[1] 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," in *Proceedings of the 19th ASPLOS*.
[2] 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.
[3] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 FPGA*, 2015.
[4] 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," in *Proc. of MICRO*, 2016.
[5] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE 23th HPCA*, 2017.
[6] 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 *Proceedings of 44th ISCA*, ACM, 2017.
[7] 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," in *Proc. of FPGA*, 2016.
[8] Y. Jia, E. Shelhamer, and e. a. Donahue, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. of MM*, 2014.
[9] M. Horowitz, "Energy table for 45nm process."
[10] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *Proc. of MICRO*, 2016.
[11] Y. H. 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*, 2017.
[12] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012.
[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *Proceedings of the 43rd ISCA*, 2016.
[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.