SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

February 2024, Vol. 67, Iss. 2, 122405:1–122405:20
https://doi.org/10.1007/s11432-022-3807-9

SWG: an architecture for sparse weight gradient computation

Weiwei WU[1], Fengbin TU[2], Xiangyu LI[1], Shaojun WEI[1] & Shouyi YIN[1*]

[1]*School of Integrated Circuits, Tsinghua University, Beijing* 100084*, China;*
[2]*Department of Electronic and Computer Engineering, The Hong Kong University of Science and Technology, Hong Kong* 999077*, China*

Received 31 August 2022/Revised 28 December 2022/Accepted 15 June 2023/Published online 23 January 2024

Abstract On-device training for deep neural networks (DNN) has become a trend due to various user preferences and scenarios. The DNN training process consists of three phases, feedforward (FF), backpropagation (BP), and weight gradient (WG) update. WG takes about one-third of the computation in the whole training process. Current training accelerators usually ignore the special computation property of WG and process it in a way similar to FF/BP. Besides, the extensive data sparsity existing in WG, which brings opportunities to save computation, is not well explored. Nevertheless, exploiting the optimization opportunities would meet three underutilization problems, which are caused by (1) the mismatch between WG data dimensions and hardware parallelism, (2) the full sparsity, i.e., the sparsity of feature map (Fmap), error map (Emap), and gradient, and (3) the workload imbalance resulting from irregular sparsity. In this paper, we propose a specific architecture for sparse weight gradient (SWG) computation. The architecture is designed based on hierarchical unrolling and sparsity-aware (HUSA) dataflow to exploit the optimization opportunities of the special computation property and full data sparsity. In HUSA dataflow, the data dimensions are unrolled hierarchically on the hardware architecture. A valid-data trace (VDT) mechanism is embedded in the dataflow to avoid the underutilization caused by the two-sided input sparsity. The gradient is unrolled in PE to alleviate the underutilization induced by output sparsity while maintaining the data reuse opportunities. Besides, we design an intra- and inter-column balancer (IIBLC) to dynamically tackle the workload imbalance problem resulting from the irregular sparsity. Experimental results show that with HUSA dataflow exploiting the full sparsity, SWG achieves a speedup of 12.23× over state-of-the-art gradient computation architecture, TrainWare. SWG helps to improve the energy efficiency of the state-of-the-art training accelerator LNPU from 7.56 to 10.58 TOPS/W.

Keywords CNN, training, gradient computation, sparsity, architecture

1 Introduction

In recent years, artificial intelligence (AI) applications are ubiquitous, and deep neural networks (DNNs) are widely applied in AI tasks. Directly deploying the DNNs trained from the cloud on devices suffers from performance degradation due to different user scenarios and specific tasks [1]. To guarantee the performance of DNNs, a common solution is to send the users' data to the cloud and receive the retrained DNNs from the cloud. However, this highly relies on communication capability and stability. Also, sending users' data to the cloud may cause privacy issues. As an alternative solution, on-device training has aroused increasing interest and several DNN training accelerators have been proposed.

The DNN training process consists of three phases, feedforward (FF), backpropagation (BP), and weight gradient (WG) update. The update phase can be separated into computing WG and updating the weights. WG computation occupies a significant portion of the training process. Table 1 lists the operation (multiplication-and-accumulation) counts of WG computation in one iteration of four typical DNN models [2–5] on the CIFAR-100 dataset. The proportion of WG computation operations is 28.96%–33.47%. Since it takes up a non-negligible part of the training process, optimization for WG computation is necessary to promote training acceleration. Nevertheless, the optimization opportunities for WG

* Corresponding author (email: yinsy@tsinghua.edu.cn)

© Science China Press 2024

Table 1 Operation counts of WG computation of DNNs

DNN model	Operations (M)	Operation (%)
AlexNet [2]	193.54	29.58
VGG-16 [5]	336.2	33.39
RES-18 [4]	580.79	28.96
GoogleNet [3]	33.69	33.47

computation are not well explored in existing training processors. These opportunities are from (1) the special computation property and (2) the full data sparsity of WG.

Special computation property. The WG computation properties are remarkably different from that of FF and BP. Compared to FF and BP computation, WG computing is 2D convolution (CONV) with a large kernel size, and there is no accumulation across the input channel dimension. However, previous training architectures usually ignore these differences. In training studies like [6, 7], designers consider the WG computation as a similar operation to FF/BP and mainly focus on the latter's optimization. Hence the WG process shares the same architecture with FF and BP in these studies. This simplification is over-optimistic on account of the differences between WG and FF/BP computation mentioned above. The distinctions lead to a mismatch between hardware parallelism and WG data dimensions, making the WG process inefficient in existing training accelerators. This mismatch problem incurs low utilization and makes the accelerator less energy-efficient. Therefore, a specific WG computation architecture is required to improve the training efficiency.

Full data sparsity. There is extensive data sparsity existing in WG computation. (1) Two-sided input sparsity. As the inputs of WG computation, feature maps (Fmaps), and error maps (Emaps) contain a large amount of zero data due to the employment of ReLU (rectified linear unit) in DNNs. The computation involved in these data can be eliminated considering they have no contribution to the final results. (2) Output sparsity. The sparsity of gradients comes from the corresponding weights' sparsity. Nowadays, model pruning techniques are broadly used in DNNs [8–12]. To maintain accuracy, the training of pruned models usually follows a prune-retrain processing pattern. During retraining, the pruned weights would not be updated. Therefore the calculation of related gradients can be eliminated, which results in gradient sparsity. The benefit of skipping the computation of these gradients is more remarkable when the batch size becomes large. Although a dedicated WG architecture, TrainWare [13], was proposed recently, it only handles dense WG and fails to exploit the inherent sparsity.

Leveraging the special computation property and full sparsity brings immense opportunities for saving computation. To fully exploit the optimization opportunities and process sparse WG computation efficiently, three types of underutilization remain to be addressed. (1) Underutilization caused by mismatch problem. The available unrolling dimensions of FF/BP and WG are different due to their distinctions. Thus there is a mismatch between WG data dimensions and hardware parallelism based on traditional dataflow which focuses on optimizing FF/BP. Processing WG with the FF/BP-tailored hardware parallelism incurs a low utilization. (2) Underutilization caused by full sparsity. Each gradient is generated by an inner-product operation, where the elements of the Fmap and Emap within a CONV window should be aligned to generate a correct dot product, i.e., a gradient. The two-sided input sparsity would incur misalignment between Fmap and Emap, leading to invalid computation, which makes the processing element (PE) underutilized. Additionally, the output sparsity would randomly make some PEs idle and cause underutilization. (3) Underutilization caused by workload imbalance. The workload imbalance induced by irregular sparsity of input and output data makes the execution time depend on the PE with the heaviest workload, which brings a performance loss.

In this paper, we propose SWG, a specific architecture designed for sparse weight gradient computation. SWG exploits the optimization opportunities provided by the special computation property and full sparsity of WG. The architecture is designed based on a hierarchical unrolling and sparsity-aware (HUSA) dataflow to overcome the underutilization resulting from the mismatch and full sparsity. In HUSA, the data dimensions are unrolled hierarchically on the hardware architecture. The Fmap and Emap channel dimensions are unrolled on the PE array to avoid the mismatch and leverage the channels' inter-reuse opportunities. The kernel plane is unrolled in the register file (RF) with a valid-data trace (VDT) mechanism to exploit the two-sided input sparsity and avoid invalid computation induced by the sparsity. The output plane is unrolled within the PE to alleviate underutilization caused by output sparsity while maintaining the data reuse opportunities. To tackle the workload imbalance problem incurred by the

irregular sparsity, we design an intra- and inter-column balancer (IIBLC), which dynamically remaps the Fmap to each PE in a column and adjusts the Emap allocation for each PE column.

The main contributions of this work are summarized as follows.

- An HUSA dataflow with a VDT mechanism is proposed to overcome the underutilization caused by mismatch and full sparsity.
- A specific architecture for SWG computation is designed to support the HUSA dataflow.
- An IIBLC is proposed to dynamically tackle the workload imbalance resulting from the irregular sparsity in WG, which further improves the performance.

2 Preliminary

2.1 DNN training

The DNN training process comprises FF, BP, and WG update. Figure 1 illustrates the three phases in a CONV layer. In the FF phase, the inputs are $N \times H \times L$ Fmaps, where N is the input channel count and $H \times L$ is the size of the Fmap. The weights are formed into M 3D kernels of which the size is $N \times K \times K$. Each kernel performs a 3D CONV on the input Fmaps with a sliding stride of S, generating an $R \times C$ output Fmap. The output Fmaps channel equals the kernel number M, thus there are $M \times R \times C$ output Fmaps. The outputs of the last layer in FF are compared with the ground truth to generate errors. In the BP phase, the errors are backpropagated through layers in a similar way to FF. As shown in Figure 1, to gain the $N \times H \times L$ output Emaps, the $M \times R \times C$ input Emaps are convoluted by $N \times M \times K \times K$ rotated kernels. In the WG phase, each of the old weights is updated by its corresponding gradient with a learning rate. To compute the gradients, each channel of the Fmaps is convoluted by each channel of the Emaps to generate $M \times N$ gradients.

The WG computation of one layer can be described by several nest loops as the pseudo code shown in Figure 2. Notably, there are mainly three differences between WG and Fmap/Emap computation. (1) While the computation results of different Fmap channels are partial sums (psums) and should be accumulated in FF/BP, these computation results are not accumulated in WG computation. The data of each Fmap channel are reused M times and that of each Emap channel is reused N times, assuming the gradient size $K = 1$. Hence the input (Fmap) channel N and the kernel (Emap) channel M are inter-reused. (2) The output plane in WG is small. Each gradient corresponds to one weight, so the output gradient plane equals the weight kernel size in the same layer. Weight kernels' sizes ($K \times K$) in DNNs are usually small. (3) Unlike the small kernel size in Fmap and Emap computation, the sliding window of the 2D CONV performed in every channel is large ($R \times C$). Within the large window, the elements of Fmap and Emap should be aligned to generate a correct gradient. In other words, each gradient is generated by an inner-product computation. The 2D large window CONV in WG requires the alignment of Emap and Fmap, meanwhile, it reduces the data reuse opportunities for the Emap plane.

2.2 Sparsity in DNNs

It is widely known that there is considerable inherent redundancy in DNNs and eliminating the redundant data helps to reduce the computation requirement. In recent years, many accelerators exploiting the sparsity in DNNs have been proposed. Some work leverage one-sided input sparsity, i.e., activation or weight sparsity, to reduce computation. Eyeriss [14] utilizes data gating logic to detect the zero input activations and skips the corresponding computation. Cambricon-X [15] employs an indexing module to skip zero weights computing. Cnvlutin [16] avoids invalid computation involving zero input activations by using hierarchical data-parallel computing units. These studies miss the chance to utilize the other input data's sparsity. Other studies like Cambricon-S [17] and SCNN (sparse convolution neural network accelerator) [18] exploit two-sided input sparsity, i.e., both activation and weight sparsity, to further eliminate redundant computation. Cambricon-S prunes the weights in a coarse-grain and skips the computation based on weight indexes and activation values. SCNN skips both zero-activations and zero-weights by adopting a dataflow based on a Cartesian product, where all the elements of a vector A are multiplied by all elements of another vector B. In this work, we exploit the full sparsity, i.e., both two-sided input sparsity (Fmap and Emap sparsity) and output (gradient) sparsity, to accelerate the computation of WG.

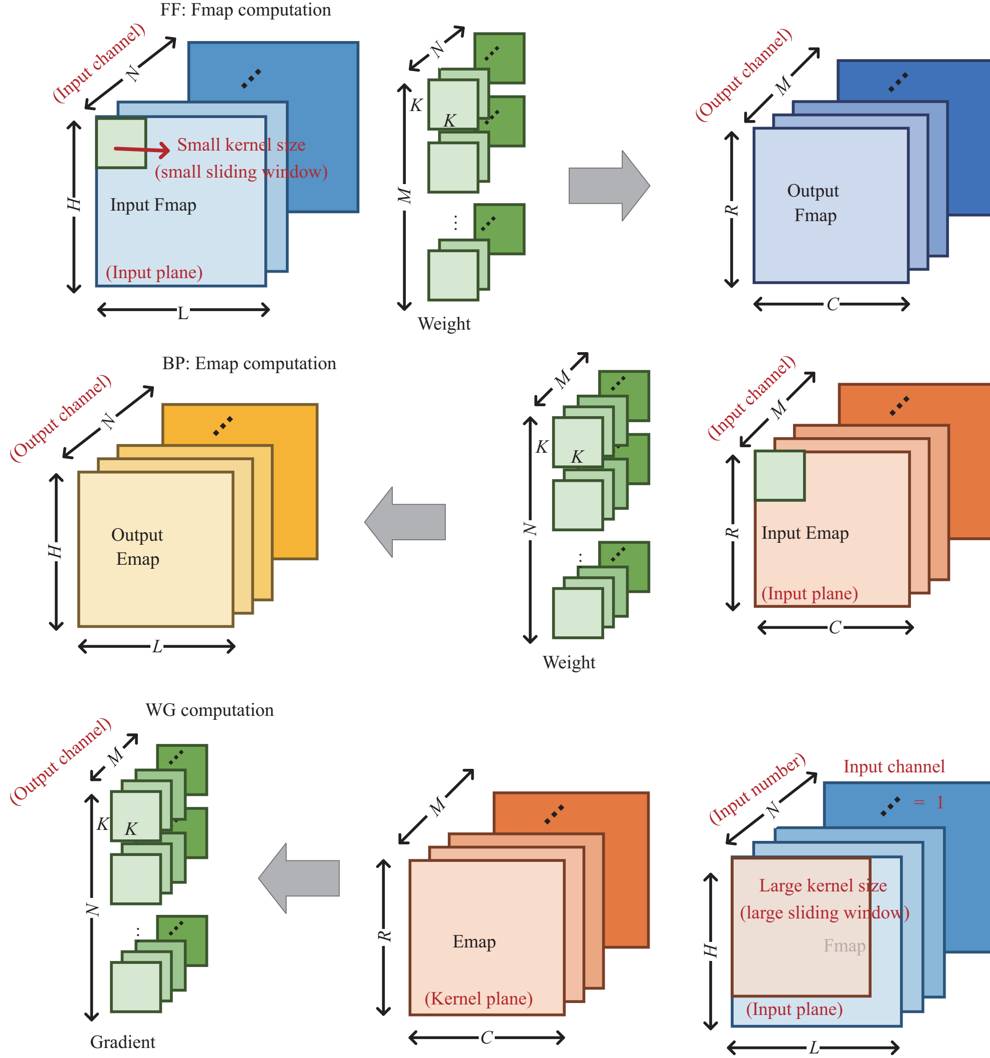


Figure 1 (Color online) Computation of Fmap, Emap, and WG in a CONV layer.

```
for(m=0; m<M; m++)                                   O-C
 for(n=0; n<M; n++)                                  I-N
  for(kr=0; kr<K; kr++)
   for(kc=0; kc<K; kc++)                             O-P
     for(i=0; i<R; i++)
       for(j=0; j<C; j++)                            K-P

         G[m][n][kr][kc]+=
         E[m][i][j]*F[n][kr+i][kc+j];
```

Figure 2 (Color online) Pseudo code of WG computation.

3 HUSA dataflow

The HUSA dataflow is proposed to overcome the underutilization caused by the mismatch and the full sparsity, where the data dimensions are hierarchically unrolled on the hardware. The Fmap and Emap channel dimensions are unrolled on the PE array, the kernel plane is unrolled within RFs, and the output plane is unrolled within each PE. In HUSA dataflow, the mismatch is addressed by Emap and Fmap channel unrolling based on the analysis of the WG computation property (Subsection 3.1). The invalid computation induced by two-sided input sparsity is eliminated by kernel plane unrolling with the

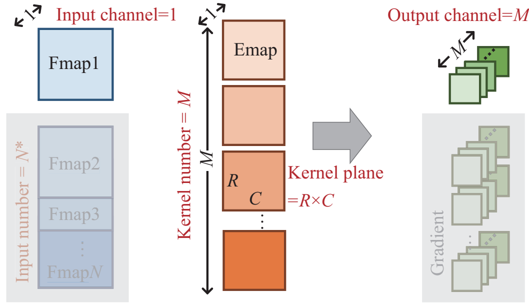


Figure 3 (Color online) WG computation is described as a combination of N large window convolutions.

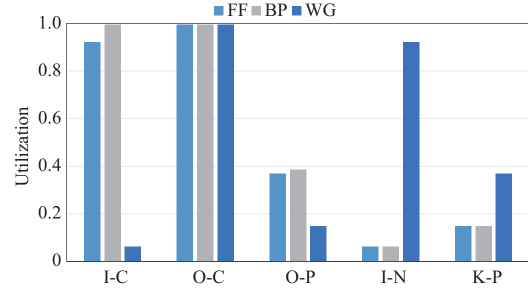


Figure 4 (Color online) Utilization of FF, BP, and WG computation with different unrolling dimensions.

embedded VDT mechanism (Subsection 3.2). The underutilization caused by output sparsity is alleviated by unrolling the output plane within the PE while maintaining the data reuse opportunities (Subsection 3.3).

3.1 Sparsity-aware channel dimensions unrolling on PE array

3.1.1 *Mismatch between hardware parallelism and WG data dimensions*

Compared to FF and BP, WG computation is 2D CONV with a large kernel size, and there is no accumulation across the input channel dimension. If these differences are ignored when designing the architecture, underutilization will occur when processing WG. To capture the computation characteristics of WG and identify the source of utilization reduction, we adjust WG computation into the 3D CONV computing pattern, and then investigate the utilization when mapping it on CONV architectures. This adjustment, as shown in Figure 3, will not change either the process or the results of WG computation. Since the channel-dimension accumulation does not exist in WG, the channels of Fmap are independent. An N-channel Fmap can be regarded as N Fmaps each with one channel. Each Fmap is reused by M channels of the Emap, hence the Emap can be regarded as M large kernels each with one channel. In this way, WG computation can be described as a combination of N 3D CONVs where each input with only one channel and the kernel size is large.

When mapping one layer on a DNN accelerator, the computation is usually decomposed into several nest loops due to the limited computing resources. The innermost is loop unrolling, which denotes hardware parallelism. In the context of this paper, hardware parallelism means the parallelism of the PE array. Unrolling a data dimension on a PE array indicates the dimension is processed in parallel. For instance, unrolling the input channel dimension is to map the data of the input channel on the PE array spatially. Any of the dimensions can be unrolled, depending on the architecture and dataflow design. In CONV architectures, which are mainly designed for 3D CONV, the unrolled dimensions are usually the ones with rich data in a 3D CONV layer, such as the input channel, output channel, and output plane.

To study the impact of each hardware parallelism on the utilization, we implement a PE array and unroll different data dimensions on it. The array has a total of 16 PEs each with one multiply-accumulator (MAC) inside. Figure 4 presents the PE utilization when processing FF, BP, and WG of dense VGG-16, RES-18, and GoogLeNet [3–5] on the array with different unrolled data dimensions. The I-C, O-C, O-P, I-N, and K-P denote the corresponding unrolled dimensions: input channel, output channel, output plane, input number, and kernel plane, respectively. These dimensions in WG computation are marked as shown in Figure 2. It is notable that the I-C mark is absent because the input channel in WG computation is 1 and there is no loop for it. Although some unrolled dimensions achieve high utilization when processing FF and BP, they are not suitable for WG due to the mismatch between hardware parallelism and WG data dimensions. The hardware parallelism of I-C, O-C, and O-P is widely used in existing training architectures. For instance, DadianNao [7] unrolls I-C and O-C dimensions, and ScaleDeep [6] adopts row stationary (RS) dataflow, which is one of the variants of O-P unrolling. Nevertheless, when processing WG with I-C unrolled, where the input channel is 1, there will be a decline in PE utilization as shown in Figure 4. Similarly, O-P unrolling leads to a waste of computing resources when dealing with WG as there is not enough data to be mapped on the PE array. The output plane of WG equals the weight kernel size which is usually small (3×3 or 1×1). As for the hardware parallelism of K-P and I-N, they

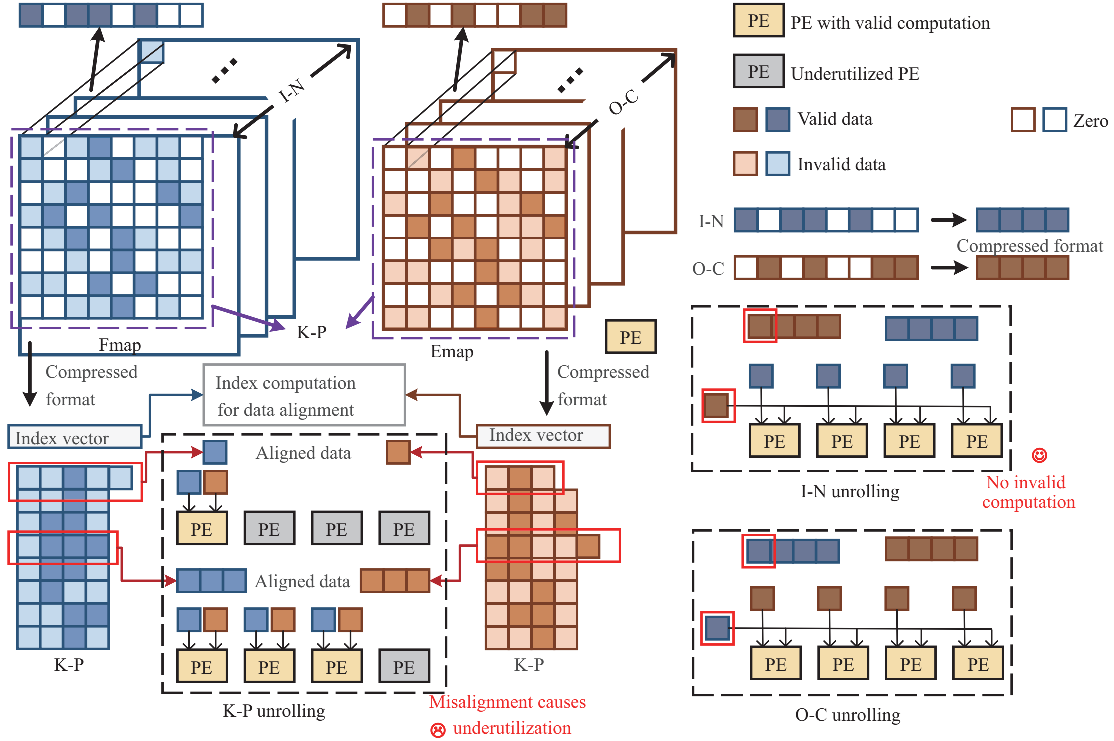


Figure 5 (Color online) Different data dimensions unrolled on PE array.

are usually not seen in CONV architectures. Typically, designers do not unroll the K-P dimension since the kernel size in 3D CONV is small. I-N dimension is generally "unrolled" temporally, which means that different inputs are processed in time order. However, for WG, the data in these dimensions are rich enough to be unrolled, which will help to achieve high utilization if the hardware is properly designed.

To summarize, the I-C and O-P hardware parallelism is common in existing accelerators, while in WG there are only a few data in these dimensions. On the other hand, the data in WG's I-N and K-P dimensions are relatively rich, while the I-N and K-P hardware parallelism is rare in training accelerators. This mismatch between hardware parallelism and WG data dimensions will cause underutilization when processing WG on existing accelerators.

3.1.2 *Hardware parallelism design for sparse weight gradient computation*

To determine the hardware parallelism, we first draw preliminary conclusions based on the utilization analysis of dense WG computation, and then further discuss the unrolling in the sparse case. Considering dense WG computation, as indicated in the utilization results of different unrolling dimensions, hardware parallelism of I-N, O-C, and K-P in WG computation contributes to relatively high utilization. Unrolling the I-N and O-C on hardware provides not only high utilization but also large opportunities for data reuse. In WG, each input Fmap, i.e., each channel of the Fmap, is reused by every channel of Emap, and per Emap channel is reused by every channel of Fmap. Notably, there remains a potential for high performance when unrolling the large kernel plane in WG computation. However, the utilization of K-P unrolling is closely related to the kernel sizes. When the kernel size is small, the PE utilization decreases. The PE number is fixed once a hardware architecture is designed while the kernel sizes of WG computation in different layers are various. Therefore, K-P unrolling cannot guarantee utilization.

Considering sparse WG computation, to discuss the performance of different unrolling dimensions, Figure 5 presents the examples of I-N, O-C, and K-P unrolling on PE array. The white grids in Fmap and Emap represent zero data and the colored grids are non-zero data. In I-N, O-C, and K-P unrolling, the data are mapped to the PE array in a compressed format to avoid zero-involved computation. A compressed format indicates that zero data are squeezed out and only the non-zero data are reserved.

The K-P unrolling of the compressed data suffers severe underutilization. The underutilization results from the misalignment of Emap and Fmap data. In WG computation, the Emap is performed as a large CONV window sliding on the Fmap. Within the CONV window, the elements of Fmap and Emap should

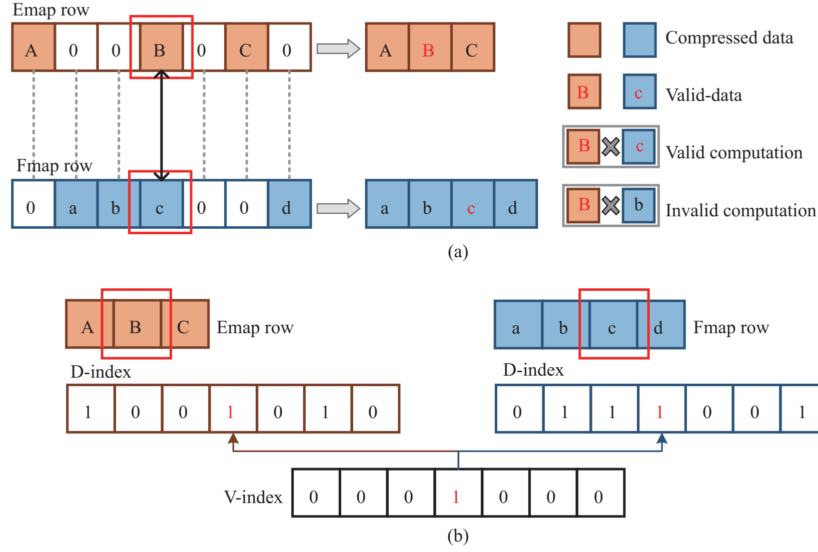


Figure 6 (Color online) (a) Difference between valid-data and compressed data; (b) examples of Fmap and Emap using the compressed data index (D-index).

be aligned to perform an inner-product computation to generate one correct gradient. Therefore, in the kernel plane of Emap and Fmap, only the aligned data (presented by deep color grids in Figure 5) involve valid computation. The misaligned data, through non-zero, fail to participate in valid computation and should be regarded as invalid data. This misalignment is inevitable in WG computation when Emap and Fmap are sparse. Since only the aligned data (valid data), which are less than the compressed data, can be mapped on the PE array, underutilization easily happens as illustrated in Figure 5. Therefore, unrolling K-P on the PE array is not suitable for sparse WG computation. On the other hand, the invalid computation could be naturally avoided without alignment in I-N or O-C unrolling. In WG computation, Fmap and Emap channels are inter-reused, each of the Fmap channels is reused by all the data of Emap, and vice versa. Hence skipping the mapping of zero data along the channel dimension merely eliminates the zero-involved computation and would not induce any invalid computation, as shown in Figure 5.

Based on the special computation property of sparse WG, to solve the mismatch between the hardware parallelism and WG data dimensions, and to leverage the channels' inter-reuse opportunities, the I-N (Fmap channel) and O-C (Emap channel) dimensions are chosen for unrolled on PE array.

3.2 Sparsity-aware kernel plane unrolling with VDT mechanism

Although I-N and O-C unrolling solve the mismatch problem and avoid invalid computation, they still suffer from the two-sided input sparsity. In I-N and O-C unrolling, the irregularity of compressed data induced by irregular input sparsity will destroy the order of output psum, resulting in index overhead. To avoid the disorder, we equip the PE array with RFs and unroll K-P by row in the RFs. Specifically, we store the Emap row (Erow) and Fmap row (Frow) in the RFs. RFs with Erows are shared by PE columns and RFs with Frows are shared by PE rows. Each PE is able to access its corresponding RFs to fetch the Erow and Frow data and compute the psums of the same gradient. In each PE, the psums are accumulated locally. Unrolling K-P in RFs not only helps to avoid the disorder but also leverages the data reuse opportunities of gradient psum. However, K-P unrolling in RFs also induces invalid computation, which should be eliminated to better exploit the two-sided input sparsity.

In WG, the elements of Fmap and Emap within the 2D CONV window should be aligned to generate a correct gradient. Failing to align the Fmap and Emap leads to invalid computation. We refer to the data involved in invalid computation as invalid-data. As mentioned before, when processing the sparse WG, the required alignment makes some non-zero data invalid. The compressed data in Fmap are valid only when the data they aligned with in Emap are non-zero, and vice versa. Figure 6(a) illustrates the distinction between compressed data and valid-data. The colored data represent no-zero data. For each of them, if the other multiplier in the aligned position is zero, the value would be invalid. The invalid computation can be avoided if the PE can selectively access the data (i.e., only fetch valid-data) in the RFs. The way to track valid-data is related closely to the encoding scheme for sparse data.

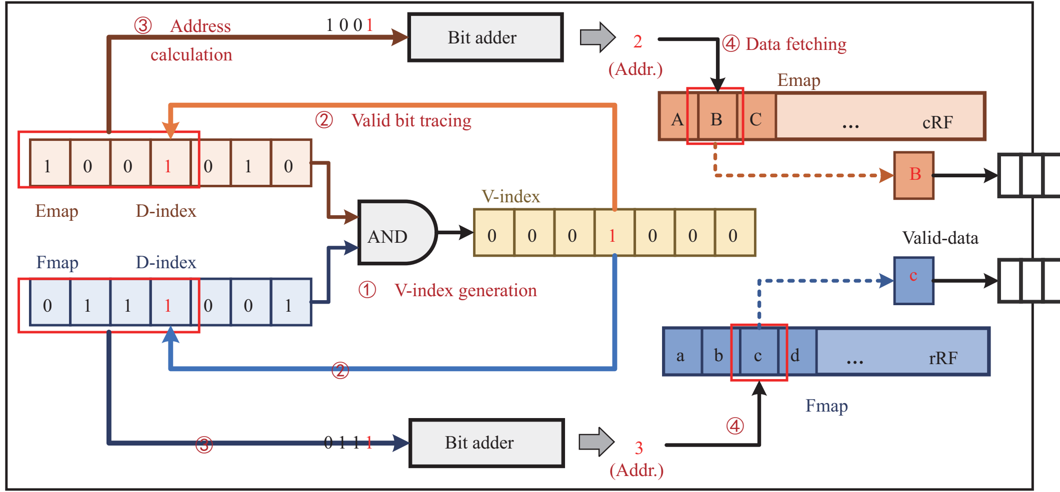


Figure 7 (Color online) Working mechanism of the VDT.

Previous sparsity architectures generally employ compressed sparse row (CSR) or compressed sparse column (CSC) as compress encoding and gain significant benefit when the sparsity is high. However, it will induce computing overhead to extract the valid-data if adopting CSR in sparse WG. To trace a valid value, the CSR coding is first decoded to obtain the absolute positions for all the compressed data in Fmap and Emap. Then the absolute positions of Fmap and Emap are compared to find the aligned data, and the invalid data are discarded. Thus when using CSR (or CSC) to compress sparse WG, all the data positions should be calculated whether the data are valid, which causes computation overhead.

Aiming at tracing the valid-data with low overhead, we design a VDT mechanism that employs binary strings for compressed data indexing. In the binary string, each bit represents a data position, indicating whether the corresponding value is zero, i.e., “1” for non-zero and “0” for zero. Examples of Fmap and Emap using the compressed data index (D-index) are given in Figure 6(b). With this indexing approach, the effective position could be obtained by bit-wise AND operations, and the generated result string is a valid-data index (V-index). This V-index can be used to retrieve the valid-data. With binary string indexing, the calculation and comparison of data positions can be eliminated.

To support the binary indexed data alignment and the valid-data fetching, we implement VDT as a micro-architecture, which is embedded in each PE. VDT takes the D-indexes of Fmap and Emap as input and outputs the aligned valid-data. Take the compressed data in Figure 6 as an example, the working mechanism of VDT is illustrated in Figure 7. VDT workflow: ① Fmap and Emap D-indexes are input to a bit-wise AND to obtain the valid-data index (V-index). ② VDT identifies the valid bit (“1”) and traces it back to the corresponding “1” in the D-indexes of Emap and Fmap. ③ For Emap (or Fmap), the valid bit is added up with all the bits in its front by a multi-in bit adder to calculate the valid-data address. ④ VDT uses the adding result to track the valid value in cRF (or rRF) and fetch it to the registers for gradient psum computation.

3.3 Sparsity-aware output plane unrolling within the PE

Although VDT helps to get rid of invalid computation, the risk of underutilization caused by output sparsity remains to be reduced. As I-N and O-C are unrolled on the PE array and K-P unrolled in the RFs, each PE is in charge of one gradient and the psums of different gradients are generated in parallel and accumulated locally. As shown in Figure 8, this would randomly make some PEs idle since the output gradient sparsity is irregular.

To alleviate the underutilization caused by the irregular gradient sparsity, we unroll the output plane within each PE. Each PE holds the computation of a gradient plane (gradients of corresponding weight kernel) instead of only one gradient, as presented in Figure 8. Although the gradient kernels are sparse, there are usually still non-zero gradients in each kernel. Therefore, the proportion of idle PEs can be significantly reduced. To support the computation, we store two Frows and two Erows in the corresponding RFs, respectively. In each PE, the computation order is rearranged so that three psum rows (i.e., a kernel of gradients) can be generated with two Frows and two Erows as input. In this way, the reuse

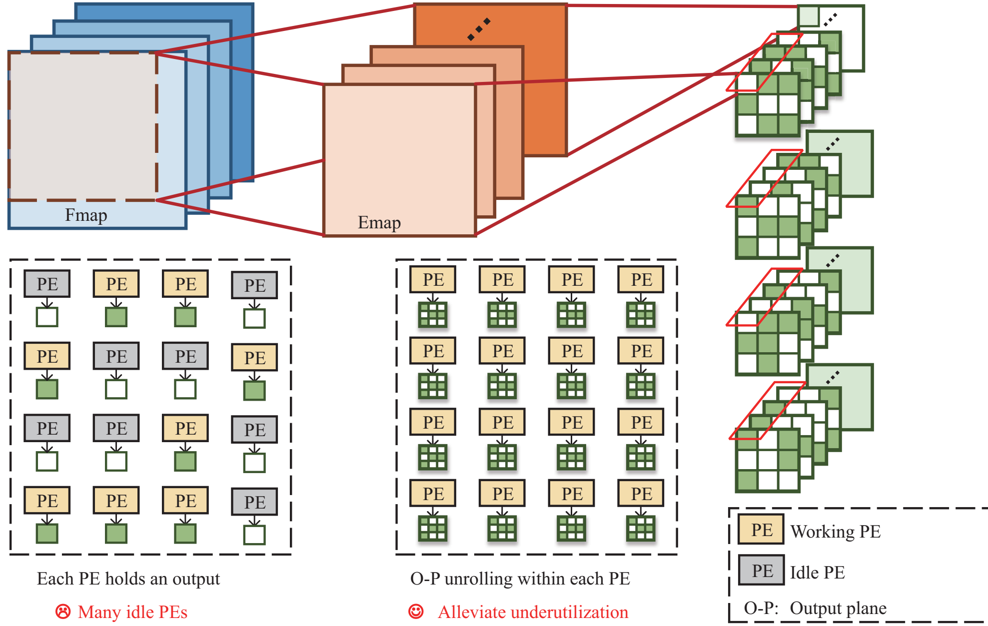


Figure 8 (Color online) Unrolling the output plane within each PE alleviates the underutilization caused by output sparsity.

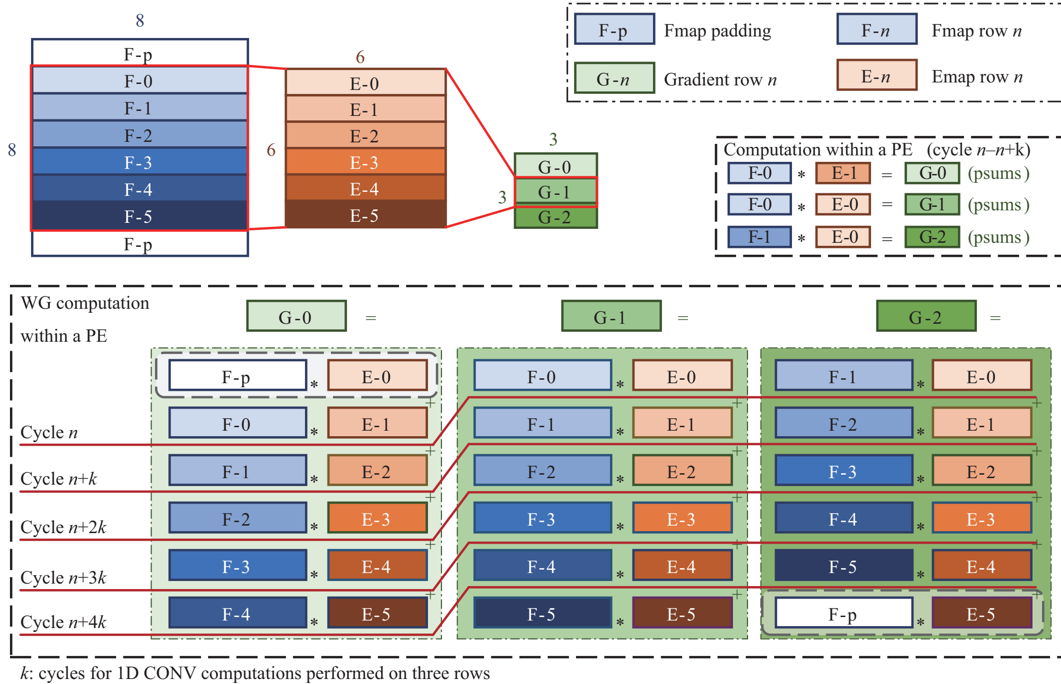


Figure 9 (Color online) Gradient kernel computing pattern for K-P unrolling within a PE.

opportunities of Fmap and Emap planes are exploited and the extra access can be reduced. Figure 9 illustrates this computing pattern using an example of 3×3 gradient kernel. When computing a gradient kernel, the 2D large window CONV can be regarded as the accumulation of several 1D CONV results. In this case, the 6×6 Emap slides on the 8×8 Fmap with unit stride to generate a 3×3 kernel of gradients. For the three gradient rows, the 1D CONV computations in the same color share the same two Erows and two Frows. For instance, the psums of G-0–G-2 could be generated by sharing E-0, E-1, and F-0, F-1. In most cases, Fmap is the same size as Emap and its first and last rows are zero paddings, of which the computation can be skipped. This modified computing pattern can be easily adapted to 1×1 gradients. For gradient kernels larger than 3×3, they will be tiled before computation.

As mentioned above, the computation of a gradient kernel in a PE can be divided into several 1D CONV

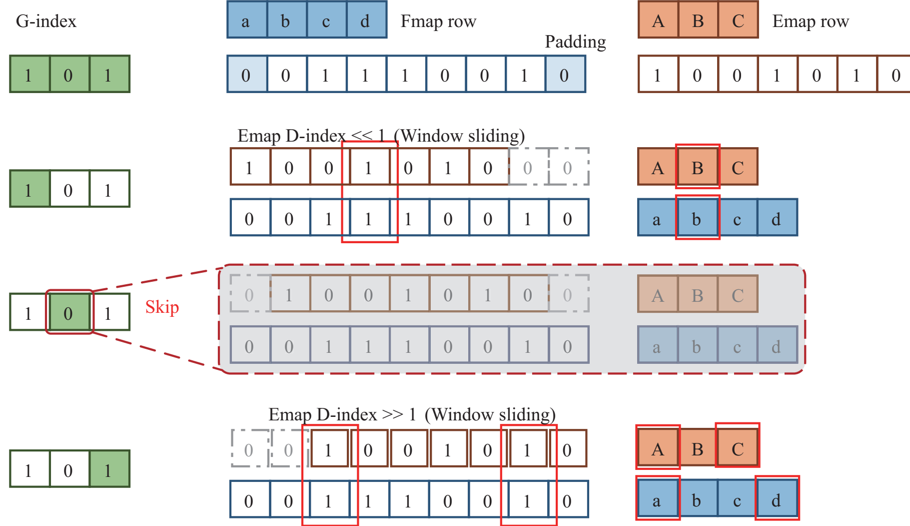


Figure 10 (Color online) Computation of a gradient row exploiting full sparsity.

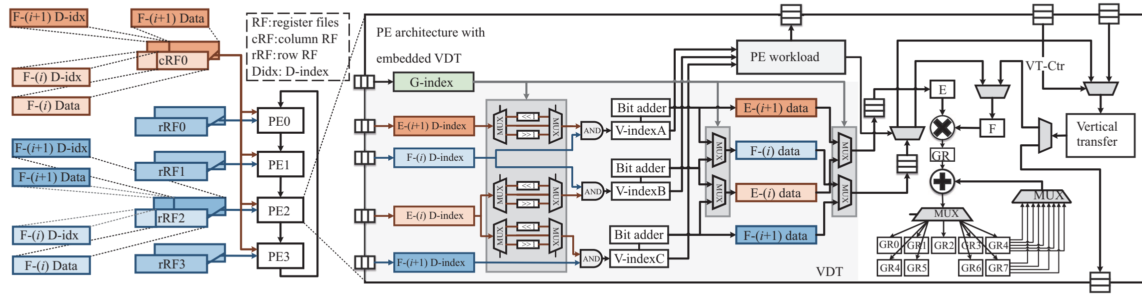


Figure 11 (Color online) Architecture of PE.

computations performed on each row. Figure 10 illustrates an example of such a gradient row computation exploiting the full sparsity. Binary strings are employed for sparse gradient data indexing, named gradient index (G-index). The G-index guides the window sliding and determines whether the computation should be skipped. The window sliding in the row is performed by shifting the D-index of Emap and the gradient psums are accumulated locally inside the PE. The PE architecture that supports this sparse gradient computation is illustrated in Figure 11. Two Frows (F_i, F_{i+1}) and two Erows (E_i, E_{i+1}) contained in the RFs are available to each corresponding PE. There are three VDTs in a PE, each for the data fetching of one gradient row. When dealing with 1×1 kernels, as only two VDTs are required, the third one would be gated. The inputs of a VDT are the D-indexes of the Frow and Erow, i.e., F_i, E_{i+1} for G_i, F_i, E_i for G_{i+1}, and F_{i+1}, E_i for G_{i+2}. Before being sent to the VDT, the D-index of each Erow may shift right, left, or maintain to perform window sliding, which is determined by the corresponding G-index in the PE.

4 Overall architecture of SWG

4.1 HUSA dataflow based SWG architecture

The overall architecture of SWG which supports the proposed HUSA dataflow is presented in Figure 12. The computation core of SWG is a 2D PE array consisting of 4 PE rows and 16 PE columns. Each PE row is accompanied by a row RF (rRF) and each PE column is accompanied by two column RFs (cRFs). The two cRFs act as ping-pong RFs for inter-column workload balance, which will be introduced in Subsection 5.2. For each PE column, there is an intra-column balancer, which will be introduced in Subsection 5.1. In each PE, besides one multiplier-accumulator and several registers to hold the psums, there are three VDTs embedded in it for valid-data fetching as shown in Figure 11.

In HUSA dataflow, the Fmap and Emap channel dimensions are unrolled on the PE array to match the

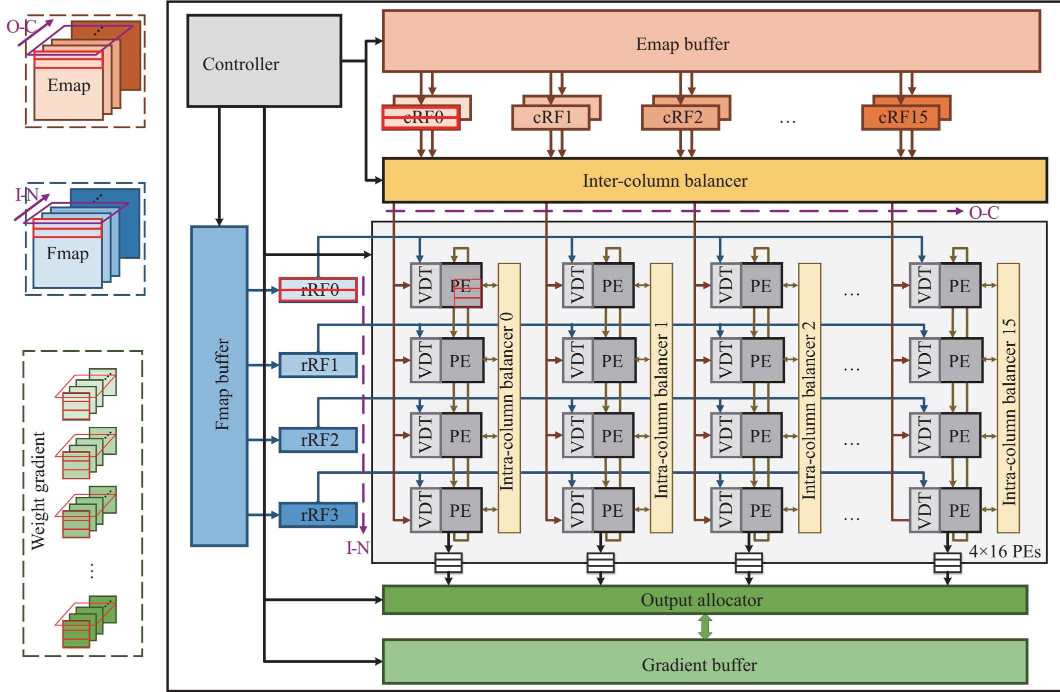


Figure 12 (Color online) Overall architecture of SWG.

hardware parallelism with data dimensions. The kernel plane dimension is unrolled in the RFs to leverage the data reuse opportunities. The output plane is unrolled within each PE. The Fmap and Emap planes are tiled by row and stored in the rRFs and cRFs attached to the PE array. Each rRF stores two Frows and each cRF stores two Erows. The rRF is shared by its corresponding PE row for Emap reuse, and rRFs of different PE rows store Frows of different channels for Fmap channel unrolling. Similarly, cRFs shared by corresponding PE columns and different cRFs store the Erows of different channels. Thus in the 2D PE array, every PE row shares an Fmap channel and every PE column shares an Emap channel. Each PE stores an output gradient plane locally. In this way, Frows are reused across the PE row and Erows are reused across the PE column, respectively.

In SWG, Fmap data from different channels are mapped on 4 PE rows and channels of Emap data are mapped on 16 PE columns to generate 64 gradient psum results in parallel. The 2D CONV within a large window is performed row by row. For each computation, PE fetches the valid-data of corresponding Frow and Erow through the embedded VDT to compute one of the gradient psums and store it locally. Take 3×3 gradient as an example, every PE performs the 9 psum results of one gradient kernel using two Erows and two Frows stored in RFs. Each psum of the gradient is accumulated locally inside the PE, and the window sliding in the row is performed by shifting the data fetching in cRF. If the Fmap/Erow is larger than the RF, it will be divided into sequential segments to fit the RF. The output allocator sends the gradient psum results to the output buffer and fetches them back to PEs when it comes to the next computing of corresponding gradients. When dealing with the full-connect (FC) layer, the feature and error vectors are tiled and stored in the rRFs and cRFs, respectively. This case is much simpler as there is no invalid computation and the computing results do not need accumulation in FC.

The total PE number in our design is relatively small compared to previous training accelerators. Considering the sparsity in WG, a large number of PE may cause a waste of computing resources. The PE row and PE column can be expanded to handle the dense WG computation.

4.2 SWG architecture's applicability to emerging models

In this subsection, we will discuss SWG architecture's applicability to emerging models such as EfficientNetv2 [19], MobileNet [20], and MobileNetV2 [21]. Compared to the widely used DNN models, these emerging models leverage depthwise (DW) CONV. In DW CONV, each input Fmap is convolved by its corresponding weight kernel separately, and the generated output Fmaps are concatenated in the channel dimension. In WG computation, each Fmap channel computes with its corresponding Emap channel to

obtain the gradient kernel. Each channel of Fmap only computes with its corresponding channel of Emap and will not compute with other Emap channels. Similarly, each Emap channel only computes with its corresponding Fmap channel. In other words, there is no reuse opportunity across the Fmap channel and Emap channel dimensions. Therefore the data mapping of DW on SWG should be different from that of the standard convolution. However, SWG does not need to be redesigned. The architecture can apply to DW CONV with appropriate data mapping. In SWG, rRFs are shared by corresponding PE rows, and cRFs are shared by corresponding PE columns. For data mapping of DW CONV, the Emap plane is tiled by row and stored in the cRF. Different cRFs store the Erows of different channels. The Fmap is tiled by the channel dimension and the same pixel of different channels are stored in the rRF. Different rRFs store the data of different Frows. The channels of Fmap data stored in the rRF respond to the channels of Emap stored in the different cRFs. Thus in the 2D PE array, the PEs of the same PE row compute different channels. Every PE column shares an Emap channel. Each PE stores one row of an output gradient plane locally. In this way, Erows are reused across the PE column by Frows of the corresponding channel. For each PE, the data of the Erow are reused by Fmap data. Since the gradient kernel size of DW is usually 3×3, the fourth PE row could be gated to make the PE array (with 3 working PE rows) match the 3 rows of gradient kernel.

5 Intra- and inter-column workload balancer

There remains an underutilization problem resulting from workload imbalance to be addressed. Due to the irregularity of input and output data, the computing task varies from PE to PE, making the execution time depend on the operation time of the PE with the largest workload. The workload imbalance caused by irregular sparsity will harm the performance badly. The sparsity caused imbalance is not a new issue, previous studies have attempted to deal with this problem. In SparTen [22], the weight kernels are sorted by their densities off-line and the kernels with similar density are assigned to one cluster for processing in parallel. However, the workload depends on both kinds of input data, the density of the weight kernel cannot determine the corresponding workload alone. In other words, balancing the weight kernels could not essentially balance the workload. Also, the off-line sorting is not suitable for dynamic sparse data like Emap and Fmap. LNPU (deep-learning neural processing unit) [23] relieves the imbalance by evenly balancing the compressed input data to each PE. Although this method works when dealing with the one-sided input sparsity caused imbalance, it fails to alleviate the imbalance in this work caused by the irregularity of full sparsity.

In SWG, the workload of each PE is the valid computation to be performed in the PE. Since the non-zero data is not necessarily the valid-data, the density of Fmap could not reflect the workload of the corresponding PE. Simply analyzing the density of each Fmap, partitioning the Fmap, and remapping them evenly on each PE will not alleviate the imbalance issue. In this work, the imbalance results from two aspects, namely the intra-column imbalance and the inter-column imbalance. To even out these two kinds of imbalance, we propose IIBLC, which consists of the intra-column balance scheme (IntraBLC) and the inter-column balance scheme (InterBLC).

5.1 Intra-column workload balancer

The intra-column imbalance is induced by the irregular sparsity of Fmap. In a PE column, PEs share the same Erows and each PE receives data from its corresponding Frows. The sparsities are various in different Frows, making the workloads within a PE column various. PEs with less workload wait till the last PE in the same column completes its work, which incurs underutilization. To even out the intra-column imbalance, the workload of each PE should be calculated first. In this work, the workload could be obtained by adding up all bits of the PE's corresponding E-index generated from VDT. The workloads of PEs within a column are compared, part of the heaviest workload is remapped to the PE with the lightest one to balance the computation tasks evenly. The remapping of workload involves partitioning the corresponding Frow and transferring the partitioned data. To support the transfer of partitioned data, each PE should be allowed to access other Frows. A direct solution is to connect every PE with all 8 rRFs. While building a fully-connected network between all the PEs and all the rRFs provides flexibility for data delivery, this also brings high wiring complexity. Instead of introducing costly connections, we exploit the interconnections between adjacent PEs within a column. By linking every two neighbor PEs, each rRF is available for the PEs within a column since the data can be transferred

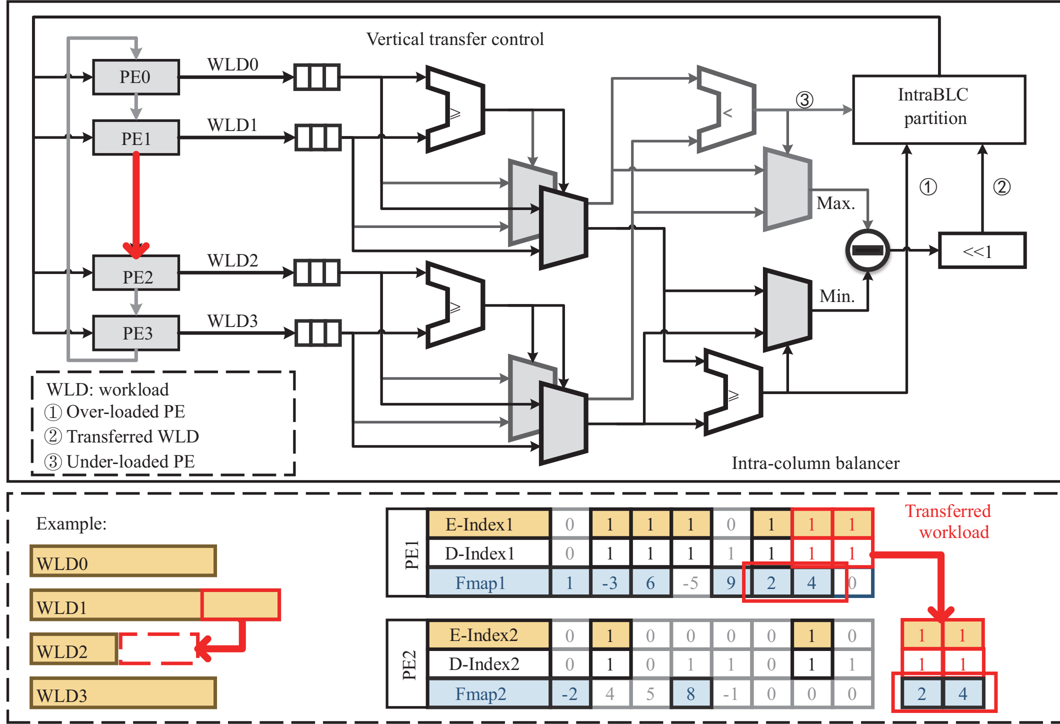


Figure 13 (Color online) Architecture and working mechanism of the intra-column balancer.

through the connections between the PEs. In this way, the connecting complexity would be reduced to $O(n)$ compared to that of the all-to-all network ($O(n^2)$). While the longest transmission time is increased from $O(1)$ to $O(n)$. Using the intra-column connections to deliver data takes slightly more time than the all-to-all network due to the longer transmission path. However, the transmission of data would not prolong the execution time as the transmission overhead can be hidden behind the data computation in PEs.

Figure 13 shows the architecture of IntraBLC and illustrates how the intra-column imbalance is tackled by IntraBLC using the 1D-systolic-like network. IntraBLC takes the workloads of the PEs in a column as input, seeking out the maximum and minimum workloads. The intra-column balance is achieved by averaging out the workloads. Intra-BLC transfers the partitioned workload from the over-loaded PE to the under-loaded PE. In the example given in Figure 13, PE1 suffers the heaviest computing tasks of 6 (WLD1), and PE2 is only in charge of 2 tasks (WLD2). To even out the imbalance, IntraBLC maps the last 2 tasks of PE1 to PE2. The path from the over-loaded PE to the under-loaded PE is enabled by the IntraBLC controller. When PE2 deals with the workloads from PE1, IntraBLC takes the V-index of PE1 to extract the corresponding valid-data of Fmap1 and Emap. In this way, the load imbalance can be reduced. The analysis and solution apply to other PE columns since each column suffers from the same imbalance situation and they are independent.

5.2 Inter-column workload balancer

While the intraBLC balances the workload within each PE column, the inter-column imbalance remains unsolved. Due to the irregular sparsity in Emap, the execution time of different PE columns are various. In traditional designs where the input data (Emap) of the PE array share a unified data controller, the overall processing time of the PE array will be determined by the PE column with the longest execution time. Therefore keeping the data allocated at the same pace makes some PE columns idle and incurs PE underutilization. To avoid the performance loss induced by idle PE columns, InterBLC dynamically adjusts the Emap allocation for each PE column. The computing state of each PE column is monitored by InterBLC. Once the computation tasks of a PE column are done, InterBLC allocates a new Erow to the cRF of the PE column and the PE column continues to work. The InterBLC architecture and its effect on speeding up are shown in Figure 14. By making each PE column process independent, the execution time of the PE array can be reduced. It is notable that it may cause extra cycles for data

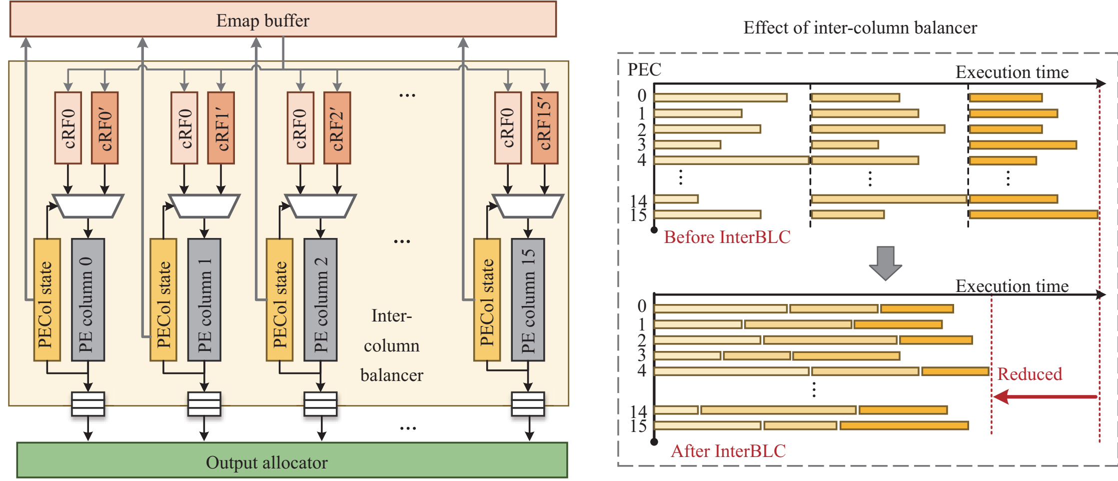


Figure 14 (Color online) Architecture of inter-column balancer and the effect on speedup.

Table 2 ASIC area and power for SWG (TSMC 28 nm)

Component	Power (mW)	Area (mm^2)	Power (%)	Area (%)
MACs	1.510	0.2045	14.10	5.79
RFs	1.067	0.2694	9.97	7.62
Buffers	5.670	1.2987	52.96	36.76
VDT	1.935	1.7566	18.07	39.17
IIBLC	0.381	0.0034	3.56	0.10
Others	0.144	0.0016	1.34	10.52
Total	10.707	3.5332	–	–

access as different PE columns may complete their tasks at the same time. To cover the potential extra access delay, we apply the ping-pong cRFs which serve alternately to each PE column. The output aggregator collects the output psums from the PE columns and sent the data to the gradient buffer after accumulating with the previous psums.

6 Evaluations

6.1 Experiment setup

Hardware evaluation. SWG is designed and synthesized using a synopsis design compiler with TSMC 28 nm LP technology. Functional simulations and performance measurements are conducted on Synopsys VCS and the chip-only power consumption is estimated by PrimeTime PX.

Benchmarks and model sparsity. We select four famous DNNs as benchmarks: AlexNet [2], GoogLeNet [3], VGGNet [5], and ResNet [4]. The models are trained on the CIFAR-100 dataset with a batch size of 128. From epoch 180 to 280 in a total of 300 epochs, we randomly extract 10 batches from each epoch at a step size of 20 epochs. In Figure 15, the average sparsity of Fmap, Emap, and WG from the selected batches are shown.

Chip characteristics. The layout and characteristics of SWG in the TSMC 28 nm LP technology are shown in Figure 16. SWG has a total of 64 MACs, 8 kB RFs, and 192 kB SRAM buffer. Each MAC supports one 16-bit multiplier and one 32-bit accumulator. The accelerator consumes a small area of only 3.53 mm^2. Working at 200 MHz under the 0.81 V operating voltage, SWG's total power range is 8.56–10.707 mW, depending on benchmarks and data sparsity. The average energy efficiency is 14.2 TOPS/W, measured on the six benchmarks. The area and power estimates for SWG are listed in Table 2.

Critical path analysis. We use a synopsis design compiler with TSMC 28 nm LP technology to synthesize the proposed SWG design and obtain the timing report. The frequency of 200 MHz under 0.81 V operation voltage is selected to achieve low power. We utilize the timing report generated by the compiler to find the critical path. In SWG architecture, the critical path lies in VDT, which is because

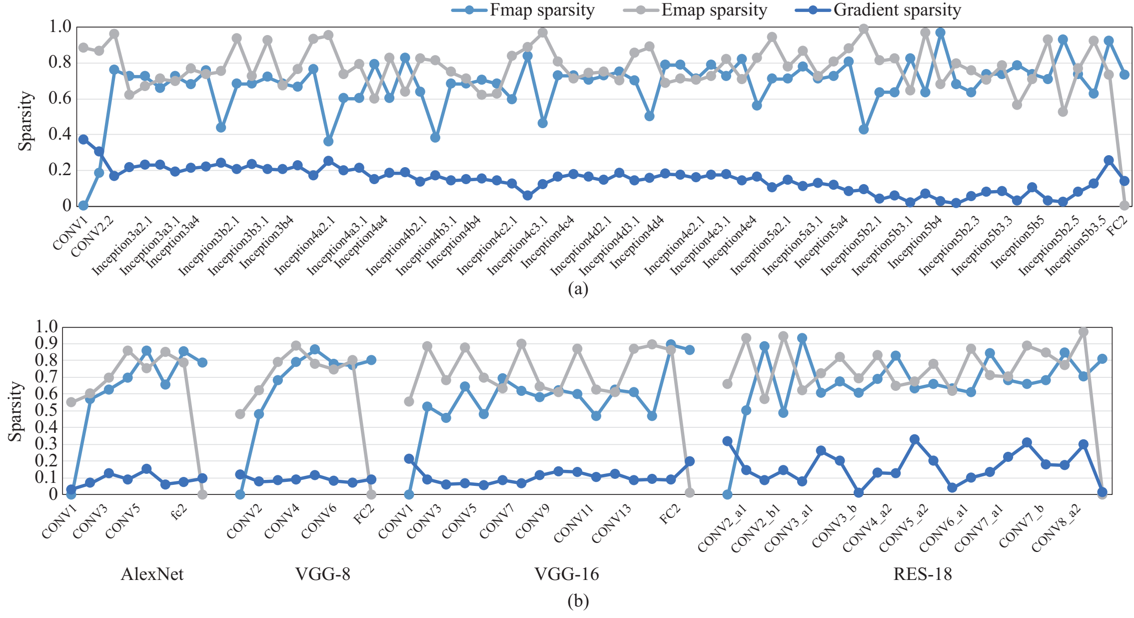
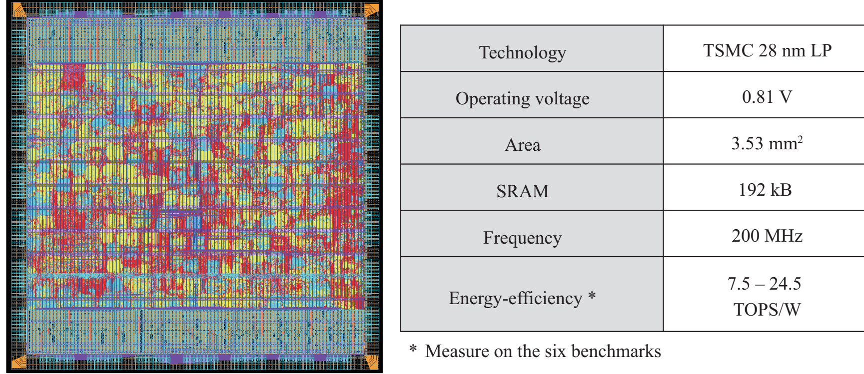


Figure 15 (Color online) Sparsity of Emap, Fmap, and WG in (a) GoogLeNet, (b) AlexNet, VGG-8, VGG-16, and RES-18.



Technology	TSMC 28 nm LP
Operating voltage	0.81 V
Area	3.53 mm^2
SRAM	192 kB
Frequency	200 MHz
Energy-efficiency *	7.5 – 24.5 TOPS/W

* Measure on the six benchmarks

Figure 16 (Color online) Layout and characteristics of SWG.

of the valid data tracing. In VDT, the valid bit ("1") in the V-index should be identified and its position should be traced to that of the corresponding D-indexes of Emap and Fmap. For each valid bit, VDT needs to go through all the bits in front of the valid bit until finds it. Therefore, VDT consumes more time than other components of the architecture.

6.2 Benefits of exploiting the special computation property

To evaluate the benefits of HUSA dataflow on PE utilization, we compare the average PE utilization of the benchmarks among SWG, RS, and no local reuse (NLR) dataflow. For a fair comparison, we modify the architecture with the same amount of computing resources (64 MACs) by adding requisite interconnections between PEs and corresponding control logic to adapt RS and NLR processing patterns respectively.

We compare the data flow without considering the sparsity. As shown in Figure 17, SWG maintains high utilization in different benchmarks, which achieves 97.2% on average. The PE utilization using RS is no higher than 67% (VGG8), and the lowest (23%) appears in GoogLeNet. The decline of utilization results from the lack of two data dimensions in WG computation, the output row and the input channel. When applying RS dataflow, the input rows and weight rows are mapped on the PE array and the output rows are reused by diagonally moving the psums and locally accumulating them. The output row is large in Fmap or Emap computation (e.g., 32) thus RS can achieve high utilization. While in WG, the output row is usually small ($K = 3$ or $K = 1$) and insufficiency of output dimension causes the degradation of

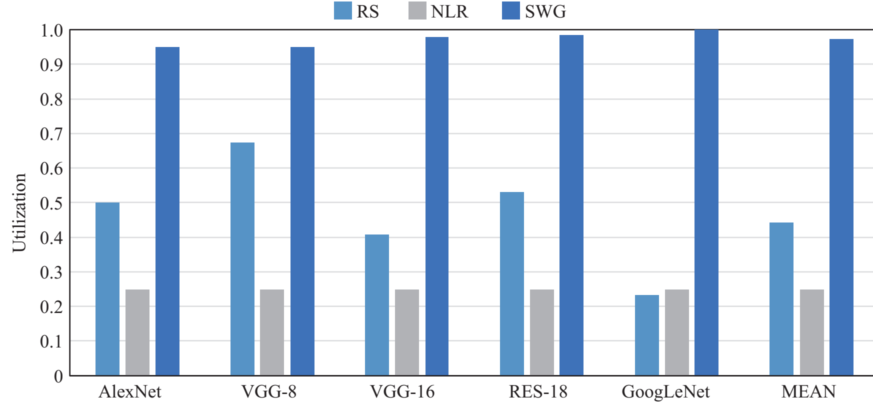


Figure 17 (Color online) Average utilization using RS, NLR, and SWG.

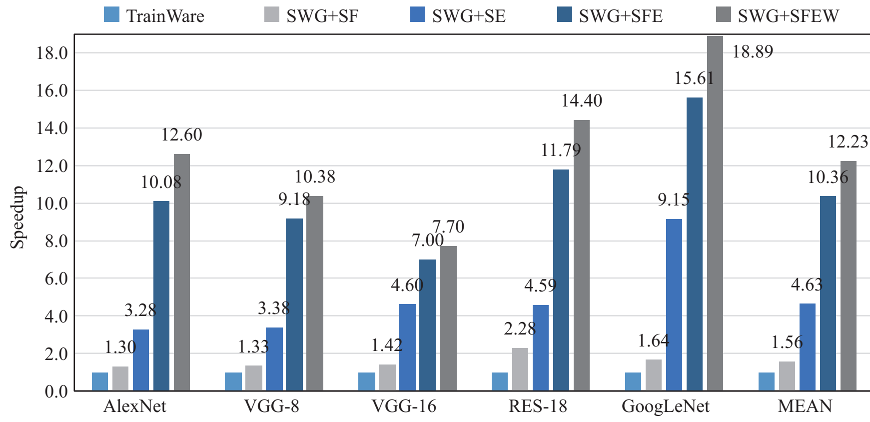


Figure 18 (Color online) SWG's speedup over TrainWare.

utilization. In addition, there is no input channel dimension in WG computation to be mapped on the PE array, which further leads to an underutilization. In NLR dataflow, the input and output channels should be spatially mapped on the PE array. Similarly, in WG computation, NLR suffers from low utilization caused by insufficient input channels.

6.3 Benefits of exploiting the full sparsity

We compare the performance of SWG with another WG computing architecture TrainWare to prove the benefits of exploiting the full sparsity by HUSA dataflow with VDT in SWG. The PE array in TrainWare is highly related to the gradient size, which refers to the kernel size in TrainWare. To fairly compare the acceleration, we modified TrainWare to fit different gradient sizes. In the modification, the PE array structure with 27 MACs presented in TrainWare is taken as the base architecture and three PE arrays with a total of 81 MACs are adopted as the MAC amount is comparable with SWG's. SWG's speedup over TrainWare is presented in Figure 18. The acceleration time of each model indicates the total amount of WG computation time in all layers. For briefness, SWG+SF, SWG+SE, SWG+SFE, and SWG+SFEW are used to represent the WG computation on SWG with sparse Fmap, sparse Emap, sparse Fmap and Emap (two-sided sparsity), and sparse Fmap, Emap and weight (full sparsity), respectively.

As presented in Figure 18, SWG+SF only reaches a speedup of 1.56×, which is relatively lower compared to SWG+SE (4.63×). The gap between SWG+SF and SWG+SE results from the difference between the balancing ranges of IntraBLC and InterBLC. In SWG+SF, the imbalance is purely induced by the irregularity of sparse Fmap. InterBLC does not make a difference in this case as the PE columns accomplish their tasks simultaneously and the computation duration depends on the sparse Fmap. Since there is no optimization space for InterBLC and only IntraBLC takes effect, in which the balancing range is relatively small (four PEs), the speedup of SWG+SF is minor. As for SWG+SE, where the imbalance comes from the irregular sparsity of Emap, InterBLC tackles the imbalance problem of the 16 PE columns. The benefit of SWG+SE is more obvious as the balancing range of InterBLC is larger.

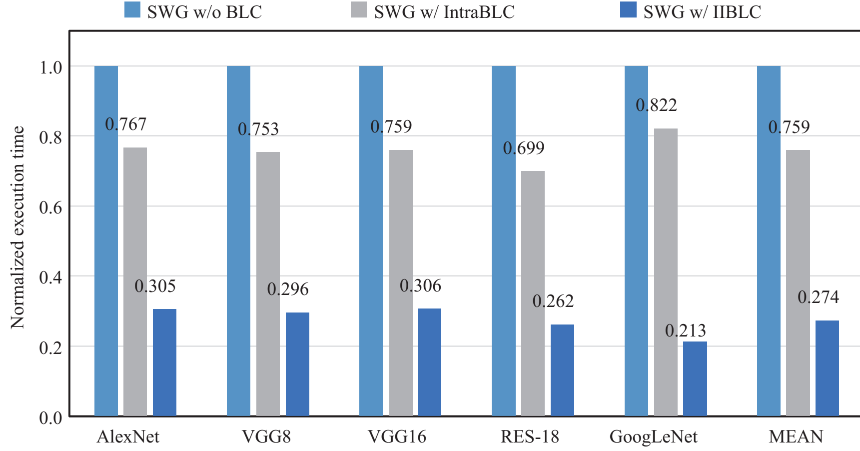


Figure 19 (Color online) Speedup of SWG with 2-level balancer.

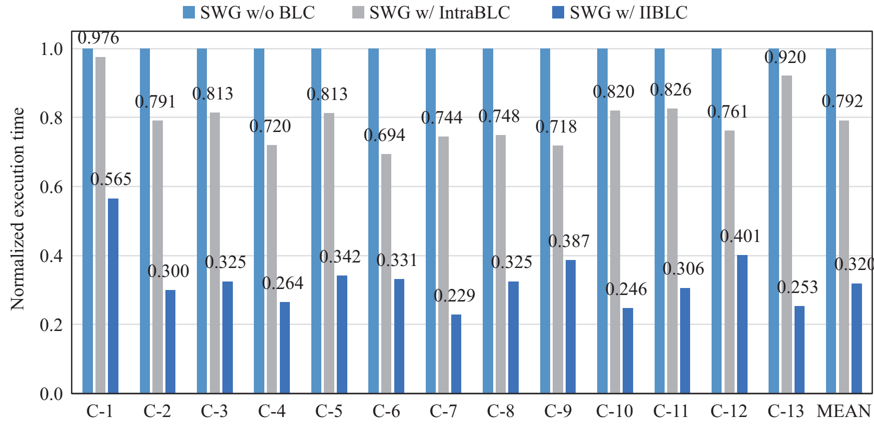


Figure 20 (Color online) Layer-wise speedup comparison in VGG16.

SWG+SFE achieves an average speedup of 10.36×. Among the benchmarks, GoogLeNet sees the highest average speedup (15.61×). In this case, both Fmap and Emap are sparse, reducing the computation and speeding up the processing. In addition, the cooperation of IntraBLC and InterBLC guarantees global balance and assists to gain a higher speedup. The speedup can be further increased by introducing weight sparsity, which helps to eliminate the computation of corresponding gradients. As shown in the results, 12.23× speedup is achieved by SWG+SFEW. In the experiments, for simplification, we merely prune 10% weights to demonstrate the benefits of SFEW. The speedup effect can be enhanced by increasing the pruning ratio since it is highly related to weight sparsity.

6.4 Benefits of intra- and inter-column balancer

To evaluate the benefits of SWG's IIBLC on the workload imbalance problem, we conduct a comparison of the designs with different balancing levels on total acceleration time, as shown in Figure 19. For better comparison, the execution time is normalized to that of SWG without the balancer (SWG w/o BLC), which is implemented by turning off both intra- and inter- column balancers. The IntraBLC in the histogram denotes the execution time of SWG with only the IntraBLC works and the InterBLC is gated.

Compared to the baseline (without balancer), the IntraBLC only gains the small benefit of 24.1% time reduction on average. The IntraBLC merely even out the imbalance within each PE column. Therefore the acceleration still suffers from the imbalance in different PE columns and the execution time depends on the PE column with the heaviest workload. Applying both IIBLC, SWG saves the acceleration time by 72.6% on average. Among the benchmarks, GoogLeNet shows the best result that 78.7% execution time is reduced.

Figure 20 shows a layer-wise speedup comparison in VGG16 to study the effect of the workload balancer.

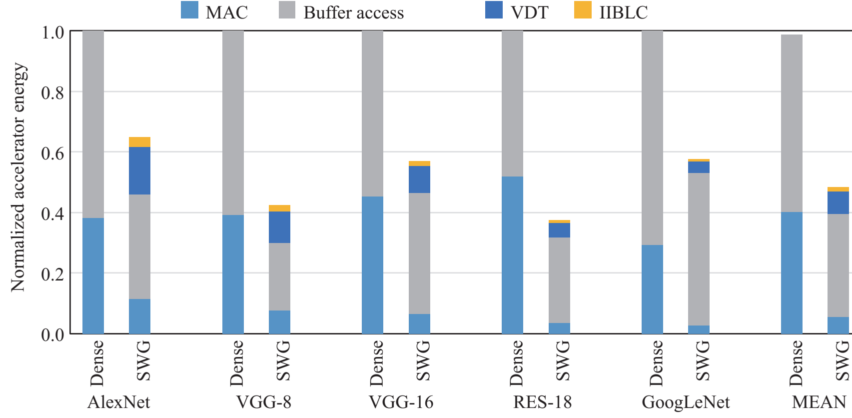


Figure 21 (Color online) SWG's benefits on energy saving.

As demonstrated in Figure 20, with the IntraBLC, SWG increases the utilization by 1.44× in the C-6 layer and 1.26× on average, which saves 30.6% and 20.8% acceleration time, respectively. With the IIBLC, SWG further reduces the execution time by 68.0% on average. In the best case (C-7 layer), up to 77.1% execution time can be saved.

6.5 Benefits on energy saving

We compare the accelerator energy of dense and sparse WG computation to demonstrate SWG's benefits on energy saving, the results shown in Figure 21. The sparse computation energy contains on-chip buffer access, computing energy, and the operating energy of VDT and IIBLC. In dense computation the energy only consists of the first two components, and the VDT and IIBLC are gated to eliminate the unnecessary energy. For better comparison, the accelerator energy is normalized to Dense mode.

Although VDT and IIBLC induce extra energy, they bring significant advantages to the total accelerator energy saving by supporting sparse WG. Compared to dense computing, SWG saves 49.27% of energy on average. SWG reduces the energy by 35.27% in AlexNet, which is the worst case. In ResNet, SWG shows the most energy reduction of 62.52%.

6.6 Benefits of SWG in state-of-the-art training work

We combine SWG with state-of-the-art (SOTA) training work, LNPU [23], to prove the benefits of SWG. In LNPU, FF, BP, and WG phases share the same architecture. To evaluate the benefits brought by SWG, we take LNPU and SWG to assemble a training architecture, named L-SWG, where FF and BP are processed in LNPU and the WG phase is accelerated by SWG. Since LNPU has 768 PEs, we scale SWG's PE number to 768 and synthesize it with a synopsis design compiler in TSMC 28 nm technology. We run VGG-16 for one training epoch to compare the performance and energy efficiency of LNPU and L-SWG.

Performance. We build a cycle-accurate simulator based on LNPU to model the execution time. In LNPU, Fmap sparsity and Emap sparsity are exploited in FF and BP, respectively. In the WG phase, LNPU only supports Fmap sparsity while the sparsity in Emap is ignored. Since the sparse information in Fmap and Emap can be obtained, we exploit both Fmap and Emap sparsity to process the WG phase on L-SWG. Figure 22 presents the layer-wise execution time comparison of VGG-16, the execution time is normalized to LNPU's. By optimizing the WG phase, L-SWG saves the training time by 21.9%.

Energy efficiency. We estimate the power consumption of LNPU and L-SWG to obtain the energy efficiency. The power consumption of LNPU is 367 mW operating at 200 MHz, 1.1 V, in 65 nm technology. For a fair comparison, we scale the voltage from 1.1 to 0.81 V (the operating voltage of SWG) at 200 MHz to estimate the power consumption of LNPU. We further normalize the power consumption to 28 nm technology. The energy efficiency of LNPU is 7.56 TOPS/W. As for L-SWG, the power consumption consists of SWG and LNPU (fixed point) power. The power consumption of SWG is estimated by PrimeTime PX. The data type in LNPU is a 16-bit floating point. The difference between floating and fixed point multiplication-and-accumulation operations is responsible for the difference between the power consumption of floating and fixed point architecture. Therefore we normalize the LNPU's power

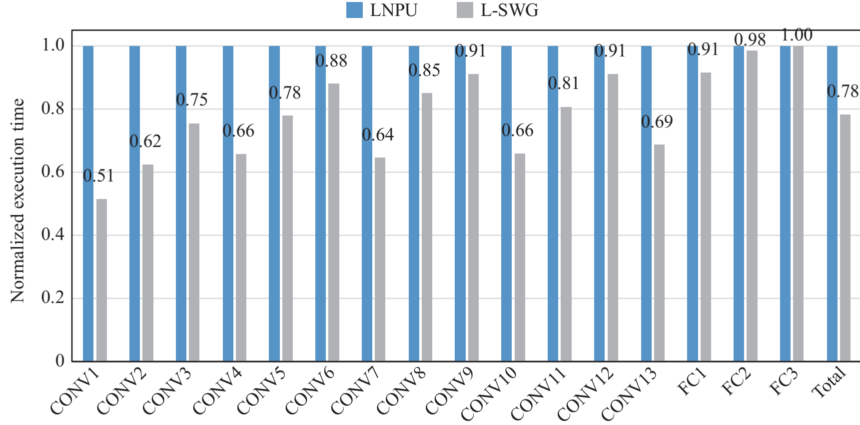


Figure 22 (Color online) Normalized execution time of VGG-16 on LNPU and L-SWG.

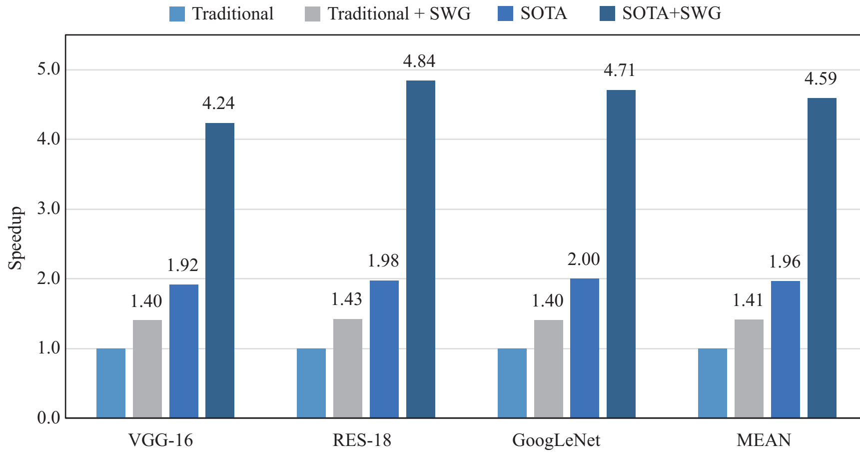


Figure 23 (Color online) SWG's speedup over traditional and SOTA training process.

in L-SWG to a fixed point based on the synthesized energy consumption of the 16-bit operation in 28 nm technology, which is 1.1 and 0.89 pJ for floating and fixed points, respectively. With SWG, L-SWG achieves a higher energy efficiency of 10.58 TOPS/W.

6.7 Benefits of SWG for the traditional and the SOTA training process

To demonstrate the benefits of SWG for the training process, we conduct experiments on the traditional and the SOTA training process to compare the performance with and without the help of SWG. In the traditional training process [6, 7], FF, BP, and WG phases perform computation without considering any data sparsity. The SOTA training process [23] only exploits the data sparsity in FF and BP phases. We build a computation simulator based on the 2D systolic array to model the computation behavior and obtain the cycle-accurate execution time. Because the systolic array has been proven efficient for DNN processing [24] and can also be adapted to handle sparse computation [25]. The training process with SWG indicates that the WG phase is executed by SWG, which fully exploits data sparsity during the WG computation. The comparisons are conducted on VGG-16, RES-18, and GoogLeNet.

Figure 23 compares the performance of the different training processes. For briefness, Traditional, Traditional+SWG, SOTA, and SOTA+SWG are used to represent the traditional training process without SWG, the traditional training process with SWG, the SOTA training process without SWG, and the SOTA training process with SWG, respectively. All the numbers are normalized to the traditional training process's performance for better comparison. With the help of SWG, the traditional process gains an average speedup of 1.41×. In traditional+SWG, only the WG phase exploits the sparsity of Fmap and Emap. While in the SOTA training process, both FF and BP phases leverage the sparsity. Therefore, the SOTA training process without SWG may be faster than the traditional one with SWG's help, as the results of benchmarks show. Notably, the speedup of SOTA+SWG over SOTA is higher

than that of Traditional+SWG over Traditional. This is because the execution time reduced by SWG contributes more to the SOTA process training time than that to the traditional one. On average, SWG brings a 2.34× speedup to the SOTA training process.

7 Conclusion

In this paper, we propose SWG, an architecture for sparse weight gradient computing, which aims at the acceleration of gradient computation and exploits the full sparsity in WG. Our design includes an HUSA dataflow with a VDT mechanism to accelerate the sparse WG computation and an IIBLC to dynamically tackle the workload imbalance resulting from the irregular sparsity. On average, SWG achieves a speedup of 12.23× over another gradient computation architecture, TrainWare. As a WG acceleration architecture, SWG helps to improve the training energy efficiency of SOTA accelerator LNPU from 7.56 to 10.58 TOPS/W.

Acknowledgements This work was supported in part by National Natural Science Foundation of China (Grant Nos. U19B2041, 62125403, 92164301), National Key Research and Development Program (Grant No. 2021ZD0114400), Science and Technology Innovation 2030 – New Generation of AI Project (Grant No. 2022ZD0115201), Beijing National Research Center for Information Science and Technology, and Beijing Advanced Innovation Center for Integrated Circuits.

References

1 Zhou Z-H, Tan Z-H. Learnware: small models do big. Sci China Inf Sci, 2024, 67: 112102
2 Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks. Commun ACM, 2017, 60: 84–90
3 Szegedy C, Liu W, Jia Y Q, et al. Going deeper with convolutions. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015. 1–9
4 He K M, Zhang X Y, Ren S Q, et al. Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 770–778
5 Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition. 2014. ArXiv:1409.1556
6 Venkataramani S, Ranjan A, Banerjee S, et al. SCALEDEEP: a scalable compute architecture for learning and evaluating deep networks. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017. 13–26
7 Chen Y J, Luo T, Liu S L, et al. DaDianNao: a machine-learning supercomputer. In: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014. 609–622
8 Hu H, Peng R, Tai Y W, et al. Network trimming: a data-driven neuron pruning approach towards efficient deep architectures. 2016. ArXiv:1607.03250
9 Li H, Kadav A, Durdanovic I, et al. Pruning filters for efficient convnets. 2016. ArXiv:1608.08710
10 Lebedev V, Lempitsky V. Fast convnets using group-wise brain damage. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016. 2554–2564
11 He Y H, Zhang X Y, Sun J. Channel pruning for accelerating very deep neural networks. In: Proceedings of the IEEE International Conference on Computer Vision, 2017. 1389–1397
12 Guo L H, Chen D W, Jia K. Knowledge transferred adaptive filter pruning for CNN compression and acceleration. Sci China Inf Sci, 2022, 65: 229101
13 Choi S, Sim J, Kang M, et al. TrainWare: a memory optimized weight update architecture for on-device convolutional neural network training. In: Proceedings of the International Symposium on Low Power Electronics and Design, 2018. 1–6
14 Chen Y H, Emer J, Sze V. Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. SIGARCH Comput Archit News, 2016, 44: 367–379
15 Zhang S J, Du Z D, Zhang L, et al. Cambricon-X: an accelerator for sparse neural networks. In: Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2016. 1–12
16 Albericio J, Judd P, Hetherington T, et al. Cnvlutin: ineffectual-neuron-free deep neural network computing. SIGARCH Comput Archit News, 2016, 44: 1–13
17 Zhou X D, Du Z D, Guo Q, et al. Cambricon-S: addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In: Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018. 15–28
18 Parashar A, Rhu M, Mukkara A, et al. SCNN: an accelerator for compressed-sparse convolutional neural networks. SIGARCH Comput Archit News, 2017, 45: 27–40
19 Tan M, Le Q. EfficientNetV2: smaller models and faster training. 2021. ArXiv:2104.00298
20 Howard A, Zhu M, Chen B, et al. MobileNets: efficient convolutional neural networks for mobile vision applications. 2017. ArXiv:1704.04861
21 Sandler M, Howard A, Zhu M, et al. MobileNetV2: inverted residuals and linear bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018. 4510–4520
22 Gondimalla A, Chesnut N, Thottethodi M, et al. SparTen: a sparse tensor accelerator for convolutional neural networks. In: Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019. 151–165
23 Lee J, Lee J, Han D, et al. 7.7 LNPU: a 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16. In: Proceedings of IEEE International Solid-State Circuits Conference (ISSCC), 2019. 142–144
24 Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA), 2017. 1–12
25 He X, Pal S, Amarnath A, et al. Sparse-TPU: adapting systolic arrays for sparse matrices. In: Proceedings of the 34th ACM International Conference on Supercomputing, 2020. 1–12