Improving Data Reuse in NPU On-chip Memory with Interleaved Gradient Order for DNN Training

Jungwoo Kim
KAIST
Republic of Korea
jwkim@casys.kaist.ac.kr

Seonjin Na*
KAIST
Republic of Korea
seonjin.na@gatech.edu

Sanghyeon Lee
KAIST
Republic of Korea
leesh6796@casys.kaist.ac.kr

Sunho Lee
KAIST
Republic of Korea
myshlee417@casys.kaist.ac.kr

Jaehyuk Huh
KAIST
Republic of Korea
jhhuh@kaist.ac.kr

ABSTRACT

During training tasks for machine learning models with neural processing units (NPUs), the most time-consuming part is the backward pass, which incurs significant overheads due to off-chip memory accesses. For NPUs, to mitigate the long latency and limited bandwidth of such off-chip DRAM accesses, the software-managed on-chip scratchpad memory (SPM) plays a crucial role. As the backward pass computation must be optimized to improve the effectiveness of SPM, this study identifies a new data reuse pattern specific to the backward computation. The backward pass includes independent input and weight gradient computations sharing the same output gradient in each layer. Conventional sequential processing does not exploit the potential inter-operation data reuse opportunity within SPM. With this new opportunity of data reuse in the backward pass, this study proposes a novel data flow transformation scheme called *interleaved gradient order*, consisting of three techniques to enhance the utilization of NPU scratchpad memory. The first technique shuffles the input and weight gradient computations by interleaving two operations into a single fused operation to reduce redundant output gradient accesses. The second technique adjusts the tile access order for the interleaved gradient computations to maximize the potential data locality. However, since the best order is not fixed for all tensors, we propose a selection algorithm to find the most suitable order based on the tensor dimensions. The final technique further improves data reuse chances by using the best partitioning and mapping scheme for two gradient computations for single-core and multi-core NPUs. The simulation-based evaluation with single-core edge and server NPUs shows that the combined techniques can improve performance by 29.3% and 14.5% for edge and server NPUs respectively. Furthermore, with a quad-core server NPU, the proposed techniques reduce the execution time by 23.7%.

*Seonjin Na is currently with Georgia Institute of Technology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MICRO '23, October 28-November 1, 2023, Toronto, Canada
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614299

CCS CONCEPTS

• **Computer systems organization → Neural networks**; **Systolic arrays**.

KEYWORDS

DNN training, accelerators, on-chip memory, scheduling

ACM Reference Format:
Jungwoo Kim, Seonjin Na, Sanghyeon Lee, Sunho Lee, and Jaehyuk Huh. 2023. Improving Data Reuse in NPU On-chip Memory with Interleaved Gradient Order for DNN Training. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614299

1 INTRODUCTION

Training model parameters of DNNs is the most resource-intensive task in applying machine learning techniques to real-world problems. Such training tasks consume a large number of servers in data centers. Recently, edge devices are also used for training with techniques such as fine-tuning and federated learning for personalization and privacy protection [41, 59]. The training tasks require costly backward passes which compute gradients of all layers. The computation of gradients accounts for the majority of costs in model training.

Traditionally, GPUs have been the primary computing engines for training tasks, but recent advancements in NPUs (Neural Processing Units) allow energy-efficient acceleration of training in addition to inference tasks [16, 30, 38, 45, 60]. In NPUs, on-chip memory, or scratchpad memory (SPM) stores tensors which are staged to be fed to processing elements typically organized as a systolic array. Unlike the last-level cache of GPUs, SPM is solely managed by the software driving NPUs. As the size of model parameters has been increasing significantly [6, 54], the importance of SPM is growing as it is an essential component to reduce expensive external memory accesses.

Efficient processing of model training requires flexible utilization of SPM, especially when dealing with gradient tensors. Recent NPUs supporting ML training allow SPM to store various tensors flexibly to accommodate the computation flows of training tasks [30, 45]. However, training computation presents new opportunities to reduce redundant off-chip memory accesses by enhancing potential data reuse chances within SPM.

The essential step in the backward pass is to compute the input gradient and the weight gradient from the output gradient in each layer. During this process, the output gradient can be potentially reused as it is the common tensor used for both the input and weight gradient computations. Although the conventional sequential computation of the backward pass does not allow such data reuse, the inherent data reuse opportunities can lead to possible optimizations of SPM.

Focusing on the backward pass of training computation with NPUs, this paper proposes a new dataflow transformation scheme called *interleaved gradient order*, consisting of three techniques to improve data reuse in SPM. The techniques exploit the potential for reuse of the output gradient by interleaving and reordering operations for the input and weight gradient computations. In typical DNNs, tensors, which are much larger than SPM, are decomposed into tile granularity, computing tiled data with asynchronous double buffering overlapped with computation. Our techniques refactor the computation of the backward pass to maximize tile reuses in SPM and reduce unnecessary tiles stored in SPM. Our scheme consists of three techniques.

The first technique interleaves the computation of weight gradients and input gradients to create data reuse of output gradients within SPM. Traditional backward pass flows process the two computation sequentially, losing potential data reuses within SPM. In our proposed interleaving technique, the two main backward computations are fused and interleaved at the tile granularity to reuse the common data stored in SPM.

Second, with the interleaving, tile access orders introduce new trade-offs in data reuse within SPM, depending on the dimensions of gradient tensors. Our second technique identifies these trade-offs and proposes a new selection algorithm to improve data reuse opportunities in SPM. The second rearrangement step transforms the code to reorder tile accesses to maximize the chance that two gradient computations use the same output gradient tiles.

The third technique determines the best data partitioning and mapping scheme which can further enhance data reuses enabled by the prior two techniques. For a single-core NPU, when dimensions of matrices are skewed in one direction and conventional data partitioning on a batch basis is used, the data reuse chances can be reduced. By decomposing them into partitions based on different dimensions and changing their mappings, the data reuse chances are improved. For multi-core NPUs, each core can process different partitions to be aggregated to the final outcome in a reuse-enhancing manner.

We evaluate the proposed dataflow transformation techniques using two NPU configurations: an edge-class NPU targeting retraining for personalization and federated learning, and a server-class NPU intended for traditional large-scale training tasks. Our simulation-based evaluation shows that the proposed techniques improve the average performance by 29.3% and 14.5% for the single-core edge NPU and the single-core server NPU, respectively. Moreover, in the quad-core server NPU runs, the proposed techniques improve the performance by 23.7% on average.

This study emphasizes the significance of optimizing SPM for the backward pass of training computation, and proposes a new technique aimed at improving data reuse. In contrast to previous data flow scheduling techniques which primarily reordered operations or refactored code within individual operations, the key distinction lies in the identification of redundant memory accesses *across* the two main gradient computations and the fusion of these two operations through interleaving and reordering. The new contributions of the paper are as follows:

- It identifies a new opportunity for data reuse in SPM during the backward pass. By interleaving gradient computation effectively, it can exploit a new type of data locality, which is absent in traditional sequential flows.
- It finds the trade-offs of tile access orders within the interleaved computation, introducing a new selection algorithm to maximize data reuse in SPM.
- It proposes a novel computation decomposition and mapping technique for interleaved gradient computations to maximize data reuse in the SPM for single-core and multi-core NPU architectures.

Forward Pass (Training and Inference)	
L_i	The i-th layer
X_i	Input feature map of L_i
W_i	Weight of L_i
Y_i	Output feature map of L_i
Backward Pass (Training Only)	
$\mathcal{L}$	Loss
dX_i	Partial derivative of $\mathcal{L}$ with respect to X_i
dW_i	Partial derivative of $\mathcal{L}$ with respect to W_i
dY_i	Partial derivative of $\mathcal{L}$ with respect to Y_i

Table 1: Symbols in forward and backward passes

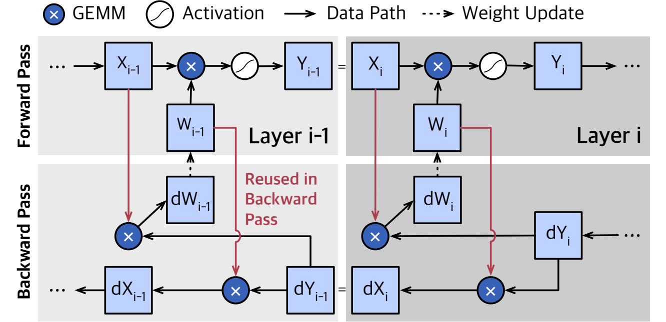

Figure 1: The computational flow of the forward and backward passes in training.

2 BACKGROUND

2.1 Computation for Training

Unlike inference in which data is propagated through layers only in a forward manner, in training, the backward pass calculation in the opposite direction is additionally required. The training procedure consists of two steps: 1) forward pass followed by 2) backward pass. We use the symbols in Table 1 in the rest of the paper.

Forward pass: In the forward pass of the i-th layer, the output Y_i is calculated from the input X_i and weight W_i. CPU pre-processes the naïve input data such as images and natural language words into a matrix or vector form as X_1. With W_1, Y_1 is calculated by mathematical operation such as general matrix to matrix multiplication (GEMM) or convolution. In a sequential DNN, the output of

the i-th layer is used as the input feature map of the (i+1)-th layer, and thus Y_1 is used as X_2. After repeating the steps across all the layers, the output of the last layer, Y_{last}, is a final prediction result to be compared to the ground truth.

Backward pass: To quantify how close the prediction Y_{last} obtained through the forward pass is to the ground truth Y_{truth}, the loss $\mathcal{L}$ is computed by the loss function such as categorical cross entropy [63]. The loss $\mathcal{L}$ is used to calculate the partial derivative of loss with respect to W_i ($\frac{\partial \mathcal{L}}{\partial W_i}$), which is required to update W_i for training the model. In this paper, we use shortened notations: input gradient (dX_i), weight gradient (dW_i), and output gradient (dY_i), which are partial derivatives of loss with respect to the X_i, W_i, and Y_i. They correspond to the following full notations: $\frac{\partial \mathcal{L}}{\partial X_i}$, $\frac{\partial \mathcal{L}}{\partial W_i}$, and $\frac{\partial \mathcal{L}}{\partial Y_i}$.

$$dX_i = \frac{\partial \mathcal{L}}{\partial X_i} = \frac{\partial \mathcal{L}}{\partial Y_i}\frac{\partial Y_i}{\partial X_i} = dY_i \times {W_i}^T \quad (1)$$

$$dW_i = \frac{\partial \mathcal{L}}{\partial W_i} = \frac{\partial Y_i}{\partial W_i}\frac{\partial \mathcal{L}}{\partial Y_i} = {X_i}^T \times dY_i \quad (2)$$

Using the output gradient dY_i, the input feature map X_i, and the parameter W_i in the i-th layer, the input gradient dX_i and the weight gradient dW_i are computed. Each gradient can be decomposed to a product of dY_i and the partial derivatives of X ($\frac{\partial Y_i}{\partial X_i}$) and W ($\frac{\partial Y_i}{\partial W_i}$) by the chain rule. Because of the linear characteristics of GEMM, each partial derivative of output can be substituted to W_i^T and X_i^T, respectively as shown in Eq (1) and (2). Therefore, the computation of backward pass can be reduced to GEMM of dY_i, X_i^T, and W_i^T. Although the bias gradient should be calculated to update the bias tensor, extra computation is not needed as the bias gradient has the same value as dY_i. A key property we will exploit in this study is the two independent computations of dX_i and dW_i, both of which use dY_i as an operand.

Figure 1 represents the entire computation flow including the forward and backward passes. As both the forward pass and backward pass must be performed in training, there are three computations in training : calculating Y_i in the forward pass and computing dX_i, and dW_i in the backward pass. Convolution layers can be performed by GEMM after the *im2col* operation [27], and GEMM occupies most of the execution time in the forward and backward passes [40, 52]. If the amount of GEMM in each computation of Y_i, dX_i, and dW_i is equal, the amount of training computation is at least three times more than that of inference. Moreover, since computed dX_i and dW_i must be stored in memory, the memory footprint is also much larger in training than inference.

Memory reuse and computation parallelism are considered as the most crucial factors for improving the throughput and power efficiency in training. On the other hand, inference requires rigid latency constraints in addition to throughput. In this paper, we focus on the training process and our approach aims to increase data reuse within on-chip memory to reduce off-chip memory accesses and to improve operational parallelism.

2.2 Accelerator for Training Neural Networks

Baseline architecture: Among several different types of training accelerators, we use a general systolic array-based accelerator in this study as it is the most common organization [19, 30, 45]. The

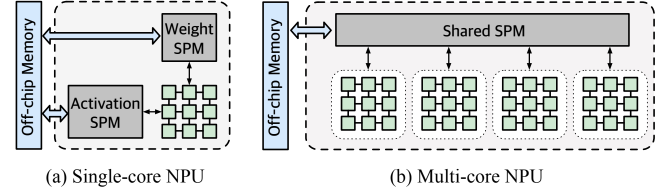

Figure 2: Architecture of a single-core accelerator and a multi-array accelerator.

data in the off-chip memory are loaded to the on-chip *scratchpad memory* (SPM) controlled by the software. Operands to the systolic array are fed from SPM. To hide the off-chip memory access latency, SPM uses the *double buffering* technique in which SPM is divided into two parts, with each half being used alternately to fill data from the off-chip memory into SPM.

The baseline architecture in this study adopts a unified SPM which can store both activations and weights, because flexibly using SPM is required for complex operations in training. Such a flexible use of SPM has been adopted in commercial NPUs [30, 45]. Unlike inference with two operands of input activations and weights, the training process involves gradient tensors as well as weights. Therefore, SPM must be able to be allocated differently for tensors for the forward and backward passes.

Edge-level training: As demand of training is not limited to the server-scale accelerator, there have been many recent improvements for DNN accelerators to support training in edge devices [1, 13, 32, 38, 48, 58, 60]. Sending private data to cloud servers can incur security vulnerabilities [38, 60]. To avoid these drawbacks, each edge device produces the model updates itself, and the cloud server aggregates model updates from edge devices (i.e., federated learning). Such federated learning is preferred when sensitive data is produced in edge devices such as medical services [41].

Multi-core NPU architecture: Recent scaling efforts for NPUs have been adopting multi-core approaches for NPUs. To efficiently support computations of tensors with various sizes, simply increasing a single systolic array can incur under-utilization of execution resources for small models. To better address a wide range of different tensor sizes, an NPU contains multiple cores which can be flexibly utilized to process different tensor sizes. Figure 2 shows the overview of single and multi-core accelerator architectures [24, 31]. In this paper, SPM is shared by all cores and each core has a systolic array.

2.3 DNN Scheduling Space

Since each DNN accelerator exhibits a distinct configuration such as different types of PE connections or various sizes of SPMs, there have been numerous studies about DNN scheduling [7, 9, 11, 19, 33, 36, 50, 61]

Tiling: To alleviate frequent data transfer between SPM and the off-chip memory in NPUs, a commonly employed strategy is the adoption of tiled linear operation which is tightly coupled with spatial dataflow [8, 19, 31, 43]. The dimension of the tiles fetched from off-chip memory impacts the degree of data reuse between SPM and the off-chip memory, as well as overall utilization. To take advantage of this locality, several studies have focused on determining an appropriate tile dimension through multi-level tiling and genetic

Prior Studies	Reuse in Independent Operations	Training	Tiling
Maestro[36]	✘	✘	✓
MARVEL[7]	✘	✘	✓
Timeloop[50]	✘	✘	✓
Interstellar[61]	✘	✘	✓
Ours	✓	✓	✓

Table 2: Prior studies for DNN scheduling space.

algorithms [33, 43], and investigating partitioning schemes that enable an efficient parallel execution [18]. Other prior work have provided novel compilation methods that facilitate the exploring tile dimensions [9, 20, 53].

Dataflow: While tiling reduces the communication between SPM and off-chip memory, mapping of spatial dataflow determines how well spatial architecture utilizes the fetched tensors from SPM during computation. To increase data reuse by considering the DNN model and accelerator's hardware together, several studies have proposed novel spatial architectures [11, 17, 31]. In addition, dataflow includes a massive DNN scheduling space, such as parallelization strategies for distributing operations to the compute units and changing the sequence of operations by loop reordering and unrolling. Consequently, there have been numerous frameworks for efficient dataflow mappings [7, 36, 50, 61]. Other prior work has provided a novel spatial accelerator that enables runtime-level dataflow selection depending on input matrices[37].

Operator Fusion: Graph-level operator fusion, which combines multiple operator together without unnecessary data transfer, has been studied for its performance improvement [2, 15, 20, 34, 44, 46]. The input of the fused operator is loaded only once from the off-chip memory, not for each operator. Reducing data transfers for intermediate tensor increases the operational intensity of input tensors, mitigating the memory bandwidth bottleneck. Moreover, it is critical to reduce the number of kernels through operator fusion, because kernel launch overheads in GPU have a significant impact on the overall performance [46, 47].

Comparison with previous studies: This study differs from the prior dataflow and tiling optimization studies, as it identifies a new type of data reuse for the training process which is not discussed in the prior work in Table 2. The prior work focus on intra-operation data reuse, concentrating on an operation that can be expressed as a single nested loop such as convolution or GEMM. On the other hand, multiple independent operations cannot be represented as a single nested loop, which is required for loop reordering or tiling. Therefore, we propose to consider a new inter-operation fusion in which there is an opportunity to reuse data across independent operations. Although there have been studies proposing inter-operation fusion for calculating attention in transformer models [15, 34], data reuse for inter-operation in convolution or GEMM, which are used in models across various fields, are not investigated by prior studies.

This study improves dataflow to maximize locality in SPM in the baseline architecture and to enhance the data reuse by interleaving and re-ordering operations for two gradient computations. From the perspective of operation fusion, this study opens a new possibility of fusing two independent gradient operations, unlike the prior fusion of dependent sequential steps. To demonstrate the importance of this study in DNN scheduling space, our baseline includes the intra-operation optimization such as tiling of previous studies in Table 2.

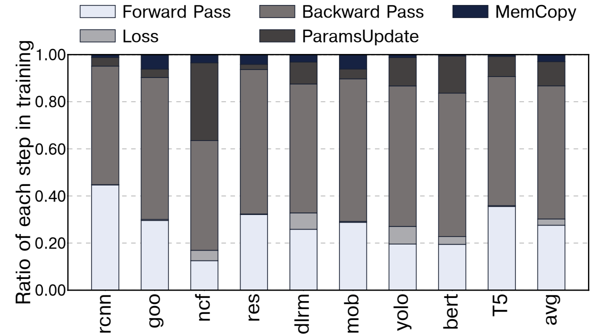

Figure 3: The execution time of each step with NVIDIA A100 GPU, normalized to the total training time.

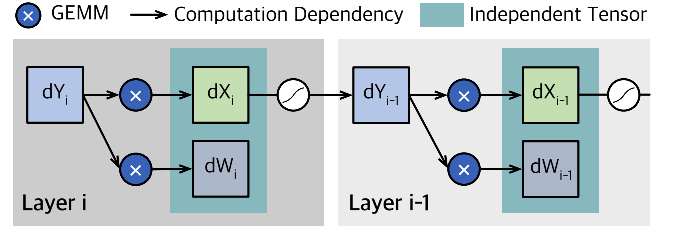

Figure 4: Dependency between computations and tensors in the backward pass. Output gradient (dY_i) is required twice to compute input gradient (dX_i) and weight gradient (dW_i).

3 MOTIVATION

3.1 Decomposition of DNN Training

To quantify the importance of the backward pass, we break down the total training time for each step in DNN training on NVIDIA A100 GPU with PyTorch [51]. The decomposition shows the times for the forward and backward passes. In addition, it also presents the decomposed times for data transfer times between CPU and GPU (MemCopy), loss computation, and parameter update. Figure 3 shows the average ratio of each step out of the total training time for 90 epochs using 256 batch sizes in a single A100 GPU. The forward and backward passes account for 27.6% and 56.5% of the total training time on average. As the forward and backward passes take more than 84% of the total time, the remaining three steps (memory access, loss function, and parameters update) account for small portions with 3.0%, 2.6%, and 10.3%, respectively. The results reaffirm that the backward pass is the most critical step in the training procedures, with 56.5% of the total execution time.

3.2 Redundant Data Accesses in Backward Pass

To improve data reuse in the backward pass, this subsection analyzes the required computations and redundant data accesses, which can be potentially eliminated.

Data access redundancy in backward pass: Figure 4 shows the dependency of tensor computations in the backward pass, in which layers are computed in the reversed order of the forward pass. X_i and W_i are not shown in Figure 4 to reduce the complexity of the

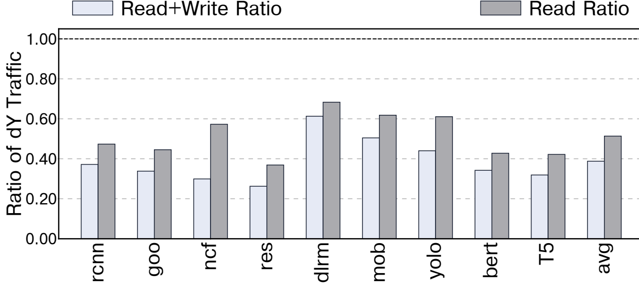

Figure 5: The proportion of output gradient (dY_i) traffic compared to the total amount of traffic in the backward pass.

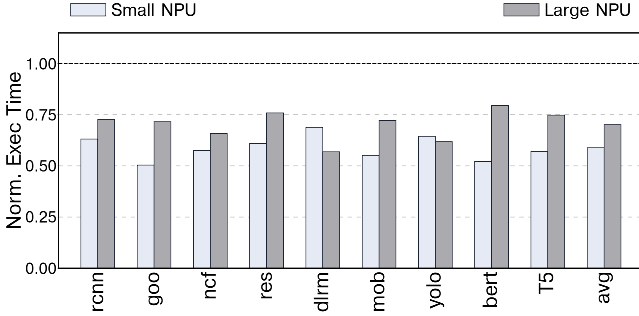

Figure 6: Execution time for training DNN models when the entire DY is reused.

figure. In the (i+1)-th layer, dX_{i+1} and dW_{i+1} are computed by using the same dY_{i+1} as an operand. The computed input gradient dX_{i+1} is required to compute weight gradient dW_i and input gradient dX_i in the layer i (i.e., $dX_{i+1} == dY_i$). In addition, an activation function is applied to dX_{i+1} before dX_{i+1} is used as dY_i in the layer i.

Since there is no dependency between the computations of dX_i and dW_i within the i-th layer, computing two gradients can be conducted in parallel. However, dX_i and dW_i are sequentially computed in the conventional training accelerators such TPUv3 with XLA. In the two gradient computations for dX_i and dW_i, a common operand (dY_i) is fetched from the external memory twice, as it is required for both computations. Such redundant accesses to dY_i in the current framework are caused by the serialized computation of dX_i and dW_i. As two operations are independent, it is possible to reorder or even fuse two operations in an interleaved manner. In addition, redundant accesses to the output gradient (dY_i) can be potentially reduced by such fusion.

The data traffic for output gradient (dY_i): Figure 5 shows the ratios of dY_i traffic in the backward pass. They are measured by the simulated `large NPU` configuration as described in Section 6.1. Among the five tensors required in the backward pass, operand tensors (X_i, W_i, and dY_i) are transferred from the off-chip memory to SPM for performing the backward pass of the i-th layer. On the other hand, result tensors (dX_i and dW_i) are transferred from SPM to the off-chip memory. Therefore, we inspect the amount of dY_i considering direction of data traffic. *Read+Write Ratio* in Figure 5 represents the ratio of dY_i traffic compared to all read and write data, which is 39.0% of total data traffic on average. Furthermore, as shown in *Read Ratio* in Figure 5, dY_i occupies 51.4% of read data on average. Especially, dY_i accounts for 68.3% of the read data traffic in `dlrm`. As shown in the result, a high percentage of dY_i shows opportunities to improve performance by reducing the dY_i traffic.

3.3 Performance Potential of Reusing dY

In this section, we quantify the performance potential of eliminating redundant reads for the output gradient (dY) in each layer during the backward pass. As shown in Section 3.2, a significant portion of memory reads during the backward pass is consumed for the redundant reads of dY. Consequently, we eliminate one of the two accesses for dY within our simulation setup to inspect the performance potential when the entire dY is re-used in SPM.

Without loss of generality, we assume dX is calculated before dW as conventional accelerators. For the computation of dX, dY tiles are simply read along with the other operands. However, for the second dW computation, we eliminate dY reads, assuming the data are hyperthetically available without any external memory access. Although this setup does not accurately include other positive effects from our proposal in Section 4, it shows the potential performance impact of eliminating one of the two redundant reads of dY.

For the evaluation, we consider a single-core NPU with two configurations (`large NPU` and `small NPU`). The detailed methodology of NPU configurations and the simulator is described in Section 6.1. It is important to note that this outcome does not offer the optimal improvement through dY reuse, as it simply eliminates the second access. The initial access for dX computation involves bursty accesses that can incur a substantial performance cost. By striking a balance between computation and memory access, further improvements in performance can be achieved.

Figure 6 illustrates the normalized execution time for training each DNN model, assuming that dY is read only once for computing both dX and dW. `large NPU` and `small NPU` are simulated as described in 6.1. On average, the speedup against the baseline is 1.43x in *Large NPU* and 1.70x in *Small NPU*. In *Small NPU* configuration, where the SPM size is limited, there are more opportunities to enhance performance by reusing all instances of dY.

4 TRANSFORMATION FOR DATA REUSE

4.1 Overview

As described in Section 2.1, during backward passes for layers involving trainable parameters, the weight gradient (dW_i) and input gradient (dX_i) need to be computed. Since the same output gradient (dY_i) is served as the operand for both gradient computations, this paper introduces a novel code transformation technique aimed at removing unnecessary memory accesses for the output gradient.

Figure 7 presents three software transformation steps aimed at enhancing backpropagation: (1) the gradient interleaving step, (2) the gradient rearranging step, and (3) the inter-core distribution step. To exploit the data reuse potential of dY in SPM during the computation of both dX and dW, the gradient interleaving step combines two computations and interleaves tiled operations for them (Section 4.2). However, the effectiveness of tile reuses can vary depending on the operand dimensions, if the conventional tile computation order is applied. To further optimize dY reuse, the second gradient rearranging step identifies the optimal tile

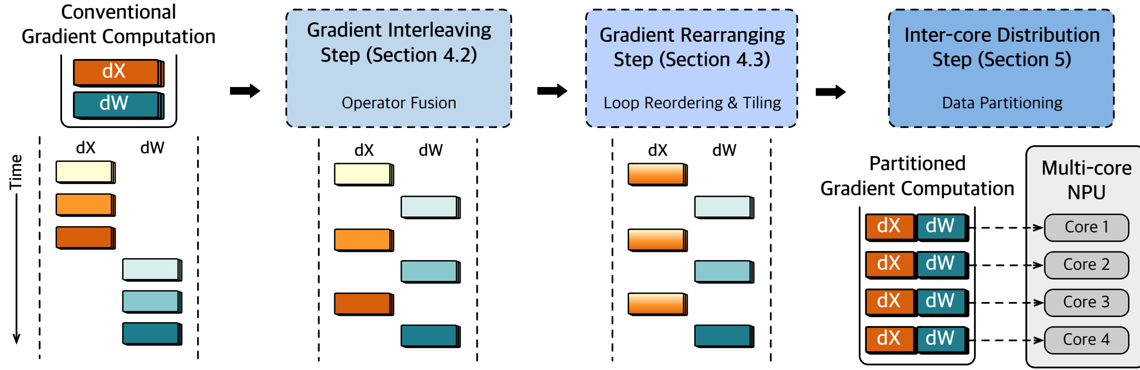

Figure 7: The overview of the proposed backpropagation transformation framework.

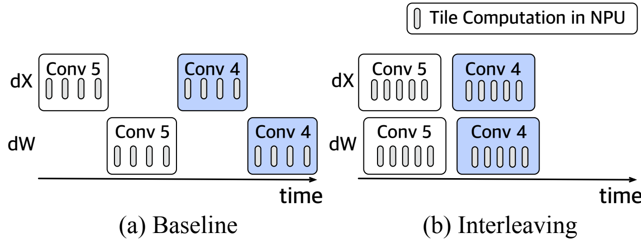

Figure 8: The tile access order of the $i = 4, 5$-th layers depending on whether interleaving input gradient (dX_i) and weight gradient (dW_i) computations or not.

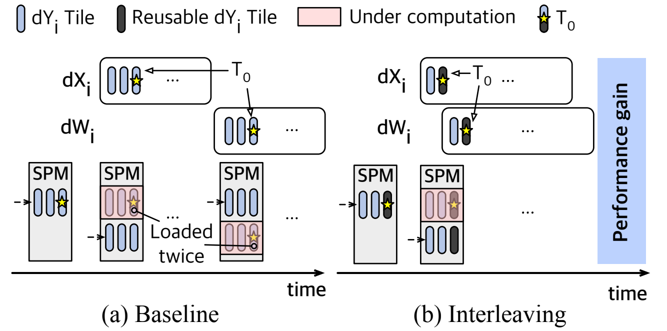

Figure 9: Reusing the output gradient (dY_i) in SPM via *interleaving* technique.

computation order and reorganizes the interleaved dX and dW computations (Section 4.3). For multi-core NPUs, operands are decomposed and assigned to different cores to maximize dY reuse. In the inter-core distribution step, the optimal method for operand decomposition is determined, and each segment is allocated to an NPU core.

The proposed code transformation can be seamlessly integrated into the existing compiler framework, requiring no modifications to the hardware design of NPUs. It is worth noting that these three steps must be applied in sequence, as the second and third steps depend on the interleaved computation of dX and dW, and the third step relies on the results from the first two steps. While the second gradient rearranging step shares some similarities with loop reordering or tiling seen in prior research [33, 43], the reorganization for interleaved dX and dW has not been previously explored.

4.2 Interleaving dX and dW

The output gradient of the i-th layer, dY_i, is necessary for computing both the input gradient, dX_i, and the weight gradient, dW_i. Since the size of the SPM is often inadequate to store all the data required for calculating dX_i or dW_i, each step is divided into multiple tiled GEMM computations to fit in the limited SPM size. The key concept of the proposed *interleaving* technique is to fuse two computations that were conventionally performed sequentially in NPUs and to adjust the computation order to optimize the data reuse of the output gradient, dY_i.

Baseline computation for dX and dW: Figure 8 (a) depicts the baseline computation sequence of the backward pass utilized in conventional training accelerators like TPUv3 with XLA. In the baseline approach, dX_i and dW_i are calculated in a sequential manner. While Figure 8 (a) illustrates tiled operations for sequentially computed dX_i and dW_i, the order of computing dX_i and dW_i within the same layer can be swapped, as there are no dependencies dictating the order.

Interleaved computation for dX and dW: In contrast to the baseline approach, the proposed technique interleaves the computation of dX with that of dW, allowing the shared dY tile to be reused for both dX and dW while it remains in SPM. Figure 8 (b) illustrates the *interleaving* transformation, which combines computations for dX_i and dW_i on a tile-by-tile basis. Since there is no dependency between dX and dW computations, the input and weight gradients in the modified code are identical to those in the previous sequential execution. Additionally, although the transformed code effectively eliminates redundant accesses to dY tiles, it does not introduce any extra computations compared to the conventional approach.

Performance potential of the interleaved computation: Since the output gradient dY_i is utilized for the computation of both dX_i and dW_i, dY_i is often transferred to SPM twice in the baseline design due to the prior loaded tile being evicted. Figure 9 (a) illustrates the duplicated loading of a tile T_0 in dY_i to compute dX_i and dW_i, respectively. In this context, the tiles marked with a yellow star in

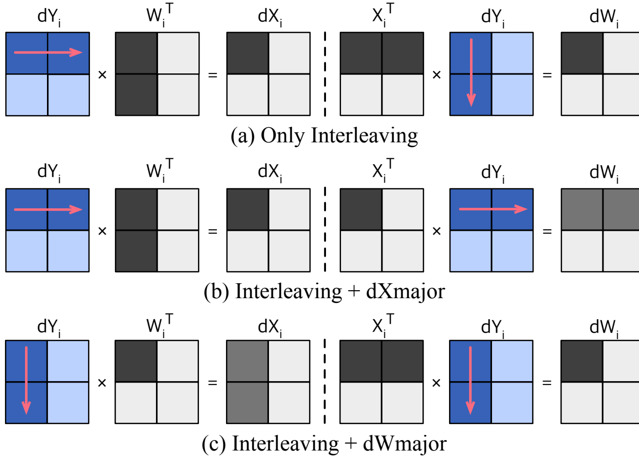

Figure 10: Three tile access orders with *Interleaving*. *Only Interleaving* follows the traditional GEMM based access order. *Interleaving + dXmajor* and *Interleaving + dWmajor* presents the row-major and column-major orders.

Figure 9 (a) represent the same tile T_0 in off-chip memory. Following the computation of dX_i using T_0, there can be numerous tiled computations before T_0 is needed again for calculating dW_i. Consequently, T_0 in dY_i is already evicted to off-chip memory before the subsequent computation (i.e., dW_i computation in this example) takes place. In general, duplicated memory traffic arises when the distance between the dX_i and dW_i calculations exceeds the number of tiled computations that can be loaded in half of the SPM. Given that the model size for training often exceeds the capacity of the SPM, a significant portion of dY_i is not reused, resulting in additional traffic.

On the other hand, the interleaving technique reduces memory traffic and enhances performance, as depicted in Figure 9 (b). The tiled computations of dX_i and dW_i are interleaved one by one, and T_0 marked with yellow stars can be reused with just one loading. Since T_0 does not necessarily need to be a specific tile of dY_i, every tile in dY_i has the potential to be reused by interleaving the tiled computations of dX_i and dW_i. Consequently, interleaving dX_i and dW_i computations significantly reduces data traffic and boosts the utilization of SPM. It is worth noting that such interleaving can be utilized for layers involving trainable parameters regardless of the type of neural networks, including transformer models.

4.3 Optimal Tile Order

While computing dX_i and dW_i in the interleaved order allows the reuse of dY_i, the *interleaving* method does not significantly improve performance in certain layers that contain non-square tensors. When one dimension (either row or column) is much longer than the other, the performance improvement through simple interleaving remains minimal. Figure 10 (a) illustrates why the performance improvement is restricted when only the interleaving technique is employed. The access patterns for dY_i differ between dX_i and dW_i computations. When computing dX_i, the access of dY_i follows a row-major access order because dY_i is multiplied with W_i^T. On the other hand, when computing dW_i, dY_i is accessed in a column-major order since dW_i is calculated by multiplying X_i^T and dY_i.

Algorithm 1: Algorithm for selecting memory access order among *Interleaving*, *Interleaving+dXmajor*, and *Interleaving+dWmajor*.

1 GEMM in forward pass is $X_i(M,K) \times W_i(K,N) \rightarrow Y_i(M,N)$
2 **if** ALMOSTSQUARECOMPUTATION() **then**
3 | Use *Interleaving*
4 **else if** $K > N$ ***and*** $K > M$ **then**
5 | Use *Interleaving+dWmajor*
6 **else**
7 | Use *Interleaving+dXmajor*

To maximize the performance improvement when utilizing the interleaving technique, we categorize the memory access orders for dY_i into three main groups, as shown in Figure 10: (a) Traditional access order which employs row-major accesses for dX_i and column-major accesses for dW_i, (b) row-major accesses for both dX_i and dW_i, referred to as *dXmajor*, and (c) column-major accesses for both dX_i and dW_i, called *dWmajor*. The appropriate memory access orders are selected based on algorithm 1. As shown in Figure 10 (a), the traditional access order does not fully capitalize on the reuse of dY_i as the required dY_i tiles differ between computing dX_i and dW_i. This results in inefficient SPM utilization, as redundant accesses to the same dY_i tile are needed when computing both dX_i and dW_i.

To maximize the data reuse of dY_i at the expense of dX_i and dW_i reuse, we apply the memory access orders called *dXmajor* and *dWmajor*, as shown in Figure 10 (b) and (c) respectively. The key concept behind *dXmajor* is that when computing dW_i, the access of dY_i follows a row-major order, just like in dX_i computation. Similarly, in the case of *dWmajor*, the access of dY_i in dX_i computation is in column-major order, similar to dW_i. By appropriately altering the memory access order, the data reuse of dY_i can be enhanced, leading to improved performance.

However, intermediate results can be generated in *dWmajor* and *dXmajor* due to the changed computation order. We assume these intermediate results are stored in the SPM to the extent possible. If not, they are stored in the off-chip memory, resulting in an additional memory traffic that is included in our performance measurement. Hence, there is no extra hardware overhead resulting from *dWmajor* and *dXmajor*. Some layers might perform better without using *dWmajor* or *dXmajor* due to the added memory traffic.

Selection algorithm: Three memory access orders based on the computation orders of dX_i and dW_i are depicted in Figure 10: *Only interleaving*, *Interleaving+dXmajor*, and *Interleaving+dWmajor*. Since the computation orders for dX_i and dW_i are predetermined according to the sizes of tensors, the optimal memory access order for each layer can be determined statically before DNN training on NPUs. Since both *Interleaving+dXmajor* and *Interleaving+dWmajor* aim to fully utilize dY_i, the choice between using *dXmajor* or *dWmajor* is made based on which of the dX_i or dW_i tensor benefits more from data reuse through a change in the order of memory accesses.

As shown in Figure 10, *Interleaving+dXmajor* and *Interleaving+dWmajor* generate additional non-reused tiles of dW_i in *Interleaving+dXmajor* and dX_i in *Interleaving+dWmajor*. Therefore, for some non-square computation, we roughly opt for *Interleaving+dXmajor* when the size of dX_i is larger than the size of dW_i, and choose *Interleaving+dWmajor* otherwise.

Algorithm 1 represents our straightforward but fairly accurate method for determining the optimal memory access order among *Only interleaving*, *Interleaving+dXmajor* and *Interleaving+dWmajor*. We select the appropriate memory access order based on the shapes of tensors for computing dX_i and dW_i. In Algorithm 1, the tensor dimensions are denoted as M and K for dX_i, and K and N for dW_i in GEMM operations during backward pass. When the shapes of the five tensors (X_i, W_i, dX_i, dW_i, and dY_i) are identified as nearly square matrices using AlmostSquareComputation() function, we choose the *only interleaving* policy since it effectively exploits data reuse of dX_i and dW_i (`line 2-3`).

We classify a tensor as nearly square when the largest dimension is less than four times the smallest dimension. As a result, AlmostSquareComputation() returns true when all five tensors are nearly square (i.e., $\frac{Max(M,N,K)}{Min(M,N,K)} < 4$), and false otherwise. For non-square tensors, where the one dimension greatly exceeds the other dimension, we determine the memory access order based on the dimensions of dW_i. We apply the *Interleaving+dWmajor* access order when the column dimension of dW_i significantly surpasses the row dimension of dW_i (`line 4-5`). Conversely, we employ the *Interleaving+dXmajor* access order in other cases(`line 6-7`).

We define *rearrangement* as the combination of (1) the interleaved gradient computations discussed in Section 4.2 and (2) the access order change for tensor computations according to the prediction from Algorithm 1. When measuring the execution time for both the forward and backward passes, Algorithm 1 can improve performance by 23.8% and 10.9% for edge and server NPUs compared to the baseline when a single systolic array is used. If the best memory access order is chosen for each layer by actually running all the three cases, the ideal performance improvement is increased to 25.1% and 12.4% for edge and server NPUs. Although there is a minor gap between the algorithm-based one and the ideal one, Algorithm 1 selects the best one mostly. Since Algorithm 1 only requires the tensor dimensions and incurs a constant execution time to determine the optimal access order per each layer, it can be applied statically. Furthermore, since *rearrangement* can only be applied to the backward pass, it is anticipated that the performance improvement would be even greater if optimization techniques for the forward pass [18, 62] are employed in conjunction with our *rearrangement* technique.

5 PARTITIONING FOR REARRANGED GRADIENT ORDER

The performance of the software transformation discussed in Section 4 significantly depends on the dimensions of the GEMM. Since a single GEMM is usually required to be divided into smaller partitioned GEMMs, we propose new data partitioning schemes to change the dimension of partitioned GEMM to better suit *interleaving* and *rearrangement* techniques discussed in Section 4.

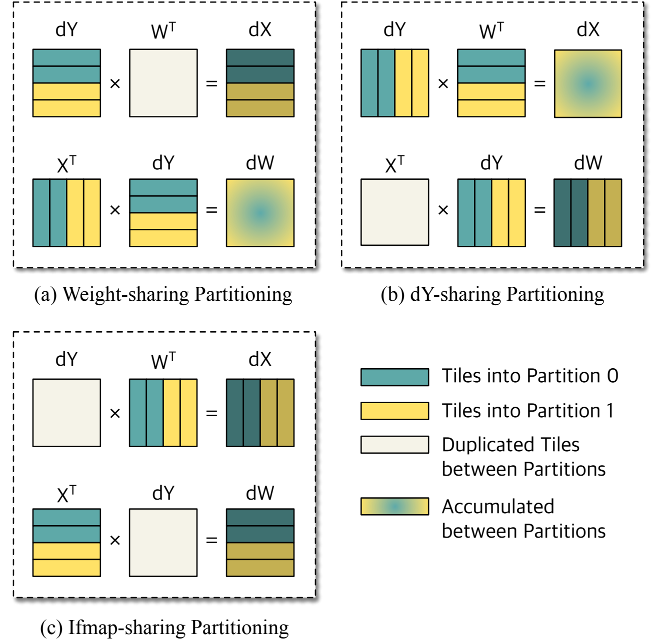

Figure 11: Partitioning schemes of interleaved gradient order.

It is common for GEMM to be partitioned on a batch basis, which is the dimension M in GEMM, where X = (M, K), W = (K, N), and Y = (M, N). When this approach is applied to the backward pass, dY and X^T are split for computing dX and dW based on batches within each partition. In Figure 11 (a), weight-sharing partitioning, in which GEMM is partitioned on a batch basis for interleaved gradient operations, is illustrated. For the computation of dX, each dY batch can be allocated to different partitions, and each partition is related to the computation of different dX with independent dY batches and shared W^T. However, dY serves as the right-hand-side operand in the computation of dW, where X^T is the left-hand-side operand. Therefore, partitioning X on a batch basis (i.e., splitting X^T column-wise in Figure 11 (a)) prevents a single partition from performing all the necessary calculations for a specific portion of dW. Instead, it necessitates additional overhead to accumulate intermediate results obtained from multiple partitions and calculate the average. Such partitioning may not be optimal within interleaved gradient operations as it can introduce overhead due to additional off-chip memory access.

Therefore, when *interleaving* and *rearrangement* techniques are utilized, partitioning GEMM in either N or K dimension rather than M dimension could be more efficient in some layers. In addition to the data parallelism of the batch unit in conventional NPUs, alternative data partitioning methods can be explored for the rearranged gradient computations. There are diverse approaches to partition interleaved gradient operations across multiple GEMM partitions, and Figure 11 (b) and (c) show two examples of such data partitioning. To change the dimension M, N, and K dimension of GEMM, we propose to partition the GEMM operation based on dimensions N and K rather than M, which would change the dimension M, N, and K of a partition.

In dY-sharing partitioning illustrated in Figure 11 (b), dY is partitioned based on its columns (dimension N), which are orthogonal to the dimension of batches (dimension M), and all X batches are duplicated in all partitions. Therefore, each partition is required to calculate an independent portion of dW, but the computation of dX requires the accumulation. On the other hand, in ifmap-sharing partitioning depicted in Figure 11 (c), all dY is duplicated in all partitions, and X is split based on the its columns (dimension K) orthogonal to batches (dimension M). Therefore, no accumulation is required for the computation of two gradients.

The choice of data partitioning scheme impacts the dimension which is split for data parallelism. Each of (a), (b), and (c) in Figure 11 respectively splits the dimension M, N, and K, where $X = (M, K)$, $W = (K, N)$, and $Y = (M, N)$. Furthermore, the tensor shared by all partitions is also different for each data partitioning scheme in Figure 11: W is shared in (a), X is shared in (b), and dY is shared in (c). These variations in partitioning can impact overall performance depending on the dimensions of tensors, much like how the optimal memory access order within a single core of the NPU changes according to the layer's dimensions, as discussed in Section 4.3. For example, when the dimension M is significant in GEMM, partitioning based on the dimension M as illustrated in Figure 11 (a) could be an efficient way to partition data. However, if the dimension M is smaller than the width of a systolic array, splitting M does not improve performance at all, and other data partitioning schemes should be considered.

The partitioned GEMMs obtained through the proposed data partitioning schemes are processed one partition at a time on a single-core NPU over time. Additionally, in multi-core NPUs, multiple partitions can be distributed across the cores and processed concurrently. Therefore, the partitioning scheme depicted in Figure 11 can further improve the rearranged gradient order in both single-core and multi-core NPUs.

Selection mechanism: Given a variety of data partitioning schemes available, it is crucial to determine the optimal data partitioning scheme for each layer. Therefore, we employ the K-nearest neighbors (KNN) algorithm to identify an efficient data partitioning scheme for each layer. We randomly select 80% of our workloads to form training set, and the objective of KNN algorithm is to predict the optimal data partitioning scheme for each layer in test set, which comprises the remaining 20%. For training set, we simulate three data partitioning schemes described in Figure 11, all utilizing interleaved gradient order. Subsequently, we empirically determine the most efficient data partitioning scheme among the three schemes for each layer in the training set.

Using the dimensions of dX, dW, and dY as features in the KNN algorithm, we predict the optimal partitioning scheme for the test set. We evaluate the accuracy of the KNN algorithm through 1000 repetitions, yielding an average accuracy of 91%. Moreover, any performance degradation stemming from incorrect predictions in the KNN algorithm is not significant. In a dual-core NPU configuration, the performance improvement achieved from data partitioning in an ideal run where all predictions are accurate is 22.4%. Using the KNN algorithm slightly reduces the performance improvement to 21.5% compared to the baseline. As long as the dimensions of dX, dW, and dY can be determined, the appropriate partitioning scheme can be predicted for new models using the KNN algorithm.

Small NPU [39]	
Compute Unit	1 x (45 x 45 PE)
DRAM Bandwidth	22 GB/s
Frequency	1 GHz
Scratchpad Memory	1MB
Large NPU [4]	
Compute Unit	1~8 x (128 x 128 PE)
DRAM Bandwidth	150GB/s per core
Frequency	1050 MHz
Scratchpad Memory	8MB per core

Table 3: Simulated NPU configurations

DNN Model	Abbr	Parameters
FasterRCNN	`rcnn`	19M
Googlenet	`goo`	62M
NCF-recommendation	`ncf`	3B
Resnet50	`res`	25M
DLRM	`dlrm`	25B
Mobilenet	`mob`	13M
YOLO (v5/v2-tiny)	`yolo`	47M/11M
BERT (large/tiny)	`bert`	340M/14M
T5 (large/small)	`T5`	770M/60M

Table 4: Evaluated DNN models

6 EVALUATION

6.1 Methodology

Simulation environment: We develop a cycle-level simulator for DNN training on NPUs, building upon SCALE-Sim [55]. In the simulator, we assume that all convolution layer computations are transformed into GEMM operations by applying *im2col*, and NPUs employ double buffering mechanism to overlap tensor computation with data transfer. Table 3 provides a description of the evaluated NPU configurations. We use `small NPU` and `large NPU` configurations based on the prior studies for our evaluation [4, 39]. The small NPU configuration models an edge-level NPU based on ARM Ethos N77 [3], and the large NPU configuration represents a server-level NPU based on Google TPU [45]. Our baseline scheduling includes relevant prior DNN scheduling techniques discussed in Section 2.3. One of the key optimizations is tiling, and we model the tiling strategies proposed in the earlier studies [22, 33, 43].

For batch sizes, the `small NPU` configuration uses a batch size of 4 to accommodate edge environments with limited computation resources. `Large NPU` uses a batch size of 8, based on the TPUv4-8 configuration. A TPUv4-8 is comprised of eight TPUv4 cores, and each core contains four systolic arrays (128x128 PEs). TPUv4-8 employs data parallelism and an ideal batch size of 256 is recommended [21]. As the large NPU configuration mimics a single systolic array of TPUv4, we use the same batch size of 8 for the configuration. However, as discussed in Section 6.5, the batch size does not significantly impact the performance improvement in this study.

We adopt the same data layout as PyTorch [51], and for other aspects such as compilation and fusion strategies, we reference

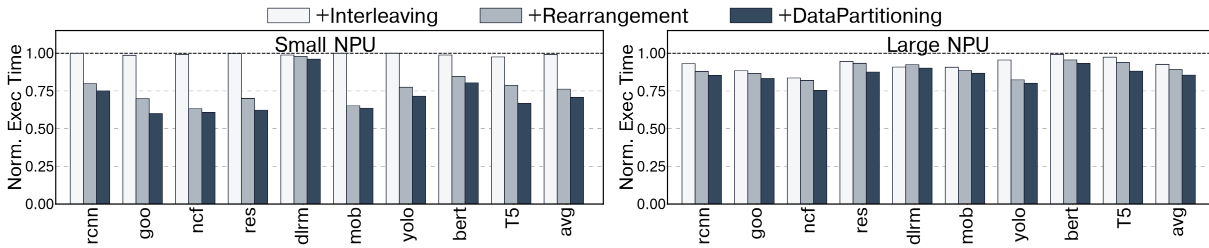

Figure 12: Execution times for *Interleaving*, *+Rearrangement*, and *+DataPartitioning* normalized to the baseline.

PyTorch [51] and XLA [20]. Our evaluation incorporates tiling strategies and double-buffering mechanism of previous studies into the baseline [33, 36]. Furthermore, given that we interleave (fuse) two gradient computations to a single linear operation (convolution and GEMM), additionally utilizing operation fusion techniques from previous studies will further improve performance.

Workloads: Table 4 provides details about the DNN models used in our evaluation, including various machine learning domains such as recommendation systems, object detection, natural language processing, and image classification. For models with different sizes, such as `yolo`, `bert`, and `T5`, we utilize different sizes for `large NPU` and `small NPU`. As mentioned in 3.1, the computation of the loss function and parameter updates takes relatively little time in the overall training process. Therefore, our focus is primarily on the forward pass and backward pass stages when investigating performance. Moreover, as described in Section 4.2, the proposed three steps are applied to layers where weight gradients and input gradients can be computed using GEMM or convolution operations.

There are prior works related to tensor rematerialization, which recalculate intermediate activations during the backward pass rather than storing intermediate activations in memory [10, 26]. While tensor rematerialization might be efficient in cases of limited memory, we do not incorporate tensor rematerialization in our NPU simulation. Instead, we store all intermediate activations from the forward pass in the off-chip memory for later use in the backward pass.

6.2 Single NPU Performance Improvement

This subsection assesses the performance improvements achieved through the interleaved gradient order in two single-core NPU configurations. Figure 12 illustrates the normalized execution time relative to the single core `baseline NPU` for each DNN workload. The results are presented separately for the `small NPU` and `large NPU` configurations. Each bar in the graph represents the cumulative outcomes obtained by applying the `Interleaving`, `Rearrangement`, and `DataPartitioning` techniques proposed in this study. For instance, `+Rearrangement` signifies the performance outcome when both `Interleaving` and `Rearrangement` techniques are applied to the baseline configuration.

The three techniques lead to significant reductions in execution times. `Small NPU` exhibits an average performance improvement of 29.3%, while `Large NPU` demonstrates a 14.5% improvement on average. `Small NPU` experiences a greater improvement than `Large NPU` due to the smaller size of its SPM, which underscores the increased importance of data reuse for achieving better performance.

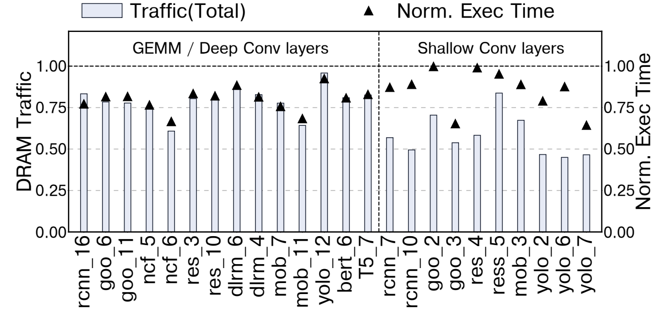

Figure 13: The amount of DRAM accesses, and execution times of *+Rearrangement*, normalized to those of the baseline.

On average, `Interleaving` results in a 0.8% reduction in execution time for `small NPU` and a 7.4% reduction for `large NPU`. As discussed in Section 4.3, applying only `Interleaving` does not lead to a substantial performance improvement for `small NPU`, mainly due to the limited SPM size that hinders efficient data reuse through `Interleaving` alone. However, when `Rearrangement` is combined with `Interleaving` and memory access order is selected by Algorithm 1, it yields significant performance improvements, especially for `small NPU`. On average, this combined approach leads to further reductions in execution times by 23.8% for `small NPU` and 10.9% for `large NPU`. Lastly, the application of `Data Partitioning`, as discussed in Section 5, along with `Interlevaing` and `Rearragement`, results in additional reductions in execution time. `Data Partitioning` leads to a 29.3% decrease in execution time for `small NPU` and a 14.5% decrease for `large NPU`.

To explain the reason behind the performance improvement achieved when both `Interleaving` and `Rearrangement` techniques are applied, we conducted a comparison of the number of DRAM accesses and execution times between the `baseline` NPU configuration and the one with our two techniques. Figure 13 illustrates the execution times and DRAM traffic of twenty four memory-intensive layers from the workloads, representing the top 15% of the longest-running layers in `large NPU`. The x-axis represents the workload names and the corresponding layer numbers for each case. For instance, *goo_3* indicates the 3rd layer of *goo*. Although the first layer consumes a significant amount of time, it is not included in Figure 13 because the interleaving technique cannot be applied in the first layer since there is no need to compute dX. The execution times and DRAM traffic are normalized with respect to the baseline. The results demonstrate a strong correspondence between the reduction in execution time and the decrease in memory traffic. These layers in Figure 13 are predominantly memory-bound, and

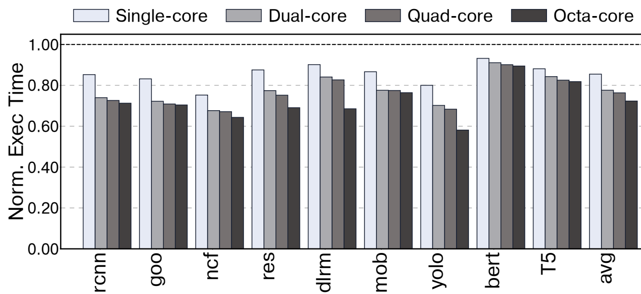

Figure 14: Execution times of interleaved gradients order in multi-core NPUs, which are normalized to the baseline NPU with the same number of cores.

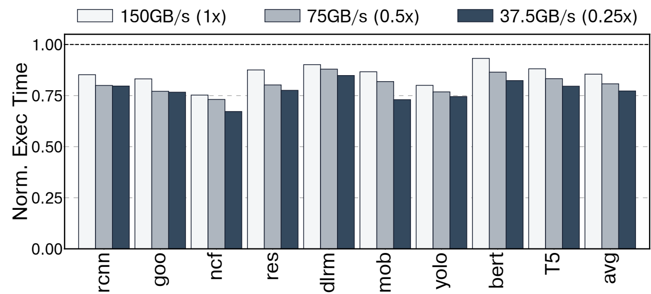

Figure 15: Execution times of `large NPU` with different bandwidth, normalized to the baseline with the same bandwidth: 150GB/s (baseline), 75GB/s (0.5x), and 37.5GB/s (0.25x) per a 128x128 systolic array.

therefore the reduction in memory traffic directly contributes to the observed performance improvement.

However, the relationship between DRAM access reduction and performance improvement varies depending on the characteristics of the layers. Layers to the left of the vertical line in Figure 13 are either GEMM (FC) layers or late-stage (deep) convolution layers with relatively small input feature maps, and there is a close correlation between DRAM access reduction and performance improvement in these layers. On the other hand, layers to the right of the vertical line are early-stage (shallow) convolution layers with significantly larger input feature maps but very small weight per channel. In these layers, calculating dW involves multiplying large-sized X and dY, but when calculating dX, smaller-sized W is used. As it is challenging to balance these two gradient computations during the interleaving process, performance improvement is insignificant compared to the reduction in memory traffic.

6.3 Multi-core NPU Scalability

To validate the efficacy of our proposed techniques in improving the performance of multi-core NPUs, we conducted an evaluation of execution times while varying the number of cores on the NPU. Given that the increasing size of emerging DNN models underscores the significance of multi-core NPUs, especially in server-class NPU hardware, we use the `large NPU` configuration for this multi-core evaluation. We assume that DRAM bandwidth, SPM size, and batch size increase proportionally with the growth in the number of cores, with all cores sharing the SPM. For example, while a batch size of 8 was employed on a single core (a 128x128 PE array), a batch size of 32 was utilized on a quad-core system (comprising four 128x128 PE arrays).

Figure 14 compares the execution time with our techniques normalized to the baseline with the same number of cores. For example, the third bar for `rcnn` compares a quad-core system including `Interleaving` + `Rearrangement` + `DataPartitioning` with a vanilla quad-core system in which none of our proposed techniques are applied. We observe a significant reduction in execution time as the number of cores increases. The reduced execution times range from 14.5% for the single core to 27.7% for the octa-core. Especially, in the quad-core similar to TPUv4's TensorCore composed of four systolic arrays sharing SPM, the performance improved by 23.7% on average.

While performance improvements generally grow with increasing numbers of NPUs due to `DataPartitioning`, there are cases where the performance improvement diminishes even with an increase in the number of cores. Although the smallest performance improvement is observed in the octa-core (`mob`), there is still a 10.5% performance improvement as shown in Figure 14. This demonstrates that our idea is not limited to a single core scenario and effectively scales with an increasing number of NPU cores.

6.4 Effect of DRAM Bandwidth

Since *interleaved gradient order* focuses on resolving memory overhead by reusing data during gradient computations, performance improvement is particularly significant when DRAM bandwidth is constrained and on-chip data reuse becomes more important. Recently, although the overall DRAM bandwidth has been increasing for server-class NPUs with HBM technologies, the number of processing elements has also been multiplied. For TPUs, the memory bandwidth increased from 700GB/s in TPUv2 to 1200GB/s in TPUv4. However, at the same time, the number of MXU (128x128 systolic array of TPU) has been increased from 2 to 8. The actual memory bandwidth per MXU has decreased from 350GB/s to 150GB/s [23, 24].

To evaluate the effectiveness of our techniques in scenarios with reduced DRAM bandwidth under such trends, we conduct experiments with the extended single-core `large NPU` configurations with 0.5 times (75GB/s per MXU) and 0.25 times (37.5GB/s per MXU) lower DRAM bandwidth compared to the baseline configuration.

Figure 15 illustrates the results of measuring the execution time in these reduced DRAM bandwidth scenarios. When our proposed techniques were applied with different memory bandwidths of 150GB/s (1x), 75GB/s (0.5x), and 37.5GB/s (0.25x), we observed performance improvements of 14.5%, 19.3%, and 22.7% respectively. As the memory bandwidth decreases, the performance impact by scarcity of memory bandwidth is amplified, and the potential for throughput improvement through maximizing data reuse in the SPM, increases significantly. The memory bandwidth has been the primary resource restricting the performance of NPUs with growing ML model sizes, and it has been difficult to scale it due to the limitation of the memory technology. Our techniques can reduce memory bandwidth requirements effectively.

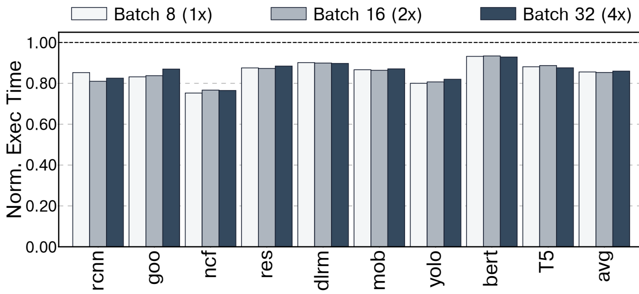

Figure 16: Execution times of `large NPU` with different batch sizes, normalized to the baseline with the same batch size. For each 128x128 systolic array, the batch size varies from 8 to 32. They correspond to batch sizes of 256, 512, and 1024 in TPUv4-8.

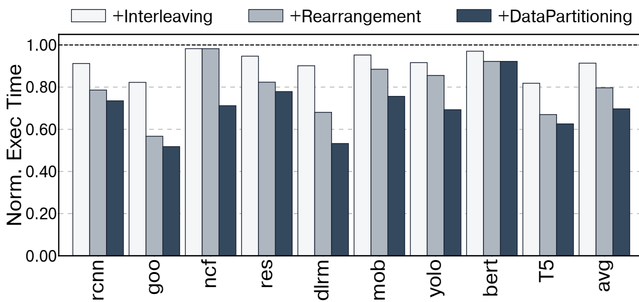

Figure 17: Performance improvement of proposed techniques in GPU with considering only backward pass.

6.5 Effect of Batch Size

Figure 16 illustrates the performance of our techniques with different batch sizes with a single-core *large NPU* configuration. Each bar represents the execution time of `Interleaving` + `Rearrangement` + `DataPartitioning`, normalized to the baseline of the same batch size. Computational resource, memory bandwidth, and SPM size remain constant as the batch size increases in this experiment. The performance improvements by our techniques are 14.5%, 14.7%, and 14.0% with a batch size of 8, 16, and 32 per each `large NPU` (256, 512, and 1024 per each TPUv4-8), and the differences in the improvements are negligible. There is no consistent trend in whether performance improvement increases or decreases with larger batch sizes. Although the optimal batch size may vary depending on the specific model's layer and NPU configuration, Figure 16 demonstrates that the performance improvement from our approach is not limited to a specific batch size.

6.6 Validating Improvement in GPU

To evaluate the effectiveness of our approach on a real system, we apply the scheduling techniques to a GPU-based implementation, using the on-chip shared memory of NVIDIA GPUs for the storage of data reuse. In the current GPU-based training implementation, dX and dW computations are serially executed with a sequence of two-input general matrix-matrix multiplications (GEMMs). To implement our techniques to GPUs, a new three-input matrix-matrix multiplication kernel (X, W, dY), which handles dX and dW in a single kernel with interleaved order, is added. Since the official code for cuBLAS is not publicly available, we modified a version of CUDA matrix multiplication implementation with optimizations such as SMEM caching and 2D Blocktiling as the baseline for computing linear layers [5]. We use NVIDIA GeForce RTX 3090 for the evaluation with the workloads in Table 4. The runs used the same batch size as *s*mall NPU. For the results, only the backward pass in the DNN training process is measured, and the time required to transfer data from the CPU to the GPU is not accounted for.

Figure 17 illustrates the performance improvement achieved when implementing our ideas on a GPU. To report the performance improvement achieved only through reusing dY, we excluded the benefit resulting from simply fusing two GEMM kernels, as fusing kernels in GPUs can reduce kernel launch overheads. For the baseline, we run each layer with two configurations: (1) sequentially executing two CUDA GEMM kernels (dX and dW) and (2) executing a single CUDA kernel that sequentially calculates dX and dW without interleaving. For the comparison excluding the effect of fusion, the baseline performance is measured by using the better one for each layer from the two possible configurations. Note that our approach always uses a fused one as it requires to interleave two operations within a kernel.

In Figure 17, the cumulative application of the three techniques results in performance improvements of 8.6%, 20.3%, and 30.3%, respectively. The result demonstrates the effectiveness of our scheme on a GPU by reducing the number of loaded dY into shared memory. Although we excluded the benefit from the kernel fusion to isolate the benefit by dY reuse, our approach can provide a new kernel fusion opportunity to reduce sequential kernel launch overheads.

7 RELATED WORK

Recent studies explored the extensive design space of DNN accelerators with the investigation of the effect of computation mapping, tiling strategies, and dataflows [7, 33, 43, 50, 61]. Marvel addressed the complexity of the mapping space for tiling and parallelization by decomposing the space into the lower-dimension off-chip and on-chip subspaces [7]. Interstellar exploited Halide's scheduling language to investigate the design space of DNN accelerators, analyzing the performance impact of tiling and replication [61]. Timeloop narrowed the vast architectural design space of DNN accelerators by analyzing tile-access counts for hardware components [50]. GAMMA automated the process of finding the optimal computation and HW mapping for DNN accelerators, taking into account tiling strategies, scheduling, and parallelism [33]. Moon et al. introduced a framework for finding the optimal mapping and tile size for various dataflow accelerators [43].

Several studies have investigated operation scheduling and data reuse in GPUs and DNN accelerators [9, 47, 49]. Out-of-order backprop introduced a scheduling technique to address GPU underutilization by re-ordering operations based on the inter-layer dependencies of gradient operations in backward passes [47]. In contrast to the technique, our approach fuses and rearranges the independent computation for two gradients in a layer to maximize the data reuse of the output gradient. TVM formalized the process of operator fusion and proposed schedule primitives to generate efficient codes under various factors, such as data layout or hardware configurations [9]. FLAT and FlashAttention have enabled

streamlined computations with effectively using the on-chip memory through inter-operation fusion for attention in transformer models [15, 34]. Layerweaver proposed a greedy scheduler that maps heterogeneous DNN models with different resource demands to multiple NPUs [49]. MOSAIC [28] and Seo et al. [56] investigated scheduler designs to map ML tasks to heterogeneous processors including a GPU and NPUs in a chip. Gpulets partition and adjust GPU resources dynamically to maximize GPU utilization for heterogeneous ML tasks [12].

Previous studies have investigated the parallelization strategy for DNN training by decomposing tensors to balance the computation and minimize the communication. As a batch size plays an important role in efficient training, optimal data parallelism methods tailored to training processes have been studied [25, 29, 35, 42]. Megatron-LM introduced tensor parallelism, which further improves upon the traditional model parallelism by incorporating intra-layer parallelism [57]. FlashAttention2 modified the tensor splitting process across warps for attention layers, such as splitting Q instead of K and V in the forward pass [14]. This adjustment reduces the synchronization overhead in accumulating intermediate results in the shared memory of GPUs.

8 CONCLUSION

This paper introduced a novel dataflow transformation scheme called *interleaved gradient order*, designed to maximize data reuse within the scratchpad memory (SPM) during the backward pass of DNN training. In contrast to the previous efforts that focused on optimizing SPM usage in each individual operations, this study demonstrates that significant performance improvements can be achieved by promoting inter-operation data sharing. Our approach proposed the interleaved gradient computation fusing two gradient operations and an algorithm for selecting the optimal tile access order. This paper also discussed a method for data partitioning schemes for single-core and multi-core NPUs and proposed an efficient approach for selecting the best partitioning policy for each layer. Our simulation-based evaluation showed that the proposed techniques lead to an average performance improvement of 29.3% for the single-core edge NPU and 14.5% for the single-core server NPU, respectively. Moreover, in the experiment with the quad-core server NPU, the proposed techniques resulted in a 23.7% reduction in execution time.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) (IITP2017-0-00466 SW StarLab, and IITP2021-0-01817 Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters) funded by the Ministry of Science and ICT, Korea.

REFERENCES

[1] Ankur Agrawal, Sae Kyu Lee, Joel Silberman, Matthew Ziegler, Mingu Kang, Swagath Venkataramani, Nianzheng Cao, Bruce Fleischer, Michael Guillorn, Matthew Cohen, et al. 2021. 9.1 a 7nm 4-core AI chip with 25.6 TFLOPS hybrid FP8 training, 102.4 TOPS INT4 inference and workload-aware throttling. In *International Solid-State Circuits Conference (ISSCC)*.

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[3] ARM. 2019. *Powering The Edge: Driving optimal performance with Ethos-N77 processor.* Technical Report. ARM.

[4] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. 2020. A multi-neural network acceleration architecture. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*.

[5] Simon Boehm. 2022. How to Optimize a CUDA Matmul Kernel for cuBLAS-like Performance: a Worklog. https://siboehm.com/articles/22/CUDA-MMM

[6] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. In *Conference on Neural Information Processing Systems (NIPS)*.

[7] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. 2021. Marvel: A data-centric approach for mapping deep learning operators on spatial accelerators. In *Journal of ACM Transactions on Architecture and Code Optimization (TACO)*.

[8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM SIGARCH Computer Architecture News*.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[10] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. In *ArXiv Preprint ArXiv:1604.06174*.

[11] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE journal of solid-state circuits (JSSC)*.

[12] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In *USENIX Annual Technical Conference (ATC)*.

[13] Seungkyu Choi, Jaehyeong Sim, Myeongggu Kang, Yeongjae Choi, Hyeonuk Kim, and Lee-Sup Kim. 2020. An energy-efficient deep convolutional neural network training accelerator for in Situ personalization on smart devices. In *Journal of Solid-State Circuits (JSSC)*.

[14] Tri Dao. 2023. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *arXiv preprint arXiv:2307.08691*.

[15] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NIPS)*.

[16] Mario Drumond, Louis Coulon, Arash Pourhabibi, Ahmet Caner Yüzügüler, Babak Falsafi, and Martin Jaggi. 2021. Equinox: Training (for Free) on a custom inference accelerator. In *International Symposium on Microarchitecture (MICRO)*.

[17] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*.

[18] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3D memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[19] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Design Automation Conference (DAC)*.

[20] Google. 2023. Tensorflow XLA. https://www.tensorflow.org/xla

[21] Google. 2023. Understanding data sharding (data parallelism) : Google Cloud TPU. https://cloud.google.com/tpu/docs/troubleshooting/trouble-tf#data-parallelism

[22] Google Cloud. 2023. Cloud TPU programming model. https://cloud.google.com/tpu/docs/tpus#programming_model

[23] Google Cloud. 2023. System Architecture: TPU Chip. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm#tpu_chip

[24] Google Cloud. 2023. System Architecture: TPU v4. https://cloud.google.com/tpu/docs/system-architecture-tpu-vm#tpu_v4

[25] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. In *ArXiv Preprint ArXiv:1706.02677*.

[26] Andreas Griewank and Andrea Walther. 2000. Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. In *Transactions on Mathematical Software (TOMS)*.

[27] Stefan Hadjis, Firas Abuzaid, Ce Zhang, and Christopher Ré. 2015. Caffe con troll: Shallow ideas to speed up deep learning. In *Proceedings of the Fourth Workshop on Data analytics in the Cloud (DanaC)*.

[28] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. 2019. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference. In *2019 28th*

International Conference on Parallel Architectures and Compilation Techniques (PACT).

[29] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. 2018. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. In *ArXiv Preprint ArXiv: 1807.11205.*

[30] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A domain-specific supercomputer for training deep neural networks. In *Communications of the ACM (CACM).*

[31] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture (ISCA).*

[32] Sanghoon Kang, Donghyeon Han, Juhyoung Lee, Dongseok Im, Sangyeob Kim, Soyeon Kim, and Hoi-Jun Yoo. 2020. 7.4 GANPU: A 135TFLOPS/W multi-DNN training processor for GANs with speculative dual-sparsity exploitation. In *International Solid-State Circuits Conference (ISSCC).*

[33] Sheng-Chun Kao and Tushar Krishna. 2020. GAMMA: Automating the HW Mapping of DNN Models on Accelerators via Genetic Algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD).*

[34] Sheng-Chun Kao, Suvinay Subramanian, Gaurav Agrawal, Amir Yazdanbakhsh, and Tushar Krishna. 2023. FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[35] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations (ICLR).*

[36] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. 2020. Maestro: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings. In *IEEE micro.*

[37] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. 2021. Heterogeneous dataflow accelerators for multi-DNN workloads. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

[38] Jinsu Lee, Juhyoung Lee, Donghyeon Han, Jinmook Lee, Gwangtae Park, and Hoi-Jun Yoo. 2019. 7.7 LNPU: A 25.3 TFLOPS/W sparse deep-neural-network learning processor with fine-grained mixed precision of FP8-FP16. In *International Solid-State Circuits Conference (ISSCC).*

[39] Sunho Lee, Jungwoo Kim, Seonjin Na, Jongse Park, and Jaehyuk Huh. 2022. TNPU: Supporting trusted execution with tree-less integrity protection for neural processing unit. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA).*

[40] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. Performance analysis of GPU-based convolutional neural networks. In *International Conference on Parallel Processing (ICPP).*

[41] Wei Yang Bryan Lim, Nguyen Cong Luong, Dinh Thai Hoang, Yutao Jiao, Ying-Chang Liang, Qiang Yang, Dusit Niyato, and Chunyan Miao. 2020. Federated learning in mobile edge networks: A comprehensive survey. In *IEEE Communications Surveys & Tutorials.*

[42] Dominic Masters and Carlo Luschi. 2018. Revisiting small batch training for deep neural networks. In *ArXiv Preprint ArXiv:1804.07612.*

[43] Gordon Euhyun Moon, Hyoukjun Kwon, Geonhwa Jeong, Prasanth Chatarasi, Sivasankaran Rajamanickam, and Tushar Krishna. 2021. Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication. In *IEEE Transactions on Parallel and Distributed Systems (TPDS).*

[44] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: Accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI).*

[45] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. 2021. The design process for Google's training chips: TPUv2 and TPUv3. In *IEEE Micro.*

[46] NVIDIA. 2023. FasterTransformer. https://github.com/NVIDIA/FasterTransformer

[47] Hyungjun Oh, Junyeol Lee, Hyeongju Kim, and Jiwon Seo. 2022. Out-of-order backprop: An effective scheduling technique for deep learning. In *European Conference on Computer Systems (EuroSys).*

[48] Jinwook Oh, Sae Kyu Lee, Mingu Kang, Matthew Ziegler, Joel Silberman, Ankur Agrawal, Swagath Venkataramani, Bruce Fleischer, Michael Guillorn, Jungwook Choi, et al. 2020. A 3.0 TFLOPS 0.62 V scalable processor core for high compute utilization AI training and inference. In *Symposium on VLSI Circuits (VLSI-circuits).*

[49] Young H Oh, Seonghak Kim, Yunho Jin, Sam Son, Jonghyun Bae, Jongsung Lee, Yeonhong Park, Dong Uk Kim, Tae Jun Ham, and Jae W Lee. 2021. Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling. In *International Symposium on High Performance Computer Architecture (HPCA).*

[50] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to DNN accelerator evaluation. In *IEEE international symposium on performance analysis of systems and software (ISPASS).*

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NIPS).*

[52] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *International Symposium on High Performance Computer Architecture (HPCA).*

[53] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Acm Sigplan Notices.*

[54] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Conference on Artificial Intelligence (AAAI).*

[55] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2020. A systematic methodology for characterizing scalability of DNN accelerators using SCALE-Sim. In *International Symposium on Performance Analysis of Systems and Software (ISPASS).*

[56] Wonik Seo, Sanghoon Cha, Yeonjae Kim, Jaehyuk Huh, and Jongse Park. 2021. SLO-aware inference scheduler for heterogeneous processors in edge platforms. (2021).

[57] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-LM: Training multi-billion parameter language models using model parallelism. In *arXiv preprint arXiv: 1909.08053.*

[58] Mingcong Song, Kan Zhong, Jiaqi Zhang, Yang Hu, Duo Liu, Weigong Zhang, Jing Wang, and Tao Li. 2018. In-Situ AI: Towards autonomous and incremental deep learning for IoT systems. In *International Symposium on High Performance Computer Architecture (HPCA).*

[59] Danilo Vucetic, Mohammadreza Tayaranian, Maryam Ziaeefard, James J Clark, Brett H Meyer, and Warren J Gross. 2022. Efficient fine-tuning of BERT models on the edge. In *International Symposium on Circuits and Systems (ISCAS).*

[60] Yang Wang, Dazheng Deng, Leibo Liu, Shaojun Wei, and Shouyi Yin. 2022. PL-NPU: An energy-efficient edge-device DNN training processor with posit-based logarithm-domain computing. In *IEEE Transactions on Circuits and Systems (TCAS).*

[61] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. 2020. Interstellar: Using halide's scheduling language to analyze DNN accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[62] Shouyi Yin, Shibin Tang, Xinhan Lin, Peng Ouyang, Fengbin Tu, Jishen Zhao, Cong Xu, Shuangcheng Li, Yuan Xie, ShaoJun Wei, et al. 2018. Parana: A parallel neural architecture considering thermal problem of 3D stacked memory. In *IEEE Transactions on Parallel and Distributed Systems (TPDS).*

[63] Zhilu Zhang and Mert Sabuncu. 2018. Generalized cross entropy loss for training deep neural networks with noisy labels. In *Conference on Neural Information Processing Systems (NIPS).*