Discovering Efficient Fused Layer Configurations for Executing Multi-Workloads on Multi-core NPUs

Younghyun Lee[1], Hyejun Kim[4], Yongseung Yu[2], Myeongjin Cho[3], Jiwon Seo[2], and Yongjun Park[4]
[1]*Dept. of Artificial Intelligence, Hanyang University,* [2]*Dept. of Computer Science, Hanyang University*
[3]*SAPEON Korea Inc,* [4]*Dept. of Computer Science and Engineering, Yeonsei University*
{younghyunlee[1], dydtmd1991[2], seojiwon[2]}@hanyang.ac.kr, brown@sapeon.com[3], {hyejunkim[4], yongjunpark[4]}@yonsei.ac.kr

***Abstract*—As the AI industry grows rapidly, Neural Processing Units (NPUs) have been developed to deliver AI services more efficiently. One of the most important challenges for NPUs is task scheduling to minimize off-chip memory accesses, which may occur significant performance overhead. To reduce memory accesses, multiple convolution layers can be fused into a fused layer group, which offers numerous optimization opportunities. However, in most Convolutional Neural Networks (CNNs), when multiple layers are fused, the on-chip memory utilization of the fused layers gradually decreases, resulting in non-flat memory usage. In this paper, we propose a scheduling search algorithm to optimize the fusion of multiple convolution layers while reducing the peak on-chip memory usage. The proposed algorithm aims to find a schedule that simultaneously optimizes execution time and peak on-chip memory usage, despite a slight increase in off-chip memory accesses. It organizes the search space into a graph of possible partial schedules and then finds the optimal path. As a result of the improved on-chip memory usage, multiple workloads can be executed on multi-core NPUs with increased throughput. Experimental results show that the fusion schedule explored by the proposed method reduced on-chip memory usage by 39%, while increasing latency by 13%. When the freed on-chip memory was allocated to other workloads and the two workloads were executed concurrently in a multi-core NPU, a 32% performance improvement could be achieved.**

***Index Terms*—Compilers, Neural networks, NPU**

I. Introduction

Artificial Intelligence (AI) has recently been successfully applied in many emerging applications, and several companies such as Microsoft [1], Google [2], and Tesla [3] have released Neural Processing Units (NPUs) to provide AI services through cloud servers. These accelerators perform a large number of Mutiply-Add-Compute(MAC) operations in parallel, resulting in high performance and low power consumption. One of the main challenges in efficiently utilizing the NPU is minimizing off-chip memory accesses. DNN operations typically have a large memory footprint because each layer requires significant data sizes for feature maps and weight parameters, and there are a massive number of sequentially connected layers. The accelerator's frequent accesses to off-chip memory (DRAM) often cause significant performance degradation [4], [5]. To address this issue, NPUs include on-chip memory units (SRAM). Some of the data from the off-chip memory is brought into the on-chip memory for computation, and the computed results are stored back into the off-chip memory via the on-chip memory.

Unlike CPU caches, the use of on-chip memory in the NPU must be manually scheduled. In addition, because on-chip memory has a limited capacity, the data required for each DNN layer computation must be partitioned and sequentially loaded into on-chip memory. Therefore, the utilization of on-chip memory is highly dependent on the computation scheduling results of the deep learning compiler [6], [7].

Operator fusion is a scheduling method for combining multiple operations to minimize off-chip memory access. It is a method of fusing and executing multiple operations by storing partial results of one operation in on-chip memory and passing them as inputs to the next operation. The combined operation does not store intermediate results in off-chip memory, allowing efficient memory access. In particular, the output of a previous layer is locally connected to the input of the next layer in consecutive convolution layers. As a result, the operation of the next layer can proceed even though only a portion of the operation result is complete. In short, this concept is typically referred to as 'fused-convolution'.

Convolutional Neural Network (CNN) is mainly used for image classification, semantic segmentation, and object detection applications, and is also used for super resolution and image generation. CNN models are first trained on large-scale data for image classification, and the front part of the trained model functions as a feature extractor and is used as a backbone for other application models. The backbone network generally extracts important features from the image by reducing the height and width of the feature map by half and doubling the channels as the layer gets deeper. The problem with applying fused-convolution to this backbone network is that the on-chip memory utilization gradually decreases during the group operation. At the beginning of the fusion group, most of the buffer is used, but later in the group, the feature map becomes smaller and most of the buffer is not utilized.

In this paper, we propose a scheduling algorithm that optimizes the latency of fused-convolution while reducing the average and peak usage of on-chip memory. First, we introduce a cost function based on the computation and memory accesses of a possible fusion schedule. Next, we organize the fusion schedules into graph edges, and the weight of each edge is calculated using the memory and compute costs. Then, the Dijkstra algorithm [8] is used on the constructed schedule graph to find the optimal schedule. Finally, we adjust the tile configuration of each fusion group in the optimal schedule to improve on-chip memory utilization.

Due to the reduced and flattened on-chip memory usage,

979-8-3503-4859-0/DATE24/© 2024 EDAA

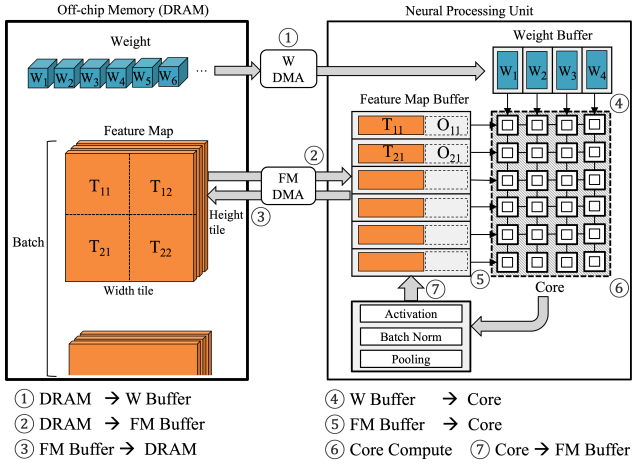

Fig. 1. An example NPU architecture and major system cycles

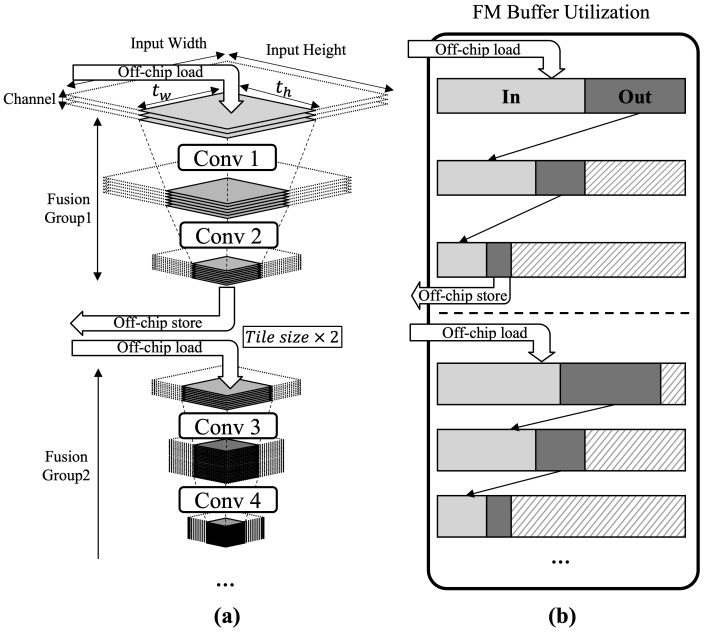

Fig. 2. (a) Fused-convolution : Group1[Conv1, Conv2], Group2[Conv3, Conv4], (b) feature map buffer utilization

the remaining on-chip memory space can be used for other DNN workloads. Therefore, we configured a multi-core NPU sharing on-chip memory on an in-house NPU simulator and evaluated the effectiveness of the proposed technique for multi-workload execution. The evaluated CNN models consist of classification, which is mainly used as a backbone, and other application models. We scheduled two target DNN workloads to run simultaneously, and our technique were able to improve the throughput by 32% over running the two workloads sequentially with the baseline method.

II. Background and Motivation

A. Conventional NPU Architectures

Fig. 1 illustrates the main components of a typical NPU system. The system consists of main memory (DRAM) and an NPU, which is equipped with a small on-chip memory (SRAM) to reduce dram accesses. The compute core includes hardware that performs DNN operations such as matrix multiplication and activation functions. Although the high-level structure appears simple, NPU have a wide design options influenced by multiple key factors such as the number of PEs, the PE layout, its dataflow, and memory hierarchy [9], [10]. For example, PE configuration options include Systolic Array [2], [11], [12] and Adder Tree [13]–[15] layouts, and data flow strategy encompasses IS (Input Stationary), WS (Weight Stationary), OS (Output Stationary), and RS (Row Stationary) approaches. Moreover, NPUs typically have a multi-level memory hierarchy to reduce off-chip memory accesses. The memory structure can be designed by considering multiple factors such as memory size, bank count, and dedicated scratchpads for weight parameters and feature map (FM) data.

Fig. 1 also shows the system cycles that occur in an NPU system due to core computation and data transfers between modules. In this figure, (①∼③) are involved in dram cycles, and (④∼⑦) are involved in compute cycles. Since the FM buffer (on-chip memory) is much smaller than dram, it is not possible to store the entire feature map at once. Therefore, it's split into multiple tiles, then transferred from dram to the FM buffer, and computed on the core. The size of each tile is represented by the tile size t_w in the width direction and the tile size t_h in the height direction. The total number of tiles is (FM_w / t_w) * (FM_h / t_h). To reduce the dram cycle, the total number of tiles should be minimized, which means the largest tile size that can fit in the FM Buffer should be determined. The compute cycle is dominated by ⑥ core compute cycles, as other compute cycles can be hidden. If the number of tiles increases, then ① can affect the compute cycle. In most CNNs, the number of filters in the weight increases as the layer gets deeper, and the Weight buffer(on-chip memory) can't hold them all at once. As shown in the Fig. 1, after computing T_{11} and $W_1 \sim W_4$ and getting a partial result, we need to load the next weights ($W_5 \sim$) and compute them with T_{11} to complete the output, O_{11}. When computing the next tile, T_{12}, $W_1 \sim W_4$ are loaded into the Weight buffer again. If the number of tiles is large, the weight reload counts will also increase, which binds the compute cycle to the weight load cycle. As you can see, the number of tiles can affect both compute and dram cycles, and if we determine the tiles for each node in the computational graph G_c, the overall system cycle is determined as follows.

$$System\ cycle = \sum_{v \in G_c} \mathrm{Max}\left(\mathrm{dramCycle}\left(v\right), \mathrm{ComputeCycle}\left(v\right)\right)$$

B. Fused-convolution

Operator Fusion is a common optimization technique that can be further applied after the tiling technique by DNN compilers [6], [7], when performing graph-level optimizations. It combines tiles of major operations, such as convolution or fully connected layer, with element-wise operations, such as batch normalization and activation, to perform operations without storing intermediate results in off-chip memory. Fused-convolution is a method of combining consecutive convolution layers as well as element-wise operations [16]. While fused-convolution can improve both latency and energy by reducing off-chip memory accesses, the optimal schedule is difficult to find. The reasons are as follows.

- When fusing n serial convolution layers, the number of possible group combinations is 2^{n-1}.

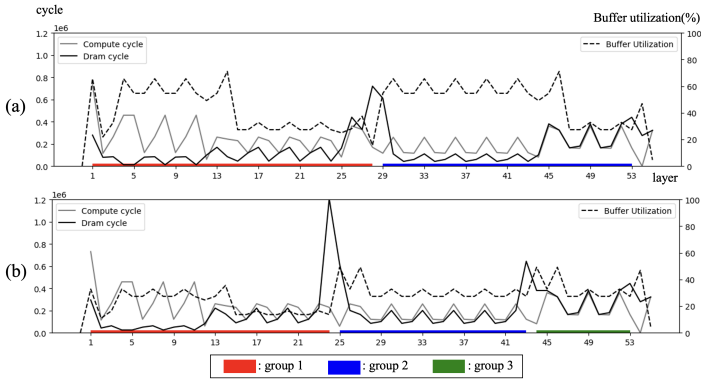

Fig. 3. Example fused-convolution schedules for ResNet-50. (a) Schedules minimizing latency and dram access and (b) schedules optimizing latency and FM buffer usage

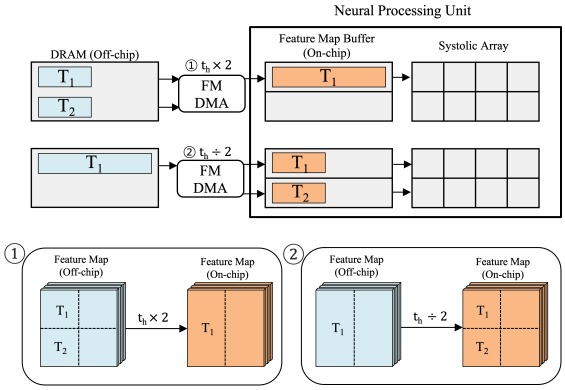

Fig. 4. Target NPU tiling constraint

- When there are multiple data dependent edges between layers, such as skip connections, there are multiple cases for the execution order.
- Since tile settings cannot be changed within a Fusion Group, the possible tile sets of each layer must be considered collectively.

Scheduling a fused-convolution is a partitioning of the entire computational graph into several fusion groups, and finding the optimal among the possible schedules is an NP-hard problem [17]. Furthermore, the optimal schedule may vary depending on the NPU structure. Several previous works have applied heuristic search algorithms to their target hardwares, aiming to find a schedule that minimizes latency while reducing dram accesses [4], [5].

Fig. 2(a) shows how four convolution layers can be fused, and Fig. 2(b) shows the FM buffer usage for Fig. 2(a). In this figure, four consecutive convolution layers are fused into two groups. Off-chip memory accesses occur only at the beginning and end of each group, and intermediate results within a group are sent directly from the FM Buffer to the core. Typically, CNNs use convolution stride and pooling layers to reduce the width and height of the feature map to extract important features. If the size of the FM becomes smaller, there are cases where the size of the tile is too small to proceed with further operations. In this case, as shown in Group 2 in Fig. 2, subsequent convolution can be performed by increasing the tile size with off-chip memory accesses. Also, as shown in Fig. 2(b), we can see that the FM buffer utilization is getting lower within the group. This is because most CNNs double the number of FM channels as the layer gets deeper, but the size of the FM is reduced by a quarter due to the pooling layer or convolution stride. Therefore, within a fusion group, FM buffer utilization tends to peak in the early part of the group and then decline, meaning that the FM buffer is underutilized later in each group execution.

C. Motivation and Insights

Fig. 3 shows the compute and dram cycles, and FM buffer utilization when two fusion schedules are applied to ResNet-50. In Fig. 3(a) schedule, the compute cycle is larger than the dram cycle in most layers and the system cycle is bounded by the compute cycle. Schedule of Fig. 3(a) is efficient because it minimizes dram accesses while optimizing overall system cycles, but FM buffers are underutilized within the fusion group. Compared to Fig. 3(a), schedule of Fig. 3(b) changes the fusion groups to three and reduces the tile size of each group. This change has increased the dram cycle but it only reaches very close to compute cycles, and the reduced tile size has also reduced the peak usage of the FM buffer. Schedule Fig. 3(b) has 7% more latency than schedule Fig. 3(a), but the peak memory usage in the FM Buffer is 22% lower, at 49%.

This result shows that there are schedules that are not optimal for latency but can reduce peak usage of the FM buffer by increasing DRAM accesses. The free space can be utilized by other compute cores sharing the FM Buffer. In this paper, we constructed an algorithm that aims to find a schedule that can significantly improve the peak usage of the on-chip memory, even if the overall system cycle is slightly increased. By applying the schedule found by the proposed method, we were able to run multi-workloads simultaneously and improve total performance in a multi-core NPU environment.

III. Buffer Efficient Fused-Convolution

This work was conducted on an in-house NPU simulator based on a typical NPU architecture similar to Fig. 1, consisting of systolic arrays and on-chip memories for FMs and weights. The target NPU has some tiling constraints as shown in Fig. 4. When a FM data is transferred from the off-chip memory to the on-chip buffer, the tile size to be stored in the buffer can be one of three options: same, 2x, and 1/2 of the previous tile size. This means that the tile size of the current layer depends on the tile size of the previous layer. In this study, we considered these constraints and implemented the search space of schedules in the form of a graph. The constructed weighted graph is used to find the schedule that can improve the peak usage of the FM buffer.

A. Graph-based Fused-convolution Scheduling

Fig. 5 is an overview of the proposed algorithm, and Algorithm 1 illustrates the creation of the schedule graph and the optimal path search processes. $[line\ 1]$ The input to the algorithm is a computational graph G_c of the target DNN

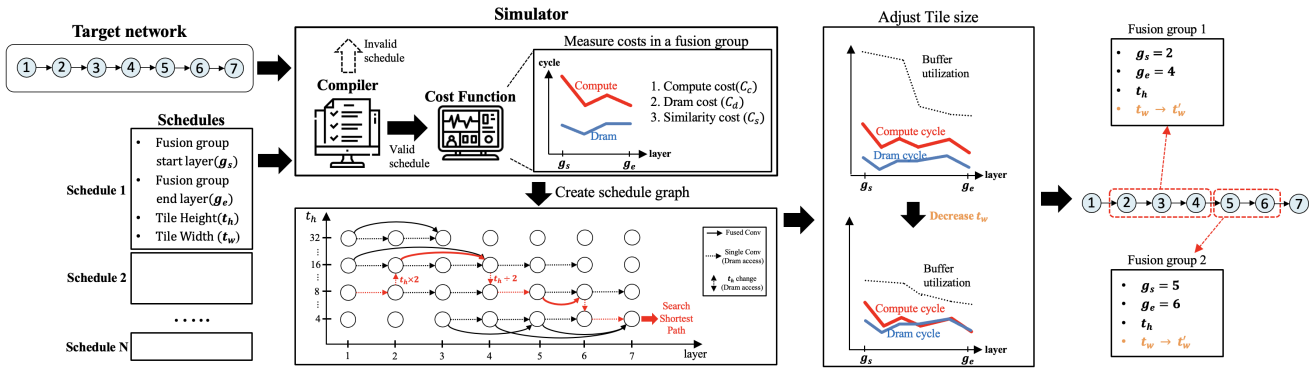

Fig. 5. Overview of buffer-efficient schedule search algorithms for high on-chip memory efficiency

workload, and G_c is topologically sorted based on depth-first search (DFS) to determine the execution order of each layer. This ensures that the intermediate results of each layer can be immediately consumed by the next layer, thus reducing the liveness of the intermediate results. [$line$ 8] It then obtains the fusion groups (g) from G_c using a sliding window technique. [$line$ 9] Once a group is selected, it gets the tile sets of T_h, T_w, which are the possible tile configurations on a layer having the smallest FM size among the layers in the fusion group. [$line$ 11] To find the largest t_w for a possible t_h, the T_w are sorted in descending order. Since t_w is relatively tunable compared to the hardware-constrained t_h, the largest possible t_w value is first explored and then reduced.

$$l = g[smallest\ tile\ layer\ index]$$
$$T_h^l = \left\{ t_h \mid t_h = \left\lceil \frac{FM_h^l}{i} \right\rceil \wedge 1 \le i \le FM_h^l \right\}$$
$$T_w^l = \left\{ t_w \mid t_w = \left\lceil \frac{FM_w^l}{i} \right\rceil \wedge 1 \le i \le FM_w^l \right\}$$

[$line$ 12 – 14] Once the group g and tile settings are determined, it tries to compile the schedule. This process checks for incorrect tile configurations and that the size of the tiles does not exceed the size of the FM buffer. [$line$ 18] If the compilation is done successfully, the compiled group g is evaluated for compute/dram cycle(cost) and similarity cost using a cost function.

$$\vec{C_c} = \text{ComputeCycles}\,(g) \qquad \vec{C_d} = \text{dramCycles}\,(g)$$
$$C_c = \text{Sum}(\vec{C_c}) \qquad C_d = \text{Sum}(\vec{C_d})$$
$$C_s = 2 - \frac{\vec{C_c} \cdot \vec{C_c}}{\|\vec{C_c}\| \cdot \|\vec{C_d}\|}$$

C_s indicates how similar the patterns of compute and dram cycles are within a group. The compute and dram cycle values for the n layers in the group can be viewed as an n-dimensional vector, and the similarity of the vectors can be measured by cosine similarity. This value is 1 if the two vectors are in the same direction, 0 if the angle between the two vectors is 90 degrees, and -1 if they are opposite. C_s is adjusted to (2 - cosine similarity value), to make the value greater than or equal to 1. A larger C_s indicates different patterns of compute cycles and dram cycles. [$line$ 19 – 20] C_d is normalized to have a maximum value of 1, and then multiplied by C_s to get w, the weight of an edge in the schedule graph. w are then written to the edge table. [$line$ 24 – 26] If no possible tile settings are found for g, we exit the loop because no possible tile settings can be found even if we increase the group size.

[$line$ 29] The completed edge table is converted to a scheduling graph G_s. During the graph conversion process, we

Algorithm 1: Graph-based Fusion Scheduling

Data: Computational Graph G_c
Result: Optimal scheduling to minimize dram cost and on-chip memory peak usage

```
1  G_c ← TopologicalSortDFS(G_c);
2  N ← n(G_c);
3  table ← InitEdgeTable();
4  for g_s ← 1 to N do
5      for size ← 1 to N-g_s do
6          CompileSuccess ← False;
7          g_e ← g_s + size − 1;
8          g ← G_c[g_s : g_e];
9          T_h, T_w ← AvailableTileSet(g);
10         for each t_h ∈ T_h do
11             for each t_w ∈ descending order of T_w do
12                 try:
13                     compile(g, t_h, t_w);
14                     CompileSuccess ← True;
15                 catch:
16                     continue;
17                 end
18                 C_d, C_s ← GetCost(g);
19                 w ← C_d * C_s;
20                 table[g_s, g_e, t_h, t_w] ← (w);
21                 break;
22             end
23         end
24         if not CompileSuccess then
25             break;
26         end
27     end
28 end
29 G_s ← ScheduleGraph(table);
30 ShortestPath ← Dijkstra(G_s);
```

Algorithm 2: Buffer Efficient Tile Schedule

Data: Fusion groups F
Result: Fusion groups F with adjusted tile settings

```
for each g ∈ F do
    cost ← SystemCycle(g);
    while True do
        DecreaseTileWidth(g);
        new cost ← SystemCycle(g);
        if new cost > cost then
            break;
        end
        cost ← new cost;
    end
end
```

construct a graph like the one in Fig. 5 such that the t_h of each layer can only be mapped to nodes that maintain, double, or 1/2 the t_h of the previous layer, reflecting the hardware constraints in Fig. 4. [$line$ 30] To find the path that minimizes the total edge cost in G_s, we used the Dijkstra algorithm [1]. The minimum path found is the resulting fusion schedule. In this optimal schedule, the dram cycle is minimized and has a similar pattern to the compute cycle, guided by C_s and C_d.

B. Buffer Efficient Tile Scheduling

The schedule found by Algorithm 1 minimizes dram cycles and tends to have a pattern similar to compute cycles. Therefore, reducing t_w in each group to increase dram cycles is unlikely to exceed compute cycles, as shown in Fig. 5. This can significantly reduce the use of the FM buffer without significantly increasing the system cycle. Algorithm 2 reduces the t_w of each group to reduce the average and peak usage of the FM buffer, while increasing the dram accesses. Algorithm 2 provides a greater reduction in FM Buffer usage for fusion groups with large FMs. The reduction of t_w is performed by when the post-adjustment system cycle becomes larger than the pre-adjustment system cycle.

IV. Evaluation

A. Experimental Setup

The evaluation was performed on an In-house NPU simulator. We configured an NPU with two cores sharing an FM buffer. Each core has 85 TOPs of processing power and can handle DNN models quantized to 8-bit fixed point. The detailed NPU specification is shown in Table I.

TABLE I
NPU Configuration

PE-array	128×64
Number of Cores	2
Dataflow	Output Stationary
FM Buffer	8 MB
Weight Buffer	3 MB
dram Bandwith	42.7 GB/s (utilization up to 80%)
Compute Precision	8-bit fixed-point

TABLE II
CNN models

VGG-19	IC(Serial model)
ResNet-50	IC(Residual connection)
MobileNet-v1	IC(DWS Convolution)
MobileNet-v2	IC(MBConv module)
GoogleNet	IC(Inception module)
Yolo-v3	Object Detection
	IC = Image Classification

[1]Dijkstra is an algorithm that finds the minimum path by gradually updating the cost table of all reachable nodes from the starting node.

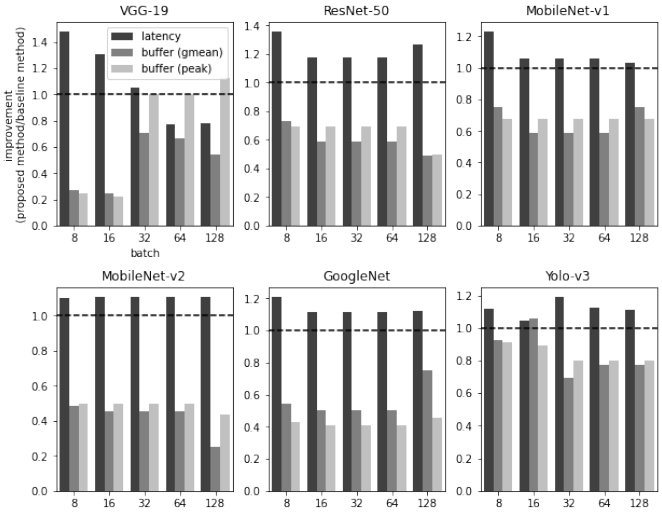

Fig. 6. Comparison of performance and FM buffer usage

For our evaluation benchmarks, we selected models that are commonly used as the backbone of CNN applications, and selected Yolo-v3 as an object detection model that can execute concurrently with them. The CNN models and the structural features of each model are shown in II. Experiments were performed with the same size of input data to balance the workload. The baseline algorithm is a heuristic search method designed for the target hardware. The algorithm searches for the largest tile configuration that can fit into the FM buffer, reflecting the constraints in Fig. 4, and then explores the fusion group using the Dynamic Programming (DP) technique. The goal of the baseline algorithm is to minimize overall system cycles and dram accesses.

B. Performance and FM Buffer Usage Comparison

Fig. 6 shows the latency and FM buffer usage of the proposed method. Several batch sizes were used from 8 to 128, which are the commonly used. Buffer(gmean) is the geometric mean usage of the FM buffer and buffer(peak) is the maximum usage. Each metric represents the percentage increase and decrease compared to the baseline algorithm. On average, the proposed method increases latency by 13%, but reduces buffer(gmean) by 39% and buffer(peak) by 38%. The proposed scheme generally performs better on lightweight networks such as MobileNet and GoogleNet than on large networks such as VGG-19. This is because reducing the FM tile size to minimize the use of the FM buffer increases the weight reloads, and lightweight models are less affected by this. Therefore, models with weight reduction techniques, such as the Depth Wise Separable (DWS) convolution used in the MobileNet series and the 1x1 convolution used in GoogleNet's Inception module, were able to reduce FM buffer usage even more with a slight increase in latency. These lightweight models increased latency by about 11% and reduced buffer(gmean) and buffer(peak) by 51% and 54%, respectively.

C. Impact of FM Buffer Usage Limits

Fig. 7 shows the increase in latency as we set a target FM Buffer usage and decrease the tile size in each fusion group

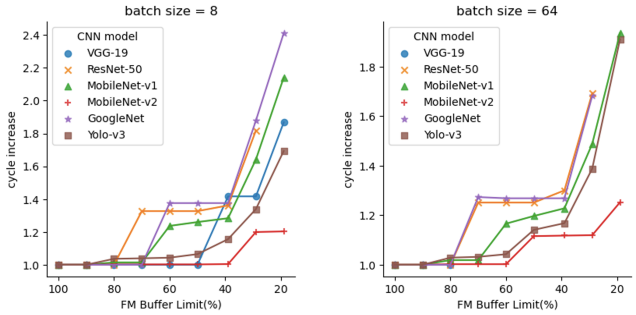

Fig. 7. Impact of different FM Buffer usage limits

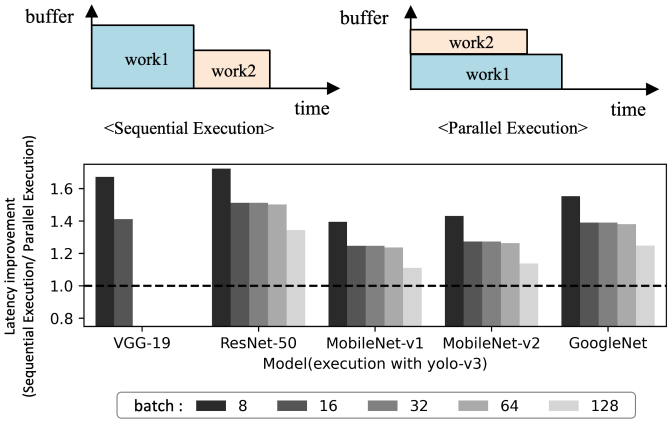

Fig. 8. Performance evaluation of multi-workload execution

until the target utilization is reached. We experimented with FM buffer limits from 100% to 20%, and no points were marked if possible tile settings could not be found. With a batch of 64, VGG-19 could not find any tile configurations below the 80% FM buffer limit. The MobileNet series and Yolo-v3 did not show a significant latency increase when the FM Limit was lowered, showing that they are relatively insensitive to changes in tile size.

D. Multi-workload Execution

Fig. 8 shows the performance improvement when the Image Classification models are run in parallel with Yolo-v3. For comparison, the baseline algorithm was applied to schedule Image Classification and Yolo-v3 models, and then run them sequentially. Note that VGG-19 does not have schedules that could run concurrently with Yolo-v3 at high batch sizes. Fig. 8 shows an average performance improvement of 32%, and smaller batches performs better on all models. The MobileNet series had the smallest difference between parallel and sequential execution due to its fast execution times.

V. Conclusion

In this paper, we proposed a novel fused-convolution scheduling technique for running multi-workloads on multi-core NPUs. While previous studies have used the method of minimizing dram access by increasing the size of tiles, in this study, we used a strategy of reducing the size of tiles until the dram cycle is close to the compute cycle to reduce latency while freeing up on-chip memory space. In our experiments, we were able to reduce on-chip memory usage by 39% while increasing latency by 13% when running one workload. By allocating other workloads to the freed up on-chip memory space and running them in parallel on a multi-core NPU, we achieved a 32% performance improvement over serial execution.

VI. Acknowledgment

This work was supported by IITP grants funded by the Korea government(MSIT) (2021-0-00310, 2020-0-01361, 2022-0-00498). This work was also supported by the Yonsei University Research Fund (2022-22-0306). Yongjun Park is the corresponding author.

References

[1] J. Fowers *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 1–14.

[2] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1–12. [Online]. Available: https://doi.org/10.1145/3079856.3080246

[3] E. Talpes, D. Williams, and D. D. Sarma, "Dojo: The microarchitecture of tesla's exa-scale computer," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 2022, pp. 1–28.

[4] X. Cai, Y. Wang, and L. Zhang, "Optimus: towards optimal layer-fusion on deep learning processors," in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, 2021, pp. 67–79.

[5] H.-J. Jeong *et al.*, "Pin or fuse? exploiting scratchpad memory to reduce off-chip data transfer in dnn accelerators," in *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 224–235. [Online]. Available: https://doi.org/10.1145/3579990.3580017

[6] T. Chen *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," 2018.

[7] Y. Xing *et al.*, "Dnnvm: End-to-end compiler leveraging operation fusion on fpga-based cnn accelerators," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 187–188. [Online]. Available: https://doi.org/10.1145/3289602.3293972

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," in *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, 2022, pp. 287–290.

[9] H. Kwon *et al.*, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of dnn mappings," *IEEE Micro*, vol. 40, no. 3, pp. 20–29, 2020.

[10] A. Samajdar *et al.*, "Scale-sim: Systolic cnn accelerator simulator," 2019.

[11] S. Ghodrati *et al.*, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 681–697.

[12] J. Lee *et al.*, "Dataflow mirroring: Architectural support for highly efficient fine-grained spatial multitasking on systolic-array npus," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 247–252.

[13] G. Zhou *et al.*, "Research on nvidia deep learning accelerator," in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018, pp. 192–195.

[14] E. Qin *et al.*, "Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020, pp. 58–70.

[15] H. Kwon *et al.*, "Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 461–475. [Online]. Available: https://doi.org/10.1145/3173162.3173176

[16] M. Alwani *et al.*, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[17] O. Moreira *et al.*, "Graph partitioning with acyclicity constraints," *arXiv preprint arXiv:1704.00705*, 2017.