Term Project Report

Group 20: Ekansh Bhatnagar (836000172), Rishal Melita Pereira (635009352)
Muskan Bhusal (335007168), Vaishnav Ganapathiraju (334008346)

***Abstract*—Deep neural network processing is extremely memory intensive due to the parameters for each model being larger than what the local high speed memory can hold, causing performance to be dependent on external memory bandwidth. In this report, we detail our attempt to recreate the results of this paper [3], which presents techniques to reduce redundant DRAM accesses during backward-pass gradient calculations by optimizing computation order to increase temporal locality. The paper under study proposes three techniques to reduce this overhead. Firstly, it introduces interleaved gradient computations. The backward computation consists of two separate computations: input gradient calculation and weight gradient calculation , both of which rely on same output gradient. Interleaving exploits locality by interleaving these two separate computations and reusing the cached output gradient. The paper then goes on to introduce tiling and rearranging. For our project we focus our scope on implementing interleaving with tiling on Small NPU configuration (Ethos-N77 [2]). We implement the proposed technique by adding custom functions to implement tiling and then modifying the computational flow control in the simulator and changing the order of access of the memory tile. We benchmark our implementation with 4 popular neural networks (GoogleNet [1], RESNET50 [4], MobileNet [5] and FasterRCNN [7]). We found that the theoretical performance benefit (not counting the second output gradient read) we can get is around 0.74 times of the original execution time. We also report the simulated execution times after implementing interleaving on the small npu which doesn't show much benefit with just interleaving as Small NPU SRAM sizes are quite small for the benefit to be substantial. We also compare these performance improvements for different available memory bandwidth (22GB/s,11GB/s, and 6GB/s).**

I. Introduction

Deep neural networks(DNN) training is a resource intensive task, also it's in high demand these days due to the requirement of applying machine learning to practical problems around the world. A typical DNN training requires evaluation of gradients for each layer, which consumes huge amount of computational resources. Neural networks are trained by doing both forward and backward passes. The input feature maps and associated weights are used to determine the output gradient for each layer during the forward pass. To determine how the results deviate from the anticipated outcomes, an inference is made in the last layer by comparing the output gradient with the loss function. Both the input gradient and the weight gradient are then determined in the backward pass using these output gradients. During the backward pass, GPUs were the most widely used hardware until recently for training and inference, but nowadays Neural Processing Units(NPUs) have also been introduced as an alternative to GPUs. The major difference between NPUs and GPUs lies with the usage of on-chip scratchpad memory, also called SPM, in NPUs. SPM is managed through software and helps in reducing expensive external memory accesses when model sizes keep increasing. Recent NPU designs are advanced enough to allow flexible tensor storage which can accommodate dynamic training computations. This flexibility has opened up new opportunities in reducing redundant off-chip memory accesses by implementing data reuse within SPM. Neural network training involves two phases: forward pass and backward pass. Backward pass involves heavy computational resources requirements, which this paper aims to optimize. Original paper [3] discusses about three data transformation schemes:

1) interleaving input and weight gradient computations to reuse output gradients within SPM
2) optimizing tile access orders based on gradient tensor dimensions
3) determining optimal data partitioning and mapping schemes for single-core and multi-core NPUs

The primary intention is to maximise tile reuse and minimise redundant data storage. In this project, we focus solely on the interleaving technique. The other two methods, which include optimizing tile access order and determining the best data partitioning and mapping scheme, are beyond the scope of this project. Gradient computations(input and weight gradients) during backward pass use the same output gradient, however traditional sequential process performs this computations separately and independently. This approach leads to duplicate memory accesses for output gradients, as it needs to be fetched from memory for each step. The interleaving technique addresses this inefficiency by combining these two operations into a single, interleaved process. The technique as used in this work involves merging these two computations by reordering. By doing so, it enables the reuse of data inside SPM, thus minimizing unnecessary data transfers from external memory. NPUs can achieve better performance and efficient resource utilization by implementing this technique. This is beneficial in the scenarios where memory size and bandwidth is limited.

II. SCALE-sim

SCALE-sim is a systolic array-based simulator designed to simulate convolutional neural networks. By mimicking the data flow through the hardware architecture, it is used to assess how well deep learning accelerators function. The Processing Elements(PEs), which carry out the Multiply-Accumulate (MAC) process, are the main computational units in Scalesim. Every PE has a register file and control logic to regulate operations and data flow. SCALE-Sim stores filter weights, input feature maps (IFMAP), and output feature maps (OFMAP) in a scratchpad memory. In order to minimize the

frequency of DRAM accesses, this memory is implemented as SRAM, which functions as a cache to store temporary data. There are two parts to the simulation: the memory system and the computing system. The computing system generates prefetch, demand, and operand matrices. The matrices used in calculations, such as input feature maps, filter weights, and output feature maps, are known as operand matrices. The necessary input, filter, and output items that must be retrieved from memory for a specific cycle are represented by the demand matrices. Prefetch matrices are used to pre-load data into SRAM before the PEs require it, hence optimizing memory access. SCALE-Sim lowers latency and increases performance by creating prefetch requests that guarantee data is available in on-chip scratchpad memory just in time for processing. This technique anticipates future data needs based on present execution patterns, reducing pauses caused by memory access delays. The memory system is made to effectively control the flow of data between the memory and the processing components. The memory is intended to have both off-chip DRAM and on-chip scratchpad memory. In order to store IFMAP, filter matrices, and output matrices, the on-chip memory is divided. The user-specified size is included in the parametrized memory. Double buffers are used to represent SRAM memory in order to control latency and memory stalls. While the other buffer partition pre-fetches data from the off-chip memory, the first partition works to obtain the necessary data into the PE array. The service memory raises requests for input and filter matrices to be read and requests for output matrices to be written after the demand matrices for PE computations are in place.

In order to maximize data mobility and reuse, Scalesim models three main categories of dataflow strategies: weight stationary (WS), input stationary (IS), and output stationary (OS). Each processing element (PE) in the systolic array is in charge of computing one output pixel for the OS dataflow. In this manner, the output remains constant throughout the PE calculation. Likewise, in input stationary and weight stationary, the feature maps for input and weight stay the same throughout many calculations.

The tool requires two input files: a configuration file and a topology file. The configuration file specifies user-defined architectural parameters, including the height and width of the array, the size of the SRAM memory buffer, offsets for IFMAP, filter, and OFMAP to indicate their starting addresses, DRAM bandwidth, and the type of dataflow to be employed. The topology file is a CSV that details the parameters for various layers within a convolutional neural network (CNN), such as the height and width of IFMAP and filters, the number of channels, the number of filters, and the stride size.

Two input files are needed for the tool: a topology file and a configuration file. The array's height and width, the size of the SRAM memory buffer, the offsets for IFMAP, filter, and OFMAP to identify their starting addresses, the DRAM bandwidth, and the dataflow type are among the user-defined architectural characteristics specified in the configuration file. The topology file, which is a CSV file, contains information about the parameters for different layers of a convolutional neural network (CNN), including the stride size, the number of channels, the number of filters, and the height and width of IFMAP and filters.

III. Problem Description

As discussed, NPU or any other vector architecture is usually throttled due to large latencies associated with off-chip memory access. Reducing these off-chip memory accesses is of huge interest in current research. On such method is explored in the paper under study [3]. As we use ARM's SCALE SIM [8] to implement this interleaving, we need to figure out how a basic SCALE-SIM simulation takes place for backward pass for each layer. We found out that SCALE-SIM simulates following steps for each layer:

1) Generating matrices with required dimensions using the topology for that layer (M,N,K values), and fills them up with values in order of its requirement and adds an offset to it (to avoid different matrices having same values).
2) Generating Prefetch matrices, that list the elements in order of their computations (meaning the first element of input, then the first element of filter and so on).
3) Generating Demand Matrices, that list the order in which the multiplication requires memory accesses.
4) Counts the computation cycles and reads (both SPM and off-chip) required.

The scope of this project is limited to interleaving. We also intent to implement tiling in both baseline and interleaving implementations to replicate the results in the research paper. Vanilla SCALE-SIM doesn't implement tiling for its simulation. Currently a basic backward pass computation consists of the following:

A. Backward Pass Computation

As show in Figure 1, in backward pass, computation starts at the last layer and goes back to the first layer. For a layer i, both input gradient (dX_i) and weight gradient (dW_i) have to be calculated. dX_{i-1} computation depends on dY_{i-1} and W_i, whereas dY_i depends on dY_i and X_i. The functions to calculate dX_i and dW_i depends on the methodology used in neural network i.e. convolution or GEMM,

$$dX_{i-1} = f(dY_i, W_i) dW_{i-1} = g(X_i, dY_i)$$

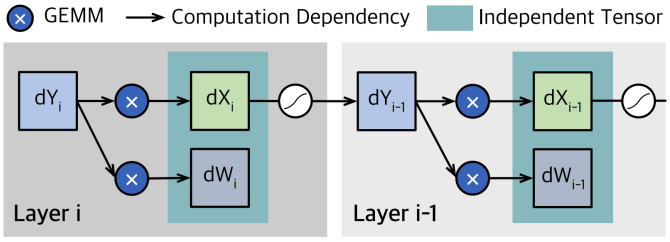

Fig. 1. Backward Pass computation and dependency between subsequent layers

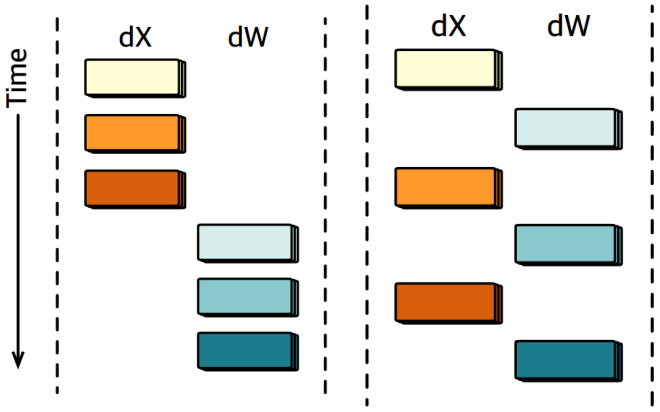

Fig. 2. Sequential computation

Fig. 3. Interleaving computation

These computations are independent for each layer. But most commonly used accelerators like TPUv3 [6] still process them sequentially.

B. Challenges

The challenges in implementing the proposed technique [3], are listed below:

1) Creating topologies for selected benchmarks in GEMM format : SCALE-SIM only provides the topologies in convolution format.
2) Generating correct configuration files : Our project focuses on implementing small NPU configuration i.e. Ethos-N77 [2]. We need to find following configuration parameters:
   a) Row Size of Processing Elements
   b) Column Size of Processing Elements
   c) Input SRAM Size
   d) Filter SRAM Size
   e) Output SRAM Size
   f) Input Offset
   g) Filter Offset
   h) Output Offset
   i) Bandwidth
   j) Dataflow
   k) Memory Banks
3) Implementing tiling for both input and filter matrices.
4) Generating Operand Matrices : Setups the row-wise or column-wise retrieval of matrices elements)
5) Generating Prefetch Matrices : Setups the order in which computation will be done.
6) Generating Demand Matrices : Setups the order in which the elements are fed to the memory system.
7) Figuring our the hit and miss latency of the SPM.
8) Figuring out how the memory system counts the reads and compute cycles after implementing tiling.
9) Calculating the Read Ratio and Read+Write Ratio for figuring out the theoretical maximum benefit we can get.
10) Simulating for both baseline+tiling and interleaving+tiling for comparison

IV. Proposed Solution

The output gradient presents an opportunity to decrease memory reads at each layer, as it is necessary for computing both the input gradient and weight gradient. Due to limited size of Scratchpad Memory (SPM), not all the data required for these calculations can be accommodated. Implementing a solution that uses the locality of SPM and only fetches the output gradient a single time for both computations should get us some performance benefits. Maximum benefit that can be achieved using this can be easily calculated by removing all output gradient reads from the total number of reads. The results of such an experiment are shown in the Section VI.

For our implementation, we decided to implement tiling as well, as the paper [3] used tiling in its baseline and interleaving implementations. Tiling also makes the interleaving more efficient to implement. To use Tiling with SCALE-SIM, we need to implement custom functions and make changes in the simulation flow of SCALE-SIM to call functions for both interleaving and baseline. Then, we passed on our custom demand and prefetches matrix to the basic SCALE-SIM memory system to ensure proper counting of latency (compute cycles) and reads. Later we also calculated results without tiling, but as expected they are found not to be ideal to compare with the results of the paper under study [3].

A. Tiling Implementation

Tiling is a technique frequently used to optimize the memory and computational efficiency in Neural network accelerators [9]. In tiling, we partition large matrices, such as input feature matrix, filter matrix and output matrix. The size of these tiles are decided by keeping the dimensions of these matrices and SPM size in mind. As tiles bigger than SRAM are not beneficial, while smaller tiles might increase execution time. These tiles can be processed sequentially or in parallel according to the implementation constraints. Creating tiles with optimized size can help facilitate data reuse. Without tiling the benefits of interleaving would be vastly minimized, because one single matrix could fill up the SPM of a small NPU. Tiling here allows us to reuse and keep tiles of dY in SPM and perform matrix multiplications as needed. Also, tiling distributes the computations across processing elements (PEs) for effective parallel execution, thus reducing latency. We used tiling technique mentioned in the paper [9]. Figure 4 explains the concept of tiling for convolution, though we are not using convolution the process remains the same..

First, filter matrices and input matrices are divided into smaller tiles; $F_{0,0}$, $F_{0,1}$, $F_{1,0}$, $F_{1,1}$ and $I_{0,0}$, $I_{0,1}$, $I_{1,0}$, $I_{1,1}$ respectively. Each tile contains a submatrix of the original matrix, this smaller chunk of data enables it to be much smaller than on-chip memory. Then each tile is used to create a partial tile of output matrix.

$$F_{0,0} \times I_{0,0} \rightarrow O_{0,0,A} F_{0,1} \times I_{1,0} \rightarrow O_{0,0,B}$$

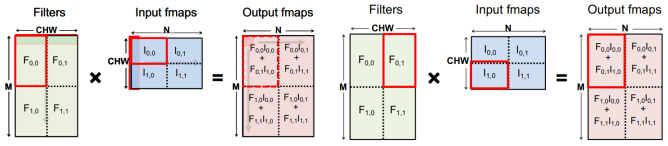

Fig. 4. Tiling process

In the example shown in figure 4, only two partial terms are used to create a single time for the output matrix.

$$O_{0,0,A} + O_{0,0,B} = O_{0,0}$$

Following this pattern of computation ensure all input and weight tiles are accounted in creation of output tile.

For our implementation, we decided on a scaling factor s to calculate the tile sizes dynamically using the SRAM size and dimensions of input, weight and output matrices. This scaling factor is calculated by using the forumula below:

$$s = \left\lceil \sqrt{\frac{mn + nk + km}{SRAM_{SIZE} \times 0.5}} \right\rceil$$

Then the tiling size for each matrix is $rows/s \times columns/s$. We do this tiling for both input and filter matrices, and get partial parts of each tile in output matrix.

We then looped over each tile of input matrices and weight matrices in a nested loop. We then figured out whether tile A (from input matrix) should be multiplied to tile B (from weight matrix) and which output tile it corresponds to.

B. Simulation Order Algorithms

1) Create Operand Matrices

Operand matrices are created based on simulation data, defining the addresses of all involved data elements.

2) Create Demand Matrices

- **For weight gradient computation**, the required tensors include:
  - Input tensors
  - Output gradient tensors

$$dW_i = X_i^T \times dY \tag{1}$$

- **For input gradient computation**, the required tensors include:
  - Output gradient tensors
  - Weight tensors

$$dX_i = dY_i \times W_i^T \tag{2}$$

In the baseline method, demand matrices are generated independently for input gradient and weight gradient calculations. This requires iterating over tiles of input feature maps, filters, and output feature maps. For input gradient computation, demand matrices are created using output gradient dY_tile and filter Wt_tile tensor tiles, while for weight gradient computation, demand matrices are created with input tensor tiles Xt_tile and output gradient tensorsdY_tile. As both of these compuations share the same output gradient tensor tiles dY_tile, they appear to be accessed separately. Since this access is sequential in nature, the dY tiles used for weight gradient computation might have been evicted from the SRAM by the time they are supposed to be reused for input gradient computation. This results in additional fetches of the same dY tiles from DRAM during weight gradient calculation, adding overhead and decreasing efficiency.

Algorithm 1: Baseline Computation After Tiling

Require: 5 input matrices: dY, Wt, Xt, dX, dW
Ensure: Generate Demand matrices for input and weight gradients

1: **Input Gradient Computation:**
2: **for** each tile in dY, Wt, dX **do**
3: Ip_dem_mat, Flt_dem_mat, Op_dem_mat = create_demand_matrix(dY_tile, Wt_tile, dX_tile)
4: **end for**
5: **Weight Gradient Computation:**
6: **for** each tile in (Xt, dY, dW) **do**
7: Ip_dem_mat, Flt_dem_mat, Op_dem_mat = create_demand_matrix(Xt_tile, dY_tile, dW_tile)
8: **end for**
=0

To address this inefficiency, an interleaving technique is used, which rearranges demand matrices such that the dY tiles used in weight gradient calculations are immediately reused for input gradient calculations. This approach ensures efficient reuse of shared data while it's present in SRAM, and reducing the need for repeated data fetching from DRAM. The interleaving algorithm processes five tensor tiles—dY_tile (output gradient), Wt_tile (weight tensor), Xt_tile (input tensor), dX_tile (input feature map), and dW_tile (weight gradient) and generates demand matrices for both input and weight gradients by iterating over each tile.

Algorithm 2: Interleaving with Tiling

Require: 5 input matrices: dY, Wt, Xt, dX, dW
Ensure: Generate Demand matrices for input and weight gradients

1: **for** each tile in dY, Wt, dX, Xt, dW **do**
2: **Input Gradient Computation:**
3: Ip_dem_mat, Flt_dem_mat, Op_dem_mat = create_demand_matrix(dY_tile, Wt_tile, dX_tile)
4: **Weight Gradient Computation:**
5: Ip_dem_mat, Flt_dem_mat, Op_dem_mat = create_demand_matrix(Xt_tile, dY_tile, dW_tile)
6: **end for**
=0

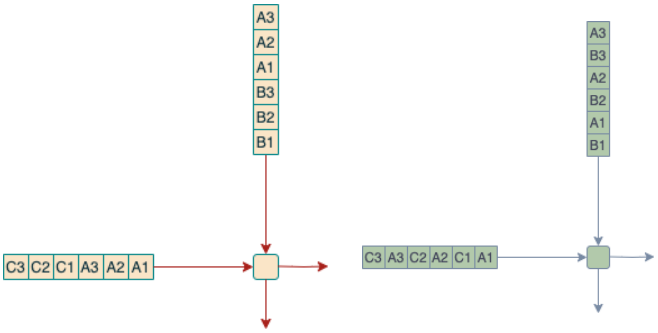

Fig. 5. Baseline

Fig. 6. Interleaving

As described in the Algorithm 2, the interleaving strategy integrates computations within the same loop iteration to ensure that the common data elements like dY tiles are reused across different gradient calculations. It not only improves memory efficiency but also enhances performance by reducing the latency associated with data transfers between different memory hierarchies. The algorithm ensures better resource utilization and faster processing times through this optimization in sequence of memory accesses in neural network accelerators. Figures 5 and 6 illustrate how the changes in the formation of demand matrices affect the dataflow during computation. In this context, the matrices A, B, and C represent the output gradient (dY), input (Xt), and weight (Wt) matrices, respectively. The baseline computation, as shown in Figure 5 operates in two steps. First, the output gradient tiles (A1,A2,A3) are sent to the processing elements from the left, and then input tiles (B1,B2,B3) are provided from the top for the computation. Once all necessary input tiles are processed for weight gradient calculation, the system proceeds to process the weight matrix tiles (C1,C2,C3), which are fed from the left. Simultaneously, output gradient tiles (A1,A2,A3) are fed from the top and input gradients calculation is completed. A significant drawback in this sequence is the reuse pattern of output gradient tiles. Although the same tiles (A1,A2,A3) are required for input gradient calculations, they are often evicted from the SRAM during the weight gradient computation phase, and then reloaded to SRAM during input gradient calculation. This eviction is caused by limited SPM capacity, where the subsequent tiles of output gradient tend to replace the tiles used at the beginning of weight gradient computation. Thus these tiles have to reloaded from DRAM during weight gradient computation, which introduces significant memory access overhead and reduces the overall efficiency. Figure 6 presents an interleaved computation strategy, which is designed to optimize the memory access by modifying the demand matrix formation step in scale sim. Here, operations sequence need to be modified such that the common dY tiles is reused immediately for input gradient computations after the weight gradient calculations, while they are in SRAM. The output gradient tiles(A1, A2, A3) are now reused as filter inputs immediately after being utilized for weight gradient computations. Since immediate reuse of tiles is prioritized in this strategy, it eliminates the overhead associated in evicting and reloading the output gradient files in SRAM.

V. Evaluation Methods

Scalesim is used for the simulation of all neural network layers. A cycle-level simulator is developed which implements tiling and performs layer wise simulation. Convolution layer computations from obtained from Pytorch are converted into GEMM operations using im2col method. NPU configuration is used as given in Table I. These configurations model small NPU based on ARM Ethos N77 [2]. THe DNN models used are listed in Table

TABLE I
NPU configurations [2]

Parameters	Values
# Compute units	1 x (45 x 45 PE)
DRAM Bandwidth	22 GB/s
Frequency	1GHz
IfMap Scratchpad Memory	341KB
Filter Scratchpad Memory	341KB
OfMap Scratchpad Memory	341KB
Batch size	1

TABLE II
List of Models

RCNN
NCF
ResNet50
GoogleNet
MobileNet

VI. Results

We modify the SCALE-sim files in order to reproduce the findings reported in the paper. In the backward pass, retrieving input and weight tensors takes less time than accessing the output gradient memory. We obtain data on output gradient traffic relative to the overall traffic involved in retrieving input and weight gradients. A thorough CSV file called a detailed access report is produced once the simulation is finished. The start and stop cycles for SRAM and DRAM associated with reading and writing activities for IFMAP, filter, and OFMAP are detailed in this report. The dy read with respect to total read of all tensors and read+write traffic with respect to total read+writes of all tensors is computed using these parameters. The baseline architecture used in SCALE-sim does not take tiling for general matrix-matrix multiplication into account. First, we obtained results for various CNN models for output stationary dataflow. The dy read ratio and dy read+write ratio accounts for 39.3% and 35.1% on average for total read and total read and writes as shown in Fig.7. Similarly from Fig.8, we obtain results for dy reads and read+write ratios for weight stationary dataflow which on average accounts for 26.7% of total read data and 20.53 of total read and writes respectively. We observe that the averages decrease from the

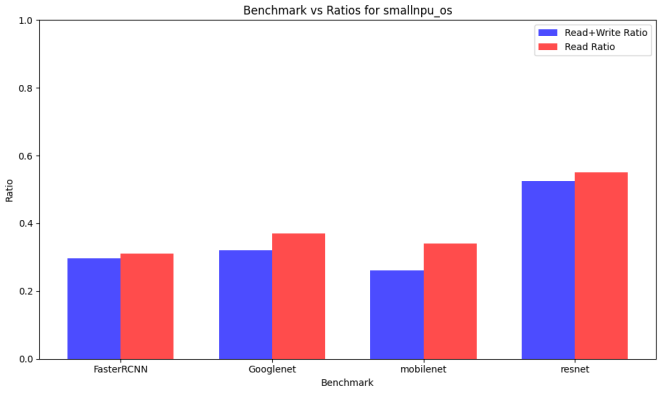

Fig. 7. dY traffic as a proportion of overall data traffic (OS dataflow)

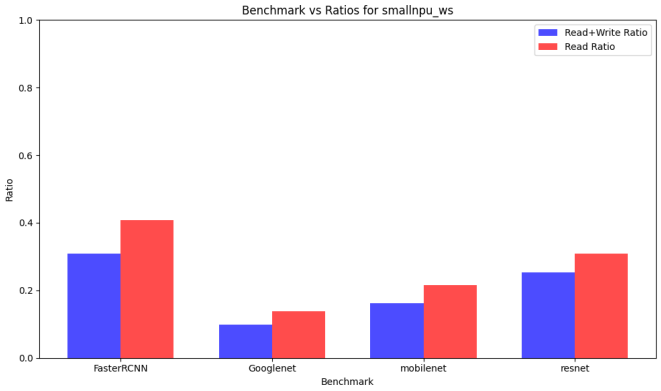

Fig. 8. dY traffic as a proportion of overall data traffic (WS dataflow)

paper because our project only uses four models and we use small NPU configurations. Next, we plot the normalized execution time that is computed for both weight stationary and output stationary. Here, we make an assumption that dY is fetched only once. From fig.10, the execution time is 0.89x the baseline architecture for WS and 0.93x in OS as shown in fig.11.

Second, we obtain the above results for dy read and read+write ratios after implementation of tiling opeartion in general matrix-matrix multiplication in our baseline architecture. From fig.11 The dy read is 42.3% of total reads and read+write ratios 33.3% of the total reads and writes. The execution time for baseline architecture with tiling is 0.72x the execution time with tiling as presented in fig.12. The average normalised execution time after interleaving process is nearly 0.98. Furthermore, we compute the effect of DRAM bandwidth changes on interleaved computation implementation. The DRAM bandwidth configuration used is 22GB/s(full bandwidth), 11GB/s(half bandwidth) and 6GB/s(quarter bandwidth). The normalised execution time improves for half-bandwidth configuration but not so much for quarter bandwidth.

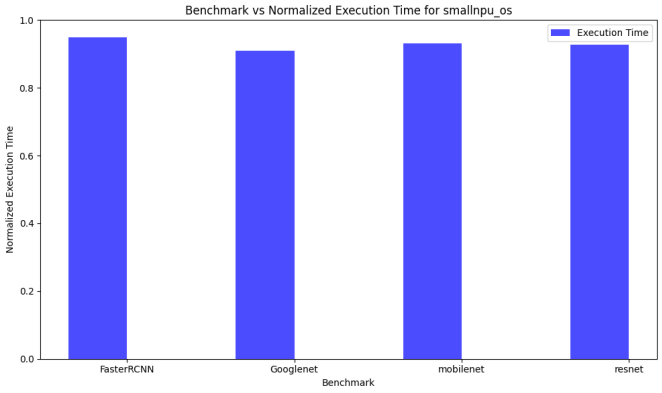

Fig. 9. Normalized Execution Time from eliminating dY DRAM reads during weight gradient calculation (OS dataflow)

A. *Inferences*

We included our results for both output and weight stationary dataflows, though the most interesting dataflow for our implementation is output stationary. We implemented the custom simulation order and tiling for output stationary data flow. Figure 7 show the ratio of output gradient traffic for both Read+Write Ratio. Our experimental results for FasterRCNN, GoogleNet and Mobilenet closely follow the trends shown in paper under study [3]. RESNET50 shows little differnce from the results in the paper. This might be due to RESNET50 being a very large model in comparison to other benchmarks we tested. Our configuration is using small NPU, hence, its very likely the larger models may not benefit as much. Similar trend is seen in Figure 9. Because of the small SRAM size, we do not observe any performance increase for execution time with merely interleaving process. Only the resNet, mobilenet, and googleNet models exhibit a decrease in the normalized execution time for half bandwidth with interleaved operation among the three bandwidth configurations. The execution time for quarter bandwidth leads to poor normalized execution time since the bandwidth is so low that the interleaved process has no or detrimental effect.

VII. Conclusion and Future Work

The results demonstrate correct trends with respect to our implementation of interleaving and tiling. The slight deviation from the results in the paper [3] is due to setup differences. As we have our own implementation of tiling, it might differ from the implementation in the paper. The modifications we made to SCALE-SIM now allows us to easily implement rearranging. We have implemented rearrange though it was beyond the scope of our project. We have included the code for rearranging in our artifacts submission. However, the simulation for rearranging implementation couldn't be completed within the limited time frame. The paper under study focused on maximizing data reuse to reduce off-chip memory accesses. There are many other techniques under current research to implement more advanced methods to achieve better results.

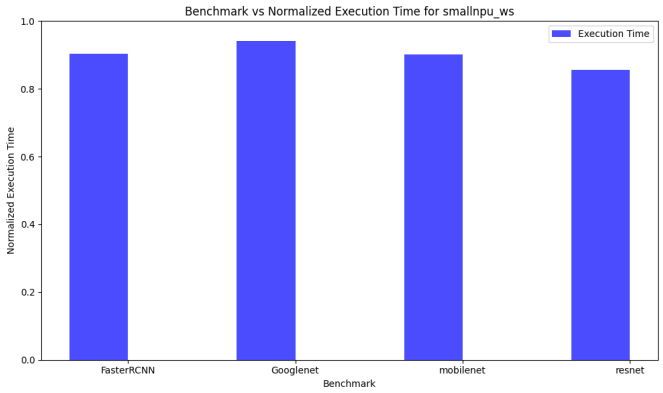

Fig. 10. Normalized Execution Time from eliminating dY DRAM reads during weight gradient calculation (WS dataflow)

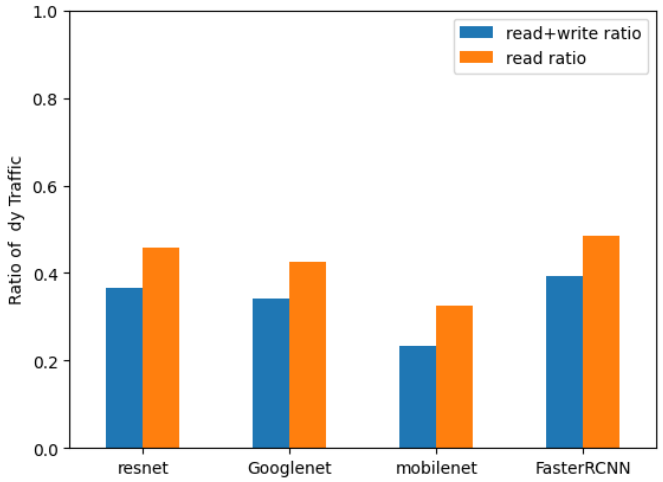

Fig. 11. dY traffic as a proportion of overall data traffic (OS with tiling)

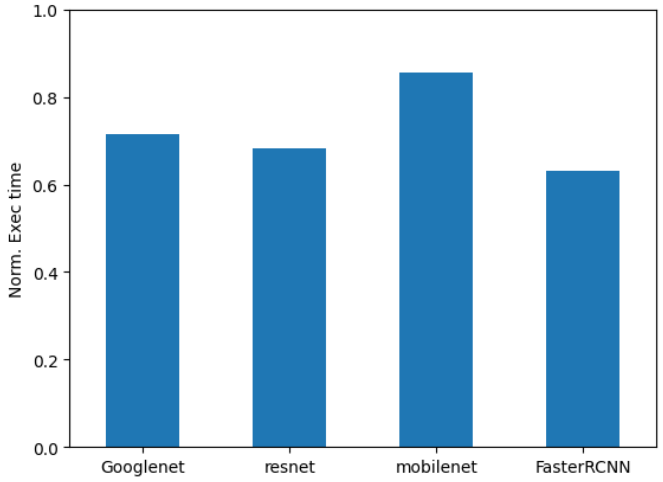

Fig. 12. Normalized Execution Time from eliminating dY DRAM reads during weight gradient calculation (OS with tiling)

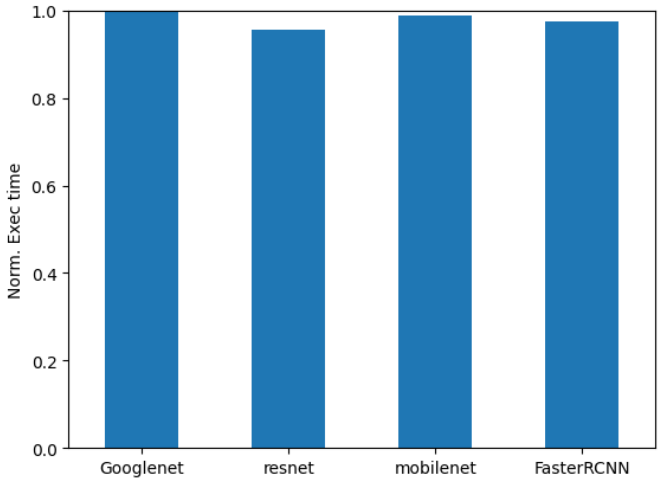

Fig. 13. Normalized Execution Time using interleaving, over baseline

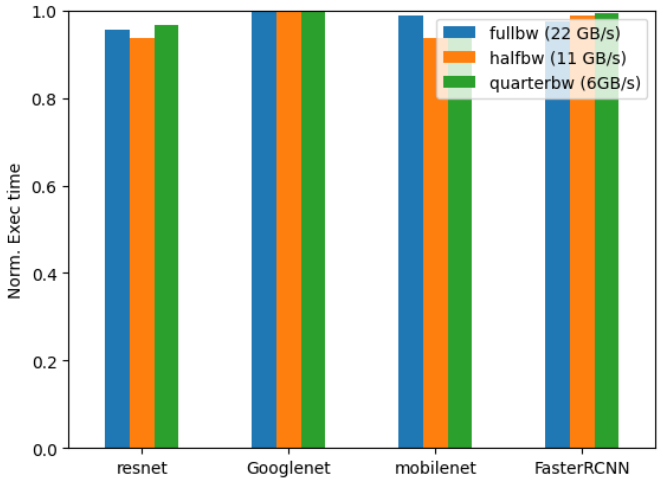

Fig. 14. Normalized Execution Time using interleaving, over baseline, for different bandwidths

Future work could focus on implementing rearranging and exploring these advanced techniques. Configuration assigned to our group was a small NPU, this also contributes in not getting much benefit from just interleaving as the on-chip memory and PEs are too small. We can easily extend our implementation to simulate other larger NPUs, and expect better performance benefits when using larger NPUs.

VIII. Work Divison

1) **Ekansh Bhatnagar (836000172): 25%**
   - Contributed to python script generating valid topologies in GEMM format.
   - Contributed in developing implementation functions for performing tiling.
   - Contributed in creation of custom functions for creating prefetch, demand and operand matrices.

- Contributed in creating automation scripts to run all configurations and topologies on Grace (HPRC)
- Contributed to report (Sections III and IV).

2) **Vaishnav Ganapathiraju (334008346): 25%**
   - Contributed to creating functions to convert convolution topologies to GEMM.
   - Contributed in creating custom simulation order flow algorithms for baseline and interleaving.
   - Contributed in combining tiling with baseline and interleaving flows.
   - Contributed in creating automation scripts to run all configurations and topologies on Grace (HPRC)
   - Contributed to report (Section V and VI)
3) **Muskan (335007168): 25%**
   - Contributed in understanding of the memory system implementation in scalesim and strategies to implement tiling
   - Contributed in creating scripts to convert convolution layer topologies to MNK format
   - Helped with creating automation scripts for running topologies on Grace
   - Contributed to report Sections IV, V and VII
4) **Rishal (635009352): 25%**
   - Contributed in understanding of SCALE-sim architecture and systolic array based computation.
   - Contributed in creating scripts to calculate output gradient read and read+write ratios
   - Contributed to the computation and ideation to derive mnk parameters from CNN input, output feature map and filter parameter.
   - Contributed to report(Sections I, II, and VI) and presentation slides

REFERENCES

[1] R. Anand, T. Shanthi, M. Nithish, and S. Lakshman, "Face recognition and classification using googlenet architecture," in *Soft Computing for Problem Solving: SocProS 2018, Volume 1*. Springer, 2020, pp. 261–269.

[2] ARM, "Powering the edge: Driving optimal performance with ethos-n77 processor," ARM, Technical Report, 2019.

[3] J. Kim, S. Na, S. Lee, S. Lee, and J. Huh, "Improving data reuse in npu on-chip memory with interleaved gradient order for dnn training," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. Toronto, Canada: ACM, October 28-November 1 2023.

[4] B. Koonce and B. Koonce, "Resnet 50," *Convolutional neural networks with swift for tensorflow: image recognition and dataset categorization*, pp. 63–72, 2021.

[5] U. Kulkarni, S. Meena, S. V. Gurlahosur, and G. Bhogar, "Quantization friendly mobilenet (qf-mobilenet) architecture for vision based applications on embedded platforms," *Neural Networks*, vol. 136, pp. 28–39, 2021.

[6] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. Jouppi, and D. Patterson, "The design process for google's training chips: Tpuv2 and tpuv3," *IEEE Micro*, vol. 41, no. 2, pp. 56–63, 2021.

[7] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 6, pp. 1137–1149, 2016.

[8] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.

[9] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks*, ser. Synthesis Lectures on Computer Architecture. Springer Cham, 2020.