Block Group Scheduling: A General Precision-scalable NPU Scheduling Technique with Capacity-aware Memory Allocation

Seokho Lee[1], Younghyun Lee[1], Hyejun Kim[2], Taehoon Kim[1], and Yongjun Park[2]
[1]Hanyang University, [2]Yonsei University
{seokholee, younghyunlee, ted6345}@hanyang.ac.kr, {gpwns3717, yongjunpark}@yonsei.ac.kr

***Abstract*—Precision-scalable neural processing units (PSNPUs) efficiently provide native support for quantized neural networks. However, with the recent advancements of deep neural networks, PSNPUs are affected by a severe memory bottleneck owing to the need to perform an extreme number of simple computations simultaneously. In this study, we first analyze whether the memory bottleneck issue can be solved using conventional neural processing unit scheduling techniques. Subsequently, we introduce new capacity-aware memory allocation and block-level scheduling techniques to minimize the memory bottleneck. Compared with the baseline, the new method achieves up to 2.26× performance improvements by substantially relieving the memory pressure of low-precision computations without hardware overhead.**

***Index Terms*—Precision-scalable MAC, NPU, Roofline**

I. Introduction

Owing to the rapid increase in the sizes and computation amounts of deep learning models, researchers have focused on reducing the memory usage of weight parameters on target models. Among them, quantized neural network (QNN) models effectively compress original models to obtain low-precision parameter data with slight accuracy degradations. Despite model size minimization, QNN-specific neural processing unit (NPU) hardware, which supports rapid low-precision computation natively, is required to obtain high inferencing performance. Precision-scalable NPUs (PSNPUs) are a specialized hardware for efficient QNN computation support, by processing multiple low-precision operations in parallel. They modulate the number of parallel computations in a cycle based on the needed precision of the target operation by adding reconfigurable logics to existing fixed-width multipliers in their basic processing elements (PEs). Therefore, the native support capability of multi-precision operations enables PSNPUs to improve processing speeds with low power and memory overhead.

However, recent model trends have changed to decrease the operational intensity of the main computation workload by reducing computational loads. For example, recent convolutional neural network (CNN) models include point- and depth-wise convolutions, and Transformers solve natural language processing and computer vision problems by adopting matrix multiplication to the main computation pattern.

PSNPU operations are more memory bound than general NPUs with fixed precision support. Specifically, the computational intensity decreases with the lower precision of the target operations, but the amount of required memory increases significantly as more data elements are fetched in each cycle to support a growing number of low-precision operations. These characteristics are not a significant problem for typical CNNs owing to the high computational intensity of convolution operations. However, matrix multiplication, which is the main computation pattern in recent model trends, has a lower operational intensity than conventional convolution operations. Therefore, matrix multiplication with low-precision data on PSNPUs can result in severe memory bottlenecks.

Previous studies pertaining to PSNPUs focused primarily on minimizing the area and power overhead of the additional reconfigurable logic. However, the fundamental problem of PSNPUs is its inability to support high memory requirements while minimizing processing performance degradations. Adding faster memory interfaces or larger on-chip memories to PSNPUs is not a feasible solution as they typically incur expensive hardware overhead costs.

In this study, we first analyze whether the memory bound problem can be solved using classical scheduling techniques currently used with fixed-precision NPUs, and if not, the amount of additionally required on-chip memory size to solve memory bottleneck. We then introduce new NPU scheduling techniques that minimize the memory bound problem with given on-chip scratchpad memory: 1) capacity-aware memory allocation (CMA) and 2) block-level scheduling. Our scheduling techniques allow performance to be improved from two aspects: 1) data reuse maximization and 2) effective data prefetching. CMA allows more elements to be stored in the on-chip memory by increasing the total utilization of the input scratchpad (IS) memory. Using the CMA technique, fewer data movements are required to perform the same number of tasks than the baseline owing to data reuse. Meanwhile, block-level scheduling improves data prefetching by consolidating multiple unit operations.

To evaluate the effects of the new scheduling techniques, we apply the PSNPU on RISC-V environment using Gemmini NPU platform [1]. Based on the FPGA-based performance, the proposed PSNPU scheduling techniques increase computation speeds by up to 2.26× over the baseline.

II. Background

Precision-scalable multiply–accumulate (PSMA) enables deep neural networks (DNNs) to process faster with less energy by performing more element operations in parallel as precision decreases. PSMA is categorized into two types of

978-3-9819263-7-8/DATE23/© 2023 EDAA

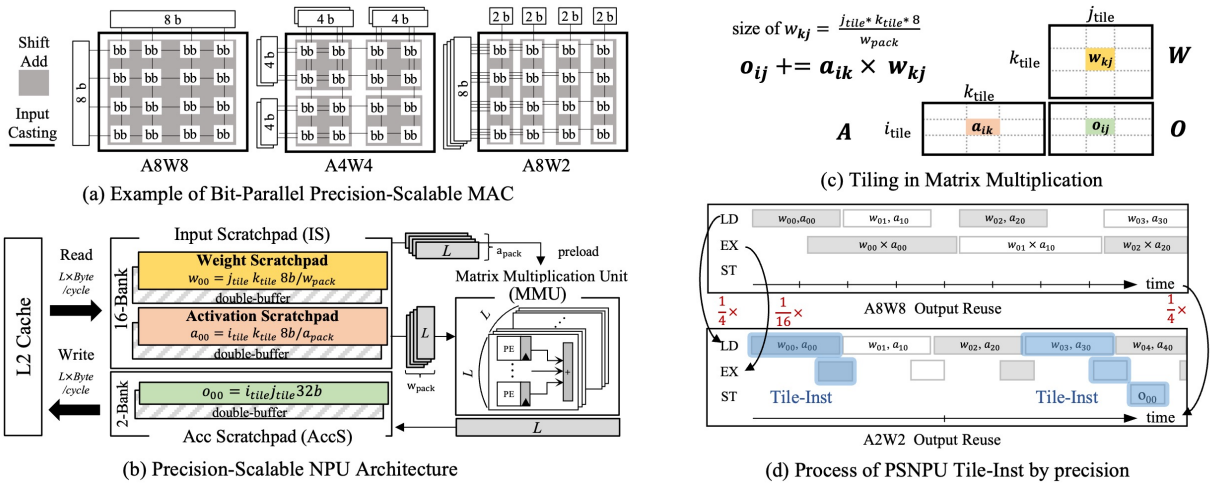

Fig. 1. (a) (n-bit, m-bit) PSNPU operation: $AnWm$, where n, m can be $\{8, 4, 2\}$. The term "8b" denotes 8-bit; (b) Number of PEs: L^2. Total size of Input and Acc scratchpads: IS and AccS, respectively; (c) Tiling in $O[I][J] = A[I][K] \times W[K][J]$ matrix multiplication; (d) a load (LD), execute (EX), or store (ST) tile composed of the same types of low-level instructions. Tile-level instructions comprise the LD and ST required to execute an EX tile.

architecture: bit-serial and bit-parallel. Bit-serial architectures perform a binary operation of multiple elements in several cycles on a single multiply–accumulate (MAC) unit, whereas bit-parallel architectures perform multiple low-precision MACs in a single cycle on a single MAC unit. Bit-parallel architectures (e.g., Bitblade [2]) have recently shown superior area and energy efficiency [3]. Therefore, we target the inefficient use of hardware resources in bit-parallel precision scalable NPUs that can operate with 8-, 4-, and 2-bit data for both activations and weight parameters.

Fig. 1a shows the PE of BitFusion [4], a representative bit-parallel PSMA. One PE comprises 16 bit-bricks (bb) that perform a 2 × 2-bit multiplication operation, and bit bricks belonging to the same shift–add region manage one activation and one weight. All shift–add outputs from the PE are added to create PE outputs. For example, if the activations are 8-bit, and the weight precision is 2-bit (A8W2), the shift–add regions are grouped by four bit-bricks, and one PE can perform four (activations × weight) MACs. In this case, the PE requires 40 bits per cycle (activation: 8-bit × 4 = 32 bits; weight: 2-bit × 4 = 8 bits). Therefore, to achieve faster low-precision PSNPU operations, more elements must be fetched than 16 bits (activation: 8-bit × 1 = 8 bits; weight: 8-bit × 1 = 8 bits) required by the A8W8.

The structure of the matrix multiplication unit (MMU) is categorized primarily into a systolic spatial array (TPU-like) [5] and a parallel vector engine (NVDLA-like) [6]. TPU-like NPUs may exhibit higher clock frequencies than NVDLA-like NPUs while consuming more power with a higher area overhead [1]. Therefore, parallel vector engine architectures have been adopted in many mobile NPUs and their corresponding PSNPUs [2], [6]. The parallel vector engine operates as a weight-stationary data flow with a separate accumulative scratchpad (AccS) to recycle the partial sums computed by the MMU.

Gemmini [1] is an open-source NPU framework that provides a complete solution spanning both the hardware and software stack, on the RISC-V environments. For effective data access latency hiding in SoC environments, Gemmini is designed to support many architectural techniques such as isolated execution access, double buffering, reservation stations, and cache bursts. Gemmini provides three types of low-level instructions (insts) (load(LD), execute(EX), and store(ST)), and tile-level instructions that can reduce instruction fetches by executing multiple low-level instructions (low-insts) per operation. These instructions can be performed separately by using the reservation station's dependency check and double buffering.

Fig. 1d presents an example of tile instruction execution for an A8W8 operation. Each LD, EX, and ST tile comprises multiple low-insts. Moreover, the tile group comprises an EX tile and other tiles (LD and ST tiles) that depend on the EX tile. A tile group can be executed in the form of a tile instructions (tile-insts), and they can operate in parallel. In Fig. 1d, different adjacent tile-insts are distinguished by the brightness of the LD, EX, and ST tiles. By allocating two buffers (double buffers) to the IS, the EX tiles of the current tile group and the LD tiles of the next one are executed simultaneously.

To exploit the advantage of Gemmini's effective data-access latency hiding, the Gemmini architecture was redesigned for the PSNPU, as shown in Fig. 1b. The PSNPU constructed in this study comprises an MMU with a Bitblade [2] architecture, and it offers 8-, 4-, and 2-bit operations for both activation and weight. It features nine operation modes. Additionally, the EX low-inst that controls the MMU operation is customized so that the precision of the activation and weight can be specified. The instruction length can be reduced by sharing multiple IS row addresses to be accessed by low-precision operations.

To support multiple low-precision operations, the number of banks in the IS was increased, and to use the IS efficiently, multiple small bit-width data were stored by packing them into 8-bit data. Specifically, the number of banks in an IS was increased from 4 to 16, and the IS areas for activation and weight were separated to avoid bank conflicts, which typically occur when activation and weight data are stored in the same bank. Additionally, we pre-packed the elements under 8-bit widths and stored partial sums (Psums) in DRAM by compressing them to one of 8-, 4-, or 2-bit precisions, thus allowing the 8-bit IS storage space and DRAM bandwidth to be used effectively. However, even if the bit width of Psums can be reduced in low-precision operations, the sums are stored as 32-bit data, same as the original Gemmini, because the

extra memory space of the AccS in low-precision operations is not as large as the extra memory space of IS. Furthermore, to exploit the extra memory space of the AccS, the memory controller must have an additional hardware logic overhead, such as having more banks.

III. Precision-Scalable NPU Analysis

In low-precision operations, as discussed, the PSMA must process not only more elements but also more bit-level data. In Fig. 1d, the characteristics of the PSMA are shown based on a comparison between the A2W2 and A8W8 operations. As shown in the examples, each A2W2 operation requires four times less data than each A8W8 operation; however, the operation speed is 16 times faster, and the data should be fetched much faster in A2W2. Therefore, as a lower precision operation requires a greater amount of data to perform the operation on the PSNPU, it becomes more memory bounded.

PSMA is designed to compute low-bit data rapidly, with the disadvantage of the reconfigurable logic (shift-add) area overhead. If low-precision operations are memory bounded, and their maximum computation speed is not achievable, the maximum performance will not be increased at the expense of the reconfigurable area overhead. Therefore, one must analyze whether conventional scheduling can solve the memory bottleneck of low-precision operations; if it cannot, then one must determine the amount of additional local memory required by PSNPUs, compared with general NPUs.

We used the roofline model [7] to analyze the performance bottleneck of the PSNPU. The roofline model provides valuable insights into performance bottlenecks and estimations of performance based on the operational intensity (OI), which can be determined at its application.

In this section, based on the system parameters defined in Fig. 1, we analyze the correlation among the tile size, precision, and OI of a matrix multiplication operation, which is frequently used in transformer-based network models.

A. Tiling and Operational Intensity of PSNPUs

Fig. 1c shows the tile group structure of matrix multiplication ($O[I][J] = A[I][K] \times W[K][J]$). A large tile group can reduce DRAM access and hide data transfer latency. Therefore, $i_{\text{tile}}, j_{\text{tile}}$, and k_{tile}, which determine the sizes of a, w, and o, should be the maximum values with limited on-chip memory. Moreover, each a, w, and o tile should use half of each IS and AccS to apply the double buffering scheme, as shown in Fig. 1b. For a_{pack} and w_{pack}, which are the ratios of (maximum n, m) and (n, m in $AnWm$) of the A and W matrices, respectively (also the ratios of the maximum and current bit widths of the matrices), and the elements are stored as 8- and 32-bit data types in the IS and AccS, respectively, Eq. (1) must be satisfied.

$$\left(\frac{i_{\text{tile}}}{a_{\text{pack}}} + \frac{j_{\text{tile}}}{w_{\text{pack}}}\right) k_{\text{tile}} \cdot 8 \le \frac{IS}{2}, \quad i_{\text{tile}}\, j_{\text{tile}} \cdot 32 \le \frac{AccS}{2} \tag{1}$$

Because PSNPUs have different computational characteristics depending on precision, we analyzed the OI based on the number of binary operations, instead of the number of integer operations. This is because a binary operation can reflect the bit width of the elements, which is related to the computation characteristics.

Because we intend to analyze the overhead of the local memory size in the PSNPU by comparing it with that of the 8-bit fixed-precision NPU, we assume that $i_{\text{tile}}, j_{\text{tile}}$, and k_{tile} are the optimal tile sizes in the A8W8 operation. We analyzed the OI, which depends on precision, without scheduling optimization based on the following OI equation:

$$\begin{aligned} OI &= \frac{\text{Binary Operations}}{\text{(A, W, Psum)Read Bits + (Psum)Write Bits}} \\ &= \frac{i_{\text{tile}}\, j_{\text{tile}}\, k_{\text{tile}} \cdot 64/(a_{\text{pack}}\, w_{\text{pack}})}{(i_{\text{tile}} k_{\text{tile}} \cdot 8/a_{\text{pack}} + j_{\text{tile}} k_{\text{tile}} \cdot 8/w_{\text{pack}} + 2 \cdot i_{\text{tile}} j_{\text{tile}} \cdot 8)} \\ &= \frac{1}{8}\left(\frac{w_{\text{pack}}}{j_{\text{tile}}} + \frac{a_{\text{pack}}}{i_{\text{tile}}} + \frac{2 w_{\text{pack}} a_{\text{pack}}}{k_{\text{tile}}}\right)^{-1} \end{aligned} \tag{2}$$

In Eq. (2) and Fig. 1, each tile of matrices A, W, and O is denoted by a_{ik}, w_{kj}, and o_{ij}, and each bitwise size of a_{ik}, w_{kj}, and o_{ij} is $i_{\text{tile}} k_{\text{tile}} 8/a_{\text{pack}}$, $j_{\text{tile}} k_{\text{tile}} 8/w_{\text{pack}}$, and $i_{\text{tile}} j_{\text{tile}} 8$, respectively. Psum is saved in AccS as $i_{\text{tile}} j_{\text{tile}} 32$, however, Psum is moved to DRAM as $i_{\text{tile}} j_{\text{tile}} 8$.

In A8W8 operation $a_{\text{pack}}, w_{\text{pack}}$ is 1, but $a_{\text{pack}}, w_{\text{pack}}$ increases as the precision decreases. Therefore, if PSNPU has the same OI as a general NPU, $i_{\text{tile}}, j_{\text{tile}}$, and k_{tile} must increase by $a_{\text{pack}}, w_{\text{pack}}$, and $a_{\text{pack}} \times w_{\text{pack}}$, respectively (based on Eq. (2)), which requiring larger IS and AccS, by ($\max(a_{\text{pack}}) \times \max(w_{\text{pack}})$), respectively (based on Eq. (1)).

B. Data Reuse Scheme Analysis of PSNPUs

We show the amount of local memory required for the lowest precision operation to achieve the same OI as the highest precision operation in conventional scheduling. Conventional scheduling can reduce the data footprint and increase OI by reusing one of the A, W, and Psum matrices that already exist in the local memory. For example, the output reuse (OR) scheme reads the input data and weight parameters of all tile-insts and reuses Psum such that the $2 w_{\text{pack}} a_{\text{pack}}/k_{\text{tile}}$ term becomes negligible in Eq. (2), and the weight-reuse (WR) reuses the weight parameters such that $a_{\text{pack}}/i_{\text{tile}}$ becomes negligible.

Herein, we show the operational intensity in a steady state. In Fig. 2b (WR case), the tile-insts in the steady state can compute two o_{ij} from one a_{ik} tile. Therefore, the average OI in the WR case is approximated by the ratio of the data movement to the computational amount of the two tile-insts with and without reading tile a_i. The equations used to estimate the OIs in the OR and WR cases are as follows:

$$OI_{\text{OR}} \approx \frac{1}{8}\left(\frac{w_{\text{pack}}}{j_{\text{tile}}} + \frac{a_{\text{pack}}}{i_{\text{tile}}}\right)^{-1} \tag{3}$$

$$OI_{\text{WR}} \approx \frac{1}{8}\left(\frac{w_{\text{pack}}}{2 j_{\text{tile}}} + \frac{2 a_{\text{pack}} w_{\text{pack}}}{k_{\text{tile}}}\right)^{-1} \tag{4}$$

As discussed, in WR or OR scheduling, the amount of IS is additionally required for A2W2 to exhibit the same OI as A8W8, which can be obtained by comparing Eqs. (1), (3), and (4): To maintain OI, the tile group size in the OR must be increased by a_{pack} and w_{pack} in i_{tile} and j_{tile}, respectively, depending on the operating precision. Furthermore, as shown in Eq. (1), the AccS must be increased by $w_{\text{pack}} \times a_{\text{pack}}$ to

prevent memory bound problems in all precision operations. In WR, $j_{\text{tile}}, k_{\text{tile}}$ must be increased by w_{pack}, $a_{\text{pack}} \times w_{\text{pack}}$, respectively, and the IS and AccS must be increased by $\max(w_{\text{pack}})\{i_{\text{tile}} + j_{\text{tile}} \max(a_{\text{pack}})\}/\{i_{\text{tile}} + j_{\text{tile}}\}, \max(w_{\text{pack}})$, respectively. Compared with the previously analyzed simple scheduling, OR and WR increase the OI and reduce the required IS size, but require more local memory or a higher memory bandwidth than an 8-bit fixed precision NPU. This implies that conventional scheduling cannot fundamentally solve the memory bound problem of PSNPUs and requires more memory bandwidth or local memory.

The memory bottleneck problem of PSNPUs has been mentioned in several previous studies, but has not been solved. According to BitFusion [4], the performance of PSNPUs is bandwidth-sensitive. HAQ [8] assumed that the memory limit of the PSNPU could not be solved and thus created a neural architecture search-based model optimized for the PSNPU by reflecting performance degradation via precision change.

IV. Precision-Scalable NPU Scheduling

Conventional scheduling can generally be applied to any NPUs, regardless of the local memory and target workload size. However, PSNPU typically exhibits substantial performance degradation owing to the memory bounded problem when typical scheduling parameters are assumed. To incorporate more elements into IS memories such that the OI can be improved, we leveraged the large local memory in mobile NPUs for DRAM access reduction and the small data size in low-precision operations. However, the number of elements fetched in conventional tile-level scheduling is still limited. The main reason is that the size of the tile group cannot be increased because the bit width of the partial sum in the AccS remains at the original bit width (typically 32-bit), and prefetch cannot be efficiently performed because of the dependency of the tile groups.

To compensate for the limitations of typical tile-level scheduling, we introduce capacity-aware memory allocation (CMA) and block-level scheduling. These offer two performance enhancement opportunities by allowing more data to be stored in the on-chip memory: 1) more effective data reuse capability and 2) data prefetching. First, CMA can improve data reuse opportunities, which increases OI by performing the same number of computations with fewer data fetches. To utilize the second opportunity, we further introduce a concept of a block group, which comprises a set of tiles that can be collectively prefetched. This allows block-level scheduling to use the memory bandwidth efficiently.

CMA and block-level scheduling are PSNPU scheduling techniques that can be applied to general data reuse-based NPU scheduling. By introducing CMA and block-level scheduling to WR-based scheduling, we successfully solved the memory bound issue of PSNPUs without hardware overhead.

A. Limitation of Tile-level Scheduling in a Large Memory Environment

In energy-constrained mobile environments, mobile NPUs feature large IS memories to minimize DRAM access [6]. In

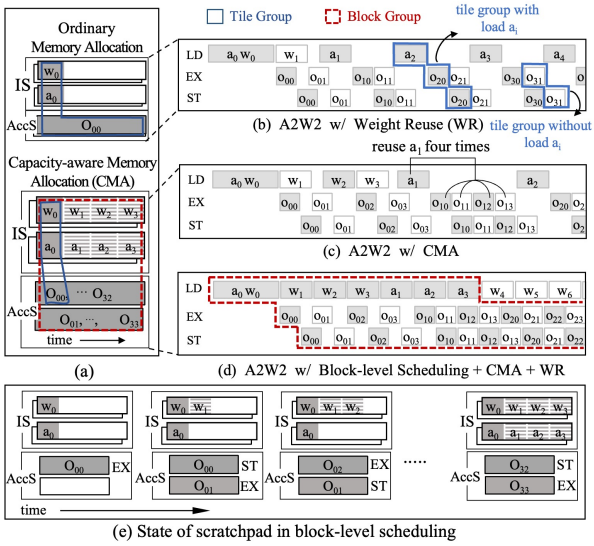

Fig. 2. (a) Memory allocation scheme; (b) A2W2 operation in weight reuse; (c) A2W2 operation in capacity-aware memory allocation; (d) A2W2 operation in block-level scheduling; (e) state of input scratchpad in block-level scheduling.

a large IS memory, most AI model layers are not required to have multiple tiles in the k dimension. Furthermore, in a low-precision condition, tiling is rarely required in the k dimension. Without k tiling, the Psum of read and write operations performed in every tile-inst in WR is not necessary, and OI may increase because the output can be calculated using only a single write to DRAM.

Even when tiling is not applied in the k dimension, WR does not fully utilize the large local memory, owing to the limitations of tile-level scheduling. TVM [9] and Gemmini achieved efficient dependency monitoring and decoupled access executions while encapsulating loads, executions, and stores into a single execution unit and verifying dependencies between groups of tiles in hardware or synchronizing them through virtual threads in software. To efficiently perform dependency verification using HW or SW techniques, the tile-inst should be allocated to half of the IS and AccS. Furthermore, once the spaces in the IS and AccS are allocated, they should not be accessed by other tile groups during the operation.

However, this limitation is unfavorable for the PSNPU for two reasons. First, even though the precision becomes low, the bit width of the output elements stored in the AccS does not change; therefore, the sizes of i_{tile} and j_{tile} cannot be increased by Eq. (1). Unless the k dimension is tiled on A8W8, k_{tile} cannot increase, as shown in Fig. 2a, and the IS will be underutilized. Second, data for the next tile-inst cannot be prefetched because the location of memory allocation is often predefined to a unique address to eliminate complex dependencies [9]. For example, in tile-level scheduling examples, as illustrated in Figs. 2b–c, the load is frequently idle even though memory underutilization is the main performance bottleneck.

B. Capacity-aware Memory Allocation

In low-precision operations, if the IS is small and the workload size is large, then the IS can be fully utilized by increasing k_{tile}. However, if tiling is not applied to the k dimension because of the sufficient IS memory size, then k_{tile} is already maximized to k. Therefore, if tiling is not performed for the k dimension, even in A8W8, the total IS usage rates

based on the precision of the activation and weight parameters are only $1/a_{\text{pack}}$ and $1/w_{\text{pack}}$ at the most, respectively.

It is not possible to increase the tile size without increasing the size of the scratchpad, but reusing previously referenced tiles in another tile-inst can increase OI. CMA creates additional data reuse opportunities by allocating data while preserving data from previous tile groups when additional memory spaces are available.

Fig. 2a shows an illustration of ordinary memory allocation and CMA. Ordinary memory allocation overwrites the current tile group data over data fetched from the previous tile group. However, CMA utilizes additional memory space to obtain more activation and weight parameter data (a_1, a_2, a_3, w_1, w_2, and w_3) to increase the data reuse opportunity. Fig. 2c shows the timeline of the faster computing process via additional data reuse. The first four tile-insts fetch w_j to be reused sequentially later, and their performance does not differ from the initial state shown in Fig. 2b. When a_i is loaded from the fifth tile-inst, as shown in Fig. 2c, it can be computed with all the prefetched w_js. Thus, the same workload can be managed with less data transfer and computed quickly. Therefore, CMA can improve performance by improving OI and reducing power consumption owing to its reduced memory access.

C. Block-Level Scheduling

CMA allows more data to be prefetched without overwriting the memory allocated by other groups of tiles. Next, we introduce a block-level scheduling technique for prefetching memory bounded workloads efficiently. A block group comprises tile groups that can be preloaded via prefetching. In Figs. 2a and d, the block group is represented by a dotted red line, and block-level scheduling is performed based on the block groups. CMA enables the loading of a_1, a_2, a_3, w_1, w_2, and w_3, in addition to a_0 and w_0 which constitute one block group. A key feature of block-level scheduling is that execute and store operations still apply double-buffering per tile group, but load operations apply double-buffering per block group. The performance improvement afforded by effective prefetching can be shown by comparing Figs. 2c and d. Although the main reason for the performance degradation of the PSNPU is a memory bottleneck, the tile-level scheduling shown in Fig. 2c often renders the load unit idle. Meanwhile, the block-level scheduling shown in Fig. 2d can always activate the load unit to maximize memory utilization.

Let us look at the correlation between the block group size and tile group size to obtain the OI of block-level scheduling (OI_{Block}). As the tile size of tile group is expressed as $i_{\text{tile}}, j_{\text{tile}}$, and k_{tile}, the size of the block group is expressed as $i_{\text{block}}, j_{\text{block}}$, and k, respectively. Here, the sizes of i_{block} and j_{block} are i_{tile} and j_{tile} multiplied by at least a_{pack} and w_{pack}, respectively, because the lower the precision, the more tile-insts (a and w) can be fetched via CMA ($i_{\text{block}} \geq i_{\text{tile}} \times a_{\text{pack}},\ j_{\text{block}} \geq j_{\text{tile}} \times w_{\text{pack}}$). Therefore, based on i_{block} and j_{block}, OI_{Block} can be expressed as follows:

$$\begin{aligned} OI_{\text{Block}} &= \frac{1}{8}\left(\frac{w_{\text{pack}}}{j_{\text{block}}} + \frac{a_{\text{pack}}}{i_{\text{block}}} + \frac{a_{\text{pack}} w_{\text{pack}}}{k}\right)^{-1} \\ &\geq \frac{1}{8}\left(\frac{1}{j_{\text{tile}}} + \frac{1}{i_{\text{tile}}} + \frac{a_{\text{pack}} w_{\text{pack}}}{k}\right)^{-1} \end{aligned} \tag{5}$$

Block-level scheduling and WR are orthogonal methods that can be applied simultaneously. The OI when block-level scheduling is applied to WR is as follows:

$$OI_{\text{Block+WR}} \approx \frac{1}{8}\left(\frac{1}{2j_{\text{tile}}} + \frac{a_{\text{pack}} w_{\text{pack}}}{k}\right)^{-1} \tag{6}$$

Through CMA, we successfully increased the OI by increasing the data-reuse opportunity, and through effective prefetching via block-level scheduling, we further increased the empirical bandwidth to improve performance.

V. Experimental Result

We evaluated the end-to-end performance of the PSNPUs using the FireSim FPGA-accelerated simulation platform [10]. We set WR as a baseline and measured the performance when CMA and block-level scheduling (Block) were applied in a full SoC system environment. To obtain target hardware configurations similar to those of flagship mobile NPUs, we tested two NPU configurations of 128KB-8KB-256 and 256KB–8KB–256 (IS size–AccS size–number of PEs). The clock frequency of the NPUs was 1 GHz, and DDR3 memory was used.

We compared the performances of WR, CMA, and Block via different visualization techniques, including the OI roofline, AI roofline, and bar plot. Figs. 3a–b show the OI and AI rooflines of matrix multiplication in the size of [512]-[512]-[2048]([I]-[J]-[K]). In Fig. 3a (OI roofline), we can compare the ratio of memory stall to computation time. In Fig. 3b (AI roofline), we can compare the actual performance between different operational precision levels. Figs. 3c–d, which show the results of [512]-[512]-[512, 1024, 2048] ([I]-[J]-[K]) matrix multiplications, allow us to determine the performance improvement of Block and the performance trends based on the size of workload k. Figs. 4a–c, which show Block performance improvement for various matrix multiplication workload sizes, allow us to determine the performance improvement yielded by computing all matrix multiplications in various BERT models (tiny, small, and base) with 128, 256, 512, and 1024 sequences. Fig. 4d shows Block performance improvement for various local memory sizes, and we compared the average total cycles in 128, 256, 512, and 1024 sequences for two BERT models (tiny and small) based on two NPU configurations.

A. Roofline Analysis

Based on the OI roofline analysis shown in Fig. 3a, the peak performance is independent of the operation precision and thus comprises only one roofline. By normalizing the data based on the peak performance of the OI roofline, the y-value of the OI roofline represents the utilization of computation resources (MMU), which can be close to 1 when memory stalls decrease. Considering the empirical bandwidth, A8W8 is in the sweet spot of the roofline for all the scheduling techniques. However, in low-precision operations, the OI of WR decreases and the memory bottleneck becomes substantial. In particular, the MMU utilization in 2-bit weight precision operations, is less than 40%. This is because WR cannot increase OI in low-precision operations, as discussed in IV-A. In CMA, OI improved compared with WR, although its performance was

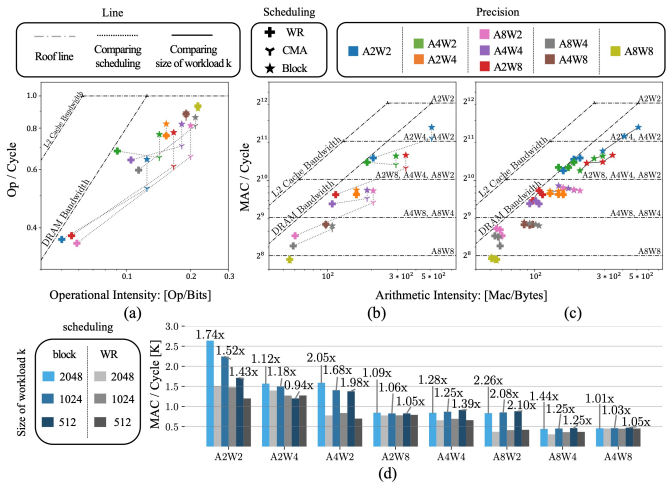

Fig. 3. (a) OI roofline analysis (single workload); (b) AI roofline analysis (single workload); (c) AI roofline analysis (multi-workload); (d) performance results for two scheduling techniques (multi-workload).

not significantly better than that of WR because an efficient prefetch was not used. Block, which can prefetch data effectively, achieved the highest performance among the three scheduling techniques at all precision levels.

Fig. 3b show the ineffectiveness of the reconfigurable logic as it depicts the unstable performance that results in WR. In WR, A2W2 indicated a performance level lower than the peak performance of A2W4, and A8W2 did not outperform A4W8. This indicates that without efficient scheduling, the reconfigurable logic in the PSNPU incurs additional power and area costs without any corresponding performance benefits. This is because the WR performance is significantly affected by the weight precision owing to the $w_{\text{pack}}/2j_{\text{tile}}$ term in Eq. (4). If the weight precision increases, then the OI becomes substantially low (based on Eq. (4)), which results in significant performance degradation. However, the performance of Block is similar to its peak performance at most precision levels and is unaffected by the various weight precision levels.

Fig. 3c shows the performance results based on the workload size through the line connecting the results of the same scheduling technique and precision level. The performance of operations where $a_{\text{pack}} \times w_{\text{pack}}$ is less than 8 did not change significantly with the size of the workload k, unlike the performance of others. This is because in Eq. (6), $a_{\text{pack}} \times w_{\text{pack}}/k$ affects the OI by more than $1/2j_{\text{tile}}$ in operations where $a_{\text{pack}} \times w_{\text{pack}}$ is greater than or equal to 8. Therefore, operations where $a_{\text{pack}} \times w_{\text{pack}}$ is greater than or equal to 8 requires a larger workload to efficiently utilize the hardware resources.

Fig. 3d shows the performance improvements of Block over the baseline. Here, Block is up to 2.26× faster than WR and 1.74× faster in the fastest operation (A2W2). It achieves substantial performance improvement in operations where the activation precision is greater than the weight precision, which is memory bounded in WR. Furthermore, Block achieves performance improvement in most cases via an efficient prefetch.

B. Performance Comparison on BERT Models

As shown in Fig. 4, even when unpredictability occurred in the actual system environments, Block achieved decent performance improvements over the baseline. In fact, its performance improvement was not mainly affected by the model

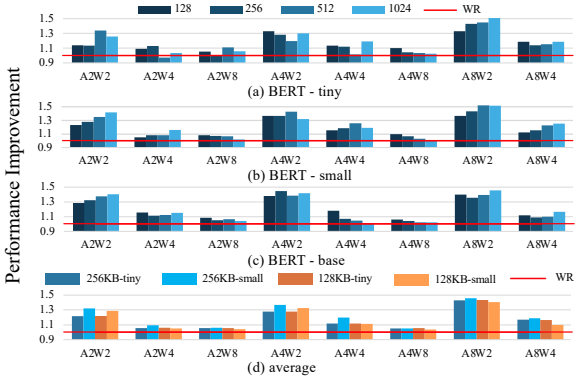

Fig. 4. Performance improvements in various input sequence lengths based on BERT models: (a) BERT-tiny, (b) BERT-small, and (c) BERT-base; (d) Average performance improvement of various input sequence lengths in BERT-tiny and small for 256KB-IS 8KB-AccS and 128KB-IS 8KB-AccS hardware configurations

or local memory size but depended on the precision level. Specifically, the geomean of the Block performance gains are 5.5%, 19.4%, and 29.4% in (weight- $>$ activation-precision), (weight- $=$ activation-precision), and (weight- $<$ activation-precision), respectively.

VI. Conclusion

In this study, we proposed an analytical model to illustrate the PSNPU performance trends. We then proposed block-level scheduling to improve OI via CMA and efficient prefetch.

Acknowledgment

This work was supported by IITP (2021-0-00310) and NRF (No. 2021R1C1C1011613, 2020M3H6A1085498, 2020M3H6A1085535) funded by the Korea government(MSIT). Yongjun Park is the corresponding author.

References

[1] H. Genc *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 769–774.

[2] S. Ryu *et al.*, "Bitblade: Area and energy-efficient precision-scalable neural network accelerator with bitwise summation," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[3] V. Camus *et al.*, "Review and benchmarking of precision-scalable multiply-accumulate unit architectures for embedded neural-network processing," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, no. 4, pp. 697–711, 2019.

[4] H. Sharma *et al.*, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 764–775.

[5] 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*, 2017, pp. 1–12.

[6] F. Sijstermans, "The nvidia deep learning accelerator," in *Hot Chips*, vol. 30, 2018, pp. 19–21.

[7] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[8] Wang *et al.*, "Haq: Hardware-aware automated quantization with mixed precision," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[9] T. Chen *et al.*, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[10] S. Karandikar *et al.*, "Firesim: Fpga-accelerated cycle-exact scale-out system simulation in the public cloud," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 29–42.