mNPUsim: Evaluating the Effect of Sharing Resources in Multi-core NPUs

Soojin Hwang* Sunho Lee* Jungwoo Kim Hongbeen Kim Jaehyuk Huh

School of Computing, KAIST

2023 IEEE International Symposium on Workload Characterization (IISWC) | 979-8-3503-0317-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/IISWC59245.2023.00018

Abstract

Multi-core neural processing units (NPUs) have emerged to scale the computation capability of NPUs to efficiently support diverse machine learning tasks. In such multi-core NPUs, workloads in different cores can affect other co-runners by incurring contentions on shared resources such as external memory bandwidth and memory management unit (MMU) for address translation. However, many recent NPU studies use a single-core NPU framework without considering dynamic effect by the shared resources. For this study, we develop a new multi-core NPU simulator to assess the effect of resource sharing accurately. Using the simulator, this paper reports the sharing behaviors of multi-core NPUs with respect to overall throughput and performance variance caused by co-runners. The evaluation of the dual and quad core NPUs shows that sharing MMU and memory bandwidth in general is beneficial for throughput, with minor degradation in fairness. The evaluation also shows that page table walkers in MMU are one of the critical shareable resources. Due to the bursty nature of NPU memory accesses, sharing of walker bandwidth across multiple cores can significantly improve the performance. The study extends the evaluation of scalability with multi-core NPUs, investigating the effect of mapping heterogeneous models to multiple NPUs.

1. Introduction

Neural processing units (NPUs) have been increasing their deployments from data centers to edge devices, partly taking over the machine learning (ML) tasks from the traditional GPUs. Such NPU architectures share a common design with a systolic array for matrix multiplication and software-managed on-chip scratch pad memory (SPM) for staging data for the systolic array. To support a diverse range of ML model sizes, a common approach is to have multiple NPU cores in a single chip. Such multi-core NPUs can avoid underutilization of a large single systolic array when it processes small tasks. As big tasks can also be flexibly mapped to multiple cores by decomposition, multi-core NPUs can adapt to a wide range of model sizes.

Such multi-core NPUs pose new challenges caused by the interference on shared resources among cores. The shared resources such as off-chip memory bandwidth and MMU (memory management unit) can potentially disrupt the performance predictability of ML tasks on multi-core NPUs as well as reducing overall throughput. The performance predictability is important to support service-level objectives (SLO) on clouds, as many current SLO-supporting schedulers assume that the latency of an inference task does not change dynamically and rely on the profiled latency for SLO supports [7, 36, 39]. Contentions on shared resources can dynamically change such latencies depending on co-runners on different cores in the same NPU.

However, most of the current NPU architecture studies do not consider such resource sharing among NPU cores, and thus the extents of potential benefit and interference have not been quantified. Such current simulation frameworks for NPUs can evaluate only a single core NPU with a deterministic memory and MMU latency without considering the runtime effects by memory bandwidth and MMU sharing. However, NPUs often generate large bursts of memory accesses from the on-chip scratchpad memory (SPM), and they can saturate the MMU and memory bandwidth, causing interference with co-runners. To address the lack of evaluation on the sharing effects, we developed a new multi-core NPU simulator, called *mNPUsim*, with a detailed memory model using DRAMsim. The new multi-core simulator can model dynamic memory latency changes due to the contentions on the shared MMU and off-chip memory bandwidth.

Using the multi-core NPU simulator, this paper analyzes the behavior of sharing MMU and memory bandwidth with co-running ML workloads, for both of the performance and fairness aspects. The first observation from the analysis is that resource sharing across NPUs on the same chip bring significant benefits. Compared to an equal static partitioning of resources across NPUs, sharing such resources improves overall throughput, helped by the bursty nature of NPU memory accesses. NPU accesses the MMU and off-chip memory in a bursty manner to fill the on-chip scratchpad memory with double-buffering. Such behaviors allow NPUs to utilize the resources more efficiently by sharing than by partitioning. Fairness for SLO support is degraded by sharing, but the degradation is relatively minor.

Among the shared resources, sharing page table walkers can bring the most benefit. As shown by the prior work [16], bursty memory accesses make the page table walker bandwidth critical for a single-core NPU. Sharing the page table walkers multiplies the available page table walkers for a core, as the bursts of memory accesses from different NPUs often do not occur simultaneously. We further investigate whether various static partitioning schemes with different

*. Equal contribution.

2835-2238/23/$31.00 ©2023 IEEE
DOI 10.1109/IISWC59245.2023.00018

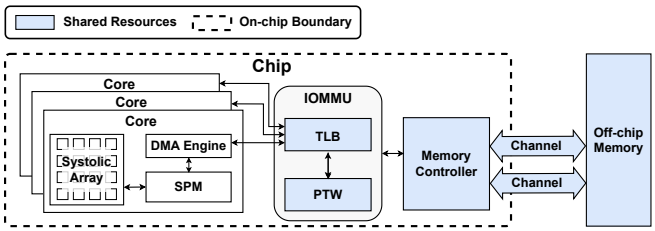


Figure 1: Multi-core NPU architecture. Memory-related resources are shared by multiple cores.

ratios between NPUs, but such partitioning did not improve the throughput. We also investigate the effect of assigning co-runners when multiple multi-core NPUs are used to serve heterogeneous models. We open-source the simulator at https://github.com/casys-kaist/mNPUsim.

2. Background

2.1. Multi-Core NPU Architecture

Neural processing unit (NPU) is a dedicated processing unit for executing machine learning workloads. An NPU includes computational units composed of multiple processing engines (PEs) and interconnection between PEs as well as the on-chip scratchpad memory (SPM) feeding data to PEs. To increase the amount of data being fetched from the SPM and forwarded to next PEs and to execute more computation in a clock cycle, several prior work have proposed NPU architectures with a large number of computational units [17]. However, large NPU cores may not be fully utilized due to characteristics of operations or dimensions of tensors [17]. For example, in the systolic array-based NPU architecture, a considerable proportion of PEs are not properly utilized if a dimension of tensors are smaller than the width or height of the systolic array.

To address the under-utilization problem of a big monolithic NPU architecture, there has been increasing interests in the multi-tenant DNN execution in NPU utilizing multiple relatively-small computational units [4, 5, 8, 10, 26]. Multi-core NPU architectures can be more efficient than the monolithic NPU architecture as both small and large workloads can utilize the resource (i.e., computation unit, on-chip SPM, off-chip memory bandwidth, etc.) of small NPUs efficiently.

Therefore, state-of-the-art NPU architectures such as Google Cloud-TPU [11] and ARM Ethos-N77 [3] are organized with multiple moderate-size cores sharing some resources. Figure 1 shows the general design of a multi-core NPU system. Each core contains a private computation unit (systolic array), a local SPM, and a private DMA engine for DMA access between SPM and off-chip memory. On the other hand, memory-related resources including TLB, page table walkers, and off-chip memory are shared among cores as displayed in blue in Figure 1.

2.2. Shared Resource in Multi-Core Systems

The management of shared resources has been a key issue in multi-core systems for making them run efficiently and fairly across multiple cores. The unbalanced distribution and monopolization of shared resources would cause performance degradation and increase of tail latencies. On the other hand, properly managing shared resources would bring a significant performance improvement compared to the design with statically-partitioned resources. In this study, we concentrate on three shared resources in multi-core NPU systems: 1) off-chip memory bandwidth, 2) page table walker (PTW), and 3) translation lookaside buffer (TLB).

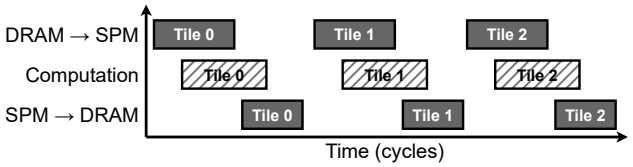
Off-chip memory bandwidth: In a multi-core system, off-chip memory bandwidth is partitioned or shared by the memory controllers. While dedicating per-core bandwidth improves fairness by resource isolation, sharing the entire bandwidth can bring performance improvement by a higher utilization. With the pros and cons of sharing, the dynamic contention of shared memory bandwidth should be considered for modeling multi-core systems.

Memory management unit (MMU): In conventional systems with NPU or GPU, IOMMU is shared among devices in the same system [12, 40]. With the shared MMU, its page table walkers (PTWs) and TLBs are shared among cores. However, such a page-table walker bandwidth for handling TLB misses is one of the key performance bottlenecks in memory intensive workloads, and thus, sharing PTW can potentially cause severe interference in co-runner performance. The prior work, DWS [28], addresses the shared PTW management for multi-tenant GPU systems. The PTW contention from inter-GPU interleaving of PTW accesses incurs a significant performance degradation in such systems. To reduce the degradation, DWS proposes a dynamic PTW stealing technique to control the sharing of PTW among GPUs.

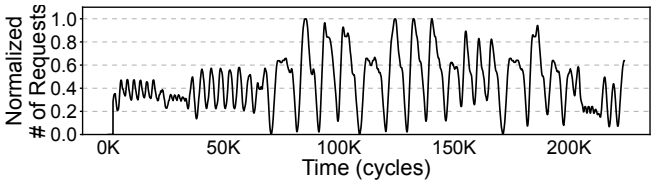
Translation lookaside buffer (TLB): To reduce TLB misses, a multi-core system can share TLB capacity among cores expecting hit rate improvement through expanded TLB coverage, while TLB conflict misses between cores would cause performance interference among co-runners. Least-TLB [21] proposed a contention-aware TLB design for multi-GPU systems. Using separated TLBs as well as a shared IOMMU TLB, Least-TLB improves the hit rate of TLBs through spilling.

2.3. Characteristics of NPU workloads

The prominent characteristic of workloads used in NPUs is their regularity. Major operations in DNN models such as convolution and general matrix-matrix multiplication (GEMM) operations can be naively represented in the form of multi-level nested loops. Since nested loops deterministically offer data locality, there have been a number of research about mapping operations into DNN accelerators for efficiently leveraging such data locality in DNN models [6, 20, 27, 43]. Especially, tiling strategies have been employed in accelerators because nested loops of convolution and GEMM operations can be reordered when there is no dependency except reduction [24, 32]. Partitioning of input data into smaller subsets can exploit parallelism in spatial accelerators to reduce data traffic [17].



(a) NPU pipelining to overlap memory access and computation.



(b) The number of memory requests for NCF in a single-core NPU.

Figure 2: Overlapping memory access and computation

Another interesting characteristic of NPU workloads is bursty memory requests due to pipelining techniques used in conventional NPUs such as double buffering [26]. The double buffering technique divides SPM into two half-sized SPMs, and allocates them for two consequent operations. Figure 2 (a) illustrates an example of pipelining widely used in conventional NPUs. When the tensor size is larger than a half of SPM capacity, it can be split into multiple tiles. With the tiling method, the memory access time and computation time are overlapped to improve PE utilization in NPUs.

Figure 2 (b) shows the moving average of memory requests between SPM and off-chip memory during 1000 cycles window when executing the NCF workload in a single-core NPU configuration of our simulator, which will be explained in Section 3 and Section 4.1 in detail. As described in Figure 2 (b), memory requests occurring when the tile read or write phase begins do not occur constantly, but a large amount of accesses may occur in a short period of time [12, 16]. Moreover, because of the virtually-addressed SPM replacing physical cache, every off-chip memory request requires address translation. For example, when the SPM size is tens of MB, thousands of pages are included in one tile as the tile size is a half of SPM size in common NPU architectures. Accordingly, thousands of address translations are required to read and write a tile in SPM. Moreover, reading or writing a tile may require many memory transactions, because a tile is a multi-dimensional tensor mapped in 1D off-chip memory address space [16].

The burstiness of address translation and memory access requests varies across layers even within a single DNN model. For example, in a convolution neural network, the computational intensity and memory intensity of each convolution layer can be different depending on various factors such as the size of the input feature map or the number of channels. To handle the large burstiness of memory-intensive layers, memory resources such as off-chip memory bandwidth and PTWs need to be fully utilized for a relatively long duration. On the other hand, in computation-intensive layers, memory resources might need to be fully utilized only for short periods or be partially utilized to handle relatively small burstiness. Similarly, the utilization of the computational units in NPU can significantly vary depending on the characteristics of layers. However, efficiently computing layers with different characteristics on the same architecture is a difficult problem because various factors such as the characteristics of workloads, configuration of the architecture, and SLO support must be considered.

3. Simulator Design

To model the dynamic contention of shared resources accurately, we developed *mNPUsim*, a C++ based cycle-accurate multi-core NPU simulator. *mNPUsim* simulates homogeneous and heterogeneous systems with multiple systolic array-based NPU cores.

3.1. Modeling Multi-core NPU System

As a multi-core NPU simulator, *mNPUsim* satisfies two major requirements: 1) dynamic modeling of shared resources with multiple levels of sharing and 2) supporting various heterogeneous core configurations.

Resource sharing: As shown in Figure 1, there are three major shareable resources in a multi-core NPU architecture: 1) off-chip memory bandwidth (DRAM bandwidth), 2) page table walker (PTW), and 3) TLB. To simulate dynamic changes in shared resource occupation, *mNPUsim* first supports out-of-order access for shared resources considering read-after-write dependency and execution delay. The non-deterministic delay of off-chip memory access is modeled by integrating DRAMsim3 [22]. Moreover, when modeling resource sharing, *mNPUsim* considers restrictions other than the total amount of resource such as TLB set index, DRAM bank and channel interleaving. We will cover the details of resource sharing in Section 4.

Heterogeneous NPU cores and clock domain: *mNPUsim* supports heterogeneous NPU cores with different per-core configurations. Different clock frequencies in NPU cores and DRAM incur asynchrony for requests between NPU and shared resources. To handle this, *mNPUsim* defines two types of clocks: 1) global clock running with DRAM clock frequency and 2) local clocks for each NPU core frequency. Shared resource access requests are synchronized to the global clock, and access latency is translated into the local clock if necessary.

Operation mapping strategy: Other than those, *mNPUsim* follows the conventions of general NPUs. For example, as most operations in DNN models including convolution layers and fully connected layers can be expressed as GEMM operations, conventional systolic array based NPUs are designed with a focus on GEMM [42, 44]. Therefore, we also consider GEMM-based systolic array designs in *mNPUsim*, adapting the image-to-column (*im2col*) algorithm that rearranges an image or a batch into a matrix, transforming convolution operations into GEMM operations to be suitable for the systolic array. Since *im2col* is for preprocessing data for GEMM, we assume that *im2col* can

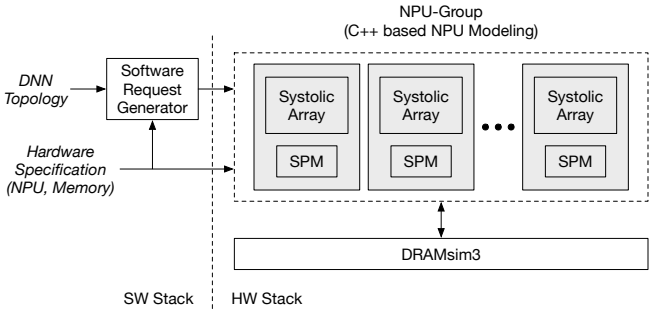


Figure 3: *mNPUsim* infrastructure.

be performed in advance on the CPU or on-the-fly if there exists the specific *im2col* module in NPU [9]. Note that we adopt the early *im2col* computation on CPU in this paper.

3.2. Simulation Details

3.2.1. Configuration. *mNPUsim* receives five kinds of configuration files as input: 1) `network_config` contains DNN topology information, including the dimension and type of each layer as well as the connection between layers. 2) `arch_config` specifies the hardware configuration for NPU computation resources such as size of systolic array, tile size, and dataflow. 3) `npumem_config` defines the additional NPU hardware configuration. It specifies the memory-related component configuration such as TLB and (PTW) configurations. 4) `dram_config` describes the shared resource configuration including DRAMsim3 configuration file path, DRAM capacity, and level of resource sharing (details in Section 4). 5) `misc_config` summarizes the execution mode of NPU core such as execution initiation time, number of iterations, and the shared partition options of page table walkers. Note that *mNPUsim* requires N numbers of `network_config`, `arch_config`, and `npumem_config` for N-core NPU architecture as they are per-core configuration, while only one `dram_config` and `misc_config` are needed because DRAM is always shared by all NPUs in *mNPUsim*.

3.2.2. Simulation steps and results. As shown in Figure 3, *mNPUsim* is composed of two parts: 1) *SW request generator* (Software stack) and 2) *HW simulator* (Hardware stack). The *SW request generator* first parses `arch_config` and `network_config` files. With the given information, the *SW request generator* and pre-run step produce the list of memory requests for every tile. Memory requests consist of address, size, and type of memory requests. The *HW simulator* then follows the *SW request generator* to simulate the multi-core NPU architecture from three inputs (memory requests, `npumem_config`, and `dram_config`). *HW simulator* runs an execution-driven simulation with given inputs, modeling constraints described in Section 3.1.

The output of *mNPUsim* contains three types of information: 1) execution cycles, 2) PE utilization, and 3) request logs of shareable resources. *mNPUsim* provides the layer-wise execution cycle of each workload in NPU clock and PE utilization information. Request logs are generated by *mNPUsim* to trace TLB, PTW, and DRAM requests. These log files include useful fields such as time (cycle), address, NPU index, channel number (optional). As described in Appendix, there are several options for additional information.

Type	Model
CNN	Resnet50 (`res`) [13] Yolo-tiny (`yt`) [31] AlexNet (`alex`) [19]
RNN	Selfish-RNN (`sfrnn`) [23] DeepSpeech2 (`ds2`) [1]
Recommendation	DLRM (`dlrm`) [25] NCF (`ncf`) [14]
Attention	gpt2 (`gpt2`) [29]

TABLE 1: Evaluated benchmark models.

Cloud-scale NPU	
Model	TPU (v4) [11]
Systolic Array	128 × 128
On-chip SPM Size	36MB
Frequency	1GHz
TLB associativity	8-way
# of TLB Entry per NPU	2048
# of PTW per NPU	8
Off-chip Memory	
Model	HBM2
Bandwidth per NPU	128GB/s
Capacity per NPU	4GB
Frequency	1GHz

TABLE 2: Basic configuration of *mNPUsim*.

4. Shared Resource Analysis

4.1. Evaluation Setup

4.1.1. Workloads. We use 3 CNNs, 2 RNNs, 2 recommendation systems, and 1 attention-based deep neural network benchmarks for evaluation as specified in Table 1. To measure the contention in multi-core NPU usage scenarios, we configure mixed workloads by composing two or four out of eight DNN benchmarks (i.e. dual-core and quad-core scenarios). All 36 ($M(8,2) = \binom{8+2-1}{2}$ for repeated combination) possible cases for a dual-core NPU and 330 ($M(8,4) = \binom{8+4-1}{4}$) mixes for a quad-core NPU are considered. We assume simultaneous execution of multiple NPU cores with the mixed workloads.

4.1.2. NPU and DRAM setup. While *mNPUsim* supports a wide range of hardware configurations, we used a representative NPU and DRAM configuration for our evaluation: a cloud-scale NPU with HBM2 as an off-chip DRAM. For PTW and TLB configuration, we use the design of NeuMMU [16]. Table 2 shows the detailed parameters for the baseline configuration with a single NPU core, and note that the amount of some resources are specified as *per NPU*. For example, the baseline configuration for the

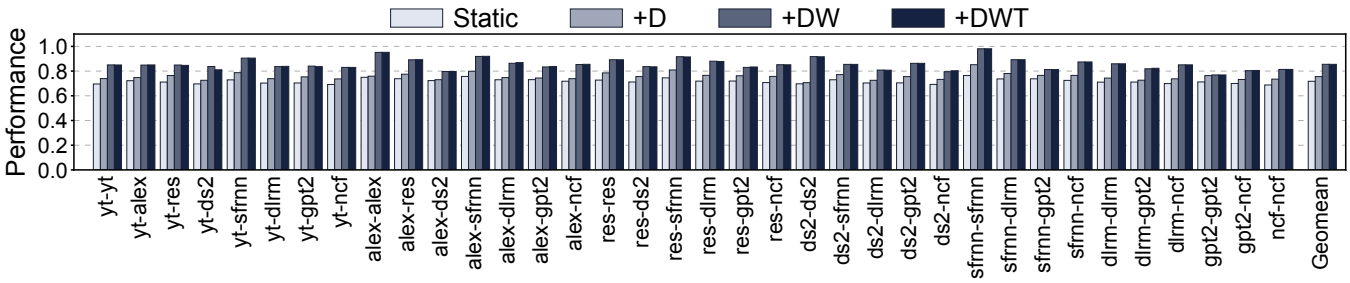


Figure 4: Performance (geometric mean of speedup) of per each workload in dual-core mix workloads with `Static`, `+D`, `+DW`, and `+DWT`, normalized to `Ideal`.

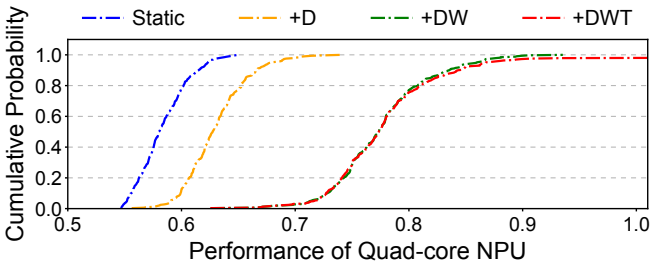


Figure 5: The cumulative distribution function (CDF) for performance of `Static`, `+D`, `+DW`, and `+DWT` per each workload in quad-core mix workloads, normalized to `Ideal`.

two-NPU system contains 16 PTWs and 4096 TLB entries in total, with 256GB/s off-chip bandwidth. We implement the operations which are primarily executed in systolic arrays such as convolution or GEMM, corresponding to the output stationary dataflow. Implementing other dataflows such as weight stationary is our future work.

4.1.3. Levels of resource sharing. As discussed in Section 3.1, there are three major shareable resources in our simulated multi-NPU system: 1) DRAM, 2) PTW, and 3) TLB. For simpler explanation, we define the abbreviation for each shared resource: Each of `D`, `W` and `T` denotes shared DRAM, shared PTWs and shared TLB respectively. In this section, we assume four levels for resource sharing: 1) `Static`, 2) `+D`, 3) `+DW`, and 4) `+DWT`. In `Static` configuration, all shareable resources (`D`, `W`, `T`) are split statically and equally to all cores, following the configuration of Table 2. For instance, assuming `Static` for a dual-core system, each core will use 128GB/s DRAM bandwidth, 8 PTWs and 2048 TLB entries without any contention. On the other hand, in `+D`, `+DW`, and `+DWT` configurations, each NPU workload cumulatively shares `D`, `W`, and `T` with co-running workloads. For example, in `+DW`, each NPU workload uses a separate `T` (TLB) by itself, but `D` (DRAM) and `W` (PTW) are shared with other NPU cores. All shared resources are shared among cores dynamically without any control on shared resource possession (i.e. first-come-first-served). In addition to the four levels of resource sharing, we also define the baseline configuration: `Ideal`. Each NPU workload monopolizes all shareable resources alone without being affected by other workloads in the `Ideal` configuration. For example, when we set a dual-core system to run with the `Ideal` configuration, each core will use the entire 256GB/s DRAM bandwidth, 16 PTWs and 4096 TLB entries exclusively.

4.2. Overall Result

4.2.1. Performance. The performance metric for shared resource analysis is a *relative speedup* compared to the ideal case (`Ideal`). When a speedup is lower than 1.0, the performance of the case is slower than `Ideal`. For mix workloads, we use the *geometric mean (geomean) of speedups for the workloads in a workload mix* to compare the overall performance of the multi-NPU system.

Figure 4 and Figure 5 present the overall performance trends of mix workloads for dual-core and quad-core NPUs, respectively. Note that we visualize the quad-core scenario using cumulative distribution function (CDF) instead of bar graphs as used for the dual-core scenario, considering the number of combinations with 330 workload mixes in total. All the three sharing scenarios (`+D`, `+DW`, `+DWT`) outperform the static equally partitioned configuration (`Static`). Figure 4 and Figure 5 illustrate three points in overall performance trends: 1) `+D`: Sharing DRAM bandwidth provides modest performance improvements over `Static`, and it achieves 75.5% and 63.0% of performance in dual and quad cores, compared to `Ideal` in geometric mean. 2) `+DW`: Shared PTW shows notable performance improvements, improving speedups by 13.2% in dual-core and 23% in quad-core from `+D` on average. 3) `+DWT`: Shared TLB has a negligible effect on overall performance, less than 1% variation compared to `+DW`. We will discuss the reason of performance trends in Section 4.3 and Section 4.4.

4.2.2. Fairness. In addition to the performance, we consider the *fairness* metric shown in Equation 1 proposed by Van Careynest et al. [41].

$$Fairness_i = 1 - \frac{\sigma_i}{\mu_i} \tag{1}$$

$Fairness_i$ represents the fairness of the i-th mix workload. μ_i and σ_i are the average and the standard deviation of slowdown (inverse of speedup) of workloads in a mix workload. For example, in the i-th mix workload composed of `alex` and `yt`, μ_i is the average of $Slowdown_i$(`alex`)

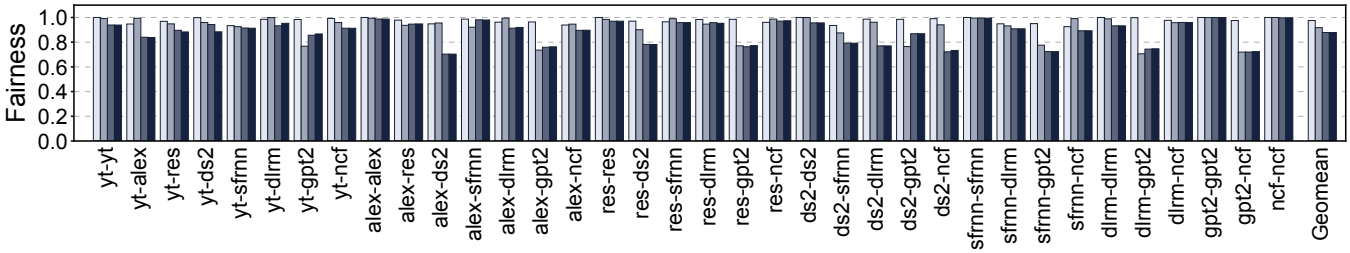


Figure 6: Geometric mean of fairness of each workload in dual-core mix workloads with `Static`, `+D`, `+DW`, and `+DWT`, calculated via Equation 1.

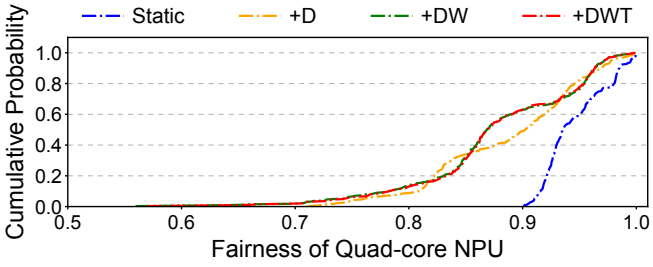


Figure 7: The cumulative distribution function (CDF) for fairness of `Static`, `+D`, `+DW`, and `+DWT` in quad-core mix workloads, calculated via Equation 1.

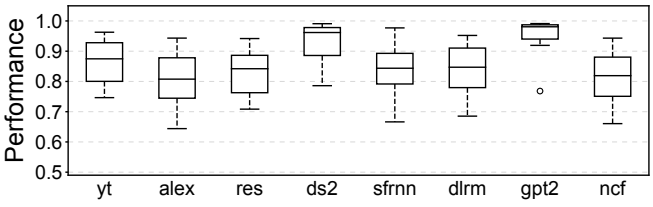


Figure 8: Performance distribution of `+DWT` affected by co-runners in dual-core, normalized to `Ideal`.

and $Slowdown_i$(`yt`) in which $Slowdown_i$(`alex`) means the inverse of speedup of `alex` compared to the `Ideal` configuration. As a smaller σ_i results in a larger $Fairness_i$, fairness closer to 1 implies better performance balance among workloads in a workload mix.

Figure 6 and Figure 7 show the overall trends of fairness with the dual and quad-core NPUs. Compared to the statically partitioned configuration, resource sharing degrades fairness although the degradation is relatively small. For the dual-core, while `Static` provides a high level of balance with fairness of 0.97 on average, the fairness drops to 0.91 with `+D`. For the quad-core, `Static` provides fairness of 0.95 on average, `+D` reduces it to 0.88. The workloads running on the quad-core system are disturbed by co-runners more extensively than the dual-core system, resulting in the higher possibility of interference between cores. Sharing PTW, however, makes system experience a similar level of imbalance in both dual-core and quad-core cases, reporting fairness of 0.87 on average. The gap between `+DW` and `+DWT` is almost negligible, as sharing TLB does not have any significant effect on fairness.

4.2.3. Contention sensitivity. Although the overall contention in mix workloads can be discussed by the result of Section 4.2.1 and Section 4.2.2, the influence of contention on each workload in mix workloads is not fully explained. Therefore, we evaluate the sensitivity of each workload in mix workloads to the contention from co-runners through the distribution of performance change.

Figure 8 visualizes the *range of performance* for each workload with various co-runners, in the dual-core NPU with all resources shared (i.e. `+DWT`). For each box, horizontal lines represent the summary of statistics. For each workload, the uppermost line is the maximum value, the lowermost line is the minimum, the upper side of each box is the 3rd quartile, the lower side of box is the 1st quartile, and the line in the middle of the box is for median. The wider range of performance distribution will result in larger gaps between lines and top/bottom sides of each box. The difference of position and size (i.e. gaps between lines and sides of the box) of each box in Figure 8 shows that every workload has a different sensitivity by the impact from the co-running workload. As all shared resources are related to off-chip memory accesses in our multi-core NPU system, memory-intensive workloads will be affected by their co-runner more than computation-intensive workloads. For example, `yt` and `res`, which only contain relatively computation-intensive convolution layers, show narrower performance distribution range compared to memory-intensive DNNs (`sfrnn` and `dlrm`).

4.3. Memory Bandwidth

In this subsection, we discuss the effect of DRAM bandwidth sharing in detail. `+D` assumes totally dynamic sharing of DRAM bandwidth among NPU cores without any limitation. Hence, we compare the performance and fairness of dynamic DRAM bandwidth sharing with static partitioning strategies with various ratios. For static partitioning schemes in the dual-core NPU scenario, we partition the total DRAM bandwidth of 256GB/s into the ratios of 1:7, 2:6, 4:4, 6:2, and 7:1, in which 1:7 implies that the first core and the second core use 1/8 and 7/8 of the total DRAM bandwidth respectively. To isolate the effect of DRAM bandwidth on overall performance, we remove address translation for experiments of this subsection.

Figure 9 shows the geometric mean of performance for each bandwidth partitioning scheme normalized to that of

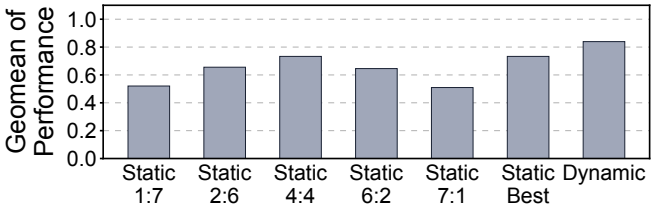


Figure 9: Geometric mean of performance for each bandwidth partitioning scheme in dual-core NPU. The normalization point is `Ideal`.

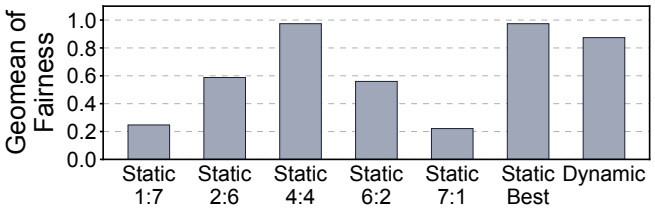


Figure 10: Geometric mean of fairness for each bandwidth partitioning scheme in dual-core NPU.

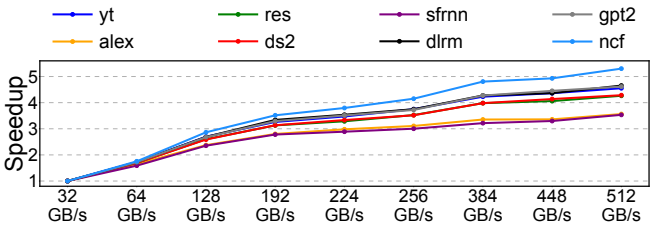


Figure 11: Speedup with respect to DRAM bandwidth change in single-core NPU, normalized to 32GB/s performance.

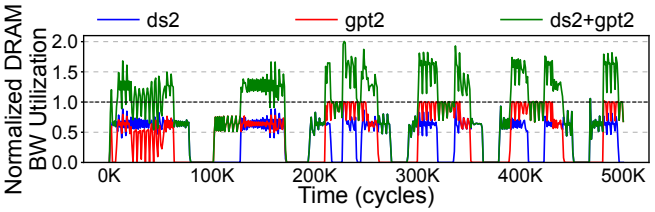


Figure 12: DRAM bandwidth utilization of executing `ds2` and `gpt2` in dual-core NPU, `Ideal` configuration.

`Ideal`, and a larger y value means a better performance (i.e. closer to the ideal performance). The scheme marked as `Static Best` represents the values obtained using the best value among the five static DRAM bandwidth partitioning schemes for each workload. In most cases, the equal static partitioning (4:4) scheme is the best among five static partitioning options. Unequal static partitioning schemes often exhibit severe performance degradation, and the equal static partitioning (4:4) shows 27% performance degradation compared to the `Ideal`. The dynamic sharing scheme achieves 84% of ideal performance on average, which is 1.14× faster than the equal static partitioning. Moreover, as shown in Figure 10, unequal static partitioning still suffers for low fairness while the dynamic partitioning scheme has a similar fairness to the equal static partitioning, which will provide the best fairness.

Such characteristic comes from the bursty nature of memory requests in NPU workloads. Figure 11 visualizes the change of speedup with respect to the change of DRAM bandwidth for the workloads of Table 1. Although a higher DRAM bandwidth would allow faster execution of NPU workload, the relationship between DRAM bandwidth and the performance is not linear. This implies that even the most memory-intensive NPU workload would not be memory-intensive for all of its lifetime, while the burstiness of its memory requests would require the ideal DRAM bandwidth to be higher than currently available peak bandwidth.

Figure 12 shows the impact of the burstiness, as an evidence of performance degradation in static partitioning. The x-axis shows elapsed cycles, and the y-axis is the DRAM bandwidth consumption normalized to the peak bandwidth (256 GB/s). Two lines, `ds2` and `gpt2` represents the bandwidth utilization when each of them run separately on the `Ideal` configuration, and `ds2+gpt2` shows the sum of bandwidth utilization of two workloads. The equal static partitioning scheme restricts the upper bound of bandwidth to be equal to the half of the peak bandwidth for each core. However, Figure 12 shows that the actual requirement of bandwidth for each core is much higher than the half of the peak bandwidth (i.e. $y \geq 0.5$) during the majority of execution time. Dynamic sharing, on the other hand, takes advantage from the time-sharing of bandwidth benefited from diverse memory access patterns of workloads and different timings of burst occurrences. Such burstiness incurs the situation shown in Figure 9 and Figure 10. Since the required bandwidth of `ds2+gpt2` exceeds the peak bandwidth (i.e. $y \geq 1.0$), even dynamic partitioning has a lower performance than `Ideal`. However, the dynamic partitioning allows effective bandwidth sharing through flexible resource management than static partitioning, resulting much less slowdowns.

4.4. Address Translation

4.4.1. Shared PTW management. In NPUs with SPM, one of the major bottlenecks in off-chip memory accesses is the page table walk bandwidth. Therefore, the sharing strategy of PTW is as important as those of DRAM bandwidth. To analyze the importance of PTW sharing schemes, we compare the static PTW partitioning with the dynamic sharing with `+DW`.

Figure 13 shows the geometric mean of performance for each PTW partitioning scheme normalized to `Ideal`, and Figure 14 represents the fairness of each partitioning scheme in the dual-core NPU system. Similar to the results of Section 4.3, the system prefers dynamic PTW sharing rather than static partitioning for performance. The bursty nature of memory requests in NPU workloads allows such performance trends as well as fairness trends.

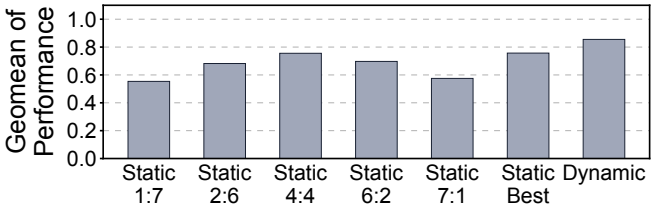


Figure 13: Geometric mean of performance for each page table walker (PTW) partitioning scheme in dual-core NPU, normalized to `Ideal`.

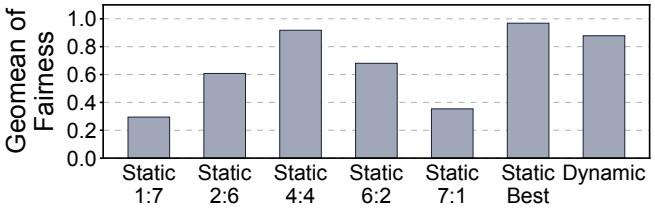


Figure 14: Geometric mean of fairness for each page table walker (PTW) partitioning scheme in dual-core NPU.

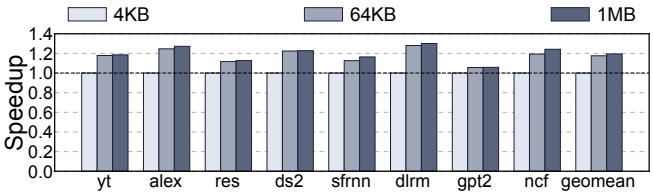


Figure 15: The speedup of 64KB and 1MB page over 4KB page in single-core NPU.

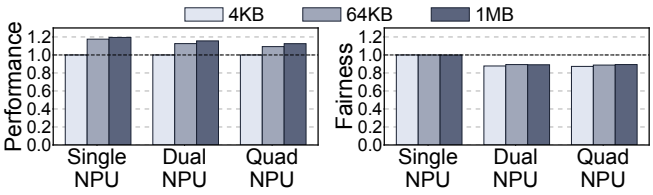


Figure 16: The geometric mean of performance and fairness for 64KB and 1MB page with single-core, dual-core, and quad-core NPU within `+DWT`. Performance is normalized to 4KB page, and the baseline of fairness is `Ideal`.

4.4.2. Shared TLB. As shown in the Section 4.2.1 and Section 4.2.2, unlike DRAM and PTW bandwidth, sharing TLB capacity does not have significant effects on both performance and fairness. Unlike DRAM and PTW whose unused bandwidth by a core can be utilized by other NPU cores quickly, each NPU core takes up a certain amount of TLB capacity constantly. Therefore, a modest increase of effective TLB capacity for memory intensive workloads by sharing TLBs is not large enough to help improve the performance. However, associativity is important when TLB is shared to avoid conflict misses. With a lower associativity for shared TLBs than 8 ways, inter-NPU conflict misses degraded the performance. The evaluated multi-core NPU system uses a 8-way set associative TLB to avoid set conflicts for the shared TLB configuration with a reasonable hardware overhead,

4.5. Scalable Page Size

Based on the analysis in Section 4.4, the shared page table walk bandwidth has a significant impact on overall performance of NPU system. The bottleneck of page table walk can be resolved in two ways: The first approach is to increasing the available PTW bandwidth by increasing the number of PTWs and/or sharing them dynamically between multiple cores. The second approach is to decrease the PTW bandwidth requirement by increasing the page size so that TLB miss rates could be decreased by orders of magnitude. The section evaluates the second approach of increasing page sizes.

4.5.1. Single-core NPU. Figure 15 shows the performance changes by three different page sizes in a single-core NPU. We use two large page sizes, 64KB and 1MB, based on the ARM64 architecture [2]. Although the increased page sizes result in performance gain, the correlation between the page size and the performance is non-linear: While the performance with 64KB page is 17.6% faster than that with 4KB page, 1MB page provides only 1.6% faster performance than 64KB page. Moreover, the sensitivity of performance to the change of page size varies depending on workload: While `gpt2` only experiences at most 5.8% speedup over 4KB page, `dlrm` runs 30% faster with 1MB page. This is because of the characteristic of page table walks caused by TLB misses. As explained in the Section 4.4.2, each TLB miss and its page table walk can have a different impact on the overall performance.

4.5.2. Multi-core NPU. We evaluate the performance and the fairness of each page size choice in multi-core NPU systems with `+DWT`. Figure 16 visualizes the evaluation result of the multi-core (i.e. dual and quad-core) NPU system. The graph on the left shows the geometric mean of performance, and the graph on the right is for the fairness. Note that the normalization point of performance is 4KB page, while the baseline of fairness is `Ideal`.

Performance: With larger page sizes, the overall performance increases for both dual and quad-core NPU systems. The dual-core NPU with 64KB and 1MB pages is 12.6% and 15.6% faster than 4KB page in geometric mean. The quad-core NPU with 64KB and 1MB pages is 9.2% and 12.5% faster than 4KB page on average, respectively. The performance gap between 64KB and 1MB page is still not large (around 3% for both cases) due to the similar reason to single-core NPU. Also, more NPU core results in less speedups since the increasing number of cores causes additional interference between cores and it reduces the effect of translation overhead.

Fairness: On the other hand, larger page size turned out to have negligible impact on fairness improvement. For both dual and quad-core NPUs, increasing page size from 4KB

to 64KB/1MB size resulted in fairness gain at most 2.3% in the geometric mean. This is because the shared TLB does not affect on balancing workload, as shown in Figure 6 and Figure 7 and explained in Section 4.4.2.

4.6. Workload Mapping in Multiple NPUs

In this section, we evaluate the effect of mapping co-runners on the system with multiple multi-core NPUs. In this analysis, we use 4 dual-core NPUs, and map 8 workloads to the total 8 NPU cores, to assess the overall performance and fairness. We evaluate the two metrics with all possible mappings. In addition to the exhaustive mapping, we apply a simple performance prediction model which estimates the performance when two models are running on a dual-core NPU. The prediction model has only a profiled information for each model and estimates the potential runtime interference.

4.6.1. Performance model. Memory intensiveness, which represents how much shared resources are utilized by each workload in each core, has significant influences on the contention between co-runners. Accordingly, this study proposes to use `PE utilization`, `memory traffic per execution`, and `ratio of execution time` as factors for considering memory intensiveness. `PE utilization` indicates how much computational units are utilized during the entire process of executing each workload, and a lower value implies a higher contention in memory resources. `Memory traffic per execution` represents the amount of memory traffic required during the entire process of executing each workload, and a higher value indicates that the workload is more memory-intensive. Finally, `ratio of execution time` is a correction factor to tune the error caused by some nondeterministic and unexpected contention, e.g. inter-core TLB conflict.

Due to the numerous factors influencing contention among co-runners, this study employs a prediction model trained using multi-factor regression. However, to prevent the overfitting from only using workloads used in Section 4.1 as a training set, the prediction model was trained with randomly generated neural networks. Similar to the approach used in Deepsniffer [15], our randomly-generated neural networks have arbitrary numbers of convolution/GEMM layers with random dimension such as output channels, stride, and kernel size in a realistic range.

4.6.2. Evaluation. We evaluate our performance model with all possible eight-workload set for four dual-core chips, total $M(8,8) = 6435$ mix workloads. Figure 17 shows the cumulative distribution function (CDF) of the speedup of our prediction model, the worst selection, random (expected) selection, and the oracle selection. The prediction model finds the better cases than the random selection for 50.04% scenarios, while mostly avoiding selecting the worst scheduling. Similarly, Figure 18 represents the fairness and 60.90% of scenarios show improved fairness by using our prediction model. With the gap between the prediction model result and oracle selection, a further possibility of improving the performance model remain as future work.

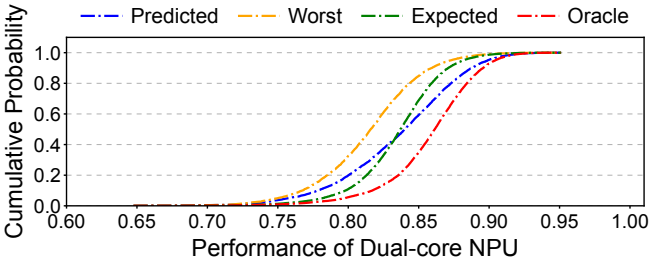


Figure 17: The cumulative distribution function (CDF) for the performance of system with mapping, normalized to the baseline without mapping.

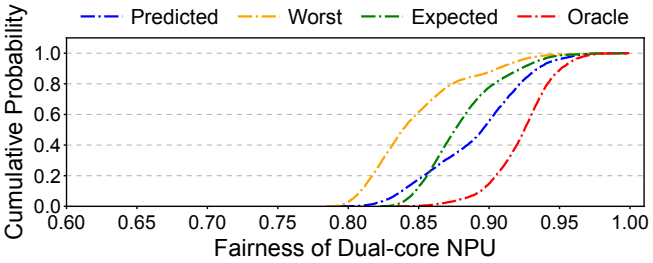


Figure 18: The cumulative distribution function (CDF) for the fairness of system with mapping, normalized to the baseline without mapping.

5. Related Work

5.1. NPU Simulators

There are several widely-used open-source simulators that model single-core NPUs. SCALE-Sim is a cycle-accurate systolic array-based DNN accelerator simulator [34, 35]. MAESTRO models various dataflows within deep-learning accelerators [20]. Gem5-Aladdin is an advanced version of Aladdin, a trace-based accelerator simulator for System-on-a-Chip (SoC) platform [37, 38].

Many prior studies on multi-NPU systems run a single-NPU simulator for multiple times, and combine their results to evaluate a multi-NPU system. However, several recent open-source software frameworks can be used as a multi-NPU simulator. Gemmini is a full-stack generator of SoC DNN accelerators implemented with Chisel, which supports generation of the cycle-accurate simulator for a given configuration [9]. Astra-Sim is a framework for design space exploration of distributed deep-learning training platforms, rather than modeling a general multi-core NPU system [30].

5.2. Multi-tenant ML Execution

There have been many recent efforts to improve supports for multi-tenancy with ML workloads on GPUs and NPUs. Choi et al. suggests the virtual GPU by leveraging spatial partitioning, and the partitioning algorithm reflects the contention of shared L2 cache and the external memory bandwidth [7]. PREMA proposes temporal sharing of NPUs for multiple DNN tasks by introducing preemption mechanism for NPUs [8]. Planaria proposes dynamic spatial

sharing of accelerators for multi-tenant DNN inference [10]. AI-MT suggests an accelerator architecture for multi-tenant DNN execution [4]. Layerweaver proposes a layer-wise scheduling technique, improving AI-MT [26]. STFusion proposes to use both spatial and temporal sharing in multiple NPUs [5]. For the multi-core accelerator, MAGMA and MoCA address the challenges of executing multi-tenant DNN inference workloads on a heterogeneous multi-core accelerator system [18, 33].

6. Conclusion

This study investigated the effect of resource sharing in multi-core NPUs. Using a new multi-core NPU simulator, it showed that memory bandwidth and page table walker sharing is beneficial for multi-core NPUs due to the bursty memory accesses, improving the overall throughput significantly from the partitioned configuration. *mNPUsim* is available on https://github.com/casys-kaist/mNPUsim.

7. Acknowledgments

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) (IITP2017-0-00466 SW StarLab and IITP2021-0-01817 Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters). This work was also supported by the National Research Foundation of Korea (NRF-2022R1A2B5B01002133). Both grants are funded by the Ministry of Science and ICT, Korea.

References

[1] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen, J. Chen *et al.*, "Deep Speech 2: End-to-end speech recognition in English and Mandarin," in *International Conference on Machine Learning (ICML)*, 2016.

[2] ARM. ARM developer. [Online]. Available: https://developer.arm.com

[3] ARM, "Powering the edge: Driving optimal performance with Ethos-N77 processor," ARM, Tech. Rep., 2019.

[4] E. Baek, D. Kwon, and J. Kim, "A multi-nural network acceleration architecture," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[5] E. Baek, E. Lee, T. Kang, and J. Kim, "STfusion: Fast and flexible multi-NN execution using spatio-temporal block fusion and memory management," in *Transactions on Computers (TC)*, vol. 72, no. 4, 2023, pp. 1194–1207.

[6] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, A. Parashar, M. Pellauer, T. Krishna, and V. Sarkar, "Marvel: A data-centric compiler for DNN operators on spatial accelerators," in *ArXiv Preprint ArXiv:2002.07752*, 2020.

[7] S. Choi, S. Lee, Y. Kim, H. Park, Y. Kwon, and H. Huh, "Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing," in *USENIX Annual Technical Conference (USENIX ATC)*, 2022.

[8] Y. Choi and M. Rhu, "PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *International Symposium on High-Performance Computing Architecture (HPCA)*, 2020.

[9] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, N. Borivoje, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Design and Automation Conference (DAC)*, 2021.

[10] S. Ghodrati, B. Ahn, J. Kim, S. Kinzer, B. R. Yatham, N. Alla, H. Sharma, M. Alian, E. Ebrahimi, N. Kim, C. Young, and H. Esmaeilzadeh, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *International Symposium on Microarchitecture (MICRO)*, 2020.

[11] Google. CloudTPU. [Online]. Available: https://cloud.google.com/tpu/docs/system-architecture-tpu-vm

[12] Y. Hao, Z. Fang, G. Reinman, and J. Cong, "Supporting address translation for accelerator-centric architectures," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2017.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[14] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *International Conference on World Wide Web (WWW)*, 2017.

[15] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, and Y. Xie, "DeepSniffer: A DNN model extraction framework based on learning architectural hints," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[16] B. Hyun, Y. Kwon, Y. Choi, J. Kim, and M. Rhu, "NeuMMU: Architectural support for efficient address translations in neural processing units," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *International Symposium on Computer Architecture (ISCA)*, 2017.

[18] S. Kim, H. Genc, V. V. Nikiforov, K. Asanovic, B. Nikolic, and Y. S. Shao, "MoCA: Memory-centric, adaptive execution for multi-tenant deep neural networks," in *International Symposium on High-Performance Computing Architecture (HPCA)*, 2023.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Conference on Neural Information Processing Systems (NIPS)*, 2012.

[20] H. Kwon, P. Chatarasi, V. Sarkar, T. Krishna, M. Pellauer, and A. Parashar, "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings," in *IEEE micro*, vol. 40, no. 3, 2020, pp. 20–29.

[21] B. Li, H. Yin, Y. Zhang, and X. Tang, "Improving address translation in multi-GPUs via sharing and spilling aware TLB design," in *International Symposium on Microarchitecture (MICRO)*, 2021.

[22] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," in *Computer Architecture Letters (CAL)*, vol. 19, no. 2, 2020, pp. 106–109.

[23] S. Liu, D. C. Mocanu, Y. Pei, and M. Pechenizkiy, "Selfish sparse RNN training," in *International Conference of Machine Learning (ICML)*, 2021.

[24] G. E. Moon, H. Kwon, G. Jeong, P. Chatarasi, S. Rajamanickam, and T. Krishna, "Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication," in *Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 4, 2021, pp. 1002–1014.

[25] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, "Deep learning recommendation model for personalization and recommendation systems," in *ArXiv Preprint ArXiv:1906.00091*, 2019.

[26] Y. H. Oh, S. Kim, Y. Jin, S. Son, J. Bae, J. Lee, Y. Park, D. U. Kim, T. J. Ham, and J. W. Lee, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *International Symposium on High-Performance Computing Architecture (HPCA)*, 2021.

[27] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A systematic approach to DNN accelerator evaluation," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[28] B. Pratheek, N. Jawalkar, and A. Basu, "Improving GPU multi-tenancy with page walk stealing," in *International Symposium on High-Performance Computing Architecture (HPCA)*, 2021.

[29] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," OpenAI, Tech. Rep., 2019.

[30] S. Rahidi, S. Sridharan, S. Srinivasan, and T. Krishna, "ASTRA-SIM: Enabling SW/HW co-design exploration for distributed DL training platforms," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[31] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[32] S. C. Kao and T. Krishna, "Gamma: Automating the HW mapping of DNN models on accelerators via genetic algorithm," in *International Conference on Computer-Aided Design (ICCAD)*, 2020.

[33] S. C. Kao and T. Krishna, "MAGMA: An optimization framework for mapping multiple DNNs on multiple accelerator cores," in *International Symposium on High-Performance Computing Architecture (HPCA)*, 2022.

[34] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of DNN accelerators using SCALE-Sim," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[35] A. Samajdar, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "SCALE-Sim: Systolic CNN accelerator simulator," in *ArXiv Preprint ArXiv:1811.02883*, 2018.

[36] W. Seo, S. Cha, Y. Kim, J. Huh, and J. Park, "SLO-aware inference scheduler for heterogeneous processors in edge platforms," in *Transaction on Architecture and Code Optimization (TACO)*, vol. 18, no. 4, 2021, pp. 1–26.

[37] S. Y. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and SoC interfaces using gem5-Aladdin," in *International Symposium on Microarchitecture (MICRO)*, 2016.

[38] Y. S. Shao, B. Reagen, G. y. Wei, and D. Brooks, "Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *International Symposium on Computer Architecture (ISCA)*, 2014.

[39] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram, "Nexus: A GPU cluster engine for accelerating DNN-based video analysis," in *Symposium on Operating System Principles (SOSP)*, 2019.

[40] S. Shin, G. Cox, M. Oskin, G. H. Loh, Y. Solihin, A. Bhattacharjee, and A. Basu, "Scheduling page table walks for irregular GPU applications," in *International Symposium on Computer Architecture (ISCA)*, 2018.

[41] C. K. Van, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2013.

[42] J. Yang, M. Wen, J. Shen, Y. Cao, M. Tang, R. Yang, J. Fei, and C. Zhang, "BP-Im2col: Implicit Im2col supporting AI backpropagation on systolic arrays," in *International Conference on Computer Design (ICCD)*, 2022.

[43] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, "Interstellar: Using Halide's scheduling language to analyze DNN accelerators," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[44] Y. Zhou, M. Yang, C. Guo, J. Leng, Y. Liang, Q. Chen, M. Guo, and Y. Zhu, "Characterizing and demystifying the implicit convolution algorithm on commercial matrix-multiplication accelerators," in *International Symposium on Workload Characterization (IISWC)*, 2021.

Appendix

1. Abstract

mNPUsim is a multi-core NPU simulator written in C++. When user inputs the several configuration files, our simulator computes per-layer (or per-tile) execution cycles, traces of PE computation, and records of memory requests.

2. Artifact Check-list (Meta-information)

- Compilation: g++ v7.5.0
- Run-time environment: Ubuntu 18.04 Kernel v4.15
- Hardware: x86-64
- Output: standard output, text files
- How much disk space required (approximately)?: 700MB
- How much time is needed to prepare workflow (approximately)?: 5 minutes
- How much time is needed to complete experiments (approximately)?: 30 seconds $\sim$ 24 hours per one configuration
- Publicly available?: Yes
- Code licenses (if publicly available)?: Creative Commons Attribution 4.0 International
- Workflow framework used?: No
- Archived (provide DOI)?: Yes (*10.5281/zenodo.8297788*)

3. Description

3.1. How to access.

- Zenodo: The artifact is published on Zenodo: https://doi.org/10.5281/zenodo.8297788.
- Github: *mNPUsim* is shared and will be updated: https://github.com/casys-kaist/mNPUsim.

3.2. Hardware dependencies. The simulator requires x86-64 architecture.

3.3. Software dependencies. The simulator is capable with Ubuntu 18.04 or later with g++ v7.5.0, and requires software prerequisites for DRAMsim3 [22].

3.4. Data sets. Since our simulator represents the multi-NPU operation flow of inference (not training), it does not require any data sets.

3.5. Models. We use 3 CNNs, 2 RNNs, 2 recommendation systems, and 1 attention-based deep neural network benchmarks for evaluation as specified in Table 1. The simulator includes Resnet50 (`res`), Yolo-tiny (`yt`), Alexnet (`alex`)), Selfish-RNN (`sfrnn`), DeepSpeech2 (`ds2`), DLRM (`dlrm`), NCF (`ncf`), and GPT-2 (`gpt2`). The architectures of models are based on the SCALE-Sim [34, 35].

4. Installation

- Load DRAMsim3 as a submodule (Optional for github distribution).
  ```
  $ git submodule update --init --recursive
  ```
- Create shared library.
  ```
  $ cd DRAMsim3
  $ make libdramsim3.so
  $ cd ..
  ```
- Run test.
  ```
  $ make
  $ make single_test1
  ```

5. Experiment Workflow

Since the detailed simulation architecture is written in Section 3, we only briefly introduce the simulator architecture here. Software request generator makes memory-ideal intermediate results. These intermediate results and several hardware configuration files are utilized to model realistic memory system by leveraging DRAMsim3 for cycle-accurate NPU core simulation.

6. Evaluation and Expected Results

We provide input files for configurations used in the evaluation of the paper. Below, we provide several examples for evaluation steps.

- Single-core NPU evaluation with test benchmark:
  ```
  $ export LD_LIBRARY_PATH=./DRAMsim3/$LD_LIBRARY_PATH
  $ ./mnpusim arch_config/core_architecture_list/tpu.txt
  network_config/netconfig_list/single/test1_network.txt
  dram_config/total_dram_config/single_hbm2_256gbs.cfg
  npumem_config/npumem_architecture_list/single.txt
  single_test misc_config/single.cfg
  $ cd single_test/result

  $ tail -1 avg_cycle_arch_tpu_small0_test1_network0.txt
  ```
- Dual-core NPU evaluation for NCF-NCF mix in `+DWT` configuration:
  ```
  $ export LD_LIBRARY_PATH=./DRAMsim3/$LD_LIBRARY_PATH
  $ ./mnpusim arch_config/core_architecture_list/tpu_tpu.txt
  network_config/netconfig_list/dual/NCF_NCF.txt
  dram_config/total_dram_config/two_hbm2_256gbs_dwt.cfg
  npumem_config/npumem_architecture_list/dual_dwt.txt
  dual_ncf_ncf misc_config/dual.cfg
  $ cd dual_ncf_ncf/result
  $ tail -1 avg_cycle_arch_tpu_small0_NCF0.txt
  $ tail -1 avg_cycle_arch_tpu_small1_NCF1.txt
  ```

By following the example, one can get the execution result (i.e. raw number of execution cycle) for each NPU core. For more information about the configuration and results, see the following section and the *README* of simulator.

7. Experiment Customization

7.1. Options. There are several options for various traces.

- *SRAM_TRACE*: Generates the SRAM trace per each PE in *result_path/intermediate/(sram files)*. If *SRAM_TRACE* equals to false, only cycle is written. (Default: false)
- *DRAMREQ_NPU_TRACE*: Option for the trace of DRAM requests in npu-side cycle. (Default: false)
- *DEBUG*: Debugging option. (Default: false)

7.2. Input customization. Users also can tuned the hardware and network configuration files. Note that, we use the list of single configuration files to support multi-NPU simulation.

- *arch_config*: NPU-core specification.
- *network_config*: Model topology.
- *dram_config*: DRAM configuration.
- *npumem_config*: Information of memory-related hyper-parameters such as page table walker (PTW).
- *misc_config*: Iteration details with the number of shared PTWs. It includes the start cycle, number of iterations, and the number of shared PTW partitions (i.e. upper and lower bound of available PTWs per core).

7.3. Parameters. Our simulator requires six parameters. Configuration parameters are list of single configuration files and result path is for output directory. The example of whole commands are in *Makefile*.

- The path of core architecture (*arch_config/core_architecture_list/tpu.txt)*
- The path of target network (*network_config/netconfig_list/single/test1_network.txt*)
- The path of target DRAM (*dram_config/total_dram_config/single_hbm2_256gbs.cfg*)
- The path of npumem config (*npumem_config/npumem_architecture_list/single.txt*)
- result path (*single_test1*)
- The path of execution mode config (*misc_config/single.cfg*)

7.4. Result files. The whole result files are stored in result path (fifth parameter). The result path contains four subdirectories to save results:

- *dramsim_output*: The access information of DRAM (*dram.log*, *dramreq.log*), TLB (*tlb<core idx>.log*), and PTW (*tlb<core idx>_ptw.log*). When DRAM requests sent, the start-cycle is written in *dram.log* and the end-cycle is written in *dramreq.log*.
- *intermediate_config*: Reconstructed configuration. Unlike original network file, it considers the GEMM translation (e.g the im2col-out convolution) and absolute address translation.
- *intermediate*: The intermediate results except the memory constraints. Based on these intermediate results, the simulator makes the memory requests. The format of output lines is *(cycle), (list of address)*. Therefore, only first values (*cycle*) in SRAM traces are remained when *SRAM_TRACE* is false.
- *result*: Text files that contains the summarization of simulation result. The name of summarization files for each NPU core is decided with following convention: *<prefix>_<core architecture name><core index>_<target network name><core index>*. There exists four kinds of summary files with intrinsic prefixes: 1) average cycles (*avg_cycle*), 2) the size of memory footprint (*memory_footprint*), 3) per layer execution cycle (*execution_cycle*), and 4) PE utilization (*utilization*).

8. Notes

Each directory has *README* file for more explanation.