A Multi-Neural Network Acceleration Architecture

Eunjin Baek, Dongup Kwon, and Jangwoo Kim‡
Department of Electrical and Computer Engineering, Seoul National University
{*ebaek, dongup, jangwoo*}*@snu.ac.kr*

***Abstract*—A cost-effective multi-tenant neural network execution is becoming one of the most important design goals for modern neural network accelerators. For example, as emerging AI services consist of many heterogeneous neural network executions, a cloud provider wants to serve a large number of clients using a single AI accelerator for improving its cost effectiveness. Therefore, an ideal next-generation neural network accelerator should support a simultaneous multi-neural network execution, while fully utilizing its hardware resources. However, existing accelerators which are optimized for a single neural network execution can suffer from severe resource underutilization when running multiple neural networks, mainly due to the load imbalance between computation and memory-access tasks from different neural networks.**

In this paper, we propose *AI-MultiTasking (AI-MT)*, a novel accelerator architecture which enables a cost-effective, high-performance multi-neural network execution. The key idea of AI-MT is to fully utilize the accelerator's computation resources and memory bandwidth by matching compute- and memory-intensive tasks from different networks and executing them in parallel. However, it is highly challenging to find and schedule the best load-matching tasks from different neural networks during runtime, without significantly increasing the size of on-chip memory. To overcome the challenges, AI-MT first creates fine-grain tasks at compile time by dividing each layer into multiple identical *sub-layers*. During runtime, AI-MT dynamically applies three sub-layer scheduling methods: *memory block prefetching* and *compute block merging* for the best resource load matching, and *memory block eviction* for the minimum on-chip memory footprint. Our evaluations using MLPerf benchmarks show that AI-MT achieves up to 1.57x speedup over the baseline scheduling method.

I. Introduction

Neural networks (NNs) have been applied to a wide range of applications thanks to their high accuracy and performance (e.g., image classification [30], [32], [35], [50], [56] and speech recognition [55]). However, executing a modern neural network can incur an extremely large number of operations and data movements due to the ever-increasing size of the network model and input data. Therefore, when modern general-purpose processors (e.g., CPU, GPU) run a large-scale neural network, they can suffer from low performance and cost-effectiveness due to their limited amount of computing resources and high power consumption [13], [16], [18].

To address this issue, researchers have made a lot of effort to design various neural network accelerators which execute a single neural network in the most cost-effective way [13], [16], [18], [27], [33], [42], [54]. For example, to efficiently support the basic operations of modern neural networks (e.g., matrix and vector operations), those accelerators often adopt two-dimensional processing element (PE) arrays [14], [15], [33]. The accelerators aim to utilize as many PEs as possible concurrently and also minimize the data movements between various units. For the purpose, researchers have also proposed various dataflow mechanisms to find the best neural network-to-PE array mapping mechanism for the maximum hardware utilization and the minimum data movements [14], [36], [60].

While many AI services benefit from the accelerators targeting single neural network executions, the cost-effective execution of multi-tenant neural networks is becoming increasingly important, in particular for cloud providers due to the following reasons. First, a modern cloud service consists of many AI algorithms executing many heterogeneous neural networks. For example, Google's search, vision, and translation services require MLP, CNN, and RNN executions, respectively [10], [19], [28], [29], [33], [49], [55], and the emerging complex services such as self-driving and natural language processing need to run many different neural networks simultaneously [5], [6], [8], [9], [21], [24], [34], [53], [58]. Second, cloud providers must minimize their huge operation costs by running as many applications on a given server as possible, while satisfying the quality of each service [2], [11], [12], [40]. Therefore, sever architects are now in dire need of a new AI acceleration architecture to enable a cost-effective, high-performance multi-neural network execution.

The existing neural network accelerators, however, cannot support a cost-effective multi-tenant neural network execution. A conventional accelerator targeting single neural network executions can naively enable a multi-neural network execution by executing different neural networks in sequence or layers from different neural networks iteratively. However, this approach creates many long periods of either compute- or memory-intensive execution which leads to the long periods of specific resource's severe underutilization. Furthermore, the existence of inter-layer dependency within a single layer prevents the following layers from even starting their execution until the current layer's completion, which also leads to a resource underutilization per layer transition.

To enable a cost-effective multi-neural network execution, this paper aims (1) to make a conventional AI accelerator support a multi-neural network execution at minimum cost and (2) to maximize its hardware utilization by creating fine-grained, dependency-free tasks with different resource intensities and executing them in parallel.

To achieve the goals, we propose *AI-MultiTasking (AI-MT)*, a novel accelerator architecture which enables a cost-effective,

‡Corresponding author.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00081

high-performance multi-neural network execution. The key idea of AI-MT is to fully utilize the accelerator's computation resources and memory bandwidth by (1) creating fine-grain compute- and memory-intensive tasks from different networks, (2) finding heterogeneous tasks with similar execution latency during runtime, and (3) executing them in parallel. AI-MT also (4) minimizes its on-chip memory capacity requirement by evicting large allocations as early as possible.

To create fine-grain tasks, AI-MT first divides each layer into multiple identical *sub-layers* at compile time. As the size of the sub-layer is statically determined by the PE array's mapping granularity, each sub-layer's SRAM requirement is small and identical. During a sub-layer execution, we define the phase of loading its weights to the on-chip SRAM as *Memory Block (MB)* execution and the phase of input processing with the loaded weights as *Compute Block (CB)* execution.

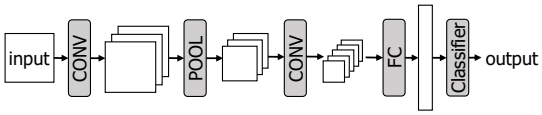
To dynamically execute MBs and CBs for the best resource load matching during runtime, AI-MT exploits a hardware-based sub-layer scheduler. The scheduler dynamically schedules MBs and CBs as long as their dependency is satisfied. First, it can fetch dependency-free MBs early (*memory block prefetching*) to fully utilize the memory bandwidth available. Second, it can group dependency-free CBs (*compute block merging*) to fully utilize the computing resources available. Lastly, it can early schedule and evict SRAM capacity-critical MBs (*memory block eviction*) to minimize the on-chip memory's capacity requirement.

To evaluate our AI-MT architecture, we use representative neural network workloads taken mainly from MLPerf benchmark. The results show that AI-MT successfully utilizes its processing elements and memory bandwidth, while minimizing the SRAM capacity requirement. Thanks to the higher resource utilization, AI-MT achieves up to 1.57x speedup over the baseline scheduling method. The sensitivity analysis for workload batching and memory capacity shows that AI-MT significantly reduces the on-chip memory's capacity requirement.

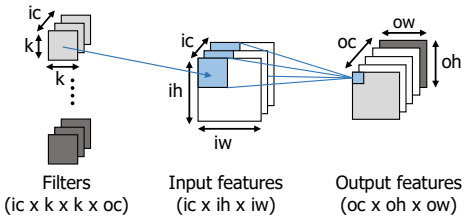
In summary, our work makes the following contributions:

- **Multi-Neural Network Acceleration:** We propose AI-MT, a novel AI acceleration architecture to enable a cost-effective multi-neural network execution.
- **Efficient Scheduling Methods:** AI-MT exploits three hardware-based scheduling methods to efficiently schedule dependency-free sub-layer tasks.
- **High Performance and Resource Utilization:** AI-MT significantly improves the hardware utilization, which also leads to the significant performance improvement.
- **Minimum SRAM Requirement** AI-MT significantly reduces its on-chip SRAM capacity requirement, which is a critical contribution for future scalability.

To the best of knowledge, this is the first work to propose an accelerator architecture to enable a cost-effective multi-neural network execution.



(a) Example neural network



(b) CONV layer

Fig. 1: Example neural network and CONV layer

II. Background

A. Neural Networks

The neural networks consist of various layers connected to each other. Figure 1a shows an example neural network which consists of convolutional (CONV), fully connected (FC), and pooling (POOL) layers. In the network, each layer's output features are passed to the next layer as its input features.

Depending on the layer's type, different layers perform different operations with their input features. For example, Figure 1b shows the operations of the example CONV layer. To produce a single output value, the layer computes the $(ic \times k \times k)$ dot products using the given input feature and filter, and adds the results with a bias value. By moving the filters to cover the entire input features, the layer generates the final output features. As another example, an FC layer performs matrix multiplications of input features and weight matrices. The FC layer can be considered as a special form of CONV layer which computes the dot product of the single-dimension input features $(ic \times 1 \times 1)$ and the single-dimension kernels $(ic \times 1 \times 1)$. Performing this operation oc times produces a $(oc \times 1)$ output. Lastly, a POOL layer reduces the input feature sizes. For example, when applying a (2×2) min pooling layer, the layer takes a minimum value in the (2×2) region of the input feature values. The pooling can be used to take the maximum value or the average of all values.

B. Baseline Neural Network Accelerator Architecture

To construct our baseline architecture, we adopt a conventional systolic array architecture based on Google's TPU [33] and scale the architecture up for our purpose (Figure 2). Each core of the latest TPU model has two processing element (PE) arrays and each array has 128×128 16-bit bfloat multiply-and-accumulate (MAC) units [1]. In addition, each TPU core incorporates HBM (300 GB/s for each TPUv2 core) and on-chip SRAM buffers to mitigate the memory bottleneck in both inference and training [20].

In this paper, we scale up the number of PE arrays to 16 as we target for server-scale neural network inference, which commonly utilizes a reduced bit precision (8-bit integer, 2x

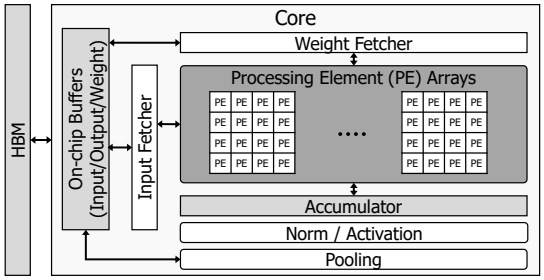


Fig. 2: Baseline neural network accelerator architecture

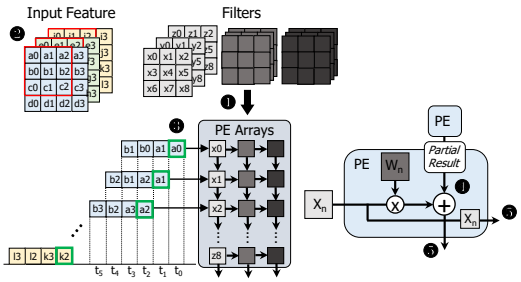


Fig. 3: Mapping example of a convolution layer over a systolic array architecture

less bit width than TPUv3) and high-end HBM technologies [4] (450GB/s for each core, 1.5x higher than TPUv2[1]). We assume that the accelerator has three physically decoupled buffers for input features, output features, and filters (or weight matrices). Each buffer has multiple banks to simultaneously feed the input and weight values to the PE array and write the output values to the buffer.

To start a neural network execution, our baseline architecture first loads the weight values from HBM to the unified buffers. After feeding the weight values into PE arrays, it performs the layer's operations using the arrays. Next, the intermediate results are passed to the dedicated units which perform subsequent operations (e.g., activation, normalization, pooling). Lastly, the accelerator passes the output values to the buffer which can be reused as the input values for the next layer. Our baseline architecture supports a double-buffering to prefetch the weights to the PE array during its computation for hiding the fetching latency [33].

C. Systolic Array Architecture

A systolic array architecture is a two-dimensional PE array to maximize the data reuse by directly forwarding them between adjacent PEs. Therefore, a systolic array architecture can reduce energy consumption by reducing data movements between the PE array and on-chip memories.

Figure 3 shows how the systolic array operates for an example CONV layer. First, a set of filters (or weights) is mapped to the PE array before starting the CONV operations

[1]TPUv3 memory bandwidth is not officially disclosed. We assume its HBM bandwidth per core is equal to TPUv2 (i.e., 300 GB/s).

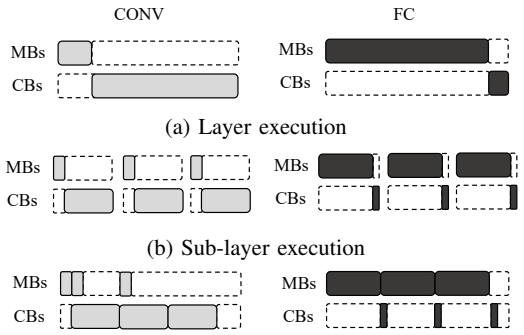


Fig. 4: Layer and sub-layer granularity executions

(❶). Next, the input values (❷) are mapped to the different columns of the PE array and fed into the array in a streamed manner (❸). Then, every PE multiplies the received input by the stored weight, and add the result to its partial result (❹). Finally, every PE passes the updated partial result to the next row, and it also passes its input values to the right column (❺). On finishing the input set, the PE array repeats this process with another filter set.

In this scheme, the layer's type and size determine the number of iterations and memory bandwidth requirements. For an FC layer, the weight matrix ($ic \times oc$) is considered as oc filters where each filter has $1 \times 1 \times ic$ size, and each filter is mapped to each column of the array. When a filter size ($1 \times 1 \times ic$) is larger than PE's row size, it repeats mapping $\lceil \frac{1 \times 1 \times ic}{PE\_dim} \rceil$ times to complete the required number of dot product operations. In addition, the total number of columns in the PE arrays ($PE\_dim \times \#PE\_array$) determines the number of filters runnable concurrently which leads to $\lceil \frac{oc}{PE\_dim \times \#PE\_array} \rceil$ iterations to execute all filters. Therefore, the systolic array performs $\lceil \frac{1 \times 1 \times ic}{PE\_dim} \rceil * \lceil \frac{oc}{PE\_dim \times \#PE\_array} \rceil$ iterations for each layer. For a CONV layer, which reuses a relatively small number of filters, we assume that all PE arrays share the same weight mapping and each PE array has partitioned input feature streams. In this case, the systolic array has $\lceil \frac{k \times k \times ic}{PE\_dim} \rceil * \lceil \frac{oc}{PE\_dim} \rceil$ iterations to execute the layer.

D. Baseline Scheduling and Prefetching Granularity

To describe our scheduling granularity, we define *sub-layer* as a single PE array mapping (i.e., single iteration). As the accelerator configuration determines the size of a sub-layer, a layer is divided into a number of *equal-sized* sub-layers. Figure 4a and 4b show example layer-granularity and sub-layer-granularity scheduling, in which MB indicates the memory bandwidth usage to fetch the weights and CB indicates the PE array usage.

In this paper, we adopt sub-layer granularity scheduling to create fine-grain tasks. We also adopt the weight prefetching to fetch the weight values for the next sub-layer mapping during the previous CB's execution (Figure 4c).

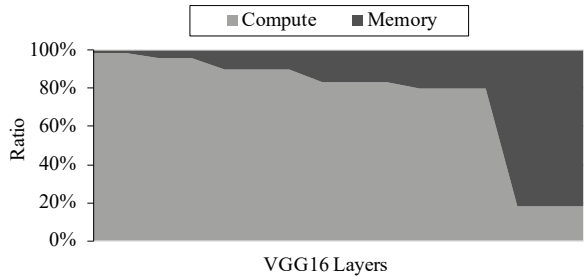


Fig. 5: Ratio of computation and memory-prefetching latency

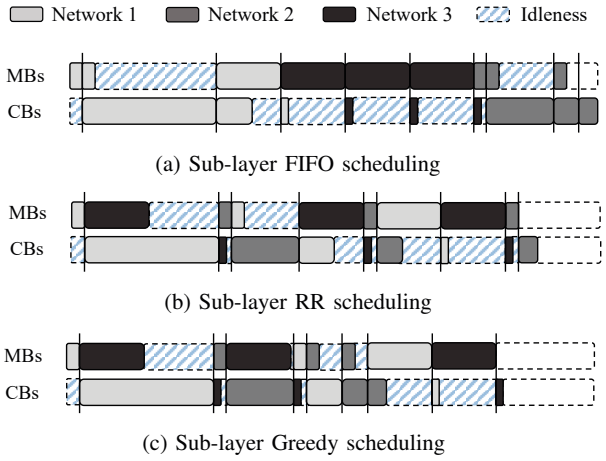


Fig. 6: Multi-neural network execution examples

III. Motivation

A. Multiple Neural Network Execution

A conventional accelerator targeting single neural network executions creates many long periods of either compute- or memory-intensive execution, which leads to the long periods of specific resource's severe underutilization. For example, Figure 5 shows the portion of computation and memory prefetching latency of each layer in VGG16 [50]. As the earlier layers are compute-intensive, the memory bandwidth utilization remains significantly low during this period. On the other hand, the later layers are highly memory-intensive, the utilization of computation resources quickly drops. In addition, the inter-layer dependency prevents the following layers from starting their execution until the current layer's completion, incurring a resource underutilization per layer transition. Note that dividing a layer to sub-layers does not improve this underutilization as sub-layers share the same resource intensity (Figure 4b).

Running multiple neural networks together has a potential to alleviate the problem as layers from different neural networks can be freely scheduled without any dependency issue. To achieve the goal, the baseline accelerator should support an optimal multi-neural network scheduling method which can create many fine-grained, dependency-free tasks with different resource intensities and executing them in parallel in a way to maximize the resource utilization. However, it is highly challenging to make a conventional accelerator to effectively support the scheduling in a cost-effective way.

B. Resource Idleness in Neural Network Execution

To analyze the resource underutilization while running multiple neural networks, we consider three reasonable scheduling mechanisms (i.e., First-In-First-Out (FIFO), Round-Robin (RR), greedy algorithm) in Figure 6. For simplicity, we assume that each scenario runs three neural networks showing only one layer each which is divided into multiple sub-layers. In each scenario, a sub-layer is shown as its MB and CB executions.

Figure 6a shows the FIFO scheduling timeline. Since the FIFO mechanism first performs the neural network which came in first, it is effectively the same as the network-wise serial execution. In the network-wise serial execution, consecutive sub-layers coming from the same layer have similar resource intensities, which is likely to incur high resource idleness.

Next, Figure 6b shows the RR scheduling timeline. The RR scheduling repeatedly selects one sub-layer from different neural networks every time in a determined order, which provides a fairness among networks. This scheduling can outperform the FIFO scheduling if two back-to-back scheduled sub-layers happen to have different resource intensities. However, this static scheduling is more likely to lead to severe resource underutilization due to the mismatching resource intensities.

To enable more flexible scheduling, we consider a simple greedy algorithm which dynamically selects the most similar size of MB to the currently executing CB (Figure 6c). This algorithm is likely to outperform the FIFO and RR mechanisms, but it can still suffer from the resource idleness due to the mismatching resource intensities.

To see resource idleness problems in the real workloads, we run neural network benchmarks included in MLPerf Inference [3] and VGG16 [50], and measure their hardware utilization using our cycle-accurate simulator. To produce various multi-neural network executions, we co-locate neural networks having distinct resource-utilization characteristics. For instance, we combine memory-intensive networks (e.g., VGG16 with large FC layers and GNMT) and multiple compute-intensive networks combined (e.g., ResNet34, ResNet50, and MobileNet). To provide a balanced distribution of CBs and MBs, we iteratively run memory-intensive workloads to properly match the amount of CBs produced by compute-intensive workloads.

We run the synthesized workloads on our cycle-accurate neural network acceleration simulator (as described in Section V-A) and measure both the performance and the resource utilization using three different scheduling mechanisms (i..e, RR, Greedy, and Shortest-Job-First (SJF)). For the SJF mechanism, we choose the next block to be the smallest one, where the size is determined by *max(MB cycle, CB cycle)*.

Figure 7 shows the resource utilization obtained with the sub-layer granularity RR mechanism. As reasoned by the

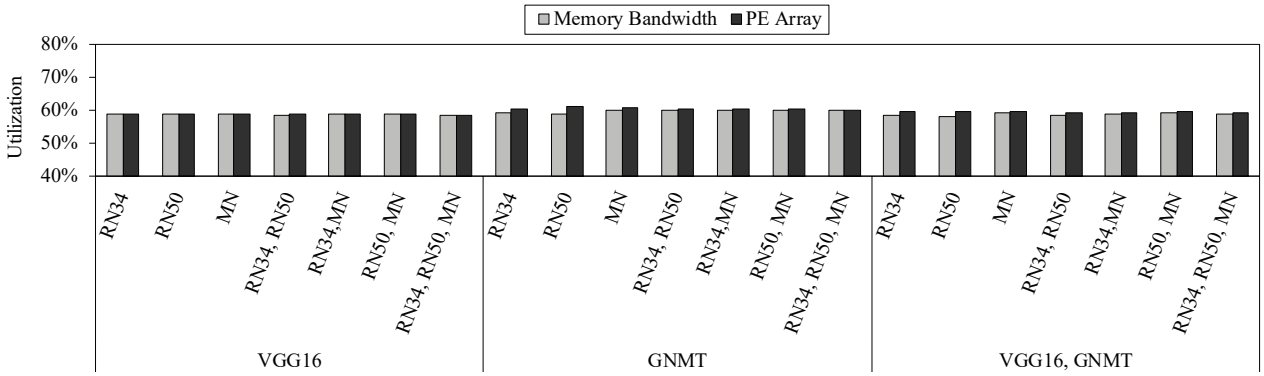


Fig. 7: Compute and memory bandwidth utilization under the round-robin scheduling algorithm (RN34: ResNet34, RN50: ResNet50, MN: MobileNet)

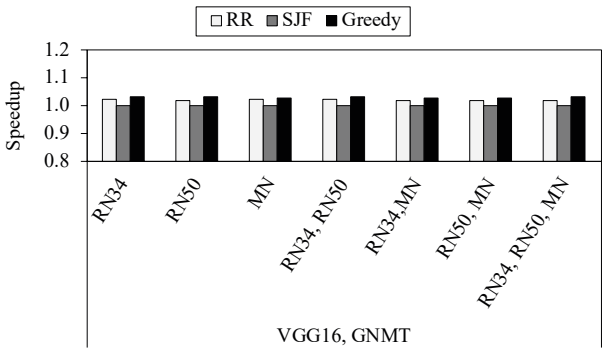


Fig. 8: Speedup of different scheduling mechanisms over the baseline sub-layer FIFO scheduling

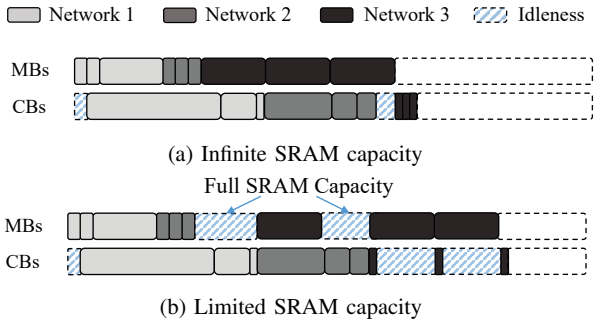


Fig. 9: Impact of the limited SRAM capacity on multi-neural network execution

simple scenario examples, the RR mechanism incurs severe resource underutilization due to the frequent mismatches of MBs and CBs, even though different resource-demand (memory- and compute-intensive) workloads are co-located. Running GNMT with other CNNs show slightly higher resource utilization, but the RR scheduling mechanism significantly suffers from its limited resource matching capability.

Figure 8 show the relative performance of three scheduling mechanisms over the FIFO scheduling mechanism. We observe that SJF also suffers from the high resource idleness because repeatedly scheduling shorts jobs from the same network eventually performs a FIFO-like network-wise scheduling. In general, the greedy mechanism outperforms the FIFO mechanism, but the improvement is far from the optimal performance due to its still limited capability to find the best-matching MB and CB pairs.

C. Limited SRAM Capacity

In Section II, we introduced the weight prefetching to fetch the weight values for the next sub-layer mapping during the previous CB's execution (Figure 4c). With such prefetching enabled, one of the easiest ways to amortize both computation and memory-bandwidth underutilization is to schedule all the compute-intensive sub-layers first and then all the memory-intensive sub-layers. Note that this method ignores the fairness.

In terms of the performance improvement only, Figure 9a shows the impact of the prefetch-aware scheduling method for the example used in Figure 6. Compared to the FIFO scheduling mechanism (Figure 6a), the proposed prefetching method significantly improves both compute- and memory-bandwidth utilization.

However, this scheduling method has a fundamental limitation due to its significant SRAM capacity requirement. The key idea of this scheme is to prefetch as many as MBs to the SRAM, until all long-running CBs are completed. Therefore, if the prefetched MBs are not evicted during the long periods of large CBs executions, the SRAM becomes full quickly which disables any further MB prefetching (Figure 9b).

Figure 10 shows the required SRAM capacity to prefetch weight values while executing the layers in MLPerf workload. We estimate the accumulated latency for all CBs and assume the accelerator prefetches MBs from later layers during each layer's execution. The result shows that even a single batch layer execution can require over 10 MB SRAM to fully utilize the memory bandwidth, and this capacity pressure can be accumulated during the multiple neural network execution. For this reason, we limit the default SRAM capacity to 1 MB in this work, and propose a solution to reduce the capacity requirement.

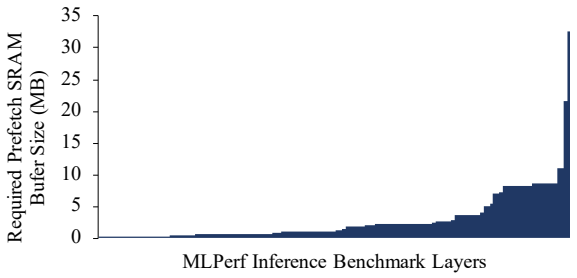


Fig. 10: Required prefetch SRAM buffer size for each layer in MLPerf inference benchmarks

D. Design Goals

From the above observations, we set three key design goals to architect a cost-effective multi-neural network execution accelerator as follows. First, it should support a cost-effective multi-neural network execution. Second, it should fully utilize the hardware resources using an effective scheduling method. Third, it should minimize the SRAM capacity requirement for future scalability.

IV. AI-MultiTasking Architecture

In this section, we propose *AI-MultiTasking (AI-MT)*, a simultaneous multi-neural network execution processor architecture. The key idea of AI-MT is to fully utilize the accelerator's computation resources and memory bandwidth by (1) creating fine-grain computation and memory-access tasks from different networks, (2) scheduling the best load-matching tasks during runtime, and (3) executing them in parallel.

At compile time, AI-MT divides each layer into multiple identical sub-layers to create fine-grain computation and memory-access tasks. Then, AI-MT generates the sub-layer scheduling table to keep the sub-layer level metadata required to the fine-grain task management at runtime. Assuming Google's TPU-like CISC instructions [33] which utilize sub-layer granularity operations (i.e., matrix multiplication based on PE array size of weight values), AI-MT can naturally divide each layer into sub-layers at compile time without sacrificing performance.

At runtime, AI-MT dynamically schedules fine-grain computation and memory-access tasks from multiple networks. AI-MT tracks the dependency-free MBs and CBs by referring to the sub-layer scheduling tables, and then applies load balancing scheduling mechanism (Section IV-B). The load balancing scheduling mechanism fetches dependency-free MBs early to fully utilize the accelerator's memory bandwidth resources, and groups dependency-free CBs to fully utilize the computing resources available. On top of the load balancing scheduling mechanism, AI-MT applies early MB eviction, which can early schedule and evict SRAM capacity-critical MBs to minimize the on-chip memory's capacity requirement.

Algorithm 1: Latency estimation model

if *LayerType = CONV* **then**
  $MB.cycle = read\_cyc\_per\_array$
  $CB.cycle = \lceil \frac{ow*oh}{\#PE\_array} \rceil * batch + filling\_time$
  $\#iters = \lceil \frac{oc}{PE\_dim} \rceil * \lceil \frac{ic*k*k}{PE\_dim} \rceil$
else if *LayerType = FC* **then**
  $MB.cycle = read\_cyc\_per\_array * \#PE\_array$
  $CB.cycle = batch + filling\_time$
  $\#iters = \lceil \frac{oc}{PE\_dim*\#PE\_array} \rceil * \lceil \frac{ic*1*1}{PE\_dim} \rceil$

A. Overview

Figure 11 shows the overview of AI-MT. To support fine-grain task management, AI-MT has three key features: *Sub-Layer Scheduling Tables*, *Candidate Queues*, and *Weight Management Table*.

First, each sub-layer scheduling table keeps the sub-layer level metadata for the fine-grain task execution of each neural network. The sub-layer scheduling table has as many rows as the number of layers, and each row includes the information of MB and CB (e.g., cycles, #sub-layers, dependencies) of the layer. AI-MT initializes the sub-layer scheduling table at compile time with the statically determining information such as the cycles taken by MB and CB of the layer.

Second, AI-MT has MB candidate queue and CB candidate queue each tracks the dependency-free MBs and CBs. AI-MT finds the dependency-free MBs and CBs by referring the sub-layer scheduling table and enters the task to the corresponding candidate queue.

Lastly, AI-MT keeps the weight address for each sub-layer using the weight management table. As AI-MT prefetches the weight values during MB's execution and consumes them during CB's execution, AI-MT needs to track the weight address for the CB's execution.

1) Sub-Layer Scheduling Table Initialization: At compile time, AI-MT initializes several columns of the sub-layer scheduling tables in Figure 11.

AI-MT estimates the cycles taken by MB and CB for each layer (i.e., *cycles* column), and total number of the sub-layers (i.e., *#iters* column) using Algorithm 1. In the algorithm, *read_cyc_per_array* indicates the cycles to prefetch weight values for a PE array and *filling time* indicates the taken cycles from the first input value injected into the PE array to the first output value is generated.

As mentioned in Section II-C, we adopt different mechanisms for a CONV layer and an FC layer as their data reuse properties are different. For a CONV layer, which reuses a small number of filters, we assume that all PE arrays share the same weight mapping and each PE array has partitioned input feature streams. In this case, the cycles required by MB is equal to the cycles to prefetch weight values from HBM to SRAM for a PE array, and the cycles required by CB is equal to the cycles to operate the partitioned input features (i.e., shared weight values, partitioned input features).

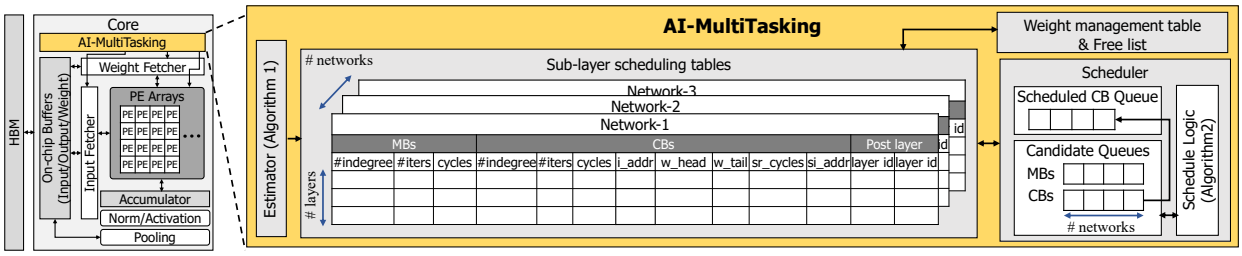


Fig. 11: Overview of AI-MT

Otherwise, for an FC layer, we assume that each PE array has individual weight values since FC layer reuses the weight values by the batch size which is relatively smaller than the reuse amount of CONV layer. In this case, the cycles required by MB is equal to the cycles to prefetch weight values from HBM to SRAM for all PE arrays, and the cycles required by CB is equal to the cycles to operate all the input features (i.e., individual weight values, shared input features).

AI-MT also updates *#indegree* and *layer_id* columns which are used to track the dependency of layers in the later section. AI-MT sets *#indegree* to the number of dependencies with previous-layers (i.e., the number of its previous layers), and *layer_id* column to the post-layer number. When a layer has multiple consequent post-layers, AI-MT sets two *layer_id* colums to the first and last post-layer's number.

2) Candidate Queue Insertion: By referring *#indegree*, *#iters* and *layer_id* columns, AI-MT finds the dependency-free MBs and CBs and inserts them to the corresponding candidate queue.

AI-MT uses *#indegree* and *layer_id* to track the dependencies between the previous layer's last sub-layer and the next layer's first sub-layer. Whenever the last sub-layer's MB or CB finishes, AI-MT refers to the *layer_id* row of the sub-layer scheduling table (i.e., the next layer's row) to resolve the dependency between the layers. Then, AI-MT decreases *#indegree* of the next layer's MB or CB by 1, and inserts the next layer's first MB or CB to the corresponding candidate queue if *#indegree* becomes 0.

AI-MT also uses *#iters* to track the dependencies between the sub-layers in the same layer. *#iters* indicates the remained sub-layer blocks for MB or CB, and AI-MT can executes the blocks sequentially by decreasing it one-by-one. Also, when *#iters* of MB is smaller than *#iter* of com, it indicates that CB's corresponding MB is already scheduled. Therefore, AI-MT can tracks both dependencies (1) between the previous MB and the next MB, and (2) between MB and the corresponding CB. For example, whenever MB or CB is scheduled, AI-MT decreases the corresponding *#iters* by 1, and adds the next block to the candidate queue. At this time, AI-MT checks if the next CB's dependency is resolved by comparing the remained *#iters* of MBs and CBs. If the *#iters* of MB is smaller than that of CB, AI-MT adds the next CB to CB's candidate queue and decreases the *#iters* of CBs.

3) Prefetched Weight Management: As AI-MT prefetches the weight values during MB's execution and consumes them during CB's execution, AI-MT needs to track the weight address for the CB's execution. AI-MT store the weight address To support this, AI-MT has three main features: (1) Free List, (2) Weight Management Table, and (3) *w_head* and *w_tail* columns in the sub-layer scheduling table. First, the free list keeps the empty block ids in the weight buffer. Next, the weight management table keeps a list where the index is the weight block id and the value refers to the next weight block id. Lastly, *w_head* and *w_tail* columns indicate the start and end block id of the prefetched weight blocks for the layer's execution.

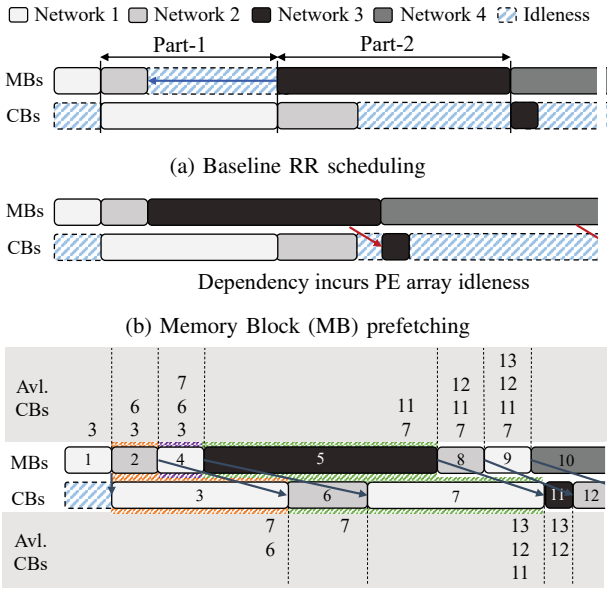
By combining three features, AI-MT can easily track the weight address for CBs to be executed. For example, when AI-MT prefetches the weight values, the memory controller allocates block ids referring to the free list and stores the weights to the blocks. After that, AI-MT refers to the weight management table using *w_tail* as an index and updates the value to the newly allocated weight block id. AI-MT also updates *w_tail* to the newly allocated weight block id to keep the last weight block id for the layer. This mechanism enables to find the CB's weight address sequentially using only *w_head* and *w_tail*. When CB is scheduled, AI-MT refers to the *w_head* to find the corresponding weight blocks for the CB, and updates *w_head* to the next weight blocks by referring weight management table for the next CB.

B. Load Balancing Scheduling

Figure 12a shows an example scheduling scenario when applying RR scheduling mechanism. The different sizes of MB and CB incur the resource idleness; memory bandwidth idleness when CB is larger than MB (*Part-1)* and PE array idleness when MB is larger than CB (*Part-2*).

To increase resource utilization in both memory bandwidth and PE arrays, we propose the balance scheduling mechanism using two schemes: *MB prefetching* and *CB merging*. MB prefetching reduces memory bandwidth idleness by aggressively prefetching MBs of the later sub-layers. It also improves CB resource utilization as the earlier MB execution resolves the dependency of the corresponding CB earlier. However, when MB prefetching is not enough to resolve the next CB's dependency, it incurs PE array idleness until the dependency is resolved. Our CB merging scheme improves PE array utilization by keeping MB and CB size at a similar pace.

1) Memory Block (MB) Prefetching: MB prefetching prefetches MBs whenever the remained SRAM capacity



(a) Baseline RR scheduling

(b) Memory Block (MB) prefetching

(c) Compute Block (CB) merging

Fig. 12: Examples of load balancing scheduling

is enough to serve them regardless of the sub-layer boundary. Figure 12b presents MB prefetching mechanism and its effect. MB prefetching prefetches MB in *Part-2* just after MB in *Part-1* finishes. It successfully amortizes the resource requirements by utilizing the idle memory bandwidth in *Part-1*. Moreover, as the earlier execution of MB in *Part-2* resolves the corresponding CB earlier, we see that the PE array idleness of *Part-2* also reduces.

2) Compute Block (CB) Merging: When the prefetched MBs become larger than the corresponding CBs, it incurs the PE array underutilization due to the dependency (Figure 12b). In this case, the PE arrays are idle until the scheduled MB finishes, resolves the dependency, and generates a new CB candidate. To further improve PE array utilization, CB merging ensures the available CBs are large enough to schedule the next MB.

CB merging schedules MBs and CBs whenever the executing MB finishes. CB merging first selects the next MB, and then schedules multiple CBs until the total cycles taken by the scheduled CBs become larger than the scheduled MB. This mechanism keeps the PE arrays busy during the selected MB's execution. To ensure the available CBs are large enough, CB merging has a variable called AVL_CB which tracks the total cycles required by the available CBs whose dependency with the corresponding MB is resolved. If AVL_CB becomes smaller than a pre-defined threshold, CB merging extends it by giving the high priority to MBs whose cycle is smaller than the corresponding CBs. Then, AVL_CB increases faster than the scheduled MB, and CB merging becomes possible to keep the AVL_CB large enough to schedule the next MB.

Figure 12c shows the example scheduling scenario of CB merging, and the numbers written in the blocks indicate

Algorithm 2: MB prefetching and CB merging

```
1  [Whenever the previous MB finishes]
2  target = None
3  for MB in MB_CQ do
4    if MB.cycle < RM_C then
5      if AVL_CB < threshold then
6        if MB's CB.cycle > MB.cycle then
7          target = MB; break;
8      else
9        target = MB; break;
10 if target == None then
11   stall until executing CB finishes
12   AVL_CB decreases for the executed CBs
13 else
14   MB_CQ.pop(target)
15   MB_C += target.cycle; RM_C -= target.cycle;
16   AVL_CB = max(AVL_CB - target.cycle, 0)
17                + target's CB.cycle
18   for CB in CB_CQ do
19     if CB_C < MB_C then
20       CB_CQ.pop(CB)
21       CB_C += CB.cycle
22       CB_SQ.push(CB)
23 [Whenever the previous CB finishes]
24 target = CB_SQ.pop(first)
25 RM_C += target.cycle;
```

the scheduling order that CB merging selects. For example, when CB merging selects *block-2*, it also selects *block-3* to make scheduled CBs are larger than the scheduled MB. Next, CB merging selects *block-4* rather than selecting *block-5* which was originally selected (Figure 12b) as the available CBs (i.e., *Block-3,6,7*) are not enough to cover *block-5*. As *block-4*'s corresponding CB (*block-7*) is larger than it, CB merging successfully extend the available CBs for the large MBs. At this time, CB merging does not select any CB as the scheduled CBs are already enough to cover *block-4*. In practice, we only track the total cycles which will be taken by the available CBs rather than tracking the list of them.

3) Combining MB Prefetching and CB Merging: Algorithm 2 shows how AI-MT's scheduler assigns MBs and CBs based on MB prefetching and CB merging. In the algorithm, the scheduler has three kinds of variables.

First, the scheduler has MB_C and CB_C to track the total cycles taken by the scheduled MBs and CBs. Whenever the previous MB or CB finishes, the scheduler adds the block cycles to MB_C or CB_C.

Second, the scheduler has AVL_CB to track the total cycles required by the available CBs. AVL_CB increases when MB resolves the corresponding CB, and decreases whenever MB finishes (*line 16,17*). Note that AVL_CB decreases by the cycles taken by the scheduled MB as it will be used to

overlap MB. If AVL_CB is small to overlap the scheduled MB, the scheduler sets AVL_CB to zero (*line 16*). Also, when the scheduler fails to find the next MB, it stalls until the executing CB finishes. In this case, AVL_CB decreases by the cycles required by the executing CB (*line 12*).

Third, the scheduler has RM_C which indicates required MB cycles to fill the remained SRAM capacity. The scheduler uses RM_C to check whether the remained SRAM capacity is enough to prefetch the next MB. At the initialization, RM_C is set to the required cycles to fill a given SRAM capacity, and then it increases or decreases whenever MB allocates or CB finishes.

Using the three variables, the scheduler schedules the next MB and CBs. First, the scheduler assigns the next MB whenever the executing MB finishes (*MB prefetching*). The scheduler iterates the MB candidate queue and selects the next MB (*target* in Algorithm 2) whose required cycle is smaller than RM_C (*line 3-4,9*). In case AVL_CB is smaller than a pre-defined threshold, the scheduler extends AVL_CB quickly by giving the high priority to MBs whose required cycle is smaller than the corresponding CB's (*line 5-7*). If none of MB in the candidate queue is smaller than RM_C, the scheduler waits until the executing CB finishes to recover the corresponding SRAM capacity (*line 11*).

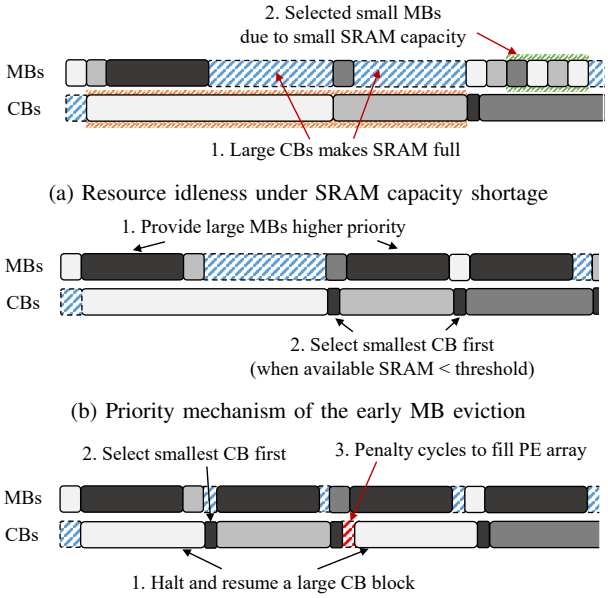
After the scheduler selects the next MB (*target*), it finds CBs to overlap the selected MB's execution (*line 18-22*). The selected CBs are inserted in CB_SQ (CB Selected Queue) and waits until the earlier scheduled CBs finish (*line 24-25*).

C. SRAM Capacity Aware Scheduling

When the SRAM is short of the free region, the balancing scheduling mechanism can incur memory bandwidth idleness during the large CB's execution (Figure 13a-1). During the large CB's execution, AI-MT can continously prefetch multiple MBs to fully utilize memory bandwidth, which requires a large SRAM capacity. Also, the large CB frees its corresponding MB after the long execution. Especially if the remained SRAM capacity is smaller than a large MB (i.e., FC sub-layer's MB), it worsens as the scheduler can select the small MBs (i.e., CONV sub-layer's MB) continuously (*line 4,9* in Algorithm 2, Figure 13a-2).

To alleviate the memory bandwidth idleness coming from the limited on-chip SRAM capacity, we introduce *early MB eviction* which schedules and evicts SRAM capacity-critical MBs to minimize the on-chip memory's capacity requirement.

When selecting the next MB and CB, Algorithm 2 simply iterates from the first element in the corresponding candidate queue (*line 9, 20*). Rather than finding the next blocks from the front, early MB eviction gives the high priority to the SRAM capacity-critical MBs whose cycles are larger than the cycles of the corresponding CBs (i.e., FC sub-layer's MB). As the large MB occupies the large SRAM capacity and the corresponding CB is relatively small, it can recover a large amount of SRAM capacity quickly. Also, AI-MT schedules the smallest CBs first when the SRAM is short of the free region. Small CBs recover SRAM capacity quickly by freeing



(a) Resource idleness under SRAM capacity shortage

(b) Priority mechanism of the early MB eviction

(c) Split mechanism of the early MB Eviction

Fig. 13: SRAM capacity aware scheduling

the corresponding weight values in the SRAM. Figure 13b shows the effects of the mechanism.

Next, when CB is too long due to the large batch size or the network configuration, SRAM capacity becomes full during CB's execution, which also incurs memory bandwidth idleness (Figure 13a-1). To alleviate this issue, early MB eviction halts the current long CB execution and schedules smaller CBs first to recover SRAM capacity quickly (Figure 13c) rather than waiting until the executing large CB finishes (*line 11* in Algorithm 2).

To halt and resume CB later, AI-MT records the status of the target CB and recover CB candidate queue. First, AI-MT inserts the target CB to the candidate queue again and pops the target's next CB which has a dependency with the target CB. It prevents the next CB's earlier execution than the target CB. Then, AI-MT records current executing input's address to the *si_addr* column and remained cycles to the *sr_cycles* in the sub-layer scheduling table (Figure 11). When the scheduler resumes the target CB later, the execution starts with referring the *si_addr*.

This mechanism has two main overheads, which can be acceptable to improve resource utilization. First, it requires the penalty cycles to fill the PE arrays again when the halted CB resumes, which can reduce PE array resource utilization. However, considering that the filling cycle is relatively small than the large CB execution, it has more potential to improve the total resource utilization. Next, CB split reads the corresponding weight values again when the scheduler resumes CB, which incurs the additional SRAM read energy consumption. Still, the additional energy consumption can be acceptable as the large CB is usually coming from the compute-intensive

layer (i.e., CONV layer) which requires a small amount of weight values by sharing the same weight values across all the PE array (Algorithm 1).

D. Alternative Software Implementation of AI-MT

In this work, we propose a hardware implementation of AI-MT, but it is also possible to implement our mechanism only using software by implementing the scheduling layer between the software framework (e.g., TensorFlow) and the accelerator. A software-based implementation is expected to achieve similar performance improvement if the following conditions are met. First, the software-based scheduling should be fast. Second, the sub-layer execution is long enough (i.e., coarse-grain sub-layer) to hide the software scheduling overhead. Third, the sub-layer split occurs infrequently enough to avoid over-populating the transfer medium (e.g., PCIe). For our current execution environment, we observe coarse-grain sub-layer executions (i.e., thousands of cycles) and infrequent sub-layer splits, which make the software-based implementation also effective as the hardware-based implementation.

Aside from the software implementation's potential, we believe that our hardware implementation will be more effective than the alternative software implementation for more aggressive multi-NN execution scenarios, where a faster accelerator would aim to run a larger number of heterogeneous neural network models. This execution model would get more benefits from finer-grain splits, but it would increase the gap between the software-based scheduling and hardware-based execution, the SRAM capacity contention, and the transfer medium contention. Therefore, the hardware-based fast scheduling without get more performance benefits without increasing the transfer medium bandwidth. But, with the hardware implementation overhead considered, we believe hardware/software collaborative approaches will be promising as another research direction.

V. Evaluation

A. Evaluation Setup

In this work, we evaluate our AI-MT by extending a systolic array cycle-accurate simulator to enable the multi-neural network execution support. We take the baseline hardware parameters from recent TPU specifications [1] and scale up the number of PE arrays from two to 16 to model the effects of the reduced bit precision (8-bit integer, 2x less than TPUv3's) and high-end HBM technologies (450 GB/s for each core, 1.5x higher than TPUv2's) [4]. Our simulation framework also implements physically decoupled buffers for input, output features and weights (Section II-B), weight-stationary dataflow (Section II-C), and sub-layer-granularity scheduling (Section II-D). We set the size of the SRAM buffer used for prefetching and storing weight values to 1 MB. The 18 MB on-chip SRAM buffers are used for storing input and output features from multiple concurrent neural network workloads. The architectural parameters are summarized in Table I.

TABLE I: Hardware and architecture parameters

Parameter	Value
Processing Element Dimension	128x128
# Processing Element Array	16
Frequency	1 GHz
Memory Bandwidth	450 GB/s
On-Chip SRAM Size (Input/Output)	18 MB
On-Chip SRAM Size (Weight)	1 MB

TABLE II: Neural network workloads and their configurations

Name	Layers		Batch Size
	FC	CONV	
ResNet34	1	36	1-32
ResNet50	1	53	1-32
VGG16	3	13	1-32
MobileNet	1	27	1-32
GNMT	6		1-32

We evaluate our AI-MT with multi-neural network benchmarks synthesized from the MLPerf Inference [3] and VGG16 [50]. Table II summarizes the configurations and characteristics of the target workloads. Our target workloads contain four CNNs and one RNN workloads, each of which has representative layer characteristics and covers typical neural network workloads running on clouds reported by service providers [33]. In the case of GNMT, we assume that embedding lookup remains on the CPU. We co-locate neural networks which have two distinct resource-utilization characteristics. For instance, we combine memory-intensive networks (e.g., VGG16 with large FC layers and GNMT) and multiple compute-intensive networks (e.g., ResNet34, ResNet50, and MobileNet). To provide a balanced distribution of CBs and MBs, we iteratively run memory-intensive workloads to properly match the amount of CBs produced by compute-intensive workloads.

B. Multi-NN Execution Latency

Figure 14 shows the speedup results for all co-located neural network workloads over the FIFO mechanism (Figure 6a) using a single batch. We first evaluate the performance impact of MB prefetching by applying it to the RR mechanism introduced in Section III-B. AI-MT achieves speedup by up to 1.34x and 1.05x over the baseline when co-locating CNNs with GNMT and VGG16, respectively. For the overall workload set, AI-MT achieves the geomean performance improvement of 1.13x. Even VGG16 has large memory-intensive FC layers, its compute-intensive CONV layers earlier than the memory-intensive layers reduce the opportunity to overlap the two different resource-demand sub-layers. On the other hand, co-locating GNMT achieves higher performance improvement as it consists of many memory-intensive FC layers which can be overlapped with different resource-demand sub-layers.

Then, we apply both CB merging and MB prefetching to the neural network workloads. CB merging achieves speedup

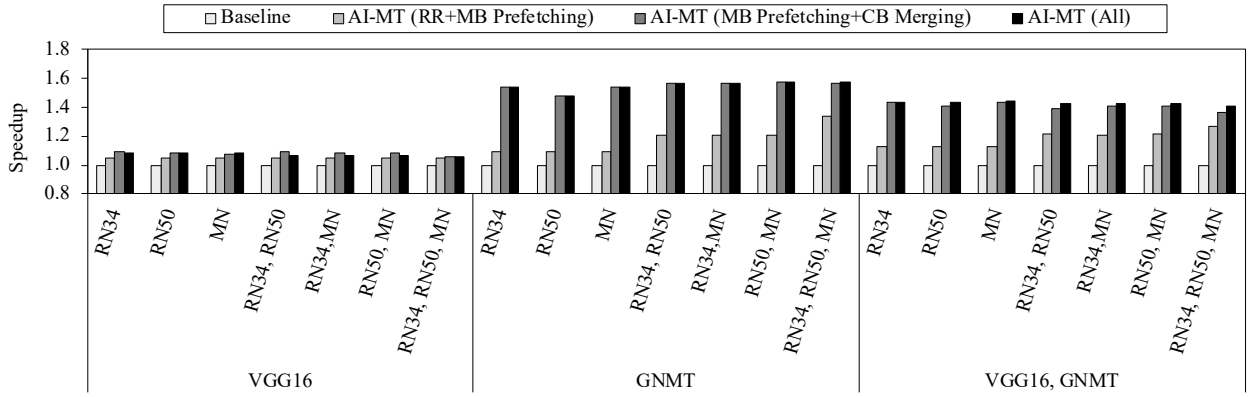


Fig. 14: Speedup of AI-MT over network-serial multi-neural network execution (Baseline)

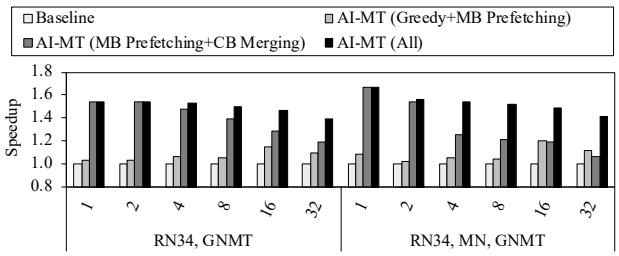


Fig. 15: Sensitivity test with different batch sizes

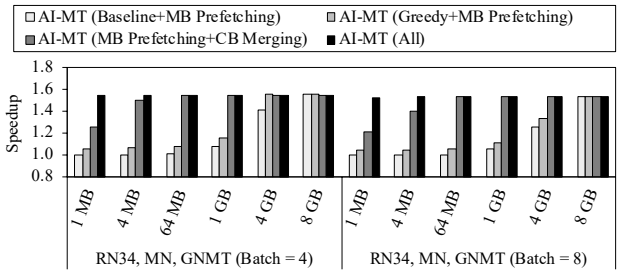


Fig. 16: Sensitivity test with different SRAM sizes

up to 1.57x compared to the baseline by fully utilizing computing resources and also improves the geomean performance improvement of 1.33x for all the workloads. Similar to MB prefetching, its effect varies depending on the co-located neural networks due to the different resource-demand requirements.

Lastly, we measure the performance impact of early MB eviction, and AI-MT achieves the geomean performance improvement of 1.33x over the baseline. With a small batch size, the cycles taken by CBs are small and the performance improvement is marginal compared to other scheduling mechanisms. To further show the impact of early MB eviction with large CBs, we show a sensitivity test of different batch sizes in Section V-C.

C. Sensitivity Test: Batch Size

Figure 15 shows the sensitivity results of the different batch sizes when multiple CNNs are running with GNMT. To verify the impact of early MB eviction, we assume that the SRAM capacity is large enough to accommodate the input and output features for the execution with large batch sizes (i.e., larger than 8). The results show that attaching early MB eviction (i.e., *AI-MT (All)*) further improves performance than just applying MB prefetching and CB merging at the large batch sizes. With the RN34 and GNMT workloads, for example, applying only MB prefetching and CB merging achieves 1.29x speedup and applying all the three schemes achieves 1.47x speedup over the baseline. When the batch size increases, CBs become larger and the limited SRAM capacity creates the bottleneck to fully utilize the memory bandwidth. As early MB eviction quickly recovers the SRAM capacity by releasing the SRAM capacity-critical MBs first, it successfully achieves the performance improvement with the large batch size. The reduction of the maximum performance improvement in the figure is coming from the PCIe transaction overhead as transferring input and output features becomes dominant in the total computation latency.

D. Sensitivity Test: On-chip SRAM Size

Figure 16 shows the speedup results over the network-serial multi-neural network execution under the various on-chip SRAM sizes. For this experiment, we assume that the input and output buffers are enough to accommodate the feature maps when the batch size is 8. We also execute our workloads iteratively to see the impact of the long-execution of neural networks like a cloud server environment where the multiple neural networks come into the accelerator continuously.

To see the required SRAM capacity, we apply two baseline scheduling mechanisms with MB prefetching. First, we apply the naive scheduling mechanism (Figure 9a) which schedules the compute-intensive neural networks first and then schedules memory-intensive neural networks. As it can prefetch all the weight values during a large amount of CBs, it achieves ideal performance when the SRAM capacity is infinitive. However,

TABLE III: Power and area overheads of on-chip memory blocks

Memory Block	Power [mW]	Area [mm^2]
Input/Output buffer (18 MB)	3575.872	119.399
Weight buffer (1 MB)	170.408	3.843
Sub-layer scheduling table (3 KB * 5)	2.897	0.0592
CQs and SQ (64 Bytes)	0.0172	0.000261
Weight management table (64 Bytes)	0.0168	0.000244
Free list (64 Bytes)	0.0168	0.000244

as Figure 16 shows, it requires more than 4 GB to achieve ideal performance improvement in our evaluation environment. Next, we apply MB prefetching to the greedy mechanism (Figure 6c). We see that the greedy mechanism also shows a similar SRAM capacity requirement due to the imbalance size of MB and CB pair. On the other hand, our AI-MT successfully achieves almost ideal speedup even with 1 MB SRAM capacity.

E. Power & Area Overheads

We estimate static power and area overheads of the memory blocks to support AI-MT using CACTI 7.0 [39] using 28-nm technology (Table III). In the estimation, we assume the five neural networks can be executed concurrently as the same as our evaluation setup. The result shows that the overheads of the additional elements to support AI-MT are negligible compared to the large input and output buffers.

The small overheads of AI-MT enable to scale the number of networks. AI-MT has the sub-layer scheduling tables per neural network and the maximum number of the entries of the candidate queue is the same as the number of the networks. The major overhead per neural network is a sub-layer scheduling table (3 KB), but it is also small enough to be scaled.

Also, our scheduling mechanism requires a negligible amount of computation overheads. To run CB, the accelerator performs 128 $\times$ 128 MAC operations per PE array at every cycle. On the other hand, our scheduling mechanism only requires dozens of addition and comparison operations for each MB scheduling (Algorithm 2), which requires much smaller operations less frequently.

VI. Discussion

A. Input and Output Buffer Capacity

This work focuses on reducing the SRAM capacity requirement by storing a minimal set of weight values when executing multiple neural networks. But, increasing the number of neural networks or increasing the batch size can increase the SRAM capacity requirement as well. We currently assume that the SRAM has a dedicated space enough to store the required input and output features. If the required size becomes larger than the SRAM capacity available, it might be useful to deploy a preemption mechanism to keep only a minimal working set of input and output features with ways to mitigate the overhead.

B. Spatial Aspects of the PE Array Utilization

This work focuses on exploiting the temporal aspects of the PE arrays to improve the resource utilization. To further improve the resource utilization, we believe that the spatial aspects of the PE array can be addressed together. Depending on the dimensions of neural networks and their mapping strategies, the resource underutilization in terms of the spatial aspect can become a non-trivial issue. For example, when CB has a small dimension to fully utilize all the MAC operations of the PE arrays, it might be possible to perform multiple CBs at the same time to improve the resource utilization within the PE array.

VII. Related Work

A. Systolic Array DNN Accelerators

Systolic array architectures are useful for computing DNN models dominated by matrix computations. For example, Google's TPU [33] comprises two-dimensional systolic arrays and embraces the idea of the weight-stationary dataflow. It preloads weight values on matrix-multiply units and performs multiply-accumulate operations by forwarding and reducing partial sums along rows or columns. Moreover, recent TPUs [1] have a high bandwidth memory (HBM) dedicated to each core and thus avoid the critical memory bandwidth bottleneck incurred by the earlier version of architectures. Similarly, Eyeriss [14] utilizes a systolic array to exploit the data reuse characteristics of modern CNNs, but leverages the row-stationary dataflow to maximize data reuse and accumulation at the RF level. Its following work, Eyeriss v2 [15], handles various layer's dimensions and sizes through a flexible on-chip network. Also, Xilinx's FPGA-based xDNN systolic-array architecture provides two operational modes, throughput- and latency-optimized modes, by adopting different mapping strategies for CNN's early layers and adjusting its pipeline stages.

Although they demonstrated using a systolic array architecture is highly effective at dealing with varying DNNs, simultaneous multi-network (or multi-threading) techniques have not been fully explored. SMT-SA [48] employs a simultaneous multi-threading technique to address the underutilization problem caused by zero-valued input. However, it still cannot handle the underutilization problem caused by layers' different resource-usage characteristics and scheduling methods.

B. Optimized DNN Dataflows

Hardware accelerators employ various dataflows to optimize data access and communication patterns. Chen et al. classified neural network accelerators into dataflow categories based on their data reuse characteristics [14]. For example, weight-stationary [7], [13], [16], [33], [43], [46], [47], [51], [52], [59], output-stationary [22], [51], row-stationary [14], [15], [26] dataflows determine spatial and temporal data mappings to PEs differently. In addition, Yang et al. [57] provided a formal taxonomy and covered more dataflows employed in recent NN accelerators [23], [25], [31], [37], [38], [41], [43]–[45].

mRNA [60] and MAESTRO [36] explore the cost and benefits of various dataflow strategies for varying hardware configurations based on the energy-performance trade-off analysis. MAESTRO [36], using the trade-off analysis results, introduces a set of compiler directives to specify the preferred DNN dataflow. However, they have the challenges of multi-network or multi-thread supports since they did not take account of the situation under which concurrent execution is allowed. We believe that our simultaneous multi-DNN supports further improve the single-network performance as well by adopting the dataflow optimizations when we split DNN layers into multiple iteration blocks.

C. Multi-DNN Scheduling

There are huge demands for system-level and architectural supports of multi-DNN scheduling to maximize the hardware utilization and reduce the cost of running large-scale production systems. For example, NIVIDA's TensorRT provides the concurrent DNN execution support, with which users can run multiple DNNs on the same GPUs simultaneously. In addition, Baymax [12] and Prophet [11] address QoS and utilization problems of current multi-DNN execution on GPUs. On the other hand, PREMA [17] proposes a preemptive scheduling algorithm under multi-DNN supports and explores various preemption mechanisms to reduce the overhead. Although they demonstrate diverse multi-DNN scheduling optimizations are effective at meeting the restricted latency requirements and increasing the hardware utilization, they fail to handle the simultaneous execution of multiple DNN models and thus cannot obtain the optimal performance under SMT supports and decoupled architectures.

VIII. Conclusion

In this paper, we propose *AI-MT*, a novel processor architecture which enables a cost-effective, high-performance multi-neural network execution. Motivated by the severe under-utilization of existing accelerators mainly due to the load imbalance between computation and memory-access tasks, AI-MT proposes *memory block prefetching* and *compute block merging* for the best resource load matching. Then, to minimize its on-chip SRAM capacity requirement during runtime, AI-MT applies *memroy block eviction* which early schedules and evicts SRAM-capacity-critical MBs. Combined with all these methods, AI-MT successfully achieves performance improvement with the minimum SRAM capacity.

ACKNOWLEDGMENT

This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1901-12. We also appreciate the support from Automation and Systems Research Institute (ASRI) and Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

References

[1] "https://cloud.google.com/tpu/docs/system-architecture," google Cloud TPU System Architecture.
[2] "https://developer.nvidia.com/tensorrt," nVIDIA TensorRT: Programmable Inference Accelerator.
[3] "https://mlperf.org/inference-overview," mLPerf Inference Benchmark.
[4] "https://news.skhynix.com/hbm2e-opens-the-era-of-ultra-speed-memory-semiconductors," hBM2E Opens the Era of Ultra-Speed Memory Semiconductors.
[5] F. Altché and A. de La Fortelle, "An lstm network for highway trajectory prediction," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2017, pp. 353–359.
[6] J. M. Alvarez, T. Gevers, Y. LeCun, and A. M. Lopez, "Road scene segmentation from a single image," in *European Conference on Computer Vision*. Springer, 2012, pp. 376–389.
[7] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer cnn accelerators," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.
[8] M. Aly, "Real time detection of lane markers in urban streets," in *2008 IEEE Intelligent Vehicles Symposium*. IEEE, 2008, pp. 7–12.
[9] D. Amodei, S. Ananthanarayanan, R. Anubhai, J. Bai, E. Battenberg, C. Case, J. Casper, B. Catanzaro, Q. Cheng, G. Chen *et al.*, "Deep speech 2: End-to-end speech recognition in english and mandarin," in *International conference on machine learning*, 2016, pp. 173–182.
[10] L. A. Barroso, U. Hölzle, and P. Ranganathan, "The datacenter as a computer: Designing warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, vol. 13, no. 3, pp. i–189, 2018.
[11] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 2, pp. 17–32, 2017.
[12] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.
[13] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.
[14] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.
[15] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.
[16] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun *et al.*, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2014, pp. 609–622.
[17] Y. Choi and M. Rhu, "Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units," *arXiv preprint arXiv:1909.04548*, 2019.
[18] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman *et al.*, "Serving dnns in real time at datacenter scale with project brainwave," *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.
[19] J. Clark, "Google turning its lucrative web search over to ai machines," *Bloomberg Technology*, vol. 26, 2015.
[20] J. Dean, "Recent advances in artificial intelligence via machine learning and the implications for computer system design," in *2017 IEEE Hot Chips 29 Symposium*, 2017.
[21] E. Derman and A. A. Salah, "Continuous real-time vehicle driver authentication using convolutional neural network based face recognition," in *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*. IEEE, 2018, pp. 577–584.
[22] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 92–104.
[23] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das, "Neural cache: Bit-serial in-cache acceleration of deep neural networks," in *2018 ACM/IEEE 45th Annual International*

Symposium on Computer Architecture (ISCA). IEEE, 2018, pp. 383–396.

[24] H. M. Eraqi, M. N. Moustafa, and J. Honer, "End-to-end deep learning for steering autonomous vehicles considering temporal dependencies," *arXiv preprint arXiv:1710.03804*, 2017.

[25] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi *et al.*, "A configurable cloud-scale dnn processor for real-time ai," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 1–14.

[26] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, "Tetris: Scalable and efficient neural network acceleration with 3d memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 751–764.

[27] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

[28] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang, "Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers," in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015, pp. 27–40.

[29] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[30] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[31] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, "Ucnn: Exploiting computational reuse in deep neural networks via weight repetition," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 674–687.

[32] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[33] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 1–12.

[34] J. Kocić, N. Jovičić, and V. Drndarević, "An end-to-end deep neural network for autonomous driving designed for embedded automotive platforms," *Sensors*, vol. 19, no. 9, p. 2064, 2019.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[36] H. Kwon, P. Chatarasi, M. Pellauer, A. Parashar, V. Sarkar, and T. Krishna, "Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 754–768.

[37] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance fpga-based accelerator for large-scale convolutional neural networks," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–9.

[38] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li, "Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 553–564.

[39] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP laboratories*, vol. 27, p. 28, 2009.

[40] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ml serving," *arXiv preprint arXiv:1712.06139*, 2017.

[41] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 27–40.

[42] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 688–698.

[43] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[44] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.

[45] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[46] Y. Shen, M. Ferdman, and P. Milder, "Overcoming resource underutilization in spatial cnn accelerators," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[47] ——, "Maximizing cnn accelerator efficiency through resource partitioning," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 535–547.

[48] G. Shomron, T. Horowitz, and U. Weiser, "Smt-sa: Simultaneous multithreading in systolic arrays," *IEEE Computer Architecture Letters*, vol. 18, no. 2, pp. 99–102, 2019.

[49] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[50] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[51] M. Song, J. Zhang, H. Chen, and T. Li, "Towards efficient microarchitectural design for accelerating unsupervised gan-based deep learning," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 66–77.

[52] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 16–25.

[53] R. Valiente, M. Zaman, S. Ozer, and Y. P. Fallah, "Controlling steering angle for cooperative self-driving vehicles utilizing cnn and lstm-based deep networks," in *2019 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2019, pp. 2423–2428.

[54] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey *et al.*, "Scaledeep: A scalable compute architecture for learning and evaluating deep networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 13–26, 2017.

[55] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv preprint arXiv:1609.08144*, 2016.

[56] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.

[57] X. Yang, M. Gao, J. Pu, A. Nayak, Q. Liu, S. E. Bell, J. O. Setter, K. Cao, H. Ha, C. Kozyrakis *et al.*, "Dnn dataflow choice is overrated," *arXiv preprint arXiv:1809.04070*, 2018.

[58] Z. Yang, Y. Zhang, J. Yu, J. Cai, and J. Luo, "End-to-end multi-modal multi-task vehicle control for self-driving cars with visual perceptions," in *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE, 2018, pp. 2289–2294.

[59] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015, pp. 161–170.

[60] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, "mrna: Enabling efficient mapping space exploration for a reconfiguration neural accelerator," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2019, pp. 282–292.