Dynamic Resource Partitioning for Multi-Tenant Systolic Array Based DNN Accelerator

Midia Reshadi*
School of Computer Science and Statistics
Lero, Trinity College Dublin
Dublin 2, Ireland
Midia.Reshadi@tcd.ie

David.Gregg
School of Computer Science and Statistics
Lero, Trinity College Dublin
Dublin 2, Ireland
David.Gregg@tcd.ie

February 22, 2023

ABSTRACT

Deep neural networks (DNN) have become significant applications in both cloud-server and edge devices. Meanwhile, the growing number of DNNs on those platforms raises the need to execute multiple DNNs on the same device. This paper proposes a *dynamic partitioning algorithm* to perform concurrent processing of multiple DNNs on a systolic-array-based accelerator. Sharing an accelerator's storage and processing resources across multiple DNNs increases resource utilization and reduces computation time and energy consumption. To this end, we propose a *partitioned weight stationary* dataflow with a minor modification in the logic of the processing element. We evaluate the energy consumption and computation time with both heavy and light workloads. Simulation results show a 35% and 62% improvement in energy consumption and 56% and 44% in computation time under heavy and light workloads, respectively, compared with single tenancy.

Keywords DNN accelerator · Multi-DNN processing · Multi-Tenancy · Dataflow.

1 Introduction

Deep neural networks have permeated applications like recommender systems [1], self-driving cars [2], and language translation [3]. The massive demand for DNN processing has led designers to develop domain-specific hardware accelerators to achieve energy efficiency and sufficient processing capacity [4][5][6]. Applications like augmented and virtual reality (AR/VR) [7] and autonomous driving cars use several DNNs for different sub-tasks in edge-side devices. For example, a VR application includes hand pose estimation and hand and eye-tracking sub-tasks as user inputs [8]. Similarly, cloud infrastructure [9] offers INFerence-as-a-Service (INFaaS) [10] for processing multiple real-life DNN-based applications in parallel. Thus, multi-tenant accelerators are shared across multiple requests on cloud-computing platforms.

There are two different methods of running multiple DNNs on the accelerator: sequential and concurrent. In the sequential approach, a layer from one DNN executes on the accelerator at any time. In the concurrent approach, multiple layers from different DNNs can be executed in parallel on the DNN accelerator. The goal of this paper is to enable parallel processing of more than one DNN layer on the systolic array accelerator, where we share storage and computing resources across multiple DNNs.

In processing each DNN layer, the dataflow procedure defines scheduling, bounding, and assignments of tensor values to memory and computation resources [11]. The main goal of the dataflow procedure is to maximize data reuse to achieve energy efficiency. Three basic dataflow approaches are commonly proposed for implementing DNNs on systolic-arrays: *weight* stationary, *input* stationary, and *output* stationary. The weight stationary approach pre-loads weight data to each of the processing elements (PEs) of the systolic array, and then streams inputs data through the systolic array to produce a stream of outputs. The input-stationary approach is similar to weight-stationary, but the

---

Figure 1: The multi-tenant DNN accelerator.

role of weights and inputs is swapped. In the output stationary approach, inputs and weights are streamed through the systolic array to compute output values at each of the PEs, after which the outputs must be drained in a separate stage.

Fig. 1 shows an example of three DNNs that are mapped to a weight-stationary systolic array. The role of the dataflow procedure is to allocate work to the hardware resources, and map weight, input, and result data to the PEs. First, the weight values are moved from the filter weight buffer into the individual processing elements of the systolic array. In the next stage, input data is streamed from the input feature map (IFMAP) buffer from left to right across the rows of the systolic array. Each input value is multiplied by the weight value stored within the corresponding PE, and the product flows downward to the PE vertically below. As the products flow downward vertically through the PEs, they are added together, so that each PE produces a partial sum of products each cycle. The partial sums move downward vertically along the columns of the PEs, until the total sum emerges at the bottom of the systolic array, and is stored to the output feature map (OFMAP) buffer.

This paper proposes a dynamic partitioning algorithm as a solution at the data-mapping level to improve resource sharing across multiple DNN layers. This algorithm needs a small change in PE logic, but it requires no expensive hardware costs, such as clusters with additional inter-cluster communication. Resource partitioning means allocating parts of the data to subsets of resources (processing and storage components). This operation is inherently data-level management and can be implemented in the dataflow procedure. We partition the systolic array vertically, so that partial sums from the same layer are added together. There is no easy way to partition our weight-stationary systolic array horizontally, because partial sums always flow downward towards the OFMAP, and partial sums from different layers must be kept separate.

Resource partitioning increases resource utilization and improves energy efficiency and computation time. We summarize the main contributions of this paper as follows:

- **Dynamic resource partitioning of multi-tenant DNN accelerator.** This paper applies dynamic resource partitioning to the PE and memory elements to increase resource utilization in multi-tenant or multi-DNN acceleration platforms.

- **Partitioned weight stationary mechanism.** This paper extends the weight stationary mechanism for supporting dynamic resource partitioning.

- **Data mapping solution with a slight hardware modification.** We simply add a tri-state gate to the processing element logic to provide the calculation control required to enable the *partitioned weight stationary* dataflow.

This paper is organized as follows: Section II provides preliminaries on DNN accelerators; Section III presents the dynamic partitioning algorithm in detail; Section IV presents comprehensive evaluations; Section V concludes.

2   Preliminaries

The primary purpose of this section is to define key concepts in a multi-tenant deep neural-network accelerator. We introduce the structures of the multi-DNN workloads and systolic array logic in detail.

Figure 2: The DNNG graph consisting of n DNNs.

2.1 Deep Neural Networks Graph (DNNG)

Multi-DNN workloads comprise several deep neural networks (DNN), including various layers with different dimensions. Accordingly, we introduce the *deep neural network graph (DNNG)* for the abstract definition of multi-DNN workloads. A deep neural network graph is a weighted directed acyclic graph (DAG), $G(V, E)$ where each vertex $v_i \in V$ corresponds to layer $l_i \in L$ that $L = \{l_1, l_2, \ldots, l_m\}$. The edge $e_i \in E$ corresponds to the flow of data between layers, and therefore defines the precedence of layer execution. Each DNNG has an *arrival time* A_t and estimated *execution time* E_t. Multi-DNN systems include n DNNs, and each DNN comprises a different number of layers, as Fig. 2 shows the pool of n DNNs with m and p layers.

Each layer consists of three convolution tensors: *filter weights (FW)*, *input feature map (IFMap)*, and *output feature map (OFMap)*, in which each tensor is a 4-dimension array with multiple shapes: $\mathbf{FW} \in \mathbb{R}^{MCRS}$, $\mathbf{IFMap} \in \mathbb{R}^{NCHW}$, and $\mathbf{OFMap} \in \mathbb{R}^{NMPQ}$. Thus, the shapes of each layer are:

$$shapes(l_i) = \{M, N, C, R, S, H, W, P, Q\} \tag{1}$$

The number of multiply and accumulate (MAC) operations required to process a layer equals the product of the shapes of FW and IFMaps:

$$Opr(l_i) = M \times N \times C \times R \times S \times H \times W \tag{2}$$

Calculating the number of MAC operations is used as a measure to estimate the execution time of each layer.

2.2 Systolic-array Based DNN Accelerator

A systolic-array-based accelerator (SA) is a domain-specific processor that consists of a 2D array of processing elements (PE) and storage components for matrix multiplication in applications like deep neural network processing.

Our design is similar to the Google TPU [12], a weight-stationary systolic array with three SRAM memory buffers: one for each of the filter weights, input feature map and output feature map. However, we use more abstract naming for buffers, that is the *load*, *feed*, and *drain* buffers, as shown in Fig. 3. For ease of presentation, we represent the dimension of each buffer as $Buffer[row, column]$, which signifies the buffer's row and column.

Each PE comprises a *load register* (LR) and MAC unit to perform multiply and accumulate operations. The PE logic has two operating modes depending on the $load$ input: Load ($load = 0$) and Calculate ($load = 1$). Fig. 3 depicts the structure of the PE with two data input-ports including *reused-data (RD)*, *feed-data (FD)* and *generated-data (GD)* as a data output-port. There is also a control input-port called $load$ that specify the PE operation. Each PE element is represented as $PE[x, y]$ where x and y refer to the PE's index in the x and y dimensions, respectively. For instance, $PE[0, 2]$ denotes PE in the first row and third column of the PE array.

In the systolic array, tensor values move through PE-array in X the dimension from the feed buffer and Y dimension from the load buffer. After completing each MAC operations in the PE, the results move downwards to the adjacent PE in the Y dimension. Note that data from the load buffer and results from MAC operations both move downwards in the Y dimension along the same inter-PE connections. Therefore, we must load data from the load buffer and compute results in separate steps. All three on-chip buffers are connected to off-chip DRAM memory.

3

Figure 3: The systolic-array based DNN accelerator. In the PE logic, LR, FD, RD, and GD stand for load register, feed data, reused data, and generated data, respectively. FD and RD are input data ports, and GD is an output data port and $load$ is input control port. $Xdim$ and $Ydim$ refer to X and Y dimensions, respectively.

Figure 4: The partitioning mechanism. A_t is arrival time and τ_0, τ_1, τ_2 are execution time (E_t)

3   Dynamic Resource Partitioning Algorithm

3.1   Systolic Array Partitioning

As shown in Fig. 3, our systolic array comprises computational and storage elements. Resource partitioning means dividing storage and processing elements into equal-sized partitions and employing them to process multiple DNN layers simultaneously. This goal is achieved by assigning tensors of a layers to a resource partition. For example, allocating a part of each storage element to tensors along with utilizing a subset of PEs to compute. Fig. 6(a) shows a simple example of a dividing a systolic array in two partitions. Two DNNs are in task queue called DNN_1 and DNN_2 with m and n layers, respectively. Two memory spaces of $load$, $feed$, and $drain$ buffers are allocated to the DNN layers. Furthermore, the PE-array is split into two 6×3 sub-arrays (six rows and three columns).

3.2   Partitioning Strategy

The systolic array structure shown in Fig. 3 allows only vertical partitioning. For example, in a 128×128 systolic array (128 PE rows and 128 PE columns), if the number of partitions is 4, then the partition shapes will be 128×32 ($128 \times \frac{128}{4}$). Using horizontal partitioning (e.g., two 64×128 partitions) causes incorrect results because the OFMap results of the upper partition enter the lower partition as inputs. Fig. 4 shows the n number of DNNs as available tasks for processing in a task queue. The first arrival time belongs to the first layer of the first DNN (L_0 of DNN_0); the

	Algorithm1: Dynamic Resource Partitioning
1:	**Inputs:** DNNG (m, n, A_t, E_t)
	SA architecture (Storage(*size*), PE(*x,y*))
2:	**Outputs:** Partition size estimation (PE(x′,y′))
3:	Task assignment($\tau_i \rightarrow PE(x', y')$
4:	Queue [*index*] ← ∅
5:	**if** (*index*==1) //First DNNG inside queue
6:	**Assign** τ_0 to PE(x,y) //Assign to all PEs with no partitioning
7:	**else**
8:	**for** All l_i with $A_{ti} \leq E_{t1}$
9:	Number of partitions= Number of DNNGs inside Queue
10:	**Call** Partition_Calculation (*Number of available layers*)
11:	**Call** Task_Assignment (*Available Layers*)
12:	**Call** Partioned_Weight_Stationery (*Available Layers*)
13:	**end_for**
14:	**end_if**
15:	**Function** Partition_Calculation (*Number of available layers*)
16:	$PE_{x'} = PE_x$ //Partition size in x dimension
17:	$PE_{y'} = \left\lfloor \dfrac{PE_y}{Number\ of\ Available\ layes} \right\rfloor$ //Partition size in y dimension
18:	**return** (PE(x′,y′))
19:	**end_function**
20:	**Function** Task_Assignment (*Available layers*)
21:	**for** All available l_i
22:	Number of operations of the layer l_i, $Opr(l_i)= \prod shapes$
23:	Sort $Opr(l_i)$ from high to low
24:	Assign l_i with the highest Opr to available partition
25:	**end_for**
26:	**return** (Assigned l_i)
27:	**end_function**
28:	**Function** Partitioned_Weight_Stationery (*Available layers*)
29:	n = Number of available layers
30:	**for** All partitions n
31:	//step ❸
32:	**Temporal_for** PE [partition column$_Y$] to Drain Buffer [partition column]
33:	**Parallel_for** PE [partition row$_X$] to Drain Buffer [partition row]
34:	//step ❷
35:	**Temporal_for** Feed Buffer [partition column] on PE [partition column$_Y$]
36:	**Parallel_for** Feed Buffer [partition row] on PE [partition row$_X$]
37:	//step ❶
38:	**Parallel_for** Load Buffer [partition row] to PE [partition row$_X$]
39:	**Parallel_for** Load Buffer [partition column] to PE [partition column$_Y$]
40:	**end_parallel_temporal_for**
41:	**end_for**
42:	**end_function**

Figure 5: Dynamic resource partitioning algorithm

partitioning algorithm assigns the first layer to the entire systolic array, so, the first layer is processed by all PEs because there are no other available layers to process in parallel with this layer.

As it is shown in Fig. 4, after finishing the layer L_0 of DNN_0 at time $At_0 + \tau_0$, there are n available layers in the queue for processing (L_1 of DNN_0, L_0 of DNN_1,.., L_0 of DNN_{n-1}).

Therefore, the accelerator is partitioned into n parts with the PE-array size equal to $128 \times \lfloor \frac{128}{n} \rfloor$ (PE row=128, PE columns=$\lfloor \frac{128}{n} \rfloor$). Some DNNs consist of only a few layers and are finished sooner than the DNNs with many layers. Hence, some partitions are freed after completing its allocated layers, and then these partitions may be merged if they are adjacent. Partition merging accelerates the processing of the remaining layers with more processing resources.

Figure 6: (a) Partitioned systolic array. In each PE partition, $PE[X, 0-2]$ means PE address in the entire X range ($X = 0\ to\ 5$) and Y in the range of 0 to 2. (b) Symbol-based representation of partitioned weight stationary dataflow (c) Loop-nest definition of of partitioned weight stationary dataflow.

3.3 Partitioning Algorithm

Fig. 5 shows the pseudo-code of the dynamic partitioning algorithm. The algorithm takes the DNN graph (DNNG) and the specs of the accelerator as inputs (Line:1) and, after estimating the partition size (Line:2), assigns layers to the partitions operated as sub-accelerators (Line:3).

All DNNs are stored in the queue, and the first layer of the first DNN is run by all PEs (Line:6). More DNNs join the queue after processing the first layer. Based on waiting layers in the queue whose arrival time is lower than the execution time of the first layer (Line:8), the *Partition_Calculation* function (Line:15-19) computes the size of partitions, and the *Task_Assignment* function (Line:20-27) assigns the available layers to the partitions.

The *Partition_Calculation* function divides PEs only in the Y dimension; the height of the partition is always equal to the height of the base PE-array, but the width of PE-array is divided by the number of available tasks (Line:17).

In the *Task_Assignment* function, all available layers are sorted based on the required MAC operations (Eq. (2)) to prioritize layers with higher processing demand (Line 22-23). Prioritizing layers is necessary because, in some cases, we may have partitions with different sizes due to merging some partitions. At the same time, there may be several layers with different dimensions ready for processing. In this case, layers with higher dimensions are assigned to the partitions with higher resources. Finally, the partitioned weight stationary function (Line:28-42) is called to perform dataflow operations on the partitioned resources.

3.4 Partitioned Weight Stationary Dataflow

Dataflow defines the temporal and spatial scheduling of operations for computation [11]. Simply put, operation scheduling is carried out in space and time. In this work, we propose a *partitioned weight stationary* dataflow to support the partitioning algorithm. We label the three steps of the dataflow with abstract names that are analogous to internal buffers: ❶ *load*, ❷ *feed*, and ❸ *drain* steps. As it is shown in Fig. 5 (Line: 28-42), all steps are defined by the loop-nests.

Data Load. In step ❶, the reused-data (filter weights) partitions are assigned to PE partitions from the load buffer. Hence, the *load* input of all PEs inside a partition is set to 1 ($load = 1$) to store the reused-data in the load register (LR). Step ❶ includes spatially assigning the rows and columns of reused data (that is *stationary*) from the load buffer

Figure 7: (a) The proposed PE with a controlled multiplier. Mul_En is connected to the En (enable) signal of the tri-state gate. In the state of $Mul\_En = 0$, the multiplier is disconnected from the adder logic, so the PE just passes data in the X dimension. When $Mul\_En = 1$, PE operates in conventional (calculate) mode. (b) The Baseline PE.

to the X and Y dimensions of PE partitions. Hence, the load operation is defined by two spatial maps as shown in Fig. 5 (Line: 38-39).

Typically, `Parallel_for` [13] denotes the spatial mapping of data in a loop-nest representation, which means assigning tensor values to the set of PEs. In contrast, `Temporal_for` denotes temporal assignment, which means assigning tensor values move across the systolic array cycle by cycle [14].

In Fig. 6(a) step ❶ involves spatial row and column assignments of reused-data to PE partitions, and Fig. 6(b) defines all assignments with details of PE-array indices during data assignments. Fig. 6(c) shows a for-loop definition, including two `Parallel_for` for each partition assignment.

Data Feed. In step ❷, the partitions of feed data (IFMap) move from feed buffers across the PE arrays in the X dimension, and PEs calculate partial sum and then send them to the neighbouring PEs in the Y dimension. As a result, the data movement is performed by spatially assigning feed data to PE arrays in rows and temporally assigning feed data to PE arrays in columns. Fig. 5 (Line:35-36) shows an abstract for-loop presentation of step ❷. In contrast to step ❶, feed data partitions pass through multiple PE partitions. Thus, a technique is required to guarantee that each feed data partition is computed solely by its matching PE partition. To create such a feature with minimum hardware overhead, we just add a tri-state buffer between adder and multiplier in PE logic. Fig. 7(a) shows our proposed PE with controlled-multiplication ability compared to the baseline PE (Fig. 7(b)). The proposed PE has a second control input called Mul_En which is used to choose to enable or disable multiplication when passing through relevant or irrelevant partitions. Therefore, in step ❷ the $load$ input of all PEs is set to 0 ($load = 0$) to operate in calculation mode and the Mul_En input of PEs is set to 1 ($Mul\_En = 1$) only when they pass trough the corresponding data partition. For example, in Fig. 6(a) the Mul_En input of partition 1 PEs is set to 0 ($Mul\_En = 0$) when passing DNN2 data in step ❷, and it is set to 1 ($Mul\_En = 1$) in partition 2 PEs when passing DNN2.

Data Drain. Finally, in step ❸, the generated OFMaps (drain data) leave PEs by spatial column and temporal row assignment, which is shown in Line:32-33 of Fig. 5. In this step, OFMaps move in Y from PE partitions to the drain buffer. Fig. 6(b) describes the symbolic representation, and Fig. 6(c) shows the loop-nest definition of the spatial column and temporal row assignment of OFMap data from two PE partitions to the drain buffer. In step ❸, the $load$ input of all PEs is set to 0 and, Mul_En is set to 1, which indicates PEs operate in the calculate mode.

4 Simulation Results

4.1 Simulation Workload

We used 12 PyTorch DNN models [15] and divided them into two groups to simulate the multi-tenancy environment in *heavy* and *light* loads. Table 1, illustrates the type and load of the workloads. The first group of DNNs is the heavy-load multi-domain type, and the second is the light-load recurrent neural network (RNN) type.

4.2 Simulation Toolchain

We simulate a systolic array similar to TPU v3 which comprises 128×128 PEs with load, feed, and drain buffers. We used the *Scale-Sim* simulator [16] to evaluate the computation time, and the energy consumption is estimated based on 45nm technology by *Accelergy* framework [17]. Scale-Sim is a dedicated systolic array simulator, and Accelergy is an open source tool for estimating accelerator area and energy consumption. As shown in Fig. 8, we employ both tools by extracting component activities in the form of a logfile from the Scale-Sim and then importing the log file into the

Figure 8: The Simulation toolchain

Table 1: Simulation workloads

Type	Domain	Training model
Multi-domain / Heavy load workload	Image classification	AlexNet[20]
		ResNet50[21]
		GoogleNet[22]
	Sentiment analysis	Sentiment analysis CNN (SA_CNN)[23]
		Sentiment analysis LSTM (SA_LSTM)[24]
	Recommendation system	Neural collaborative filter (NCF)[25]
	Intelligent search	AlphaGoZero[26]
	Natural language processing	Transformer [27]
Recurrent Neural Network / Lightload workload	Melody extraction	Melody LSTM [28]
	Language translation	Google Translate [29]
	Text to speech	Deep voice[30]
	Handwriting recognition	Handwriting LSTM [31]

Accelergy framework as a component activity. Accelergy uses *Cacti* [18] and *Aladdin* [19] plugins to estimate energy usage based on component activity.

4.3   Results

Fig. 9(a) and 9(b) show the computation time of our two workloads in the baseline systolic array with no partitioning algorithm and a systolic array with the dynamic partitioning algorithm. As we can see in Fig. 9(a) and 9(b), all DNNs run sequentially in baseline scenario but concurrently in the dynamic partitioning algorithm. DNNs such as AlexNet in multi-domain workloads or Google Translate in RNN workloads are completed later than other DNNs. However, these DNNs have been running in parallel with the other DNNs from the beginning, but the processing of DNNs with smaller dimensions is completed earlier.

Employing the dynamic allocation algorithm increases resource utilization, which reduces the total computation time due to the parallel execution of the layers. Fig. 9(c) and 9(d) show more details about assigned resource partitions to DNN layers.

According to Fig. 9(c), the processing of AlphaGoZero, NCF, and SA_CNN were completed using 128×16 partitions due to their low dimensional layers. The reason that all layers of NCF are processed by 128×16 partition, while 128×16 and 128×32 partitions process some layers of SA_LSTM or Transformer, is that the NCF layers are lighter (have lower dimensions) compared with SA_LSTM or Transformer DNNs.

As per our algorithm in Fig. 5, the allocated partitions become free once each layer is finished and can be merged with other adjacent free partitions to create a partition with more resources to speed up the remaining DNNs.

More complex DNNs with high dimension layers like ResNet50, GoogleNet, SA_LSTM, and Transformer use both 128×16 and 128×32 partitions, and the last two fully connected layers of AlexNet use all PEs to complete the processing. In the RNN workload (Fig. 9(d)), we see a similar case in which the handwriting LSTM and Deep voice DNNs are processed by a 128×32 partition. The last layer of Melody LSTM uses a 128×64 partition size, and the last six layers of Google translate use all PEs to finish the processing.

Fig. 9(e) and 9(f) show energy consumption of the baseline systolic array and the dynamic partitioning technique respectively. As shown in Fig. 9(e) and Fig. 9(f), utilizing a dynamic partitioning mechanism saves 35% in multi-domain

Figure 9: (a) The computation time of multi-domain workload (b) The computation time of RNN workload (c) The detailed computation time with assigned partition sizes of multi-domain workload (d) The detailed computation time with assigned partition sizes of RNN workload (e) The energy consumption of multi-domain workload (f) The energy consumption of RNN workload

workload and 62% in RNN workload compared to the baseline systolic array. Energy saving is achieved due to the efficient resource utilization of our dynamic partitioning algorithm.

5   Related Work

Multi-tenancy has been addressed for DNN accelerators in server and edge platforms. There are two main types of hardware accelerators for DNNs. *Spatial accelerators* consist of a network of processing elements connected by some sort of network on chip (NoC). Spatial accelerators provide a lot of flexibility in the division of work between PEs. In contrast, systolic arrays provide a fixed pattern of execution and communication between PEs. Systolic arrays can be extremely efficient because the PEs and intercommunication network are extremely simple and deterministic. In contrast, spatial accelerators provide much greater flexibility, at the cost of more complex hardware to support the flexibility.

Spatial accelerators are arguably much better suited to multi-tenancy in edge devices because they provide the flexibility for groups of PEs to work independently of others. Supporting multi-tenancy on a systolic array is much more difficult because the PEs work together in lock-step with a fixed data movement pattern. Where systolic array-based accelerators support multi-tenancy, it is almost always by providing separate systolic arrays for each tenant, rather than splitting one systolic array between tenants. In contrast, our work focuses on dividing a single systolic array between multiple tenants.

Edge-side multi-DNN processing by spatial accelerators. In edge platforms, DNN types and numbers are commonly fixed and predefined. Hence, a designer can modify an accelerator design according to the type of DNNs. For example, Herald [32] consists of multiple accelerators called sub-accelerators tuned for different dataflows to support multi-DNN processing. The sub-accelerators are divided among the tenants so that there is no need to share sub-accelerators.

Server-side multi-tenancy by systolic array based accelerators. Planaria [33] and AI-MT [34] are examples of server-side acceleration. Planaria comprises sixteen systolic arrays that are connected via an on-chip bi-directional ring bus. Planaria uses an extension of the systolic array called *omni-directional*. In the omni-directional type, the PE logic is modified to send the calculated partial sum to the output ports in all directions. Thus, the modified PE logic routes the tensor values in four directions, which turns the systolic array into a more general full spatial accelerator. Lee et al. proposed dataflow mirroring [35] for an omni-directional systolic array, to compute multiple DNN layers by different inter-PE communication patterns. As previously mentioned, we partition a single systolic array, and we require only a very small additional hardware cost compared with solutions like utilizing an omni-directional structure.

AI-MT enhanced the AI-multitasking hardware to a systolic array-based accelerator to divide each layer into sub-layers and categorize them into compute or memory-intensive types. Subsequently, the scheduler schedules sub-layers based on their type and runs them sequentially. In other words, AI-MT performs task scheduling at the hardware level.

Google performs multi-tenant inference and training operations on a supercomputer of connected Tensor Processing Units (TPU) [36]. Each TPU is a systolic array, but there is no support for partitioning the TPU. So, multi tenancy is performed by allocating different tenant DNNs to different TPUs. TPU v2 and v3 [37] comprise pods where each pod consists of eight TPU boards, and each TPU board includes eight systolic arrays. Thus, a TPU board is the primary cluster of systolic chips.

6   Conclusion

Multi-tenancy and multi-DNN processing are the new challenges in cloud and edge devices that affect computation time and energy consumption. This paper proposes a dynamic resource partitioning algorithm with a slight hardware modification and uses a data-level method to share an accelerator across several DNNs. Therefore, multiple DNNs use partitions of memory and processing components as sub-accelerators. We divide the PEs of the weight-stationary systolic array into vertical partitions that achieve data reuse in multi-tenant processing. At the hardware level we add a tri-state gate to the PE logic to allow data to flow through PEs without triggering computation. We apply two different types and loads of multi-DNN workloads to simulate the proposed idea. Simulation results show significant improvements in computation time and energy consumption due to increased resource utilization.

7   Acknowledgment

---

[2]https://lero.ie/

References

[1] Alexandros Karatzoglou and Balázs Hidasi. Deep learning for recommender systems. In *ACM RecSys*, pages 396–397, 2017.

[2] Sauhaarda Chowdhuri, Tushar Pankaj, and Karl Zipser. Multinet: Multi-modal multi-task learning for autonomous driving. In *IEEE WACV*, pages 1496–1504, 2019.

[3] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE JSSC*, 52(1):127–138, 2016.

[5] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices*, 53(2):461–475, 2018.

[6] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *ISCA*, pages 92–104, 2015.

[7] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *HPCA*, pages 331–344, 2019.

[8] Computational directions for augmented reality systems. *VLSI Symposium on Circuits Plenary Talk*, 2019.

[9] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *ISCA*, pages 1–14, 2018.

[10] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. INFaaS: Managed & model-less inference serving. *arXiv preprint arXiv:1905.13348*, 2019.

[11] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Using dataflow to optimize energy efficiency of deep neural network accelerators. *IEEE Micro*, 37(3):12–21, 2017.

[12] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *ISCA*, pages 1–12, 2017.

[13] Midia Reshadi and David Gregg. LOCAL: Low-complex mapping algorithm for spatial dnn accelerators. In *NorCAS*, pages 1–7, 2021.

[14] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *IEEE/ACM Microarchitecture*, pages 754–768, 2019.

[15] Pytorch. URL https://pytorch.org/.

[16] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim. In *ISPASS*, pages 58–68, 2020.

[17] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *ICCAD*, pages 1–8, 2019.

[18] Sheng Li, Ke Chen, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques. In *ICCAD*, pages 694–701, 2011.

[19] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *ISCA*, pages 97–108, 2014.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *CACM*, 60(6):84–90, 2017.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE CVPR*, pages 770–778, 2016.

[22] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE CVPR*, pages 1–9, 2015.

[23] Igor Santos, Nadia Nedjah, and Luiza de Macedo Mourelle. Sentiment analysis using convolutional neural network with fasttext embeddings. In *IEEE LA-CCI*, pages 1–5, 2017.

[24] Jin Wang, Liang-Chih Yu, K Robert Lai, and Xuejie Zhang. Dimensional sentiment analysis using a regional cnn-lstm model. In *ACL*, pages 225–230, 2016.

[25] Wanyu Chen, Fei Cai, Honghui Chen, and Maarten De Rijke. Joint neural collaborative filtering for recommender systems. *ACM TOIS*, 37(4):1–30, 2019.

[26] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[27] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NIPS*, 30, 2017.

[28] Hyunsin Park and Chang D Yoo. Melody extraction and detection through lstm-rnn with harmonic sum loss. In *IEEE ICASSP*, pages 2766–2770, 2017.

[29] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[30] Sercan Ö Arık, Mike Chrzanowski, Adam Coates, Gregory Diamos, Andrew Gibiansky, Yongguo Kang, Xian Li, John Miller, Andrew Ng, Jonathan Raiman, et al. Deep voice: Real-time neural text-to-speech. In *ICML*, pages 195–204. PMLR, 2017.

[31] Victor Carbune, Pedro Gonnet, Thomas Deselaers, Henry A Rowley, Alexander Daryin, Marcos Calvo, Li-Lun Wang, Daniel Keysers, Sandro Feuz, and Philippe Gervais. Fast multi-language lstm-based online handwriting recognition. *IJDAR*, 23(2):89–102, 2020.

[32] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *HPCA*, pages 71–83, 2021.

[33] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, et al. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *IEEE/ACM Microarchitecture*, pages 681–697, 2020.

[34] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A multi-neural network acceleration architecture. In *ACM/IEEE ISCA*, pages 940–953, 2020.

[35] Jounghoo Lee, Jinwoo Choi, Jaeyeon Kim, Jinho Lee, and Youngsok Kim. Dataflow mirroring: Architectural support for highly efficient fine-grained spatial multitasking on systolic-array npus. In *ACM/IEEE DAC*, pages 247–252, 2021.

[36] Norman P Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *CACM*, 63(7):67–78, 2020.

[37] Thomas Norrie, Nishant Patil, Doe Hyun Yoon, George Kurian, Sheng Li, James Laudon, Cliff Young, Norman Jouppi, and David Patterson. The design process for google's training chips: TPUv2 and TPUv3. *IEEE Micro*, 41 (2):56–63, 2021.