Layer-Puzzle: Allocating and Scheduling Multi-task on Multi-core NPUs by Using Layer Heterogeneity

Chengsi Gao[1,2,3], Ying Wang*[1,4], Cheng Liu[1,4], Mengdi Wang[1,2], Weiwei Chen[1,2], Yinhe Han[1,4], and Lei Zhang[1]

[1]SKLP, Institute of Computing Technology, CAS [2]University of Chinese Academy of Sciences
[3]Beijing Key Laboratory of Mobile Computing and Pervasive Device, Institute of Computing Technology, CAS
[4]State Key Laboratory of Computer Architecture, Institute of Computing Technology
{*gaochengsi17z, wangying2009, liucheng, wangmengdi17s, chenweiwei, yinhes, zlei*}*@ict.ac.cn*

***Abstract*—In this work, we propose Layer-Puzzle, a multi-task allocation and scheduling framework for multi-core NPUs. Based on the proposed latency-prediction model and dynamic parallelization scheme, Layer-Puzzle can generate near-optimal results for each layer under given hardware resources and traffic congestion levels. As an online scheduler, Layer-Puzzle performs a QoS-aware and dynamic scheduling method that picks the superior version from the previously compiled results and co-runs the selected tasks to improve system performance. Our experiments on MLPerf show that Layer-Puzzle can achieve up to 1.61X, 1.53X, and 1.95X improvement in ANTT, STP, and PE utilization, respectively.**

I. Introduction

The extraordinary accuracy and performance of deep neural networks(DNNs) make them widespread in many applications, such as autonomous driving [1] and augmented reality and virtual reality (AR/VR) [2]. Typically, real-world scenarios, whether in cloud AI or edge infrastructure, either face the concurrency of multi-tenant deep learning(DL) request [3]–[6], or invoke multiple DNN models to accomplish different sub-tasks of an application [7], [8], and these concurrent multi-tenant requests and multi-tasks need to be completed within a specified deadline so that the user experience can be guaranteed.

To support the efficient execution of DNNs, a plethora of neural processing units(NPUs) [9], [10] have been developed and deployed in the cloud and at the edge to accelerate the inference. However, as the scale and number of DNNs in applications continue to increase, single-core NPU can no longer meet the performance requirement, so the multi-core NPUs architecture has been proposed to address this problem [5], [11], [12]. In order to utilize the computational power of multi-core NPUs, the DNN model needs to be split into multiple parts and mapped to multiple NPUs to accomplish the computation. During the execution, multiple NPUs can process the models in parallel to achieve high throughput.

When supporting multiple DNN tasks on multi-core NPUs as in the cloud or edge servers, the resource allocation and scheduling among multiple DNN models needs to be re-investigated, which are critical factors that affect system performance. To support concurrent DNN models, two representative approaches have been proposed: temporal-sharing [4], [6], [12], in which DNNs occupy the entire hardware resources layer by layer and network by network, and spatial-sharing [5], [8], in which each DNN model is allocated fixed hardware resources. These two methods comprise the basis of accelerating multi-task on multi-core NPUs, however, both will result in low hardware resource utilization when individually and statically applied. This is because there exhibits a high degree of heterogeneity between different DNN models, as well as between successive layers within the same DNN model, resulting in different hardware utilization for the same hardware resources. For example, Fig 1 shows a PE utilization comparison between MobileNet [13] and Resnet18 [14] layer by layer and their average. They are both allocated 5x5 NPUs(see more details in Sec IV-A) and scheduled in a temporal-sharing manner. Both networks are parallelized in feature map (see more details in Sec II-A). As shown in Figure 1, the average PE utilization of MobileNet is lower than that of Resnet18 for the same hardware resources. Meanwhile, we can notice that different layers in the same network also exhibit varying PE utilization. However, previous works ignored these differences and assigned the network to the same or fixed resources, resulting in low hardware utilization and system performance [4], [11], [12]. In addition, we also observe that static or fixed allocation methods can not achieve optimal performance in dynamically varying traffic congestion scenarios, which is widespread while running multi-tasks on the multi-core NPUs architecture.

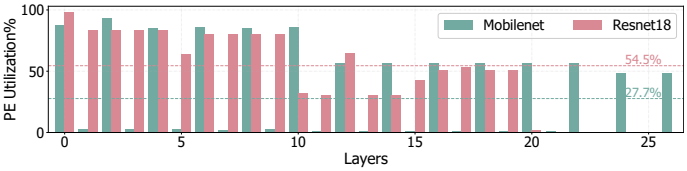


Figure 1. PE utilization comparison between MobileNet and Resnet18

Therefore, differing from previous work (see Table I for details), we propose Layer-Puzzle, a fine-grained multi-task allocation and scheduling framework for multi-core NPUs. Layer-Puzzle consists of an offline compiler and an online scheduler. When offline compiling, according to each layer's characteristic, Layer-Puzzle can generate near-optimal compilation results suitable for varying system states. Then, based on the detected runtime system states, Layer-Puzzle will perform a quality of service (QoS) aware and dynamic scheduling method that picks the superior version from the previous compiled results and co-run the selected tasks in a spatial-sharing way.

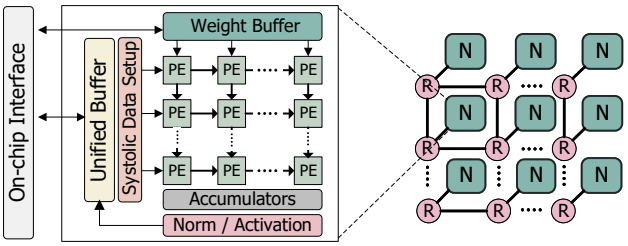
*Corresponding Author.

978-3-9819263-7-8/DATE23/© 2023 EDAA

TABLE I
COMPARISON DETAILS OF THE RELATED WORKS

Name	Allocation Method	Schedule Scheme	Schedule Granularity	Design Goal*	Hardware Utilization%	Qos Guaranteed
AI-MT [15]	Fixed	Temporal-sharing	Model	Latency, MT	Low for light NNs	Not mentioned
PREMA [4]	Fixed	Temporal-sharing	Layer	System Performance,MT	Low	YES
Planaria [5]	Static	Spatial-sharing	Layer	System Performance,MT	Low	YES
VELTAIR [3]	Dynamic	Spatial-sharing	Layer Block	System Performance,MT	Middle	YES
FGSpMt-NPU [8]	Fixed	Spatial-sharing	Model	System Performance,MT	Middle	Not mentioned
MAGMA [16]	Fixed	Spatial-sharing	A minibatch of a layer	System Performance,MT	Middle	Not mentioned
Layerweaver [6]	Fixed	Temporal-sharing	Layer	Resource Utilization, MT	Low	Not mentioned
NN-Baton [12]	Fixed	Temporal-sharing	Model	Energy, ST	Low	Not mentioned
Ours	**Dynamic**	**Spatial-sharing**	**Layer**	**System Performance, Resource Utili, MT**	**High**	**YES**

* MT and ST mean supporting for multi-task and single-task workload. System performance represents the performance metrics for multi-task systems, such as STP and ANTT.



(a) NPU architecture (b) Multi-core NPUs architecture

Figure 2. Architecture of NPU(a) and multi-core NPUs(b)

In brief, the main contributions of this paper are as follows:

1) We found that the neglected inter-layer heterogeneity is the main cause of low hardware utilization. Thus, we propose a layer-wise allocation method to better balance the PE utilization and performance speedup.

2) When online scheduling, Layer-Puzzle will detect the runtime system states and dynamic select the proper pending layers to execute.

3) We evaluated our framework with the MLPerf benchmark on a multi-core NPUs platform and compared it with three representative works. The results show that our framework can achieve up to 1.61X, 1.53X, and 1.95X improvement in ANTT, STP, and PE utilization, respectively.

II. BACKGROUND AND MOTIVATION

A. NPU and Multi-core NPUs

NPUs are designed to support the efficient execution of DNN models, and most NPUs are optimized for DNN inference [9], [10]. Fig 2(a) depicts the baseline NPU architecture(TPU [9]) in this work. It mainly consists of a systolic array that completes the convolution computation, a weight buffer that stores the weight, a unified buffer that stores the input feature maps(ifmaps), and output feature maps(ofmaps). When starting a computation, the NPU loads the data from the DRAM into the on-chip buffer and orchestrates the data into dataflow and flow through the PE array to complete the computation. Of all the the described dataflows in work [17], we adopt the commonly used output stationary as the dataflow for this paper. To meet the growing computation needs of DNNs, researchers scale up NPUs to multi-core architecture [5], [6], [11], [12]. A typical multi-core NPUs architecture is shown in Fig 2(b), which is also the baseline architecture of this work. It consists of multiple homogeneous NPUs, which are connected to routers. The data of each NPU is transmitted through the NoC. Therefore, when multiple NPUs execute in parallel, the data transmission congestion on NoC and DRAM is also a considerable factor in system performance. To utilize the computation power of multi-core NPUs, several DNN parallelization schemes have been proposed [11]. For a DNN layer, these parallelization schemes partition the corresponding data dimension and assign these sub-tiles to different NPUs for processing, which is called intra-layer parallelism [15]. Different parallelization schemes result in various PE utilization and data traffic, while the latency of the latter is significantly impacted by run-time traffic congestion. Thus, determining an optimal parallelization scheme for a layer also requires considering the system run-time status, which is not considered in previous work [5], [6], [11].

B. Supporting Multi-task on Multi-core NPUs

As multi-task applications that employ multiple DNNs become increasingly popular, prior works have proposed some methods to support multiple DNNs on the multi-core NPUs to speed up computations [4]–[6], [8], [12], [15]. Table I lists the detail of the related works, including their allocation method, schedule scheme and granularity. According to Table I, they can be summarized into two categories. (a) Temporal-sharing [4], [6], [12], [15]. In this approach, each layer of the DNNs will be allocated the entire hardware resource and is scheduled for execution in a sequential manner or interleaved with layers of other tasks. By using the entire hardware resources, the execution latency of a single DNN will be significantly reduced, while the multi-core NPUs will suffer from low hardware utilization, especially for lightweight DNNs. (b) Spatial-sharing [5], [8]. In this method, the entire hardware resources will be allocated to multiple DNNs for spatial co-running. The allocated resources of each DNN will be fixed and smaller. Thus, the PE utilization of each NPU will be improved than temporal-sharing. However, this method still ignored the high degree of heterogeneity in layer shape and operation exhibited in the individual DNN. For example, in Fig 1, the PE utilization varies widely for each layer in MobileNet. Thus, this method cannot fundamentally solve the problem that exists in temporal-sharing.

To compensate for the shortcomings of the above methods, researchers have proposed several improvements. Work [8] proposed an online resource re-allocation method, but it ignored the significant overhead of DNN online compiling caused by resource re-allocation. MAGMA [16] developed an optimization framework to map jobs on multiple accelerators, but it only supports mapping one layer on a accelerator core, and ignores compilation overhead and optimization. VELTAIR [3]

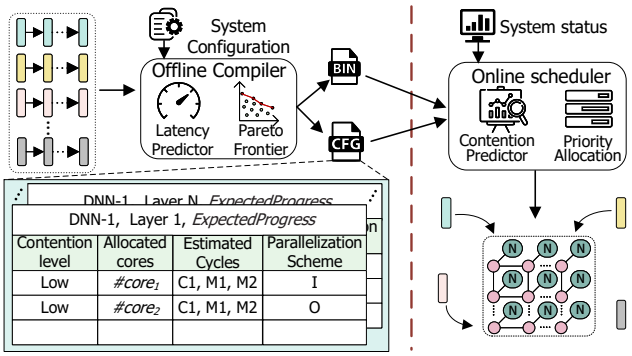


Figure 3. Overall workflow of our Layer-Puzzle

alleviates the problem of low hardware utilization through a layer-block granularity scheduling strategy, while the inter-layer heterogeneity is still underutilized. In contrast, our work takes full account of layer heterogeneity and adopts a dynamic allocation method and layer-wise scheduling strategy to improve hardware utilization and system performance.

III. Architecture of Layer-Puzzle

In this section, we will introduce Layer-Puzzle in detail. The overall workflow of Layer-Puzzle is shown in Fig 3. It consists of an offline compiler that generates compiled instructions and data, an online scheduler for performing layer-wise and QoS-aware multi-task scheduling in spatial-sharing manner.

A. Offline Compiler

Because of the significant overhead of DNN compilation, DNN inference tasks are typically compiled offline and then deployed online. The biggest challenge for the offline compiler is to determine the optimal parallelization scheme for each layer without knowing the run-time available resources and system states. To solve this problem, we adopt three novel methods.

1) Analytical prediction model

To estimate the latency and PE utilization of a DNN layer for a given hardware resource and parallelization scheme, we design an analytical prediction model. Although previous work [4], [15] proposed some latency prediction methods for DNNs, they were only applicable to a single NPU scenario and did not consider the data partition across multi-core NPUs under different parallelization schemes and the data transmission delay over NoC and DRAM due to traffic congestion, which accounts for a considerable portion of the total latency [18]. Therefore, both the computation and transmission latency are taken into account in our prediction model. The detail of our model is illustrated in Algorithm 1, and it shows how to estimate the total latency of a layer in the DNN. First, given the number of allocated NPUs $\#core$ and the parallelization scheme $paralle\_shcme$, the algorithm will compute the partitioned data dimensions of sub-tiles for a DNN layer. Then, according to the assigned sub-tile of each NPU($tiles_{perNPU}$), the height, width, and channel of the ofmap can be calculated. With the shape of the ofmap, we can get the number of times that ofmap will expand on the PE array vertically and horizontally (line 4-5) and the time required to drain the systolic array from vertical and horizontal (line 6-7). With PX_{perwin}, which represents the cycles needed for one output element, the computation

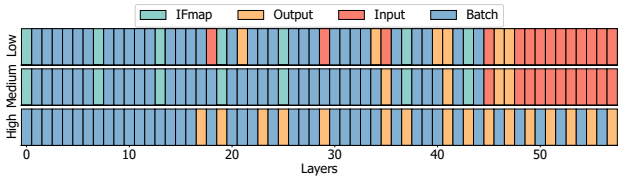


Figure 4. Optimal parallelization schemes of Googlenet under varying traffic congestion levels

latency(C_1) of this tile can be calculated (line 8-9). Line 10-11 are calculating the memory access latency of loading input and storing output. We equate the impact of traffic congestion as bandwidth BW multiplied by a discount factor mem_{factor}, since we only need the impact of traffic congestion on total memory access latency, so a precise and complex memory model is cumbersome and unnecessary, and our method can achieve a better trade-off between complexity and accuracy. At last, the total latency of this $tiles_{perNPU}$ can be estimated as shown in line 12. Based on the prediction model, the compiler can compare the performance between different parallelization schemes. To verify the accuracy of our prediction model, we compare it with a multi-core NPUs platform (see in Sec IV-A). Since mem_{factor} is determined by the run-time system states, here we only compare the results where neither has traffic congestion (mem_{factor}=1, experiment platform is occupied exclusively). The case with traffic congestion will be discussed in Sec III-B. We generated 8000 DNN layers with different shapes and numbers of allocated NPUs, and every layer is arbitrarily assigned a parallelization scheme. The results show that the prediction latency of our model never deviated more than 9%(average 4%) of the experiment platform.

2) Dynamic parallelization scheme

There are two major flaws that make previous work [4]–[6], [11], [15] difficult to achieve optimal parallelization schemes. First, they ignore the high degree of inter-layer heterogeneity in layer shape, and thus their fixed parallelization scheme will lead to low PE utilization for some layers. For example, assuming that all layers adopt the fmap parallelization [11], for later layers in the same DNN with a small ifmap size and a large output channels, the size of the ifmap allocated in each parallel core will be very small, occupying only a tiny fraction of the PE array. Thus, output parallelization is a superior choice for these layers. Second, They do not consider the extra memory access latency caused by traffic congestion, which can greatly impact the final latency. This is because different parallelization schemes [11] lead to variable amount

Algorithm 1: Latency Prediction Model

1 $TotalLatency = 0$;
2 $tiles_{perNPU}$ = **CalculateSubTiles**($\#cores$, $paralle\_scheme$);
3 H_{of}, W_{of}, C_{of} = **OutputShape**($tiles_{perNPU}$);
4 $fold_h = \lceil (H_{of} * W_{of})/ArrayH \rceil$;
5 $fold_w = \lceil C_{of}/ArrayW \rceil$;
6 $drain_h = (H_{of} * W_{of} - 1)\%ArrayH$;
7 $drain_w = (C_{of} - 1)\%ArrayW$;
8 $PX_{perwin} = H_{filter} * W_{filter} * C_{filter}$;
9 $C_1 = PX_{perwin} * fold_h * fold_w + max(drain_h, drain_w)$;
10 $M_1 = tiles_{perNPU}.inputsize()/(BW * mem_{factor})$;
11 $M_2 = tiles_{perNPU}.outputsize()/(BW * mem_{factor})$;
12 $TotalLatency = sum(M_1, C_1, M_2)$;
13 return $TotalLatency$

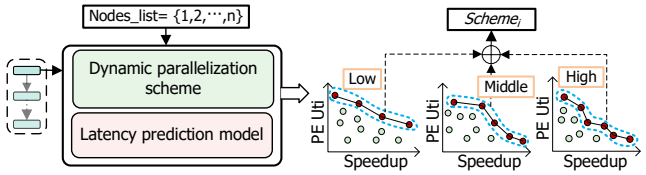


Figure 5. Detail of the allocation method

of memory access and on-chip data communication [12]. In high traffic congestion states, the performance of parallelization schemes with fewer data accesses may outperform the previous optimal one. Fig 4 shows the optimal layer-wise parallelization scheme for Googlenet(Batch=5) at different traffic congestion levels. Googlenet is assigned 5 NPUs and the platform's traffic congestion is manually configured to 3 levels. As the figure shows, a significant proportion of layers will adopt distinct parallelization schemes after the congestion level changes. Thus, to overcome the above problem, we propose a dynamic parallelization scheme to identify the optimal parallelization strategy for DNN layers under varied traffic congestion levels. For a DNN layer assigned with $\#core$ NPUs, we will use our prediction model to calculate the PE utilization of different parallelization schemes at varied congestion levels. Then, for each congestion level, the parallelization scheme with the highest PE utilization will be selected. Through experiments on a large number of common networks, we find that 3 congestion levels(low, medium, and high) are sufficient to cover most cases where the optimal parallelization scheme of the layers will change. Thus, in this work, we set 3 congestion levels, corresponding to mem_{factor} being set to 0.8, 0.5, and 0.2, respectively. Finally, each DNN layer i obtains a set $S_i^{\#core} = \{s_1, ..., s_n\}$ indicating selected parallelization schemes under given $\#core$ NPUs.

3) Allocation along the Pareto frontier

With the prediction model and dynamic parallelization scheme, we can easily obtain the optimal or near-optimal compilation results for a given DNN model for all allocation cases. However, simply saving all compiled versions will result in large storage overhead(25x3 times larger on a 5x5 scale). Besides, we discovered that some compiled versions consistently performed worse than others and were thus not worth saving. Thus, to achieve a better trade-off between the storage overhead and system improvement, we propose a novel NPU allocation method, as shown in Fig 5. When compiling each DNN, the compiler needs a $nodes\_list$ that contains the number of NPUs can be allocated to this DNN(default value of n is the scale of the platform). When processing each layer in the DNN, the compiler will use our prediction model and dynamic parallelization scheme to compute PE utilization and latency for each allocation scheme in $nodes\_list$ under different congestion levels. The latency is normalized to a single NPU to obtain the speedup ratio. Thus, each compilation result can be represented by a coordinate $(speedup, PE\ utilization)$. Since we have three congestion levels, the compiler will generate three corresponding coordinate systems for each DNN layer. After obtaining all results of a layer, the compiler will calculate the Pareto frontier in each coordinate system. Thus, for layer i, the compiled versions that lie on the Pareto frontier are preserved to form the set $Scheme_i$. Collecting the $Scheme_i$ of all layers in the DNN model yields the final results. Our experiments with numerous representative DNNs show that this compilation method can save about 44% store overhead.

As shown in Figure 3, in addition to the detailed compilation information of each layer, the compiler generates an $ExpectedProgress(EP)$ for each layer, which indicates the expected progress of the computation when reaching that layer. EP is defined in Equation 1 and will be used in online scheduling to calculate the task's priority.

$$EP = \frac{Operations\ elapsed\ until\ the\ current\ layer}{Total\ operations\ of\ current\ DNN\ model} \quad (1)$$

B. Online Scheduler

When online scheduling, Layer-Puzzle will schedule the pending layers in each DNN to meet their QoS. First, Layer-Puzzle calculates the priority of each DNN task. Then, based on the detected traffic congestion level of system, it dynamically chooses the superior version from compiled results. First, we introduce how to detect the system traffic congestion.

1) congestion Estimator

To select the superior version from the compiled results, we need to identify the runtime traffic congestion of the system, that is, to determine the value of mem_{factor}, which is equal to $\frac{ideal\ acess\ time}{real\ access\ time}$. Because the runtime traffic congestion level of the system is influenced by a variety of different factors, such as the number of active cores and the characteristics of the running layers, manully building a equation between mem_{factor} and these factors is complex and laborious task. Thus, we propose a learning-based congestion estimator with light overhead. We generate 10000 DNN layers(8000 for training, 2000 for testing) with different shapes and numbers of allocated NPUs, and every layer is arbitrarily assigned a parallelization scheme. Then, we deploy them in turn on the multi-core NPUs platform(see in Sec IV-A), while selecting arbitrary cores from the remaining free cores and deploying layers with various attributes on these cores. During the experiment, we discover that the value of mem_{factor} is highly connected to allocated cores of the generated layer, memory access intensity and active cores of the system. Thus, depending on these parameters, we construct several regression models to predict mem_{factor}. The memory access intensity is represented by the inverse of the average operational intensity [19](obtained at compilation phase) of the running layers. Finally, the test results show that RandomForestRegressor achieves a better accuracy(mean absolute error=5.1%) while incurring trivial overhead. Thus, we adopt RandomForestRegressor as the congestion estimator. For different platforms, the congestion estimator needs to be trained only once before it can be applied.

2) QoS-aware Scheduling

One of the main advantages of Layer-Puzzle as a scheduler is that it performs QoS-aware scheduling, capable of assigning an appropriate priority to each DNN task and scheduling the proper task for execution. The detail of the scheduling algorithm is shown in Algorithm 2. Whenever a layer accomplishes its computation, the scheduler will invoke the function $SchedulingTask$ to obtain new pending layers to

Algorithm 2: Multi-task Scheduling for Multi-NPUs

1 **Function** `CalculatePriority(`*pendingnets*`)`**:**
2 **for** *net* ***in*** *pendingnets* **do**
3 $Currentprogress = \frac{net.elapsedtime}{net.deadline}$;
4 $priority = net.Expectedprogress - Currentprogress$;
5 $prioritylist[net] = priority$;
6 return $prioritylist$;
7 **Function** `SchedulingTask(`*Tasklist*`)`**:**
8 **for** *net* ***in*** *Tasklist* **do**
9 **if** *net.notbusy* **then**
10 $pendingnets.append(net)$;
11 $prioritylist$ = `CalculatePriority(`*pendingnets*`)`;
12 $sort(prioritylist)$;
13 **for** *net* ***in*** *prioritylist* **do**
14 **for** $S_i^{\#core_j}$ ***in*** $Scheme_{net.currentlayer}$ **do**
15 **if** $\#core_j > sys.freeNPUs$ **then**
16 *continue*;
17 $mem_{factor} = CongesEsima(sys.status, \#core_j)$;
18 $s = Findbestversion(mem_{factor}, S_i^{\#core_j})$;
19 $pendinglayers.append(s)$;
20 $sys.freeNPUs -= s.\#core$;
21 return $pendinglayers$;

schedule. First, the scheduler verifies the state of each task and determines its priority (line 9-11). The priority of a task is defined as the difference between $ExpectedProgress$ and $Currentprogress$ (line 3-5), which means that the task will have a higher priority when the current progress is behind the expected progress. After acquiring the priority, starting with the highest priority task, the scheduler estimates the mem_{factor} for each $S_i^{\#core_j}$, who requires less NPUs than available resources system has, using the congestion estimator(line 13-17). Then, the scheduler recalculates the latency of each compiled version and selects the one with the lowest latency(line 18). At last, the scheduler obtains $pendinglayers$, a set of layers to be scheduled. Additionally, to avoid starvation of a task, when the number of times a task is continuously skipped scheduling reaches a threshold(3 in our experiment), the scheduler will stop scheduling other tasks until the system has sufficient NPUs for this task.

IV. Evaluation

A. Experimental Setup

To show the effectiveness of our framework, we developed a multi-core NPUs platform that consists of SCALE-sim [17], a cycle-accurate DNN accelerator simulator, and BookSim2 [20], a detailed cycle-accurate network-on-chip simulator. The simulation of the memory subsystem is the same as the previous work [4]. To present the scalability of Layer-Puzzle, we set up two scales of multi-NPUs architecture and two deadline factors under corresponding scales. We define the deadline of each task as ($Time_{isolated} * deadline\_factor$), where $Time_{isolated}$ refers to the task's isolated execution time on a single NPU. The detail of the experimental setup is shown in Table II. The workloads we used are five representative MLPerf [21] NNs with batch sizes of 1 and 4, and the details of each NN are presented in Table III. To quantify the effectiveness of Layer-Puzzle, we adopt four metrics: average normalized turnaround time(ANTT, lower-is-better), system throughput(STP, higher-is-better), service level agreement violated rate(SLA violated rate, lower-is-better), and PE utilization. We set up three baselines in our experiments: PREMA [4], a temporal-sharing multi-task method, Planaria [5], a spatial-sharing multi-task work with software-hardware co-design while we only apply its software part to our platform, and VELTAIR [3], a spatial-sharing multi-task method with layer-block schedule granularity. To apply VELTAIR on our experimental platform, we simulated its single-pass compilation with our compilation method and replace its interference proxy with our congestion estimator, while retaining its scheduling method.

TABLE II
EXPERIMENTAL SETUP DETAILS

Parameter	Value	
Systolic-array dimension	32x32	
PE frequency	700MHz	
Dataflow	output-stationary	
On-chip SRAM size(unified & weight)	8 & 4MB	
Multi-core NPUs scale	3x3	5x5
Deadline_factor	0.4	0.2
DRAM channels	4	
Bandwidth	175GB/s	
Router virtual channels & Buffer size	4 & 4	

TABLE III
DETAILS OF BENCHMARKS

Area	Name	# of layers	Input size
Vision	Googlenet	58	224x224x3
Vision	MobileNet-v1	27	224x224x3
Vision	Resnet50	54	224x224x3
Commerce	NCF	6	1x1x138000
Language	Transformer	891	1x1x33708

B. Experimental Results

1) ANTT and STP comparison

ANTT indicates the turnaround-time slowdown of a task when executing with other tasks compared to its isolated execution. The comparison of the ANTT between baselines and Layer-Puzzle under varying experiment setups is shown in Fig 6(a) and Fig 6(e). The results demonstrate that Layer-Puzzle can achieve the lowest ANTT across all experimental settings. More specifically, Layer-Puzzle can achieve an average of 1.61X, 1.33X and 1.22X improvement over PREMA, Planaria and VELTAIR across all experiments, respectively. This means that every task can execute on the multi-NPUs platform more efficiently and can be scheduled more fairly, thanks to our near-optimal compilation method and QoS-aware scheduler. In terms of STP, which quantifies accumulated single-task progress under multi-task execution, Layer-Puzzle can achieve the highest results in all experimental settings, as shown in Fig 6(b) and Fig 6(f). Compared with PREMA, Planaria and VELTAIR, Layer-Puzzle can improve STP by an average of 1.53X, 1.32X and 1.16X across all experiments, respectively. In addition, the results also reveal that Layer-Puzzle can achieve better system performance, including ANTT and STP, on bigger scale multi-NPUs. With more hardware resources, temporal-sharing method PREMA results in worse hardware utilization for the whole DNN model. Despite Planaria and VELTAIR can improve the system performance by co-running multi-task, the inter-layer heterogeneity is still underutilized. In contrast, Layer-Puzzle can obtain near-optimal compiled versions for each layer under varying hardware resources and adopt a layer-wise scheduling method to maximize the utilization of system resources.

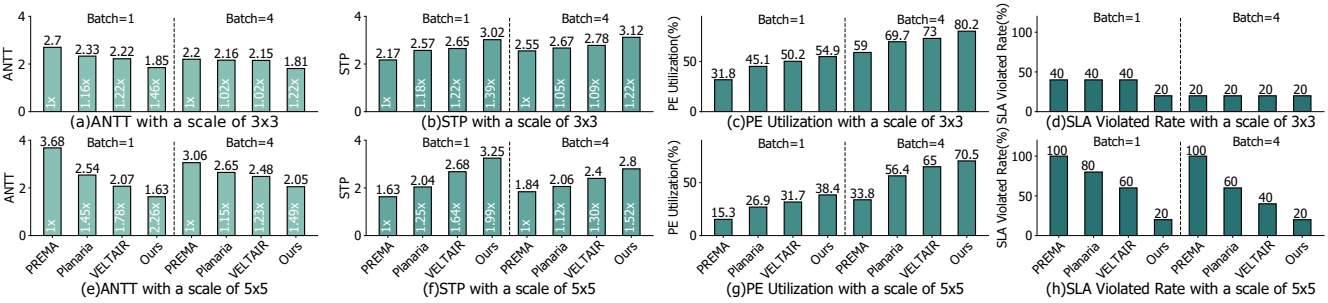


Figure 6. ANTT, STP, PE utilization, and SLA violated rate of the baselines and Layer-Puzzle with multi-task workloads

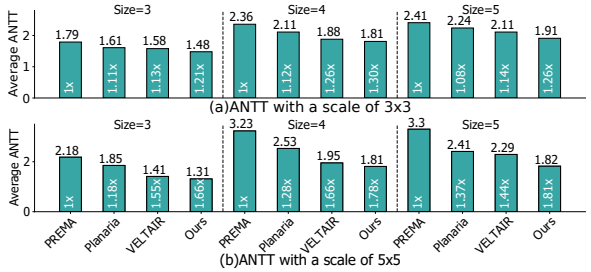


Figure 7. Average ANTT results under various multi-task workloads.

2) PE utilization and SLA comparison

The comparison of PE utilization between Layer-Puzzle and the baselines under varying experimental settings is presented in Fig 6(c) and Fig 6(g). Benefiting form the dynamic parallelization scheme, Layer-Puzzle can improve PE utilization by an average of 1.95X, 1.28X, and 1.13X over PREMA, Planaria, and VELTAIR, respectively. Fig 6(d) and Fig 6(h) illustrate the SLA violated rate comparison between Layer-Puzzle and baselines. With the expansion of the multi-NPU scale, the SLA violation rate of all baselines is increasing, while Layer-Puzzle can achieve a stable and the lowest SLA violated rate. We find that task NCF is the reason for the high SLA violation rate of baselines. NCF has the shortest deadline while it has poor parallel performance on multiple NPUs. Thus, during scheduling, PREMA schedules NCF first and assigns all hardware resources to it, and other tasks will be halted and delayed, resulting in high SLA violated rates. Although Planaria and VELTAIR can mitigate the impact of NCF by co-running other tasks, they still allocate additional needless NPUs to NCF. Instead, Layer-Puzzle assigns NPUs to NCF along its Pareto frontier, leaving more NPUs available for other tasks.

3) Workload sensitivity

To further verify the performance of Layer-Puzzle, we construct various multi-task workloads from the benchmarks. The workloads consist of different numbers of DNN models randomly picked from Table III with varying batch sizes. We set up 3 distinct sizes of workloads and generate 30 workloads in total. Due to length limitations, we only show the average ANTT results for each experimental setup, which is shown in Fig 7. As the figure illustrates, Layer-Puzzle can obtain the lowest ANTT under all experiments, indicating that Layer-Puzzle can achieve better performance for different multi-task workloads.

V. Conclusion

In this study, we present Layer-Puzzle, a multi-task allocation and scheduling framework for multi-core NPUs. By leveraging the dynamic parallelization and proper allocation scheme, it can generate near-optimal compilation results for each DNN task. By exploiting the fine-grained and QoS aware scheduler, Layer-Puzzle dynamically selects superior versions from compiled results for execution. According to the evaluation, Layer-Puzzle can significantly improve the system performance.

Acknowledgment

This work is supported by the National Natural Science Foundation of China under Grant 62222411 and Zhejiang Lab under Grants 2021PC0AC01.

References

[1] S. Lee, S. W. Oh, D. Won, and S. J. Kim, "Copy-and-paste networks for deep video inpainting," in *ICCV*, pp. 4413–4421, 2019.
[2] C.-J. Wu *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *HPCA*, pp. 331–344, IEEE, 2019.
[3] Z. Liu, J. Leng, Z. Zhang, Q. Chen, C. Li, and M. Guo, "VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling," in *ASPLOS*, pp. 388–401, ACM, 2022.
[4] Y. Choi *et al.*, "Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units," in *HPCA*, pp. 220–233, IEEE, 2020.
[5] S. Ghodrati *et al.*, "Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks," in *MICRO*, 2020.
[6] Y. H. Oh *et al.*, "Layerweaver: Maximizing resource utilization of neural processing units via layer-wise scheduling," in *HPCA*, IEEE, 2021.
[7] H. Kwon *et al.*, "Heterogeneous dataflow accelerators for multi-dnn workloads," in *HPCA*, pp. 71–83, IEEE, 2021.
[8] J. Lee *et al.*, "Dataflow mirroring: Architectural support for highly efficient fine-grained spatial multitasking on systolic-array npus," 2021.
[9] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *ISCA*, pp. 1–12, 2017.
[10] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," in *MICRO*, pp. 609–622, IEEE, 2014.
[11] M. Gao, X. Yang, J. Pu, *et al.*, "Tangram: Optimized coarse-grained dataflow for scalable nn accelerators," in *ASPLOS*, pp. 807–820, 2019.
[12] Z. Tan *et al.*, "Nn-baton: Dnn workload orchestration and chiplet granularity exploration for multichip accelerators," in *ISCA*, IEEE, 2021.
[13] A. G. Howard *et al.*, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.
[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, pp. 770–778, 2016.
[15] E. Baek, D. Kwon, and J. Kim, "A multi-neural network acceleration architecture," in *ISCA*, pp. 940–953, IEEE, 2020.
[16] S.-C. Kao *et al.*, "Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores," in *2022 HPCA*, IEEE.
[17] A. Samajdar *et al.*, "Scale-sim: Systolic cnn accelerator simulator," *arXiv preprint arXiv:1811.02883*, 2018.
[18] L. Yang, W. Liu, P. Chen, N. Guan, and M. Li, "Task mapping on smart noc: Contention matters, not the distance," in *DAC*, pp. 1–6, 2017.
[19] S. Williams *et al.*, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, 2009.
[20] N. Jiang *et al.*, "A detailed and flexible cycle-accurate network-on-chip simulator," in *ISPASS*, IEEE, 2013.
[21] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, *et al.*, "Mlperf inference benchmark," in *ISCA*, pp. 446–459, IEEE, 2020.