TLP Balancer: Predictive Thread Allocation for Multitenant Inference in Embedded GPUs

Minseong Gil, Jaebeom Jeon, Junsu Kim, Sangun Choi, Gunjae Koo, *Member, IEEE*, Myung Kuk Yoon, *Member, IEEE*, and Yunho Oh, *Member, IEEE*

***Abstract*—This letter introduces a novel software technique to optimize thread allocation for merged and fused kernels in multitenant inference systems on embedded graphics processing units (GPUs). Embedded systems equipped with GPUs face challenges in managing diverse deep learning workloads while adhering to quality-of-service (QoS) standards, primarily due to limited hardware resources and the varied nature of deep learning models. Prior work has relied on static thread allocation strategies, often leading to suboptimal hardware utilization. To address these challenges, we propose a new software technique called thread-level parallelism (TLP) Balancer. TLP Balancer automatically identifies the best-performing number of threads based on performance modeling. This approach significantly enhances hardware utilization and ensures QoS compliance, outperforming traditional fixed-thread allocation methods. Our evaluation shows that TLP Balancer improves throughput by 40% compared to the state-of-the-art automated kernel merge and fusion techniques.**

***Index Terms*—Embedded graphics processing unit (GPU), inference, multitenancy.**

I. Introduction

RECENTLY, embedded systems equipped with graphics processing units (GPUs) run various deep learning workloads [1], [2]. These systems have become crucial in maintaining stringent quality-of-service (QoS) standards. However, the inherent limitation of hardware resources in embedded GPUs and the diverse nature of deep learning models pose significant challenges in workload management [3]. These challenges involve optimizing hardware utilization without compromising service latency and queuing delay.

In embedded GPUs, effectively utilizing thread-level parallelism (TLP) is crucial for maximizing performance and efficiency, especially in resource-constrained environments. Fully exploiting TLP in GPUs allows for the simultaneous execution of multiple threads, enabling GPUs to make full use of their computational resources. Prior work, such as kernel merging and fusion, has been proposed to improve throughput and reduce latency in the systems that serve requests requiring inference with various AI models [4]. However, the prior work often relies on static thread allocation strategies that fail to adapt to the varying needs of different models. This inability to dynamically allocate threads based on workload characteristics can lead to inefficient hardware utilization.

We propose a novel software technique called TLP Balancer to address the above challenge. We design a TLP Balancer to determine the best-performing thread allocation in multitenant inference systems. The proposed technique bridges the performance gap left by static thread allocation strategies by automatically determining the best-performing number of threads for the active tasks. TLP Balancer performs the kernel merge and fusion process by monitoring the inference request queue to minimize the number of concurrent kernels inspired by the prior work [4]. Unlike prior work, our technique predicts the thread count based on model characteristics and workload, aiming to minimize inference time and maximize throughput while merging and fusing kernels. Through a comprehensive evaluation conducted on the NVIDIA Jetson Orin platform and deep learning models, the proposed technique achieves a 40% improvement in throughput compared to the kernel merge and fusion technique.

II. Motivation and TLP Balancer

A. Why TLP Balancer?

Embedded GPUs often serve as multitenant inference systems, processing numerous requests concurrently given a strict QoS requirement [5]. These systems run various deep learning models, each distinguished by characteristics, such as the number of layers and the size of tensors. Unlike discrete GPUs that support parallel executions of multiple GPU kernels with a dedicated technique, such as the MultiInstance GPU, embedded GPUs run multiple kernels by sharing GPU hardware in a time-sharing manner [6]. As such, embedded GPUs incur frequent context switches, which increase queuing delay and average inference time and throughput [7].

Prior work has proposed vertical and horizontal kernel fusion techniques that can run multiple kernels in parallel [8], [9], [10]. Vertical fusion combines sequentially

Received 21 August 2024; revised 21 October 2024; accepted 4 November 2024. Date of publication 14 November 2024; date of current version 13 June 2025. This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT under Grant NRF-2022R1C1C1011021 and Grant NRF-2021R1C1C1012172, and in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korea Government (MSIT) through Artificial Intelligence Innovation Hub under Grant RS-2021-II212068. This manuscript was recommended for publication by F. Merchant. *(Corresponding authors: Myung Kuk Yoon; Yunho Oh.)*

Minseong Gil, Jaebeom Jeon, Junsu Kim, Sangun Choi, and Yunho Oh are with the School of Electrical Engineering, Korea University, Seoul 02841, Republic of Korea (e-mail: yunho_oh@korea.ac.kr).

Gunjae Koo is with the Department of Computer Science, Korea University, Seoul 02841, Republic of Korea (e-mail: gunjaekoo@yonsei.ac.kr).

Myung Kuk Yoon is with the Department of Computer Science, Ewha Womans University, Seoul 03760, Republic of Korea (e-mail: myungkuk.yoon@ewha.ac.kr).

Digital Object Identifier 10.1109/LES.2024.3497587

invoked kernels into a single kernel, eliminating global memory accesses for intermediate results. Horizontal fusion distributes the execution of multiple kernels across different threads within a single kernel. This technique allows kernels that use different resources to execute in parallel, thus better utilizing GPU resources. In multitenant inference systems, the consecutive kernel requests are typically independent. Thus, the successive kernels do not share intermediate results, which diminishes the advantage of vertical fusion for multitenant systems. As such, horizontal fusion is more suitable for multitenant systems than vertical fusion [11]. Horizontal fusion can distribute the computation of consecutive and independent kernels either across threads within the same block or across different blocks.

To address the challenge above, an adaptive kernel merge and fusion technique has been proposed [4]. The kernel merge and fusion technique consolidates kernels that run the same model into a single kernel and applies kernel fusion to these merged kernels. During kernel merging, a kernel is repeatedly executed using inputs from multiple requests or outputs from previously executed kernels. Kernel fusion is applied to the merged kernels, reducing the number of concurrent kernels and the overhead linked to kernel launches. This approach minimizes memory access demands by reusing the same weights across different requests.

Even in the kernels created by the merge and fusion technique, inefficiencies arise if the thread configuration is determined without considering various critical factors related to the launched kernels. Key factors, such as the number of merge requests, input size, and tensor dimensions, significantly influence the optimal thread configuration. These factors affect the computational workload and memory operations required for each of the merged kernels being fused. As such, allocating an appropriate number of threads to each merged kernel is crucial to maximize GPU hardware utilization.

We investigate the performance impact of thread allocation using the previously proposed kernel merge and fusion technique [4]. Our study employs ResNet50 and ViT-Base, generating 1000 requests that randomly demand either of the two models. Initially, we manually determine the thread allocations that yield the shortest execution time (Oracle) by varying the number of merge requests. We determine the Oracle thread configuration by measuring the execution time for all possible thread configurations and selecting the one with the shortest execution time. We then compare the execution times of the Oracle thread configurations to those of the static thread allocation, as exactly implemented in the prior work [4], which evenly allocates threads within an SM to inference tasks in a merged and fused kernel (e.g., consistently allocating 24 warps per kernel). Our methodology involves two models, which are fused into a single kernel by applying the inner thread block fusion technique. We use the NVIDIA Jetson Orin system for the experiments [12]. Table I describes the detailed configurations. We configure the number of thread blocks to 24 and the thread block size to 768, enabling the GPU to fully utilize the hardware. Also, considering the GPU architecture, we allocate the number of threads for a kernel in units of warp size (i.e., 32). We allocate a maximum of 48 warps per SM and distribute these warps to the merge and fused tasks within a kernel.

TABLE I
EXPERIMENT CONFIGURATION

Platform	NVIDIA Jetson Orin AGX		
GPU architecture	Ampere architecture		
GPU cores	8 SMs, 1792 CUDA cores, 56 Tensor cores		
Max. # of threads per SM/block	1536/1024		
CPU	8-core Arm Cortex-A78AE v8.2		
Memory	32GB LPDDR5, Bandwidth: 204.8GB/s		
DNN models	ViT-Base	ResNet50	VGG16
Number of Parameters	86M	25.6M	138M

TABLE II
FIVE EXAMPLES OF FUSED LAYERS, EACH WITH TWO KERNELS EXECUTING DIFFERENT MODELS. ORACLE THREAD CONFIGURATION IS THE NUMBER OF THREADS ALLOCATED TO EACH MODEL THAT YIELDS THE SHORTEST EXECUTION TIME. ALL RESULTS ARE NORMALIZED TO THE ADAPTIVE KERNEL MERGE AND FUSION TECHNIQUE THAT EVENLY ALLOCATES THREADS TO BOTH TASKS

Layer Name	**Layer 1**	**Layer 2**	**Layer 3**	**Layer 4**	**Layer 5**
Oracle Thread Configuration (ResNet50, ViT-Base)	24, 24	16, 32	12, 36	5, 43	2, 46
Speedup	1.00	1.30	1.47	1.78	1.92

TABLE III
FIVE EXAMPLES OF THE DIFFERENT NUMBERS OF MERGE REQUESTS. ALL RESULTS ARE NORMALIZED TO THE ADAPTIVE KERNEL MERGE AND FUSION TECHNIQUE

Number of Merge Requests (ResNet50, ViT-Base)	**1, 1**	**1, 2**	**1, 3**	**2, 1**	**3, 1**
Oracle Thread Configuration (ResNet50, ViT-Base)	12, 36	11, 37	9, 39	14, 34	16, 32
Speedup	1.47	1.53	1.60	1.39	1.31

Table II shows the experimental results for five out of fifty fused layers, including those with the maximum and minimum speedup. As each layer computes matrices of different sizes, the five layers achieve optimal performance with varying thread configurations. Layer 5 requires 46 warps for ViT-Base and two warps for ResNet50 to achieve the best performance. This thread allocation results in a 1.92× speedup compared to the prior work. In contrast, Layer 1 achieves maximum performance with an equal number of warps allocated to both models, identical to the thread allocation approach used in prior work. Therefore, the Oracle thread configuration in Layer 1 does not achieve any performance gain. Such layers represent a small fraction, comprising only one out of 50 layers. The effectiveness of the prior work diminishes as the disparity in the number of warps allocated to each model increases.

Table III shows the results for Layer 3 with varying numbers of merge requests. Even for the same layer, the number of merge requests alters the computational load and the number of memory operations. The Oracle thread configurations change, leading to speedups ranging from 31% to 60%.

Based on the experimental results, we observe that embedded GPUs can achieve better throughput in multitenant inference environments not only through kernel merge and fusion techniques but also with a carefully designed thread allocation policy.

B. TLP Balancer

We propose a new software technique called TLP Balancer to address the challenges in the previous section. The key idea

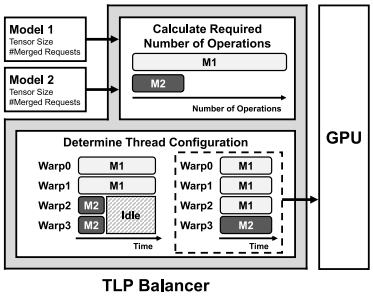

Fig. 1. Thread allocation technique of TLP balancer.

of TLP Balancer is automatically identifying the number of threads for all active tasks running on a GPU. We exploit the behavioral characteristics of inference to identify the thread counts of all the tasks. Inferences with the same AI model exhibit regular and therefore, predictable behaviors. TLP Balancer determines the best-performing thread count using a prediction mechanism before a kernel launch.

Fig. 1 depicts how TLP Balancer works. TLP Balancer exploits the implementation of the adaptive kernel merge and fusion [4]. The prior work has a software request queue. It monitors every five requests in the queue and analyzes them to identify how to merge and fuse them. In this technique, a software kernel merger inserts the codes that iterate the same kernel with inputs from each request. The kernel merger repeats the above process for the kernels of all the layers in the model. After the kernel merger creates the new kernels for all the models, the kernel fuser determines the models to fuse based on the parameter counts of the merged kernels. The kernel fuser appends conditional statements that use thread ID as a condition, allowing different requests to be run in a kernel.

TLP Balancer ensures that each merged kernel has a similar execution time, maximizing the overall TLP during execution. As shown in the right part of Fig. 1, the threads for the merged kernel with a shorter execution time complete first and remain idle until the other merged kernels complete their tasks. With the proposed prediction technique, TLP Balancer enables the GPU to operate kernels at peak TLP, reducing inference time and enhancing throughput.

We design the prediction technique by mathematically modeling the required number of operations for merged and fused kernels. Our performance model quantifies the number of operations per thread, denoted as M_{ops}, as follows:

$$M_{\text{ops}} = \lceil \frac{M_{\text{out\_size}}}{\#blocks \times M_{\text{th}}} \rceil \times M_{\text{in\_width}} \times \left(1 + M_{\text{merge}}\right) \quad (1)$$

where M_{th} represents the number of threads assigned within a block for a Model M. $M_{\text{out\_size}}$ and $M_{\text{in\_width}}$ represent the output size and the width of the input matrix of Model M, respectively. M_{merge} represents the number of merge requests.

The first factor on the right-hand side of (1) indicates the number of output parameters each thread computes. We implement the kernels to distribute the computation of output parameters evenly among each thread. Therefore, the number of output parameters each thread computes is equal to the output size divided by the total number of threads. The second factor represents the operations needed to compute each output parameter, including fetching each input data element and its corresponding weight from memory and multiplying them across the width of the input matrix. The third factor reflects the number of memory fetches, which varies depending on the number of merge requests. In inference, the number of input data fetches increases proportionally with the number of merge requests. Unlike the input data, the merged and fused kernel shares the same weight across multiple inputs, so the number of weight fetches does not change with the number of merge requests. As such, we multiply the third factor, 1, for weight fetch and M_{merge} for input fetch.

As the fused kernel executes the models in parallel, a kernel with the highest operation count becomes the critical path. For a kernel fusing n models, TLP Balancer calculate the number of operations (K_{th}) determining the inference time as follows:

$$K_{\text{th}} = \max\left(M1_{\text{ops}},\ M2_{\text{ops}},\ \ldots,\ Mn_{\text{ops}}\right) \quad (2)$$

where Mn_{ops} represents the number of operations per thread for Model n. To predict the best-performing number of threads, the TLP Balancer calculates the K_{th} for all possible thread configurations. It selects the thread configuration that minimizes the K_{th} as the best-performing number of threads.

With the merge and fusion technique, we implement TLP Balancer as follows. After the kernel fuser determines the models to fuse, the TLP Balancer calculates the number of threads allocated to each model. TLP Balancer calculates the required number of operations for all possible thread configurations using (1) and (2). Then, the TLP Balancer finds the thread configuration with the minimum required operations. After the TLP Balancer determines the thread configuration, the kernel fuser sets the conditions for the conditional statements to ensure threads are allocated following the determined configuration.

TLP Balancer introduces software overhead while predicting the best-performing thread configuration. For a fused kernel that fuses two models, the prediction takes 1 μs. While the overhead increases with more fused models, it takes under 1% of inference time for fusing three and four models, with overheads of 30 μs and 500 μs, respectively, so it is negligible.

III. Evaluation

For evaluation, we measure average inference time, throughput (requests per second), average queuing delay, and the 99th percentile queuing delay. We implement a request generator that evenly generates 1000 inference requests for ResNet50, ViT-Base, and VGG16 in random order with a Poisson distribution by referring to the prior work [4], [13]. We evaluate a sequential kernel execution that executes a single request at a time as the baseline and the kernel merge and fusion technique for comparison [4]. In our initial study, we find that the two methods show no significant difference in performance. Therefore, we distribute the workloads across threads within the same block. We conduct all the experiments on the NVIDIA Jetson Orin platform.

First, we compare the average inference time of all the implementations. To measure the execution time of each request, we divide the time taken from the kernel launch to its completion by the number of requests being merged and fused. Fig. 2(a) shows the experimental results. TLP Balancer

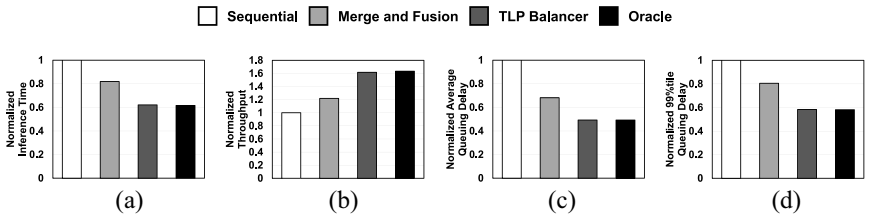

Fig. 2. Experimental results with 1000 requests. All the results are normalized to the results of sequential execution. Merge and Fusion allocate the thread to models evenly. For throughput, higher is better. For other metrics, lower is better. (a) Average inference time. (b) Throughput. (c) Average queuing delay. (d) 99%tile queuing delay.

TABLE IV
FIVE EXAMPLES OF KERNELS AND PERFORMANCE OF TLP BALANCER AND ORACLE CONFIGURATION. ALL RESULTS ARE NORMALIZED TO THE ADAPTIVE KERNEL MERGE AND FUSION TECHNIQUE

Layer Name	Layer 1	Layer 2	Layer 3	Layer 4	Layer 5
Number of Merge Requests (ResNet50, ViT-Base)	1, 1	2, 3	2, 2	3, 2	4, 1
Oracle Thread Configuration (ResNet50, ViT-Base)	2, 46	9, 39	24, 24	28, 20	33, 15
Oracle Speedup	1.92	1.61	1.00	1.14	1.23
TLP Balancer Thread Configuration (ResNet50, ViT)	4, 44	8, 40	24, 24	28, 20	33, 15
TLP Balancer Speedup	1.79	1.57	1.00	1.13	1.23

reduces the average inference time by 38% and 20% compared to the baseline and the merge and fusion, respectively. TLP Balancer finds the best-performing thread configuration for each layer, resulting in an iso-Oracle average inference time. Although the merge and fusion technique reduces the inference time, the unbalanced thread allocation incurs a 20% longer inference time than the oracle.

Fig. 2(b) presents the throughput (requests per second without violating QoS requirements) of all the techniques. The TLP Balancer significantly improves throughput by 62% and 40% compared to baseline and the adaptive kernel merge and fusion technique, respectively. Both the prior work and TLP Balancer improve throughput by eliminating memory access for merged requests and minimizing the number of concurrent kernels. However, the throughput of the prior work is 40% lower than the Oracle incurred by a longer inference time.

Fig. 2(c) and (d) presents the average and 99%tile queuing delay results, respectively. TLP Balancer exhibits the same average and 99%tile queuing delays as the Oracle results, which are 51% and 42% shorter than the baseline, respectively. If many requests arrive in a short period, parallel execution of multiple requests and reducing inference time allow requests to be processed quickly, decreasing the queue waiting time. Also, the TLP Balancer achieves a 22% shorter 99%tile queuing delay than the adaptive kernel merge and fusion technique due to a shorter inference time.

Table IV presents the thread configuration for the Oracle and TLP Balancer, along with their perspective speedups of kernels. We select five layers with different thread configurations, including those with maximum and minimum speedups, and present them along with their number of merge requests. In three out of five layers, the TLP Balancer calculates thread configurations identical to the Oracle cases. As such, the TLP Balancer achieves the same performance as the Oracle. TLP Balancer shows a 13% error in Layer 1, but this layer results in only a 2% increase in the overall inference time. These experimental results demonstrate that TLP Balancer can accurately calculate the thread configuration regardless of the layer type or the number of merge requests.

The evaluation shows that TLP Balancer performs very similarly to the Oracle results by adjusting TLP based on the size of the fused models and the number of merge requests.

IV. CONCLUSION

In this letter, we propose a TLP Balancer that significantly advances the efficiency and performance of multitenant inference systems on embedded GPUs. TLP Balancer predicts optimal thread allocation for diverse deep learning workloads, enhancing throughput while ensuring robust adherence to QoS standards. TLP Balancer achieves a 40% improvement in throughput over the prior work.

REFERENCES

[1] E. Jeong, J. Kim, S. Tan, J. Lee, and S. Ha, "Deep learning inference parallelization on heterogeneous processors with TensorRT," *IEEE Embed. Syst. Lett.*, vol. 14, no. 1, pp. 15–18, Mar. 2022.
[2] J. Xu, B. Wang, J. Li, C. Hu, and J. Pan, "Deep learning application based on embedded GPU," in *Proc. 1st Int. Conf. Electron. Instrum. Inf. Syst.*, 2017, pp. 1–4.
[3] N. Bouhali, H. Ouarnoughi, S. Niar, and A. A. El Cadi, "Execution time modeling for CNN inference on embedded GPUs," in *Proc. Drone Syst. Eng. Rapid Simul. Perform. Evaluat., Methods Tools*, 2021, pp. 59–65.
[4] J. Jeon, G. Koo, M. K. Yoon, and Y. Oh, "Adaptive kernel merge and fusion for multi-tenant inference in embedded GPUs," *IEEE Embed. Syst. Lett.*, early access, Jan. 9, 2024, doi: 10.1109/LES.2024.3351753.
[5] W. Pang, X. Luo, K. Chen, D. Ji, L. Qiao, and W. Yi, "Efficient CUDA stream management for multi-DNN real-time inference on embedded GPUs," *J. Syst. Archit.*, vol. 139, Jun. 2023, Art. no. 102888.
[6] H. Zhou, S. Bateni, and C. Liu, "S^3DNN: Supervised streaming and scheduling for GPU-accelerated real-time DNN workloads," in *Proc. IEEE Real-Time Embed. Technol. Appl. Symp.*, 2018, pp. 190–201.
[7] Z. Bai, Z. Zhang, Y. Zhu, and X. Jin, "PipeSwitch: Fast pipelined context switching for deep learning applications," in *Proc. 14th USENIX Symp. Oper. Syst. Design Implement.*, 2020, pp. 499–514.
[8] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, "Optimizing CUDA code by kernel fusion: Application on BLAS," *J. Supercomput.*, vol. 71, pp. 3934–3957, Oct. 2015.
[9] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 191–202.
[10] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic horizontal fusion for GPU kernels," in *Proc. IEEE/ACM Int. Symp. Code Gener. Optim.*, 2022, pp. 14–27.
[11] H. Zhao et al., "Tacker: Tensor-CUDA core kernel fusion for improving the GPU utilization while ensuring QoS," in *Proc. IEEE Int. Symp. High-Perform. Comput. Architect.*, 2022, pp. 800–813.
[12] S. Karumbunathan, *NVIDIA Jetson AGX Orin Series: A Giant Leap Forward for Robotics and Edge AI Applications*, Santa Clara, CA, USA, 2022.
[13] V. J. Reddi et al., "Mlperf inference benchmark," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 446–459.