# Adaptive Kernel Merge and Fusion for Multi-Tenant Inference in Embedded GPUs

Jaebeom Jeon, Gunjae Koo, *Member, IEEE,* Myung Kuk Yoon*, *Member, IEEE,* Yunho Oh*, *Member, IEEE,*

***Abstract*—This paper proposes a new scheme that improves throughput and reduces queuing delay while running multiple inferences in embedded GPU-based systems. We observe that an embedded system runs inference with a fixed number of deep learning models and that inference requests often use the same model. Unlike prior work that proposed kernel fusion or scheduling techniques, this paper proposes a new software technique that merges and fuses kernels by monitoring the requests in a queue. The proposed technique first monitors a fixed number of requests and groups the requests running the same model. Then, it creates the kernels that iteratively process the grouped requests. We call such a technique kernel merging. After that, the proposed technique performs kernel fusion with merged kernels. Eventually, our idea minimizes the number of concurrent kernels, thus mitigating stalls caused by frequent context switching in a GPU. In our evaluation, the proposed kernel merge and fusion achieve 2.7× better throughput, 47% shorter average kernel execution time, and 63% shorter tail latency than prior work.**

***Index Terms*—Embedded GPU, inference, multi-tenancy.**

## I. Introduction

EMBEDDED systems run Artificial Intelligence (AI) inference for various purposes. For example, automotive vehicles are supposed to perform real-time car and pedestrian detection for safety [1]. Embedded systems should deal with numerous inference requests given stringent requirements on Quality-of-Service (QoS). To satisfy QoS, a system should mitigate queuing delay and improve throughput.

Recent embedded systems equip Graphics Processing Units (GPUs) to offer high throughput. Embedded GPUs should be able to run multiple tasks efficiently. Conventional discrete GPUs support parallel execution of multiple kernels, such as multi-instance GPUs and multi-process service [2]. Unlike discrete GPUs, embedded GPUs do not support such functionalities. Both discrete GPUs and embedded GPUs support multi-stream executions. However, multi-stream executions still cannot overlap the executions of multiple kernels in embedded GPUs. So, if an embedded GPU launches multiple kernels simultaneously, it runs them concurrently (i.e., a GPU runs a single kernel in a time-sharing manner). Such a mechanism does not improve throughput as the number of active kernels given a fixed time window becomes the same as the sequential execution of the kernels. Also, concurrent kernel execution incurs frequent context switching, thus suffering from stalls due to context switching overhead. As such, it increases the average kernel execution time. Prior work has proposed a technique that schedules operators (or kernels) of concurrent tasks with a breadth-first issuing [3]. Such a scheme could reduce queuing delays. However, prior work eventually runs a single task in a period, so it still suffers from underutilization of the GPU hardware resources. Also, if thousands of inference requests are congested in an embedded system, the prior work still suffers from stalls due to frequent context switching and a long queuing delay.

To address the challenge above, we propose a new software technique that improves throughput and reduces queuing delay, called automatic kernel merge and fusion. The key idea is to minimize the number of concurrent kernels with the following two techniques: kernel merge and fusion. We define 'kernel merge' as a new technique that iterates a single kernel with the inputs from multiple requests or the results from previously executed kernels. Unlike concurrent kernel execution, the kernel merge scheme reduces off-chip memory access as it fetches filters of a deep learning layer only once, thus reducing the average execution time of a layer. For the requests running different models, we proposed to fuse the kernels. In inference, depending on the number of parameters, a single task execution time varies. With this insight, we propose a new software technique that determines kernels to be fused. Among the requests in a queue, our technique monitors a fixed number of requests from the queue head. Based on a predetermined number, the proposed software distinguishes target models into large models and small models. After the kernel merge, the software fuses kernels running the small and large models, respectively. We implement a software framework that performs kernel merge and fusion before the system starts processing inference.

In our evaluation, the proposed technique achieves 2.7× throughput improvement, a 50% reduction in tail latency compared to prior work.

## II. Motivation and Proposed Idea

### A. Why Merge and Fusion?

Embedded systems often run inference based on a concept of server-client systems that serve and respond to requests

Jaebeom Jeon and Yunho Oh are with the School of Electrical Engineering, Korea University, Seoul, Republic of Korea (e-mail: 414dragon@korea.ac.kr and yunho_oh@korea.ac.kr).

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

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

*Myung Kuk Yoon and Yunho Oh are the co-corresponding authors.

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2022R1C1C1011021, and NRF-2021R1C1C1012172) and in part by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-02068, Artificial Intelligence Innovation Hub).

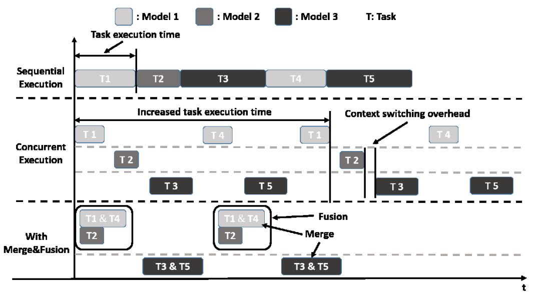


Fig. 1. Sequential execution, concurrent execution, and the proposed kernel merge and fusion on embedded GPU. GPUs perform context switching once they detect a stall.

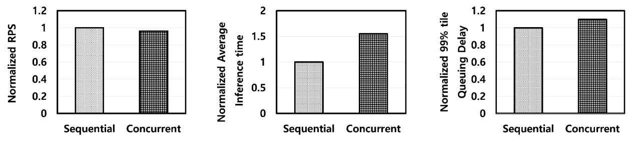
TABLE I
EXPERIMENT CONFIGURATIONS

| Platform | NVIDIA Jetson Orin AGX | | |
|---|---|---|---|
| GPU architecture | Ampere architecture | | |
| GPU Cores | 1792 CUDA cores, 56 Tensor cores | | |
| CPU | 8-core Arm Cortex-A78AE v8.2 | | |
| Memory | 32GB LPDDR5, Bandwidth: 204.8GB/s | | |
| Storage | 64GB eMMC 5.1 | | |
| Total number of request | 1,000 | | |
| Request generation scenario | Poisson distribution | | |
| DNN models | AlexNet | ResNet18 | VGG16 |
| Number of Layers | 8 | 18 | 16 |
| Number of Parameters | 61M | 11M | 138M |

given a stringent QoS requirement. Recent embedded systems equip GPUs and offer a higher peak throughput than embedded CPUs [4]. In the NIVIDA Jetson Orin AGX platform, a GPU exhibits 6.5× higher peak throughput (INT8) than a CPU in our initial analysis. While discrete GPUs support parallel kernel execution, such as multi-instance GPU and multi-process service [2], [5], embedded GPUs cannot run multiple kernels efficiently. Prior work has proposed a multi-stream kernel execution technique for embedded systems [6]. The prior work could reduce the response time by scheduling kernels towards better hardware utilization. While such a scheme can reduce queuing delay, it does not improve throughput compared to the sequential kernel execution. Figure 1 shows how an embedded GPU runs multiple kernels. Embedded GPUs run kernels concurrently, so active kernels share a GPU in a time-sharing manner. Also, the concurrent kernel execution incurs frequent context switching. GPUs contain all the register values of thousands of threads in the hardware register file [7]. To perform context switching, a GPU should copy all the register values of active/inactive kernels from/to the register file. As a result, the context switching of GPUs requires a long latency, so it incurs an increase in queuing delay and inference time.

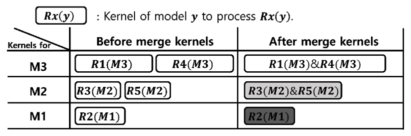
We run 1,000 inference requests and measure throughput (requests/second), average inference time, and tail latency (99%tile queuing delay + inference time, we call the sum of queuing delay and inference time 'service latency'). For the experiments, we employ three models: AlexNet [8], ResNet-18 [9], and VGG16 [10]. Table I describes the specification of the NVIDIA Jetson Orin system that we use. We assume that each request runs inference with either of the pre-deployed deep learning models. We implement a request generation scenario that receives an inference request based on the Poisson distribution [11].

Figure 2 shows the experimental results. The concurrent kernel execution exhibits the same throughput as the sequential execution as mentioned above. While the concurrent kernel execution is not effective in throughput, its average inference time is 50% longer than the sequential execution. Also, the concurrent kernel execution also exhibits 10% longer tail latency than the sequential execution.

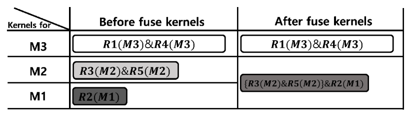


(a) Throughput. (b) Inference time. (c) Queuing delay.

Fig. 2. Comparison with sequential execution and concurrent execution. All the results are normalized to sequential execution.



(a) Merge kernels of requests for the same model.



(b) After merge kernels, fuse kernels running small model.

Fig. 3. Merge and fusion design policy.

Prior work has proposed a software kernel scheduling technique that runs kernels from different tasks in a breadth-first order [3]. Once the software technique schedules a kernel from a task, it prioritizes another task and runs a kernel of the prioritized task. While the prior work could reduce queuing delay, it eventually runs kernels in sequential order. So the prior work could not improve throughput compared to the sequential kernel execution.

## *B. Merge and Fusion*

We propose a new software technique called automatic kernel merge and fusion. Unlike the prior work on kernel scheduling and kernel fusion, we propose a new idea of kernel merging and adaptively apply it along with kernel fusion. With the proposed idea, embedded systems can run multiple tasks with a minimized number of kernels. So a system can serve requests with shorter queuing latency and kernel execution time than existing mechanisms in an embedded GPU. TensorRT includes a technique that vertically/horizontally fuses the layers within a single model [12]. In contrast to TensorRT, the proposed technique merges and fuses layers across models. Therefore, TensorRT and our proposal are orthogonal. We believe that combining the proposed technique with TensorRT's fusion technique can further improve performance.

Figure 3 shows the scheme of kernel merging and fusion. Kernel merge (Figure 3a) is a technique that runs a single kernel to process multiple requests that run the same model. Concurrent kernel execution and the prior work (the breadth-first operator scheduling explained in the previous section [3]) eventually run the kernels as many as the number of

**Algorithm 1** Merged Kernel Generator Algorithm

```
    Input :$K_{ij}$, $window\_size$.
        $K$={$K_{ij}$|for each model i, for each kernel j}.
    Output :A Merged kernel $M$.
 1: function M_GEN($K_{ij}$, $window\_size$).
 2: Initialize $M$ with local variable declarations from $K_{ij}$.
 3: Append input arguments of $M$ as many as $window\_size$.
 4: Copy the code to the last loop header of the $K_{ij}$'s compute part to $M$.
 5: Append "for(int i = 0; i < $window\_size$; i++) {" to M.
 6: Append "output[i] += input[i] * weight" to M.
 7: Append "}" to $M$.
 8: Copy the remaining codes of $K_{ij}$ to $M$.
 9: return $M$.
10: end function
```

(a) Merged kernel generator algorithm.

**Algorithm 2** Fused Kernel Generator Algorithm

```
    Input :$K_1$ and $K_2$ are two input kernels. $d_1$ and $d_2$ are the block dimensions of $K_1$ and $K_2$.
    Output :A fused kernel $F$.
 1: function F_GEN($K_1, K_2, d_1, d_2$).
 2: Initialize $F$ with local variable declarations from $K_1$ and $K_2$.
 3: Append "if ($tid \geq d_1$) goto $l_1$;" to $F$.
 4: Mark the end of $K_1$ with label $l_1$.
 5: Append $K_1$ to $F$.
 6: Append "if ($tid \geq d_2$) goto $l_2$;" to $F$.
 7: Mark the end of $K_2$ with label $l_2$.
 8: Append $K_2$ to $F$.
 9: Set the dimension of the fused kernel to $\max(d_1, d_2)$.
10: return $F$.
11: end function
```

(b) Fused kernel generator algorithm.

Fig. 4. Merge and fusion algorithm.

the requests. Unlike them, the kernel merge iterates the same code as many as the number of requests running the same model for each kernel of a model. While iterating the code, the proposed idea uses input from each request. The input may be either input data (image or voice) or intermediate results from a previous kernel. Such an idea improves throughput by eliminating context switching overhead and memory access to fetch the same filters for multiple requests.

We also propose to employ a kernel fusion technique inspired by prior work [13]. The proposed technique performs kernel fusion with the kernels created by the kernel merging. Unlike the prior work, the kernel fusion used in our idea reduces the number of concurrent kernels to one or two.

We implement the automatic kernel merge and fusion as shown in Figure 3b. The proposed idea consists of three key parts in software: Request analyzer, kernel merger, and kernel fuser. Like servers, we implement a software queue that keeps inference requests. After creating the merged and fused kernels, the request analyzer monitors a fixed number of requests from the queue head (also called in the request window in this paper). The request analyzer first finds the requests that can be merged for each deep learning model. In our implementation, the request analyzer monitors five requests from the queue head as a request window. The request analyzer finds that two requests run model M3, another two requests run model M2, and one remaining request runs model M1. We assume that model C contains more than 2× parameters than models A and B. After analyzing the requests, the kernel merger creates new kernels with the kernels of all the layers in the three models.

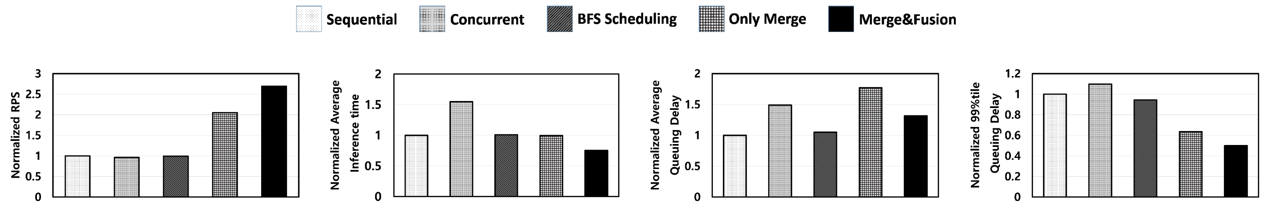
Both the kernel merger and kernel fuser create new kernels with existing kernels before the system actually serves incoming inference requests. Figure 4 depicts the detailed algorithms of the kernel merger and the kernel fuser. The kernel merger creates a kernel that can run multiple requests with the same models by inserting the codes that run a loop to each kernel of a model. The kernel merger adds the codes that perform loops to kernels of the layers in deep learning models. Our technique analyzes the existing kernel codes at the intermediate representation level (e.g., PTX or SASS) and adds the codes for looping as described in Figure 4a.

After the kernel merger runs for all the models, the kernel fuser classifies the deep learning models and creates merged kernels with all possible combinations as we explain above. The criteria of the task execution time in our design is the total number of parameters of a deep learning model as the inference time depends on it. The kernel fuser analyzes all the parameter counts of the models in a system and calculates their arithmetic mean. After that, the kernel fuser performs the fusion with the models whose parameter counts are below the average and those whose counts are above the average. In our example in Figure 3b, the kernel fuser creates new kernels with the merged kernels from models M1 and M2. It does not perform kernel fusion with the kernels from model M3 as it has an exceeding number of parameters than the average. The kernel fuser creates new kernels with every n$_{th}$ merged kernel from models M1 and M2. If either model M1 or M2 has more kernels than another model, the kernel fuser creates fused kernels as many as the least number of kernels between the two models and leaves the remaining kernels as they are. The kernel fuser creates a new kernel with the merged kernels by modifying them at the PTX level.

## III. Evaluation

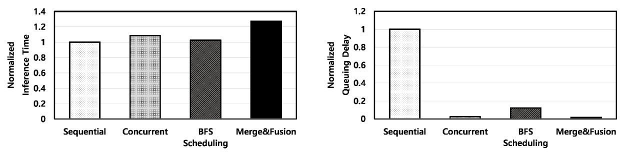
We evaluate throughput (requests per second), average inference time, average queuing delay, and 99%tile queuing delay under the following environments. As explained in Section II, we model a request generation with a Poisson distribution and generate 1,000 requests (333 requests for AlexNet, 333 requests for ResNet-18, and 334 requests for VGG16) in a random order. Second, we generate only a single request for AlexNet, ResNet-18, and VGG16, respectively. We compare the performance of the proposed technique to that of the sequential execution, concurrent execution, and the prior work that schedules operators of multiple DNN models [3]. We implement the proposed software framework on top of the NVIDIA Jetson Orin platform [14].

Figure 5 shows the experimental results with 1,000 random requests. The automatic kernel merge and fusion achieves 2.7× better throughput than the sequential kernel execution and prior work. The proposed technique runs multiple tasks only with two kernels within a short time window. Also, for each group of tasks merged and fused by the proposed technique, it reduces the number of global load instructions by 27% to 40% compared to the sequential execution. The more kernels are merged by the proposed technique, the more load instruction count is reduced at runtime. Due to such an advantage, the proposed technique achieve a better throughput than the prior work. We also observe that applying only the kernel merge achieves 2.1× better throughput than the baseline, but it is 23% lower than the merge and fuse. The merge only often executes multiple kernels if the grouped five requests run multiple models, thus exhibiting lower hardware utilization. The concurrent execution and the prior work



(a) Throughput. (b) Inference time. (c) Average queuing delay. (d) 99%tile queuing delay.

Fig. 5. Experimental results with 1,000 requests. All the results are normalized to the results of sequential execution. BFS Scheduling is the technique that schedules operators in a breadth-first manner [3]. For throughput, higher is better. For other metrics, lower is better.



(a) Inference time. (b) Average queuing delay.

Fig. 6. Experimental results with 3 requests. All the results are normalized to the results of sequential execution.

exhibit the same throughput as the baseline because both run a single kernel concurrently.

The proposed technique achieves the shortest average inference time (time taken from an inference start to end) among all the techniques. The proposed technique mitigates context switching overhead with kernel merge and fusion, thus resulting in a 47% shorter inference time than the prior work and the kernel merge only. The prior work and the kernel merge only show only a 3% shorter inference time than the baseline. We observe that concurrent execution, even with advanced operator scheduling, cannot contribute to improving throughput and reducing inference time.

The proposed technique achieves the shortest average and 99%tile queuing delay, which are 50% (average) and 63% (99%tile) shorter than the prior work. By significantly reducing the average inference latency and running multiple tasks in parallel, the proposed technique achieves such results. The prior work can reduce the queuing delay if the request comes sparsely. However, embedded systems often receive numerous requests given a short time window. As such, rather than scheduling operators, running them in parallel by reducing redundant memory operations is more effective.

Figure 6 shows the experimental results of the sequential execution, the concurrent execution, prior work, and the proposed technique only with three requests. As those experiments run only three requests, we do not measure 99%tile queuing delay in these experiments. The proposed technique and all other techniques achieve almost the same throughput, so we do not show the results. The proposed technique achieves only a 5% better throughput than the baseline. The concurrent execution and the prior work eventually run a single kernel at once, thus resulting in the same throughput.

The proposed kernel merge and fusion shows 25% longer inference time than the prior work. However, our technique still achieves a 70% shorter queuing delay than the prior work. The prior work shows better average inference time and average queuing delay than the baseline in these experiments. However, the effect of context switching is not critical as only three requests run. We believe that such results are not critical as the experiment environment is not a realistic situation.

## IV. Conclusion

For inference, embedded GPUs are not running multiple kernels in an efficient way, thus suffering from throughput degradation and long inference latencies. Addressing those challenges, we propose a new software framework that adaptively minimizes the number of concurrent kernels while running a greater number of tasks simultaneously. The key idea is monitoring a fixed number of incoming requests, classifying the requests running the same model, and running the models whose inference latency is expected to be similar. The proposed technique achieves 2.7× better throughput, 47% shorter inference time, 50% shorter average queuing delay, and 63% shorter tail latency than prior work.

## References

[1] Z. Kim, “Robust lane detection and tracking in challenging scenarios,” *IEEE Transactions on intelligent transportation systems*, vol. 9, no. 1, 2008.
[2] N. Corporation. (2023) Nvidia multi-instance gpu user guide.
[3] F. Yu, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu, and X. Chen, “Automated runtime-aware scheduling for multi-tenant dnn inference on gpu,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021.
[4] M. Kim, “Guaranteeing that multilevel prioritized dnn models on an embedded gpu have inference performance proportional to respective priorities,” *IEEE Embedded Systems Letters*, vol. 14, no. 2, 2022.
[5] N. Corporation. (2023) Nvidia multi-process service.
[6] 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,” *Journal of Systems Architecture*, vol. 139, 2023.
[7] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, “Unifying primary cache, scratch, and register file memories in a throughput processor,” in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.
[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” *Advances in neural information processing systems*, vol. 25, 2012.
[9] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.
[10] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.
[11] S.-D. Poisson, *Recherches sur la probabilité des jugements en matière criminelle et en matière civile: précédées des règles générales du calcul des probabilités*. Bachelier, 1837.
[12] N. Corporation. (2023) Nvidia tensorrt.
[13] G. Wang, Y. Lin, and W. Yi, “Kernel fusion: An effective method for better power efficiency on multithreaded gpu,” in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications*, 2010.
[14] N. Corporation. (2022) Nvidia jetson agx orin series: A giant leap forward for robotics and edge ai applications.