ParallelFusion: Towards Maximum Utilization of Mobile GPU for DNN Inference

Jingyu Lee*
jingyu.lee@hcs.snu.ac.kr
Seoul National University
Seoul, Korea

Yunxin Liu*
liuyunxin@air.tsinghua.edu.cn
Institute for AI Industry Research
(AIR), Tsinghua University
Beijing, China

Youngki Lee
youngkilee@snu.ac.kr
Seoul National University
Seoul, Korea

ABSTRACT

Mobile GPUs are extremely under-utilized for DNN computations across different mobile deep learning frameworks and multiple DNNs with various complexities. We explore the feasibility of batching and it improves the throughput by up to 35%. However, real-time applications in mobile have a limited amount of requests to get a benefit from batching. To tackle the challenge, we present *ParallelFusion* technique that enables concurrent execution of heterogeneous operators to further utilize the mobile GPU. We implemented *ParallelFusion* over the MNN framework and evaluated on 6 state-of-the-art DNNs. Our evaluation shows that *Parallel Fusion* achieves up to 195% to 218% throughput with fused execution of 2 and 3 operators compared to single DNN inference.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing**.

KEYWORDS

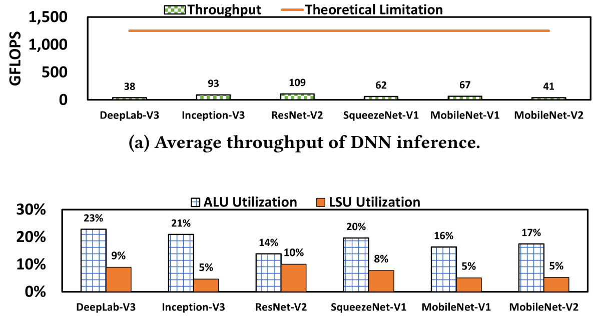
mobile GPU, mobile deep learning, under utilization, parallel fusion

ACM Reference Format:
Jingyu Lee, Yunxin Liu, and Youngki Lee. 2021. ParallelFusion: Towards Maximum Utilization of Mobile GPU for DNN Inference. In *5th International Workshop on Embedded and Mobile Deep Learning (EMDL '21), June 25, 2021, Virtual, WI, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3469116.3470014

1 INTRODUCTION

Emerging mobile applications require real-time, continuous execution of deep neural networks (DNNs) on resource-constrained mobile processors. For example, future augmented reality and augmented virtuality applications execute multiple DNNs over the streaming first-person-view video for accurate physical scene analysis (e.g., surface/object/face detection), photo-realistic virtual object generation, and user interaction (e.g., hand tracking) [19, 20]. Besides, they need to run at the 15-30 ms latency for an immersive user experience [4].

Mobile GPU is the most commonly available accelerator for commodity devices. It has been widely used for on-device DNN inferences. However, its utilization for various DNN tasks has not been thoroughly explored. A rich body of prior work attempts to improve the efficiency of DNN execution using GPUs (e.g., optimizing convolution operations and matrix multiplications and combining adjacent operators [3, 6]), but they have been mainly studied in the context of desktop-scale GPUs. A recent study reports the under-utilization of mobile GPU on DNN tasks [11], but focus on single DNN execution.



(a) Average throughput of DNN inference.

(b) Average utilization of arithmetic logic unit and load-store unit.

Figure 1: Utilization of Mobile GPU in DNN Inference (MNN over Qualcomm Adreno 650 - Samsung Galaxy S20).

In this paper, we first report our interesting observation that mobile GPUs are extremely under-utilized for DNN inference tasks. In particular, average ALU (Arithmetic Logic Unit) and LSU (Load Store Unit) utilization were under 18% and 7%, respectively, for six different state-of-the-art DNN models (See Figure 1 and Table 1). This phenomenon has been observed across different mobile deep learning frameworks [6, 11, 12] and DNN models with varying complexity, requiring in-depth study in the future.

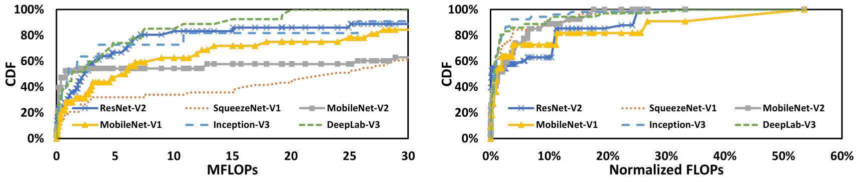
We then explore the feasibility of batching (i.e., grouping multiple inference requests and executing them at one go) as a plausible way of addressing the mobile GPU under-utilization problem. We find that the batched execution of an operator (e.g., over multiple consecutive video frames) can improve the GPU throughput by up to 35% with batch sizes of 5. However, this temporal batching method is not adequate for real-time mobile applications where the input needs to be processed right away without waiting for additional inputs to arrive for batched processing. Also, this can be only applicable to the same operators with the same weights, reducing the opportunities of batching significantly. This is different from server-side batching [7, 14] where many concurrent requests arrive within a short time window.

*This work was done in part when the authors were with Microsoft Research as an intern or a principal researcher.

EMDL '21, June 25, 2021, Virtual, WI, USA
2021. ACM ISBN 978-1-4503-8597-8/21/06…$15.00
https://doi.org/10.1145/3469116.3470014

Model	DeepLab-V3 [2]	Inception-V3 [16]	ResNet-V2 [8]	SqueezeNet-V1 [10]	MobileNet-V1 [9]	MobileNet-V2 [15]
FLOPs	734M	5.73G	3.87G	833M	575M	300M
# Params	39M	23M	25.56M	1.25M	4.2M	3.4M
Input size	257x257x3	299x299x3	224x224x3	224x224x3	224x224x3	224x224x3
Inference time	19.1 ms	61.5 ms	35.5 ms	13.4 ms	8.5 ms	7.4 ms

Table 1: List of evaluated DNN models.



(a) Distribution of FLOPs in operators.

(b) Distribution of Normalized FLOPs in operators.

Figure 2: Distribution and computational complexities of operators in DNNs.

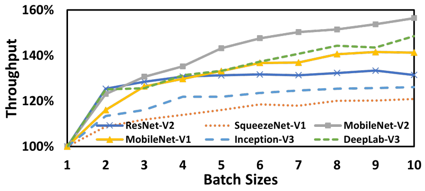


Figure 3: Throughput of batching relative to executing them individually.

In this paper, we propose the *Parallel Fusion* technique that enables parallel execution of independent operators to further utilize the mobile GPU, which exploits operator parallelism among independent operators from the multi-DNN workload of emerging AR DNNs with multiple branches. The key ideas of *Parallel Fusion* are 1) Construction of unified kernel code and 2) Branch divergence-aware thread workload assignment in run-time, to accumulate workloads from multiple kernels for better latency hiding.

The summary of our contributions is as follows:

- We show that the mobile GPU is seriously under-utilized (e.g., 18 % ALU utilization and 7 % LSU utilization) for 6 types of DNN models.
- We explore the feasibility of batching to address the GPU under-utilization problem. Also, we suggest a new *Parallel Fusion* technique to enhance the GPU utilization for real-time, multi-DNN application scenarios.
- We implement the *Parallel Fusion* method based on the OpenCL backend of the MNN [12] mobile deep learning framework. Our evaluation shows that our parallel fusion technique between 2 operators and 3 operators achieves 36% ∼ 195% and 31% ∼ 218% throughput, respectively.

2 MOBILE GPU UNDER-UTILIZATION

Experiment setting. To understand the GPU utilization for various models with different computational complexity, we consider 6 state-of-the-art DNNs listed in Table 1. We execute these models on the MNN framework [12] with the OpenCL-based GPU backend. We conduct experiments with Samsung Galaxy S20 (USA version) equipped with an Adreno 650 GPU and octa-core Kryo 585 ARM CPU. We use Qualcomm Snapdragon Profiler to measure GPU utilization (e.g., ALU and LSU utilization).

Results. Figure 1 illustrates the average utilization of ALU and LSU is around 18% and 7% across the 6 DNN models, respectively. The overall tendency of hardware utilization is weakly associated with the overall complexity of the model. However, exceptional cases such as Inception-V3 worse utilizes LSU compared to the MobileNet-V2, which has 19× smaller FLOPs. We observe that the component analysis of the model better explains the hardware utilization of the GPU. For example, Figure 2 shows the distribution and computational complexities of operators in DNN. Inception-V3 is composed of operators that have smaller FLOPs than the MobileNet-V2 regardless of its overall heavy complexity. GPU-based framework interprets the DNN inference task as a sequence of kernel execution representing the individual operator's computation. Consequently, utilization of the hardware is closely related to the computational characteristics of individual operators in DNN.

3 IMPROVING MOBILE GPU UTILIZATION

3.1 Approaches

Hardware-based solution. Advances of modern GPU architectures such as NVIDIA Hyper-Q of Kepler and AMD GCN enables concurrent execution of multiple kernels on different GPU streams to mitigate the resource under-utilization. However, concurrent kernel execution is not available in mobile yet since recent market-dominant mobile GPU architectures such as Qualcomm Adreno, ARM Mali, and Imagination Technologies PowerVR still rely on a single stream execution.

Software-based solution. DNN is a computationally intensive and easily parallelizable workload. Despite its overall heavy complexity, most of the building blocks of the deep learning model are light-weight operators requiring small and short GPU computation. As shown in Figure 2, almost 60% to 80% of operators are under 5 MFLOPS. Furthermore, more than 50% of operators have 1% or less proportion of model's operation across the 6 models. Therefore, most DNN frameworks [7, 14] utilize *batching* to process multiple identical computations at a time to increase the size of workloads for better hardware utilization. In addition, both of the two distinct computation phases of DNN (training and inference) can benefit from batching.

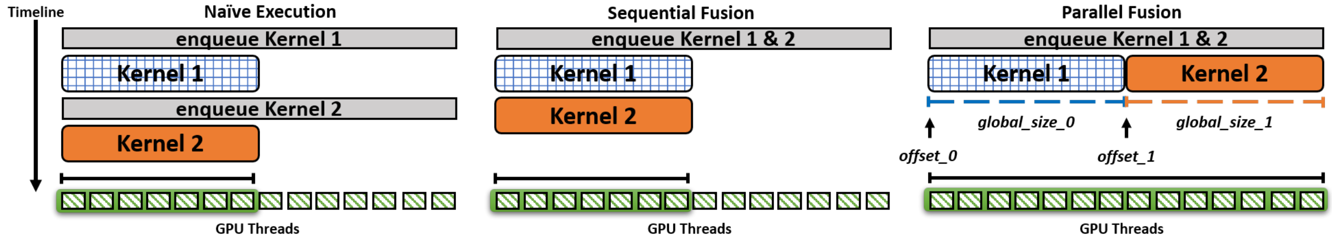


Figure 4: An illustration of kernel execution on GPU. Sequential fusion assumes element-wise case that requires no block-wise synchronization. Green area illustrates threads that activated for the computation.

3.2 Opportunities of Batching for Mobile GPU

Recent studies on desktop GPU show that batching is an effective solution to improve throughput. We conduct preliminary evaluations to explore the effectiveness of batching in mobile GPU.

We first evaluate the model-wise effectiveness on the 6 DNNs listed in Table 1 across different batch sizes. Figure 3 shows that the throughput of mobile GPU can be further increased by processing multiple inputs at a time. In batch sizes of 5, MobileNet families [2, 9, 15] [1] can achieve more than 30% of improvement and it saturates as batch size increases. Similar to the GPU component utilization, the effectiveness of the batching is also related to the computational characteristics of individual operators. As shown in Figure 2, $\approx$ 80% of operators in the models that achieve high throughput gains are below 5 MFLOPs. Furthermore, SqueezeNet-V1 shows the worst throughput gain from batching because of its heavy operators.

One of the key benefits of batching is latency hiding. GPU is known to have heavy memory latency bound due to the memory access cost. For example, global memory access of the GPU takes 14-64× of the simple arithmetic operation (add/sub/min/max) latency from NVIDIA Pascal architecture [1]. GPU simultaneously executes multiple threads, and this enables context switching between active threads to hide latency [17]. Batching increases the number of workloads per GPU invocation, which leads to better occupancy and latency hiding. This enables desktop-scale GPU[2] to improve the system throughput up to 13.3× for batch sizes of 32 compared to the individual execution of DNN. However, mobile GPU has smaller capacity to hold active threads at once compared to desktop-scale GPU. Nvidia RTX 2070 Super has a 20× number of execution units(streaming multiprocessors) compare to the Qualcomm Adreno 650 GPU and each execution unit can hold 2× more active set of threads(wave) simultaneously. This explains the earlier saturation with smaller batch sizes and why batching cannot achieve dramatic improvement in mobile GPU. Nevertheless, this simple technique still could achieve up to 50% throughput improvement and requires in-depth study for further utilization of the mobile GPU.

3.3 Challenges of Batching for Mobile GPU

The key challenge of the batching lies in the preparation of the inputs. In the training phase of the DNN, a framework samples a set of data(minibatch) from the existing database per each iteration. In addition, minibatch-based stochastic gradient descent algorithmically utilizes batching for faster convergence, which naturally leads to higher throughput.

Unlike the training phase, inference of DNN processes input of the application request as they arrive. Which requires low latency along with high throughput as multiple clients expect a real-time response. To solve this problem, server-level deep-learning serving frameworks consider inputs that are identical in graph-level [7, 14] and aggregate requests over time to increase the system-level throughput. Despite recent progresses that utilize the finer-grained level of user-defined operators [21] or recurrent blocks [5] to increase the opportunity to batch, the unit of batching is limited to the same models or specific operators. However, cloud-based deep learning frameworks are designed to serve requests from multiple applications simultaneously. This alleviates the current limitation of batching in the server-level DNN framework. For instance, if multiple clients send processing requests of facial recognition DNN that deployed on a social network service app, these frameworks can guarantee a sufficient amount of requests for the same model for batching.

However, mobile deep-learning framework targets a single foreground application, and existing techniques in server-level frameworks cannot mitigate the limitation of the batching in mobile scenarios due to the limited number of requests. Furthermore, it is also not adequate to use a wider time window for further aggregation due to the strict real-time constraint of mobile applications.

4 PARALLEL FUSION

We propose *Parallel Fusion* technique to enable concurrent execution of different operators to better utilize mobile GPU in DNN workloads. Figure 4 shows the key differences between existing execution models and *Parallel Fusion*. First, *Parallel Fusion* accumulates workloads of sub-kernels to span more threads in one kernel execution for better latency hiding. Second, assign each thread to the corresponding sub-kernel in the execution phase of the kernel.

4.1 Use Cases

Given DNNs that represented as directed acyclic graph (DAG)[3], *Parallel Fusion* utilizes independent kernels from single or multiple DAGs for concurrent execution. There are two plausible ways to

[1] Note the backbone network of Deeplab-V3 is MobileNet-V2.
[2] Inception-V3 on Nvidia GTX 1080

[3] Vertices are kernels that represent a set of layers (single or sequentially fused set of layers, e.g., Convolution-Batch normalization-ReLU), and edges specify connections between layers input and output.

```
__kernel void depthwise_conv2d_conv_2d_1x1(
    __private const uint2 offset_0,
    GLOBAL_SIZE_2_DIMS(0)
    ... // Arguments of depthwise_conv2d
    __private const uint2 offset_1,
    GLOBAL_SIZE_2_DIMS(1)
    ... // Arguments of conv_2d_1x1
    )
{
    if GLOBAL_ID_CONDITION_2_DIMS(0) {

        const int output_channel_width_idx = get_global_id(0) - offset0.x;
        const int output_batch_height_idx     = get_global_id(1) - offset0.y;
        SKIP_ID_2_DIMS(0, output_channel_width_idx, output_batch_height_idx);

        ... // Remaining body of depthwise_conv2d
    }
    else if GLOBAL_ID_CONDITION_2_DIMS(1) {

        const int output_channel_width_idx = get_global_id(0) - offset1.x;
        const int output_batch_height_idx  = get_global_id(1) - offset1.y;
        SKIP_ID_2_DIMS(1, output_channel_width_idx, output_batch_height_idx);

        ... // Remaining body of conv_2d_1x1
    }
}
```

Figure 5: The OpenCL code for a fused kernel.

```
1 #define SKIP_ID_2_DIMS(i, input_0, input_1) \
2     if (input_0 >= global_size_dim0##i || input_1 >= global_size_dim1##i) { \
3         return; \
4     }
5
6 #define GLOBAL_ID_CONDITION_2_DIMS(i) \
7     (offset##i.x + global_size_dim0_##i > get_global_id(0) && offset##i.y + ↩
           global_size_dim1_##i > get_global_id(1))
8
9 #define GLOBAL_SIZE_2_DIMS(i) __private const int global_size_dim0_##i, ↩
       __private const int global_size_dim1_##i,
```

Figure 6: Pre-processor definitions for branch generation.

infer operator-level parallelism from DNN workloads in mobile devices.

- DNNs with branches. Unlike traditional DNNs, several recently proposed architectures[10, 16, 22] are based on having internal branches. In addition, recent neural architecture search approaches consider multi-branches in their search spaces [13]. Existing DNN framework sequentially executes the DNN operators in topological order. However, independent layers from internal branches enable different operators of the model to be executed in parallel.
- Operators from different models. Future augmented reality, and virtual reality applications involve real-time multiple DNN tasks such as scene understanding (e.g., object recognition, depth estimation), photo-realistic virtual object generation (e.g., style transfer), and user interaction (e.g., hand tracking) [19, 20].

4.2 Parallel Fusion Technique

Parallel Fusion fuses multiple kernels to create the unified kernel code for parallel execution. This process requires the following considerations in two steps. Note that we use the terminology of the Qualcomm Snapdragon Mobile Platform OpenCL to describe the algorithmic details, but most of the concepts are easily applicable to other GPU platforms such as CUDA.

Fused Operator Preparation. Given a set of DNNs from the above use cases, it is essential to prepare kernels in advance to enable parallel execution of every possible combination of individual kernels in run-time.

OpenCL	ParallelFusion
get_global_size	global_size_dim_i
get_global_id	get_global_id - offset_i
get_num_groups	global_size_dim_i / get_local_size
get_group_id	get_group_id - offset_i / get_local_size

Table 2: Conversion table for representative OpenCL work-item built-in functions for kernel k_i.

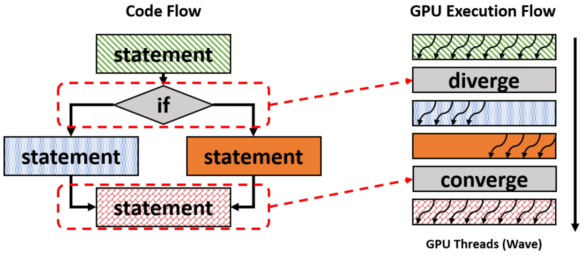


Figure 7: Branch divergence of wave. If threads from same wave diverges, GPU executes every execution paths with masked operation.

Figure 5 illustrates the fused OpenCL kernel code. Given any arbitrary set of independent kernels K, we first parse each kernel k_i (sub-kernel) to extract function arguments arg_i and bodies b_i. Also, add an additional suffix (*_i*) to duplicate identifiers to prevent any collision. Kernel code formulates the address of the memory based on its index (*global_id*) out of the entire workload (*global_work_size*). As we accumulate workloads from multiple kernels, we add *offset_i* and *global_size* definition (L9 from Figure 6) to arg_i, then update OpenCL work-item built-in functions in b_i based on Table 2 to preserve memory addresses in execution time.

Ultimately, re-organize the unified kernel code based on the following steps. First, list each argument in arg_i for a new kernel. Second, place each b_i along with statements (definitions in L1, L6 from Figure 6) to let the thread select or skip its own execution logic in run-time.

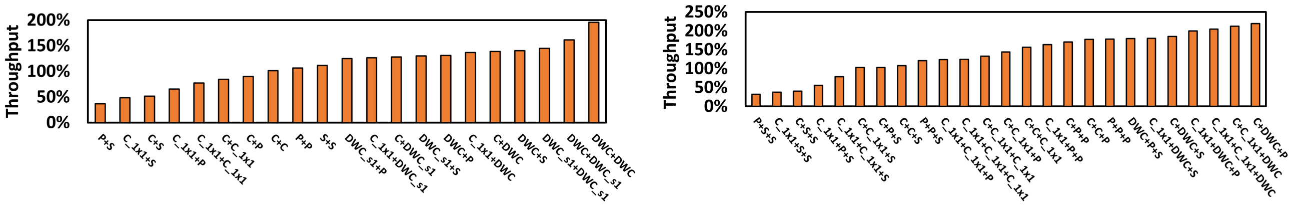
Fused Operator Execution. As shown in Figure7, executing branches in on wave degrades the performance. As we utilize branches to assign threads to its statements in run-time, it is crucial to avoid branch divergence of threads locked as a wave in workload boundaries between sub-kernels. We round each workload and offset of kernel up to the hardware-specific wave size to prevent performance degradation from branch divergence. Finally, Bind (*clSetKernelArg*) all function arguments including workload and offset in order before kernel execution.

$$global\_work\_size = \sum_{i=0}^{n-1} WS\lceil \frac{global\_work\_size_i}{WS} \rceil$$

$$offset\_i = \sum_{j=0}^{i-1} WS\lceil \frac{global\_work\_size_j}{WS} \rceil$$

5 EVALUATION

Experiment setting. We implement *Parallel Fusion* on top of the OpenCL backend of the MNN [12]. Except for the 3k framework-level code change, We only added a few code changes per kernel



(a) with 2 Operators

(b) with 3 Operators. Uniformly sampled 25 combinations from list sorted by throughput.

Figure 8: Performance of fusion between different operators compared to sequential execution of individual operators. The reported values are averaged value of all possible combination of operators from DNNs listed in Table 1.

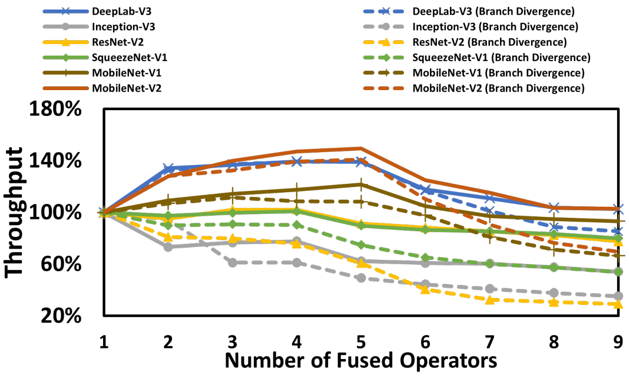


Figure 9: Performance of fusion between identical operators from same model.

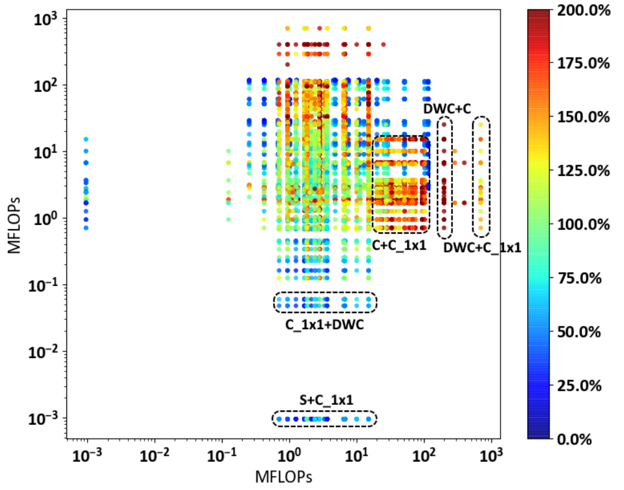


Figure 10: Performance of fusion between different operators. Color of the marker corresponds to the throughput compared to individual executions.

to fully reuse the existing OpenCL kernel of the MNN. We evaluate *Parallel Fusion* for the 6 DNNs in Table 1 based on evaluation settings in §3. Reported numbers are averaged results of 100 consecutive inferences with 5 warm-up iterations.

Fusion of identical operators. To analyze the performance of *Parallel Fusion* and branch divergence on identical operators without considering the combination or computational complexity of individual operators, we evaluate the fusion of various numbers of identical operators across the same model.

Figure 9 shows the throughput of fused execution of identical operators. *Parallel Fusion* increases the throughput up to 40% higher throughput in evaluations on MobileNet-V2 and DeepLab-V3. The effectiveness of the technique varies across different types of model and shows a similar tendency to the result of batching in Figure 3.

The dotted series show the throughput of *Parallel Fusion* without consideration of the branch divergence. Overall, branch divergence incurs performance degradation which reaches up to 50% in case of fusion with more than 5 numbers of kernels.

Fusion of heterogeneous operators. We study the effectiveness of *Parallel Fusion* on every combination of 2 or 3 operators[4] from 6 DNNs compared to the single kernel execution.

Figure 8a shows the performance on fused execution of 2 operators. *Parallel Fusion* achieves 36% ∼ 195% throughput depends on a combination. Our method shows better throughput than single operator execution from 66% out of all combinations. Similarly, Figure 8b illustrates the performance of three operators. *Parallel Fusion* performs 31% ∼ 218% throughput with 3 operators. Similar to the result with 2 operators, fused execution is beneficial in ≈ 80% out of all combinations. We notice that the fusion of more than 2 squeeze operators incurs huge performance degradation.

Overall, the effectiveness of the *Parallel Fusion* technique highly dependent on the combination of operators. Figure 10 illustrates the in-depth analysis of 2 operator fusion associated with the FLOPs of each operator. As shown as dotted groups, the throughput of the *Parallel Fusion* shows a similar tendency depends on a combination of components.The effectiveness depends on the computational characteristics (type of the operator) than the algorithmic complexity of the individual component since each group stretched over 10-100× scales of MFLOPs.

GPU Utilization analysis. Finally, we report GPU utilization of *Parallel Fusion* on parallel execution of 2 MobileNet-V2.

Figure 11 shows GPU utilization of all operators across the entire inference timeline. Parallel execution of two DNN increases by 30% compared to the separate executions (12.1 ms vs. 15.6 ms). The major factors of this throughput gain are operators in the dotted box. Specifically, parallel execution of the first 1x1 convolution of depth-wise separable convolution increased the ALU utilization by 45% (11% to 16%), and depth-wise convolution increased the LSU utilization by 61% (6.5% to 10.5%).

[4] Abbreviations for operator types (C: convolution, C_1x1: 1x1 convolution, S: squeeze, P: pooling, DWC: depth-wise convolution, DWC_s1: DWC with stride of 1).

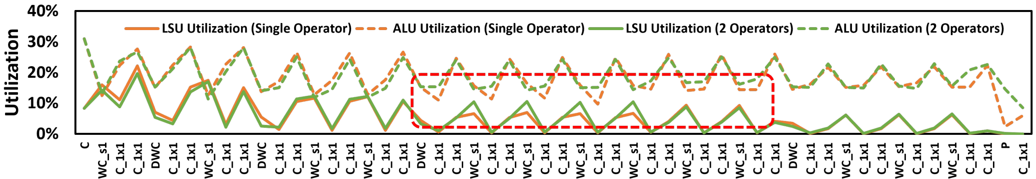


Figure 11: GPU Utilization of 2 identical operators of MobileNet-V2. Operators are listed in an execution order.

6 RELATED WORK

Fusion of different kernels is a well-known technique for DNN inference from existing frameworks[3, 6, 12, 18]. It can reduce the kernel launch overhead as well as avoid redundant memory access requests. For example, element-wise operations such as batch normalization or activation layers can be easily computed with simple arithmetic operations on the output of preceding convolution layers without extra GPU invocation and memory access. However, it is only applicable to simple arithmetic or element-wise operator since GPU does not support global synchronization.

7 CONCLUSION AND FUTURE WORK

We presented *Parallel Fusion*, a kernel fusion technique that enables parallel execution of heterogeneous operators of DNN. To further utilize the GPU, *Parallel Fusion* invokes multiple workloads from different kernels at one request to enable better latency hiding. Our experiment showed that *Parallel Fusion* achieves up to 195% to 218% throughput with parallel execution between 2 and 3 operators, respectively. We envision that our *Parallel Fusion* could utilize operator-level parallelism from recent DNNs with branches and future AR/VR scenarios requiring simultaneous execution of multiple DNNs to further utilize the mobile GPU.

Extension of the *Parallel Fusion* towards model-level execution scenarios stated in §4.1 requires modeling effectiveness of the technique. Specifically, we need a policy that maximizes the end-to-end throughput to group independent operators to avoid exhaustive search. The essential factors for performance modeling are as follows. First, in-depth study on the relationship between computational characteristics of kernel and effectiveness of the technique. Second, theoretical occupancy[5] of the mobile GPU to estimate the potential latency hiding from the accumulated workload.

8 ACKNOWLEDGEMENTS

This work was supported by Research Resettlement Fund for the new faculty of Seoul National University, and MSIT(Ministry of Science, ICT), Korea, under the High-Potential Individuals Global Training Program)(2020-0-01649) supervised by the IITP(Institute for Information & Communications Technology Planning & Evaluation).

REFERENCES

[1] Yehia Arafa, Abdel-Hameed A. Badawy, Gopinath Chennupati, Nandakishore Santhi, and Stephan Eidenbenz. 2019. Low Overhead Instruction Latency Characterization for NVIDIA GPGPUs. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC.2019.8916466

[2] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. 2017. Rethinking Atrous Convolution for Semantic Image Segmentation. *CoRR* abs/1706.05587 (2017). arXiv:1706.05587 http://arxiv.org/abs/1706.05587

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[4] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler. 2018. Toward Low-Latency and Ultra-Reliable Virtual Reality. *IEEE Network* 32, 2 (2018), 78–84. https://doi.org/10.1109/MNET.2018.1700268

[5] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[6] Google. Accessed on 20.12.2020. Tensorflow Lite. https://www.tensorflow.org/lite/

[7] Google. Accessed on 20.12.2020. Tensorflow Serving. https://github.com/tensorflow/serving

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[9] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[10] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016). arXiv:1602.07360 http://arxiv.org/abs/1602.07360

[11] Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu. 2020. Profiling and Optimizing Deep Learning Inference on Mobile GPUs. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems* (Tsukuba, Japan) *(APSys '20)*. Association for Computing Machinery, New York, NY, USA, 75–81. https://doi.org/10.1145/3409963.3410493

[12] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lyu, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/287.pdf

[13] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[14] NVIDIA. Accessed on 20.12.2020. Tensor RT. https://developer.nvidia.com/tensorrt

[15] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 4510–4520.

[16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.

[17] Vasily Volkov. 2016. *Understanding latency hiding on GPUs*. Ph.D. Dissertation. UC Berkeley.

[18] XiaoMi. Accessed on 20.12.2020. MACE. https://github.com/XiaoMi/mace

[19] Juheon Yi, Sunghyun Choi, and Youngki Lee. 2020. EagleEye: wearable camera-based person identification in crowded urban spaces. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*. 1–14.

[20] Juheon Yi and Youngki Lee. 2020. Heimdall: Mobile GPU Coordination Platform for Augmented Reality Applications. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking* (London, United Kingdom) *(MobiCom '20)*. Association for Computing Machinery, New York, NY, USA, Article 35, 14 pages. https://doi.org/10.1145/3372224.3419192

[21] Sheng Zha, Ziheng Jiang, Haibin Lin, and Zhi Zhang. 2019. Just-in-Time Dynamic-Batching. *CoRR* abs/1904.07421 (2019). arXiv:1904.07421 http://arxiv.org/abs/1904.07421

[22] Hang Zhang, Chongruo Wu, Zhongyue Zhang, Yi Zhu, Zhi Zhang, Haibin Lin, Yue Sun, Tong He, Jonas Mueller, R Manmatha, et al. 2020. Resnest: Split-attention networks. *arXiv preprint arXiv:2004.08955* (2020).

[5] Which can be inferred from hardware specifications such as the maximum number of active waves and the size of the wave.