

© 20XX IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Yu, Fuxun, Zirui Xu, Longfei Shangguan, Di Wang, Dimitrios Stamoulis, Rishi Madhok, Nikolaos Karianakis, et al. "Rethinking Latency-Aware DNN Design With GPU Tail Effect Analysis." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2024.

<https://doi.org/10.1109/TCAD.2024.3404413>.

<https://doi.org/10.1109/TCAD.2024.3404413>

Access to this work was provided by the University of Maryland, Baltimore County (UMBC) ScholarWorks@UMBC digital repository on the Maryland Shared Open Access (MD-SOAR) platform.

Please provide feedback

Please support the ScholarWorks@UMBC repository by emailing scholarworks-group@umbc.edu and telling us what having access to this work means to you and why it's important to you. Thank you.

Rethinking Latency-Aware DNN Design with GPU Tail Effect Analysis

Fuxun Yu, Zirui Xu, Longfei Shangguan, *Member, IEEE*, Di Wang, Dimitrios Stamoulis, Rishi Madhok, Nikolaos Karianakis, Ang Li, *Member, IEEE*, ChenChen Liu, *Member, IEEE*, Yiran Chen, *Fellow, IEEE*, and Xiang Chen, *Member, IEEE*,

Abstract—As the size of Deep Neural Networks (DNNs) continues to grow, their runtime latency also scales. While model pruning and Neural Architecture Search (NAS) can effectively reduce the computation workload, their effectiveness fails to consistently translate into runtime latency reduction. In this paper, we identify the root cause behind the mismatch between workload reduction and latency reduction is *GPU tail effect* — a classic system issue caused by resource under-utilization in the last processing wave of the GPU. We conduct detailed DNN workload characterization and demonstrate the prevalence of GPU tail effect across different DNN architectures, and meanwhile reveal that the unique deep structure and the light-weight layer workload of DNNs exacerbate the tail effect for DNN inference. We then propose a tail-awareness design space enhancement and DNN optimization algorithm to optimize existing NAS and pruning designs and achieve better runtime latency and model accuracy performance. Extensive experiments show 11%-27% latency reduction over SOTA DNN pruning and NAS methods.

I. INTRODUCTION

Deep Neural Networks (DNNs) have achieved remarkable progress on cognitive applications [3], [16], [6] and have drawn attentions from both industry and academia on deploying DNN inference service on the cloud, edge and mobile devices [19], [39]. However, the superior performance of DNNs is largely built upon growing model volume and structure complexity. Consequently, how to run these bulky DNNs on accelerators efficiently becomes the key to meet the ever-growing user experience, *e.g.*, DNN model inference *latency*.

To relieve the computation cost and improve runtime performance, many research efforts have been made in different design perspective and implementation levels. These works include model-level optimization [11], [9], [10], compiler-level computing optimization [25], [35], [1], [30], and architecture-level optimizations [28], [27], [26]. Among these works, model-level design solutions are usually the most flexible and straightforward way, since such methods can reduce model volume and computation workload like FLOPs, which can directly translate to latency reduction. However, recent hardware-aware works questioned the effectiveness of these

optimization works by revealing a widespread concept, *i.e.*, the lower amount of FLOPs does not equal to small latency during actual deployment on certain hardwares [38], [36], which underlines the importance of understanding of the hardware execution mechanisms for DNN design optimizations.

In this work, we re-examine the effectiveness of model design works for latency optimization on the general processing units (GPUs). By delving into the DNN to GPU deployment and execution mechanisms, we reveal a series of findings that current model design fails to recognize and thus lead to sub-optimal latency optimization performance. Figure 1 illustrates such an ineffective latency optimization case with a structural filter pruning example. Specifically, two layer configurations (config#1, #2) are compared, in which the config#2 conducts filter pruning and reduces partial workloads of this layer. Surprisingly, although config#2 reduces the filters, it ends up having the exact same runtime latency as config#1. In fact, as we keep pruning filters, Figure 1 (c) shows that the execution time exhibits an interesting non-linear *latency staircase* phenomenon. Why does filter pruning not always lead to latency reduction? In this paper, We identify the root cause of this phenomenon as the lack of awareness of the *GPU tail effect* — a classic parallel system issue that causes the resource under-utilization in the last processing cycle [20].

In particular, during execution on GPUs, a DNN layer's computation workload will be translated into massive parallel threads and allocated to GPU for processing on CUDA cores (Figure 1 (a)). However, due to the resource limit (*e.g.*, number of cores, maximal concurrency), the excessive parallel threads would be partitioned into multiple *waves* (Figure 1 (b)). Threads in each wave run concurrently while these multiple waves execute sequentially. Due to such an execution mechanism, although layer config#2 reduces the number of threads in the last wave, it takes the same runtime latency as config#1. Figure 1 (c) verifies the above execution analysis by showing a *latency staircase* pattern. Comparing layer config#2 to config#1, pruning certain amounts of filters brings no latency gain, but rather significantly lower down the GPU utilization and throughput due to the last partial wave.

Motivated by such an observation, **we conduct comprehensive benchmarking analysis and reveal the systematic mechanism** that causes such inefficiency in DNN inference: (i) Due to massive CUDA core parallelism, GPU presents a large *hardware-level compute granularity*. This is one major factor that causes the non-linear DNN workload to GPU runtime latency translation, and lead to the *latency staircase*

Fuxun Yu, Di Wang, Dimitrios Stamoulis, Rishi Madhok, Nikolaos Karianakis, are with Microsoft (contact email: fuxunyu@microsoft.com).

Zirui Xu is with CVS Helath.

Longfei Shangguan is with University of Pittsburgh.

Ang Li is with University of Maryland, College Park.

ChenChen Liu is with University of Maryland, Baltimore County.

Yiran Chen is with Duke University.

Xiang Chen is with George Mason University and Peking University.

Manuscript received 2023; revised May, 2024.

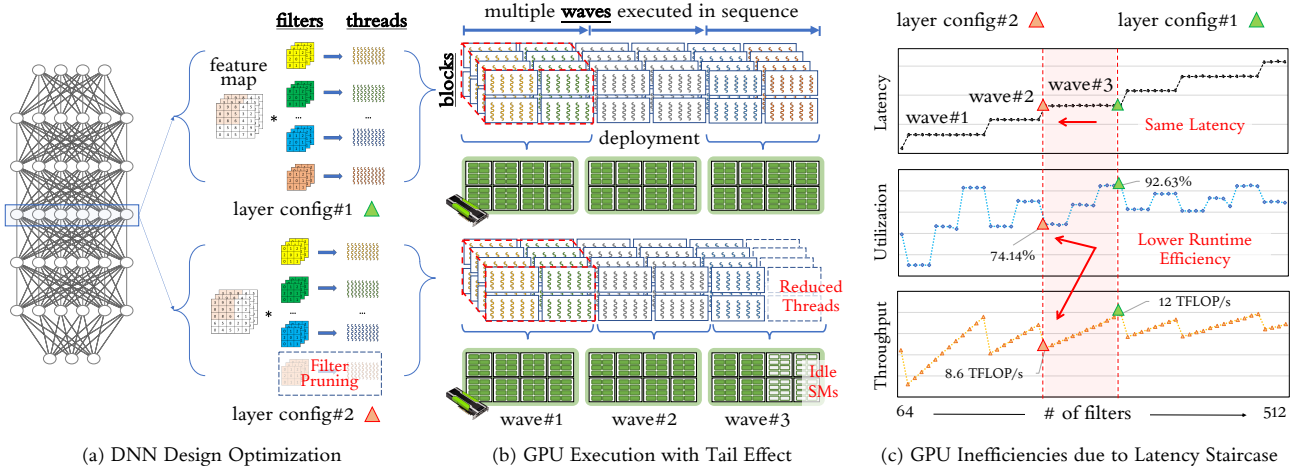


Fig. 1. An Example of Ineffective Latency Optimization by Filter Pruning. Specifically, by analyzing the GPU execution mechanism, we identify that software-level DNN optimization algorithms such as pruning and NAS algorithms fail to consider the native GPU runtime and thus can easily cause *GPU tail effect* that under-utilize the GPU during runtime, which makes these methods ineffective. Therefore, we propose a tail-aware software-hardware DNN co-optimization method to achieve ultimate DNN optimization effectiveness and efficiency.

phenomenon; (ii) Most current works are non-aware of such a GPU compute granularity and their DNN optimization can easily lead to the runtime *tail effect*; (iii) Furthermore, due to common DNN's deep layer structure, such *tail effect accumulates and amplifies with hundreds of layers*, resulting in significant overall resource under-utilization.

Based on such tail effect analysis, **we rethink the current DNN design optimization and propose following insights:** (i) For DNN structure design, the observation of latency staircase implies *DNN architecture design should take GPU compute granularity into consideration*, which would help enable more efficient hardware utilization for model optimization, *e.g.*, help defining the search space in NAS solutions; (ii) For DNN structure optimization, the tail effect implies that there are *free runtime resources for potential accuracy boost*, *e.g.*, by fulfilling each layer's tail wave of workload to utilize idle SMs but without increasing the latency; (iii) For complex DNNs with hundreds of layers, accumulated tail effects within these layers implies great chances for *non-trivial latency reduction* while maintaining accuracy.

Performing the design insights above could guide both DNN pre-design and DNN post-optimizations. We demonstrate our optimization approaches regarding two common design-level solutions, neural architecture search and structural pruning: (i) For NAS solutions, we enhance the current design space with expected GPU granularity awareness. Such a GPU granularity-aware configuration discretizes previous continuous layer width design space, thus greatly reducing the search candidates and the searching complexity. Meanwhile, as the design space already avoids the tail effect, the DNN structure could also achieve better GPU runtime efficiency and thus better latency performance. (ii) For structure post-optimization like pruning, we can also leverage the GPU-aware layer design space to fine-tune the sub-optimal pruned models. Specifically, we adopt layer growing and layer pruning two operations to utilize the free resources and to eliminate the GPU tail effect. Featured with such flexibility, our structural post-

optimization can achieve both accuracy lifting and latency reduction objectives, which is one major difference from previous works.

Through comprehensive evaluations, we demonstrated that our DNN design and optimization methods have exceptional latency performance and model generality with optimal accuracy. Across benchmarks, our design achieves 11%–27% latency reduction and 2.5%–4.0% accuracy improvement over state-of-the-art (SOTA) DNN structural pruning and NAS techniques.

The rest of this paper is organized as follows: Section §II provides the research preliminaries for this work. Section §III presents the observed DNN latency issue with the underlying GPU computing granularity; Section §IV then rethinks current DNN design pitfalls regarding latency-awareness; and Section §V proposes effective latency-aware DNN design insights and demonstrates different design and optimization approaches, following comprehensive evaluations in Section §VI; Section §VIII concludes this work.

II. BACKGROUND AND RELATED WORK

This section introduces the background for the primary research focus of this paper. A GPU workflow overview is first given and then the state-of-the-art DNN computing latency optimization works are reviewed. Specifically, their *pros and cons* are cross-compared to highlight this work's optimization perspective and motivation.

A. GPU Computing Preliminary

GPU Architecture. We take NVIDIA's GPU architecture as the primary investigation object in this work, which also applies to other GPU products. The GPU architecture is featured with an array of *Streaming Multiprocessors* (SMs), each of which is formed by a group of CUDA Cores. For example, NVIDIA Titan-V GPU [28] has 80 SMs and can process 5120 threads

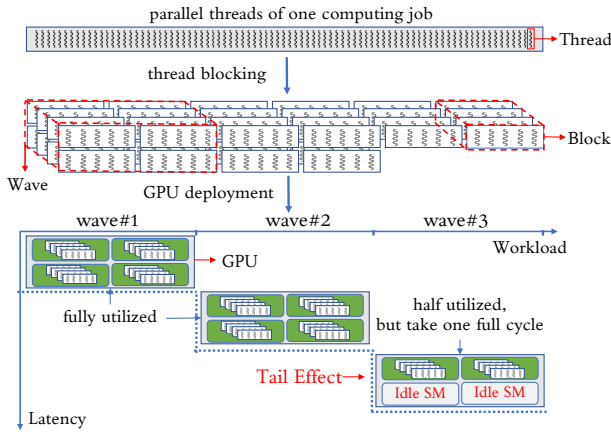


Fig. 2. GPU Execution Model. A computing workload will be split into massive threads and then split into Blocks and Waves for parallel processing. When the final wave comes unfulfilled, the GPU tail effect happens that under-utilizes the GPU resources.

simultaneously. Such a physical parallelism with massive computing units also defines the distinct GPU execution model.

GPU Execution Model. As shown in Figure 2, when a multi-thread job is deployed to a GPU, it will be first parsed into multiple logical workload partitions — **Blocks** with identical thread capacity. Depending on the physical capacity of the SM, the number of deployed thread blocks per SM varies.

For a large computing job, whose total workload exceeds the overall SM capacity in a GPU, the GPU needs to take multiple processing cycles. In such as case, the blocks are further clustered into logical **Waves** for sequential deployment. Specifically, wave describes the workload that can be concurrently processed per GPU cycle, which equals to the number of SMs \times the number of blocks per SM. And the **GPU Latency** can be roughly estimated by the overall number of waves \times the time per GPU computing cycle.

GPU Tail Effect. In practical deployment, the job workload usually will not be perfectly clustered into complete waves; in other words, the last wave could contain fewer blocks, as shown Figure 1. Nevertheless, such an incomplete tail wave will still consume a full GPU cycle, due to the GPU concurrent execution mechanisms as mentioned above, resulting in under-utilized SMs and sub-optimal runtime efficiency, namely, the **GPU Tail Effect**.

The tail effect generally exists in parallel systems [20] and is considered a trivial issue with conventional GPU workload (e.g., streaming graphic rendering), since the impact of the incomplete tail wave tail can be amortized with dozens of total wave amounts. *However, when it comes to DNN computing, particular DNN design characteristics (e.g., the deep layers) will significantly exacerbate the negative impact of the tail effect with considerable latency issues.*

B. Related Works on Latency Optimization

Software Perspective — DNN Design. In early methods, the volume of the DNN model or FLOPs used to be considered as a linear factor of the workload and therefore the computing latency. Thus, early latency optimization works from the DNN

design perspective are through either lightweight structure designs, such as neural architecture search (NAS) [12], [34]; or post-design model architecture tuning, such as structural DNN pruning [11], [9], [14].

However, recent works show the workload measured in FLOPs fail to consistently translate into latency due to different hardware execution mechanisms [38]. Therefore, **latency-aware** DNN design is expected to consider the actual hardware runtime [38]. Some recent works leverage off-line profiling of DNN models for latency estimation, such as latency-aware NAS works [10], [2].

Hardware Perspective — GPU Compiling. Currently there are also compiler optimization works that specifically target at optimizing the GPU kernel computing efficiency, such as TensorFlow-XLA and TensorRT [35]. These works tuned the low-level kernel computing primitives (e.g., tiling, unrolling) for each DNN model and each target computing hardware, and thus could achieve optimal hardware efficiency, such as maximal SM utilization, memory throughput. Recently, many ML-based auto-tuning frameworks are proposed, which leverage machine learning to search scheduling factors automatically, such as TVM, TASO [1], [13], etc..

It is worthy to note that these mentioned software and hardware optimizations are current mainstream, but not all. Beyond GPU compiler-level, there are also architecture-level works, such as GPU resource management and scheduling [27, 28], memory system optimizations [15], etc. However, due to their optimization cost and method generality issues, they will not be addressed in this work.

C. Pros and Cons of Different Perspectives

Latency optimizations from the software and hardware perspectives have their respective advantages and disadvantages.

From the DNN design perspective, as mentioned above, recent latency-aware DNN designs are more direct, which optimizes the profiled latency metric already, instead of indirect metrics like FLOPs [38]. However, most latency-aware optimizations treat the underlying runtime mechanism as a black box and only utilize rough look-up-table (LUT) based latency modeling [2], [36]. Although being able to approximate the end-to-end DNN inference latency, they reveal limited understanding of how varied structures perform differently on the hardware. Therefore, the searching performance can be easily limited with an improper design space configuration or insufficient sampling density¹.

Compared to that, hardware-perspective optimizations mainly focus on GPU computing efficiency and provide complementary computing speedup.

Take the TensorRT or TVM compiler for GPU as an example. In compiler-level, re-compiling the hardware-specific kernels for pruned layers can renew the shape-specific optimization (e.g., rescheduling the thread blocking/tiling settings), thus

¹For example, most NAS works [36], [34] heuristically set or hand-pick layer width search space such as {16, 32, 64, 128, 192}, which can miss optimal layer settings in terms of GPU runtime efficiency.

reducing/removing the tail effect and achieving the optimal efficiency. However, due to the practical cost issue, the current limitation still exists even in the most popular CuDNN/TensorRT lib since TensorRT only develops a static kernel base with around 100+ kernel optimization templates [22], [21] in order to support all conv shapes and many of them are also legacy ones for GEMM-based MatMul kernels. Although such kernel templates take heavy hardware engineering tuning efforts, they are still heavily tuned towards and thus restricted to the most common layer shapes (like ResNet50 layer shapes, EfficientNet layer shapes), which cannot fit various pruning operations with different shapes. To remediate this inefficiency, pytorch and TensorRT also have a “cudnn_benchmark” function that will exhaustively test all kernel shapes’ runtime in advance and rank the relative performance with these kernel templates. However, a lot of different pruning/NAS methods’ customized configuration cannot find optimal matched templates and thus tail effect problems still exist in such static kernel bases.

D. Our Design Perspective and Motivation

Targeting on optimization efficiency and generality, our work mainly rethink the software-level approaches, *i.e.*, targeting at the DNN model design. But instead of treating GPU as a black box, we incorporate its computing mechanism awareness into the DNN design by diving into the DNN model-to-GPU workload mapping during compiling for an explicit GPU computing granularity analysis, and therefore guide our latency-aware DNN model optimization. Our later proposed solution and evaluation also reveal an interesting finding: DNN designers could effectively alleviate the GPU computing inefficiency (*i.e.*, tail effect) directly from the early design stages, delivering optimal computing performance with low costs, and meanwhile achieving expected generality and feasibility.

III. REVEALING THE GPU COMPUTING GRANULARITY FOR RUNTIME LATENCY

In this work, we first dive into the DNN latency analysis to reveal the underlying GPU computing mechanisms. A particular DNN latency issue is identified through detailed observation as the *latency staircase* phenomenon, which triggers our rethinking of the latency-aware DNN model design.

A. DNN Latency Staircase Phenomenon

We profile the inference latency of two representative DNN model settings on an NVIDIA Titan-V GPU²: (a) VGG16 on CIFAR10 dataset; (b) ResNet50 on ImageNet dataset. For each DNN model, we investigate the latency impact from different model volumes by gradually pruning the layer width step by step and measure the runtime latency in each layer width setting. All experiments are conducted using CUDA 10.2 and CuDNN 7.6.5 backend.

Through the analysis, we identify a distinct DNN runtime issue, *i.e.*, the **latency staircase** phenomenon. Specifically, Figure 3 illustrates the latency of five randomly selected

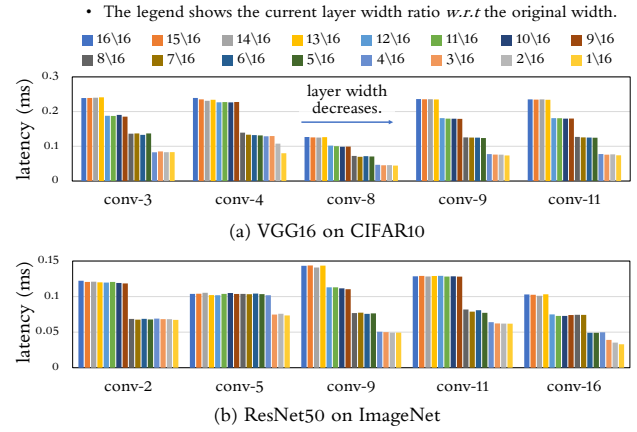


Fig. 3. DNN Runtime Latency Profiling.

DNN layers (denoted by conv- N) of VGG16 and ResNet50, respectively. For each layer’s dataset, although the layer volume is gradually shrunk by filter pruning, runtime latency doesn’t reduce linearly but demonstrates a “staircase” trend, are changes occurring only after a certain plateau period. Taking the left-upper corner dataset of as an example (conv-3 on VGG16), we observe that the runtime latency does not change as the layer width shrinks gradually to 13/16 (4th yellow bar from the left) *w.r.t* its original size (1st blue bar). However, the latency suddenly drops as the layer width shrinks from 13/16 further to 12/16, after which stays the latency still again until the layer width drops to 9/16 of its original size. And such a staircase dropping pattern repeats through the whole latency profiling.

Such results reveal several findings in DNN latency optimization: (i) *the layer workload reduction by conventional structural pruning cannot directly translate to GPUs’ runtime latency optimization*; (ii) *such an issue is prevalent across DNN models and layer configurations*; (iii) *DNN latency optimization requires a deeper investigation of the hardware computing mechanism than a pure software perspective*.

B. Reflection of GPU Computing Granularity

Through dedicated analysis, we find that the above latency staircase is a reflection of specific GPU computing granularity due to the huge-parallelism execution mechanism. Specifically, we dive into fundamental GPU computing primitives (*e.g.*, number of GPU thread blocks B and waves W) using NVIDIA Nsight Compute [24], and reveal the DNN deployment and GPU computing mechanism.

Latency Staircase Analysis. Figure 4 shows the GPU computing changes when we vary a layer’s width from 64 to 512 filters³. As we can see, the number of thread blocks B grows with increasing number of filters F (top figure), which is expected. Correspondingly, the number of waves also increase. However, due to the granularity mismatch (*i.e.*, one filter’s workload doesn’t equal to one block, and not equal to one wave.), the number of wave increase in a discrete way (step-by-step), forming the staircase growing trend and with a larger granularity (middle and bottom figure).

²We will show in later experiments that such phenomenon is general for both high-capacity and low-capacity GPUs like Jetson Nano.

³Specifically, the filter shape and input feature map sizes are set to 3×3 and $64 \times 64 \times 512$. And we keep the batch size to 1 throughout the experiment.

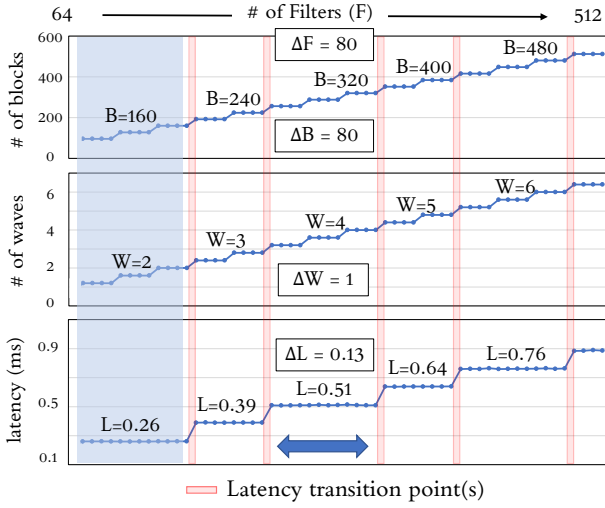


Fig. 4. GPU Computing Analysis w.r.t. Blocks, Waves, and Latency with Increasing DNN Layer Size.

As highlighted by the blue box in Figure 4, the number of waves W grows gradually from 1 to 2 with increasing workload. However, the latency stays still until the workload grows to above 2 waves, and then GPU consumes a new processing cycle to process the new wave of workload. We thus see a jump in the runtime latency as highlighted in the red box, i.e., the latency transition point.

C. Latency Formulation with GPU Granularity

Similar staircase patterns continuously emerge with growing workloads, which reflects a GPU-specific compute granularity. We can thus model the latency staircase based on the above analysis. For a given DNN layer, its GPU runtime latency L can be represented by:

$$L = \Delta L \times \lceil \#W \rceil, \quad \text{where } \#W = \frac{B}{S}, \quad (1)$$

where ΔL represents the latency for the GPU to finish one wave of workload, $\#W$ is the number of waves and B is the number of blocks that the overall workload being translated to. S is processing capacity of the GPU, e.g., the maximal number of blocks that can be concurrently executed. $\lceil \cdot \rceil$ is the rounding-up operation that returns the least integer greater than or equal to the input (e.g., $\lceil 2.1 \rceil = 3$), which reveals the impact of **tail effect** on the runtime latency, that is, one underutilized tail wave still leads to the latency of one full computing cycle for each DNN layer.

Eq. 1 demonstrates that the DNN execution latency on GPUs is highly dependent on a GPU-specific compute granularity, i.e., the wave W . Such a granularity is not only a hardware-level concept to describe the GPU capacity, but also an important factor for latency-aware DNN design. Without being aware of this, DNN structure design can easily generate granularity-unmatched models, thus resulting in sub-optimal GPU efficiency and DNN inference latency.

IV. RETHINKING THE DNN DESIGN PITFALLS WITH GPU TAIL EFFECTS

The previous observation corresponds to the **GPU tail effect** mentioned in §II-A. As the DNN structure is usually composed

of massive layers, the latency staircase and the tail effect can thus be prevalent across most DNN layers. Whereas, unlike ordinary GPU computing tasks where a single tail effect can be amortized, DNN inference with deep sequential layers incurs massive accumulated tail effects. And even worse, each tail wave may occupy a larger proportion of runtime since each layer has relatively lightweight workload.

In this section, we reveal the latency impact of the tail effect, and rethink the DNN design with GPU hardware-awareness.

A. Excessive Tail Effects within DNN Design

We characterize the workload of mainstreaming DNN models and reveal the excessive tail effects' existence in DNN structures due to two factors: (i) *Massively sequential layers*: while DNN models come in a wide variety of shapes and sizes, they all share the similar deep structure containing massive fragmented runtime units as layers. The tail-induced inefficiency in each layer thus accumulates with the feed-forward inference; (ii) *Limited per-layer workload*: While the entire DNN workload can be large, the workload of each layer is light-weight and thus requires much fewer waves to finish, amplifying the per-layer tail wave's negative impact.

Massively Sequential Layers. Figure 5 illustrates the workload of each layer (termed as Layer workload) in three different DNN models. To facilitate the presentation, we also plot the actual GPU workload in the figure, both of which are quantified by the number of waves. The gap between the layer workload and the actual GPU workload reflects the GPU under-utilization ratio. We observe these three models all contain more than 50 layers, and more importantly, the GPU under-utilization can easily occur in almost every layer of these models. During inference, such per-layer GPU under-utilization accumulates with the model execution, and thus severely undermines the GPU runtime efficiency. Beyond such lightweight models, higher under-utilization may appear in even deeper DNN models such as ResNet101 [7].

Limited per-Layer Workload. We further measure the number of waves incurred by each layer's workload in different batch size settings, and show their cumulative distribution function (CDF) in Figure 6. Here, we can observe that the majority of layers incur less than five waves for all three different models. For ResNet50, we can see over 80% of layers only introduces less than three waves from batch size=1~4. Similarly, over 80% of layers in both MobileNet and MNasNet requires less than three GPU waves when the batch size is relatively small (i.e., $B=1$). The maximum 80-percentile per-layer workload with batch size=4 only reaches 5-6 waves, which is far limited compared to the aforementioned same-sized GPGPU workload (30-80 waves). Due to the limited workload per layer, under-utilization in the tail wave causes more significant resource under-utilization (e.g., up to 33% in a layer with three waves).

It is worthy to note that, the GPU tail effect can potentially become increasingly prominent with more light-weight DNN design (such as in the edge domain), while GPUs are becoming increasingly powerful with higher capacity even for the edge GPUs like NVIDIA Jetson [23]. Therefore, without dedicated

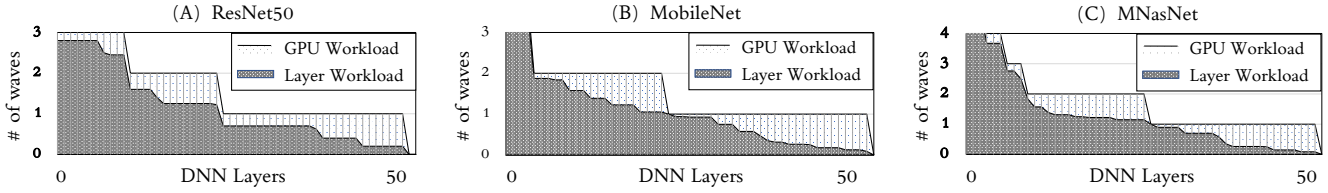


Fig. 5. The offset between the runtime GPU workload and the layer workload illustrates the GPU under-utilization caused by the tail effect (BS=1). Such under-utilized happens in each layer and thus accumulates to a non-negligible amount.

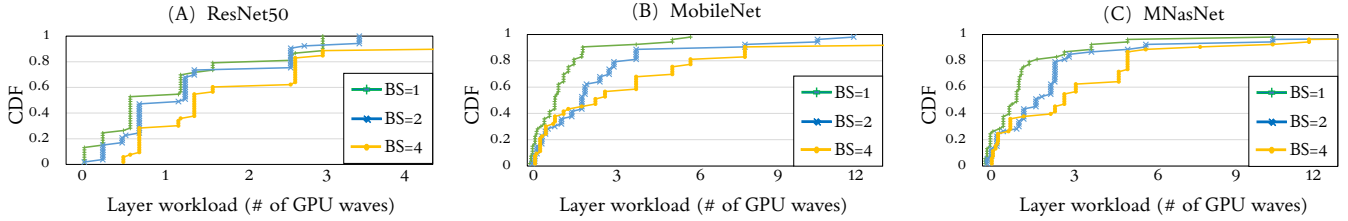


Fig. 6. The CDF of the layer workload shows that most (>90%) DNN layers consume less than 3~9 GPU waves with varied batch sizes (BS=1~4). In such cases, the relative influence of tail effect (affecting the last wave) is amplified.

optimization, such DNN latency sub-optimality can be exacerbated with the increased tail effects caused by mismatched GPU and DNN design granularity.

B. Pitfalls in DNN Optimization

We then demonstrate that conventional DNN optimizations such as filter pruning and architecture search (NAS) can even exacerbate the tail effect, leading to ineffective DNN optimization pitfalls.

In the beginning of Figure 1 (c), we have shown a real case of GPU runtime efficiency variation when layer workload is not matching the GPU granularity. Specifically, as the latency exhibiting a staircase pattern, the two GPU runtime metrics (GPU utilization and throughput) also show significant oscillation. The highlighted red box in Figure 1 (c) annotates a concrete example: the GPU utilization and throughput peak at the local maxima (92.63% and 12.0 TFLOPS/s respectively) when the workload fully occupies the current wave, while decreases to the local minima for the lower-bound workload (74.14% and 8.6 TFLOP/s). Caused by ineffective optimization, the GPU runtime efficiency gap can reach up to $1.4\times$.

Such "ineffective pruning" pitfalls, that is, blindly pruning a DNN layer can lead to small runtime latency reduction but rather cause sub-optimal GPU utilization and throughput, can also be common in general pruning algorithms. For example, by reproducing the model pruning result using one of the recent SOTA pruning work [14], Figure 7 shows the runtime latency

and throughput information of two resulted DNN layers. The performance result achieved by [14] is highlighted in orange. As we can observe that, without such tail awareness, two pruned layers did not achieve the optimal GPU throughput, but rather leading to the *ineffective pruning*. Especially in the first case that pruning 40% of filters doesn't lead to latency reduction, the gap between the throughput achieved by [14]'s configuration and the optimal one can reach up to $1.5\times$.

V. APPROACHING LATENCY-AWARE DNN DESIGN WITH GPU AWARENESS

In this section, we summarize the lessons learned from our analysis and characterization to guide DNN design on GPUs. We then demonstrate that by eliminating the GPU tail effect, we can enhance current DNN optimizations with better GPU awareness and achieve better performance.

A. DNN Design with Latency-Awareness

By understanding the GPU granularity-based latency modeling and the inefficiency caused by the tail effect, we can illustrate a series of DNN design insights:

- *Latency-aware DNN optimization on GPUs should consider a new basic unit, i.e., the GPU computing granularity (§III-B).* Previously, many DNN design and latency optimization works consider filters as the primary structural unit. However, our latency staircase analysis shows that, as one filter's workload is usually much smaller than the GPU computing granularity, workload measured by filters fails to translate to the latency consistently. By understanding this, latency-oriented DNN design could improve both their latency modeling effectiveness and design efficiency. For example, given a targeted GPU, each DNN layer's design only needs to consider a discrete design space, i.e., with a step size of a certain number of filters that matches the GPU granularity.

- *For existing DNN structures, the existence of tail effects is prevailing, which exposes many opportunities for both accuracy and latency performance improvements.* The reason of prevalence is that, as the GPU platform (and other factors like compilers) can vary, the unified DNN structure design

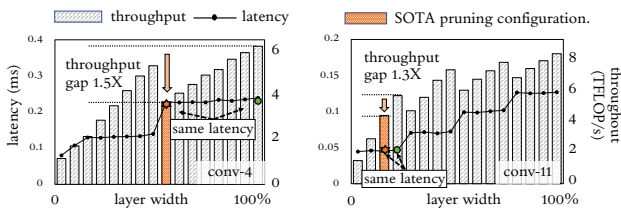


Fig. 7. The "ineffective pruning" pitfall. Without being aware of tail effect, the shown filter pruning configuration is sub-optimal, i.e., $1.3\times$ - $1.5\times$ less GPU throughput under the same latency. The DNN model is VGG16 [32] on CIFAR10 dataset as per the official open-sourced repository [15].

can hardly avoid tail effects in all different settings. In such a case, tail effect accumulation in DNNs can incur huge inefficiency, meanwhile exposing many optimizations potentials. Specifically, the GPU resource under-utilization enables us to increase DNN workload and utilize the “free” tail resources⁴ to boost accuracy while without affecting the runtime latency. In addition, as tail effect can emerge in many layers, by simultaneously pruning certain tail waves and filling up other ones, we can also obtain certain latency gains while maintain the overall DNN capacity and accuracy.

The above design insights can guide various DNN design optimization approaches. In this work, we mainly include approaches: GPU-aware NAS design space enhancement (§V-B) and lightweight DNN structure fine-tuning (§V-C).

B. GPU-aware NAS Design Space Enhancing

The first optimization we can apply is to enhance the design space of NAS methods with the proposed GPU awareness.

As discussed in related work (§II-C), previous NAS works mainly heuristically choose layer width in the design space, which may have sub-optimal GPU runtime performance. Therefore, one direct way of utilizing our GPU granularity-aware modeling is to find the optimal layer configurations and integrate into the NAS design space. The benefits are twofold: (i) our granularity-based modeling discretizes a continuous layer width design space and reduces the amount of search candidates for each layer, reducing the searching complexity; (ii) meanwhile, as each configuration already avoids the tail effect, our design space brings better GPU runtime performance without the tail effect related inefficiency.

Profiling-Guided Identification. We take a profiling strategy, *i.e.*, profiling the GPU latency staircase offline to identify the optimal design space for each layer.

Based on the layer runtime modeling in Eq. 1, the tail effect diminishes when DNN layer workload matches the GPU compute granularity, *i.e.*, the last wave of workload fully occupies the GPU. In such case, the GPU achieves optimal utilization, and the throughput also reaches local maximum in the range of current wave. Therefore, we leverage the GPU throughput T to identify the existence of tail effect in each layer and further detect its optimal layer configurations. Taking a retrospect of Figure 1 (c), we can see the throughput always peaks its local maximal (denoted by the green triangle) at the optimal layer size setting without tail effect and thus can serve as a reliable clue to track the optimal configurations. Specifically, we obtain the optimal configuration C_i^* for each layer i based on the following rule:

$$C_i^* = \arg \max_m (T_{i,m}) \quad (2)$$

where $T_{i,m}$ denotes GPU throughput of layer i in the m^{th} layer width configuration. We then use Eq. 2 to identify optimal configurations in different layer width intervals.

Overhead Analysis. Table I shows the offline profiling overhead on different DNN models. For each layer, we sample 16

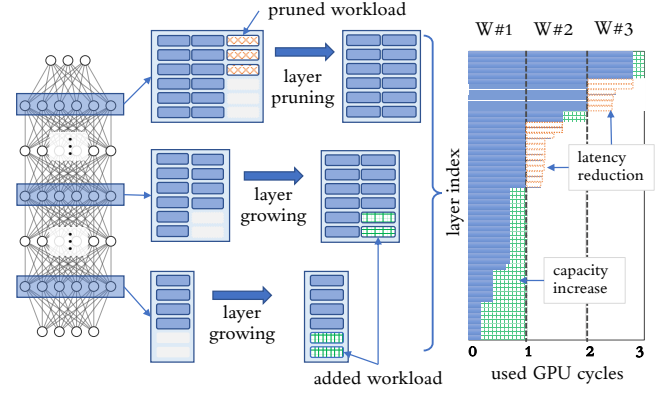


Fig. 8. We leverage two operations (*i.e.*, pruning and grafting) to eliminate the tail effect. Such two operations provide complementary latency gain and accuracy gain and facilitate the DNN accuracy-latency optimization.

different layer width configurations (from 1/16 to 16/16 with respect to the default layer size) and record their throughput. We observe the profiling overhead stays in the range of one to three minutes, which is generally acceptable considering the NAS overall search time.

C. Lightweight DNN Structure Fine-Tuning

Besides the end-to-end DNN design approach of NAS, we can also conduct lightweight DNN structure fine-tuning to obtain the optimal pruned models in structural pruning methods.

Structure pruning without GPU awareness can generate pruned models with excessive tail effects (§IV-B). Therefore, we can leverage **pruning and grafting** two operations to eliminate the tail effect. An overview of our method is shown in Figure 8. Specifically, *layer pruning* reduces the number of filters to remove the remaining workload on the last wave and thus can effectively harvest latency gain by an entire GPU wave processing cycle. *Layer grafting* instead increases the number of filters in other layers to fill up the GPU capacity in the last wave, which can compensate the accuracy loss of layer pruning without sacrificing the latency (*i.e.*, accuracy gain). These two operations provide us flexible opportunities to match the GPU granularity and thus eliminate the tail effect. Meanwhile, they also provides DNN optimization flexibility towards two optimization targets: latency reduction and accuracy lifting, which is one major difference from traditional pruning works.

Here one challenge remains, *i.e.*, how to select two operations between layer pruning and grafting across multiple layers regarding different optimization objectives? We propose

TABLE I
OFFLINE PROFILING OVERHEAD.

	Model	BS=32	BS=64	BS=128
	CIFAR (32x32x3)	VGG16 2m22s	2m23s	2m25s
	ResNet56	1m27s	1m30s	1m40s
	Model	BS=1	BS=2	BS=4
	ImageNet (224x224x3)	VGG16 2m14s	2m15s	2m18s
	ResNet50	2m50s	2m51s	2m56s

⁴Here we mainly denote latency-wise “free”. The power consumption may still show differences, which is out of scope of this work.

Algorithm 1 Lightweight DNN Structure Fine-tuning.

```

1: Input: Model's initial layer configs  $r[l]$ , latency  $L[l]$  and throughput profiling  $T[l]$ .
2: Output: New config  $R_{new}$ .
3: Identify design space  $C_l$  for each layer  $l$  by Eq. 2.
4: for layers  $i = 1$  to  $l$  do
5:   Get  $LG_i$ ,  $PG_i$  estimation through profiling results.
6:   Sort the layer index list  $[1, 2, \dots, i, \dots, l]$  by  $LG_i$  or  $PG_i$ .
7:   while layer index list is not empty do
8:     Pop out layer  $j$  with  $\text{argmax}_j LG[j]$ .
9:     Prune layer  $j$  with one step in design space  $C_l$ .
10:    while  $\Sigma^l PG(R_{new}) \notin (-\tau, \tau)$  do
11:      Pop out layer  $k$  with  $\text{argmin}_k LG[l]$ .
12:      Grow layer  $k$  with one step in design space  $C_l$ .
13:   Get runtime latency evaluation  $L_{new}$  of config  $R_{new}$ .
14:   if  $L_{new}$  achieves the target latency then
15:     Train and evaluate the model accuracy.
16:   else
17:     Set  $\tau *= 2$  and repeat the algo. from line 9.
18: Return Fine-tuned config  $R_{new}$ .

```

a lightweight fine-tuning algorithm to flexibly balance the corresponding operations to reach different objectives.

Optimization Objectives Formulation. We first define the DNN inference accuracy and runtime latency optimization objectives. We define the latency gain LG to indicate the latency being saved, and parameter gain PG (amounts of parameters been deleted or added) to indicate the model capacity as a guidance for retaining accuracy. The two metrics could be estimated by the change in different layer widths R_i and their latency profiling L_i .

Accuracy-Oriented Optimization aims to improve the model accuracy without extra latency overhead. Therefore, it can be formulated as maximizing the parameter gain while constraining latency gain:

$$\text{Maximize } \sum_i^l PG_i, \quad \text{s.t. } \sum_i^l LG_i \geq 0. \quad (3)$$

Latency-Oriented Optimization aims to reduce the runtime latency without accuracy drop, *i.e.*, maximizing latency gain while maintaining parameter gain in a tolerable range (τ):

$$\text{Maximize } \sum_i^l LG_i, \quad \text{s.t. } \sum_i^l PG_i \in (-\tau, \tau). \quad (4)$$

Algorithm Walk-through. Without loss of generality, we take the latency-oriented optimization as an illustrate example to explain the algorithm. The accuracy-oriented optimization follows the same principle by simply switching the optimization objectives as defined in Eq. 3–4.

Step 1. Layer Candidates Identification (Line 3): Given a DNN model structure, we first identify the optimal layer configurations, *i.e.*, design space C_i by Eq. 2.

Step 2. Intra-Layer Performance Gain Estimation (Line 4–6): For each layer, we get the latency and parameter gain estimation assuming conducting layer pruning and layer growing operations by one step in the design space C_i . These performance estimation will guide the inter-layer adjustment.

Step 3. Inter-Layer Adjustment Strategy (Line 7–13): To achieve latency optimization, we maintains a bi-directional queue of layer indexes, and greedily prune layers with maximal LG_j (max latency gain) and minimal LG_k for growing (balanced capacity). Same procedure applies to accuracy optimization by switching the objective/constrain in Line-9, 11.

Step 4. Model Structure Determination (Line 14–18): Finally, we return the new configuration if it reaches the targeted latency. Otherwise, we can adjust the constraints, *e.g.*, with larger parameter gain tolerance ($\tau* = 2$), to allow more aggressive layer pruning for latency reduction.

Algorithm Discussion. Note that different from traditional pruning-from-scratch method, our method is designed as an orthogonal pruning “post-calibration” method on top of well-searched pruning/NAS configurations, which could be easily combined with any global pruning method that finds optimal accuracy-aware pruning configurations. In fact, our pruning and grafting operations usually yiled minor pruning size calibration within one staircase as shown in Figure 8, for example, adjusting from pruning 40 filters (an accuracy-aware best pruning configuration but NOT latency-aware) to 48 filters (both latency- and accuracy-aware best setting) in a 512-filter layer. Such a practice incurs minimal accuracy change to the optimal pruning configurations. Meanwhile, with the small filter number changes, we adopt the simple but effective guidance metrics that are the parameter gain (PG) and latency gain (LG). Even though the PG indicator is not mathematically proven to be related to accuracy modeling like traditional from-scratch pruning method, it is more designed as a lower-bound indicator that our pruning post-calibration method is not hurting the model capacity (*e.g.*, the parameter gain remains largely positive/non-negative), thus are shown to achieve fairly stable accuracy improvement and latency reduction, which we will show later in our experiments.

VI. EXPERIMENTAL EVALUATION

In this section, we evaluate our latency-aware DNN design methodology on a series of common benchmarks. Specifically, we demonstrate our performance gain in different approaches.

Experimental Setup. We conduct experiments using the common software stack including CUDA 10.2 and CuDNN 7.6.5. For GPU hardwares, we select four GPUs from high-end (Ampere 3080Ti, Volta Titan-V, Pascal P6000) to embedded ones (Jetson Nano). The detailed specifications of these GPUs are shown in Table II, including number of SMs and CUDA cores, as well as the peak FP32 performance.

We apply our method into two common model optimization approaches, *i.e.*, *structural filter pruning* and *NAS*. For pruning, we compare with several state-of-the-art pruning methods including HRank [14], SoftPruning [8], Network Slimming [18] and Dependency Graph [5]. For NAS, we apply our model optimization method onto the EfficientNet series of structures [34], which are one type of the SOTA efficient model architectures. For model performance evaluation, we compare the model's accuracy and runtime latency. Without specific mentioning, the batch sizes for latency evaluation on CIFAR10

TABLE II
THE EVALUATED GPUS AND SPECIFICATIONS.

GPU Name	Architecture	#SMs	#Cores	Peak FLOP/s
3080Ti	Ampere	80	10240	29.8T
Titan-V	Volta	80	5120	14.9T
Quadro P6000	Pascal	30	3840	12.0T
Jetson Nano	Maxwell	1	128	0.24T

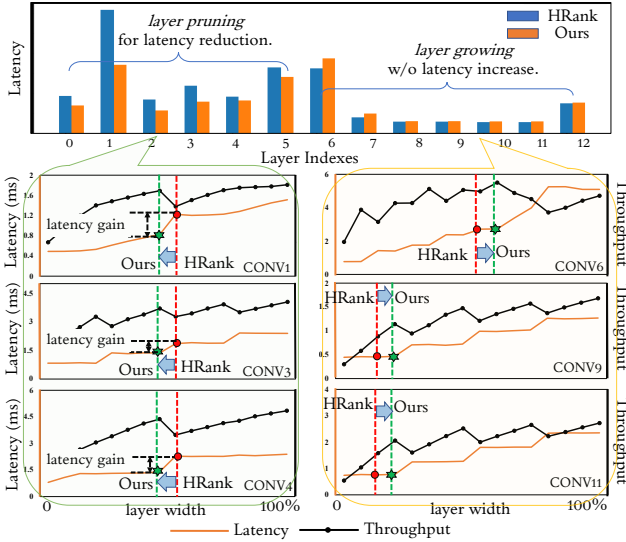


Fig. 9. Our DNN optimization illustration. We conduct layer pruning in Conv1-5 to get latency reduction, and layer growing in Conv6-12 to recover model capacity but without obvious latency increment. During this, we eliminate tail effects across layers and get latency reduction.

and ImageNet are set to 128 and 1 to observe the latency variations. Meanwhile, we also collect the GPU throughput (FLOP/s) information as a indicator to demonstrate the GPU runtime efficiency comparison.

A. Evaluation on the Pruning Approach

A Proof-of-Concept Case Study. As a proof-of-concept, we first apply our optimization to a prevalent DNN benchmark model VGG16 and compare the results with one SOTA pruning competitor HRank [14] to illustrate our algorithm mechanism. The result is shown in Figure 9.

Our results (orange bar) consistently outperform HRank (blue bar) on run-time latency reduction, as shown in Layer 0~5. We further showcase the latency and throughput of three randomly selected layers for both layer pruning and growing operations (bottom figure). As shown in CONV-1,3,4, we successfully remove one tail latency by conducting layer pruning operation. At the same time, the GPU throughput achieves its local maximal. The layer growing actions, on the other hand, brings little to no latency overhead, as shown in CONV-6,9,11. Taking a further scrutiny on their latency results, we can see that the layer width is increased by a proper amount such that the layer's runtime latency stays similar with HRank. Overall, we can achieve **17.7%** latency reduction compared

TABLE III
LATENCY OPTIMIZATION AND GPU THROUGHPUT ANALYSIS (TITAN V).

	Method	Params	#FLOPs	FLOP/s	Acc.%	Time (ns)
VGG16 (CIFAR10)	HRank [14]	1.90M	67.0M	2.41T	93.1	2.50E6
	Ours	2.85M	104.1M	3.90T	92.9	2.05E6 (-17.7%)
ResNet56 (CIFAR10)	HRank [14]	0.48M	65.9M	0.83T	93.6	4.20E6
	Ours-1	0.50M	79.1M	1.00T	93.8	3.72E6 (-11.3%)
	Ours-2	0.50M	75.1M	0.95T	93.5	3.51E6 (-16.3%)
ResNet56 (CIFAR10)	SOFT-1 [8]	0.53M	68.8M	0.88T	93.1	4.08E6
	Ours-1	0.43M	71.4M	0.91T	93.2	3.52E6 (-13.7%)
	SOFT-2 [8]	0.45M	53.1M	0.68T	92.3	3.64E6
	Ours-2	0.43M	66.0M	0.79T	92.3	3.01E6 (-17.3%)

to HRank with only **0.2%** accuracy drop (93.1%→92.9%), as shown in the first two rows in Table III.

Pruning Accuracy and GPU Throughput Analysis. We then apply our model optimization method in more pruning benchmarks and analyse both the latency and GPU throughput. The result comparisons are shown in Table III. Note that, SOFT method [8] has two pruning configurations (denoted by SOFT-1 and SOFT-2) with varied pruning degrees, which we compare separately. The latency evaluation is conducted on Titan-V.

Latency Reduction: As Table III presents, by optimizing the model structure configurations, our method achieves consistently lower latency than baseline methods (**11.3%-17.7%** latency reduction in all settings). Different from previous work that focuses on reducing workload to reduce latency, our method's latency reduction mainly comes from the higher GPU runtime efficiency, *e.g.*, GPU throughput improvement.

GPU Throughput Improvement: As Table III shows, our method shows consistently higher GPU throughput than the baselines, *i.e.*, in the (FLOP/s) column. For VGG16, our optimized model achieves 3.90 TFLOP/s, **1.6×** throughput than the baseline method, 2.41 TFLOP/s. For ResNet56, our models also achieve **1.2×** throughput than HRank (1.00 vs. 0.83 TFLOP/s) and SOFT-2 (0.79 vs. 0.68 TFLOP/s). Such throughput enhancement indicates the GPU runtime efficiency improvement, and shows the advantages of our configuration optimization by tail effect elimination.

Accuracy Maintenance: In addition to lower latency, our method also maintains the model accuracy change within negligible ranges ($\pm 0.2\%$). This is benefited by constraining the parameter gain (PG) in the algorithm. As (Params) column in Table III shows, our optimized models have maintained the amount of parameters either in positive range or very small negative range. By doing so, we ensure the models' capacity to be well-maintained, thus keeping the accuracy during the optimization.

Fine-Grained Comparison with SOTA Benchmarks. To further illustrate the advantages of our methods, we conduct a fine-grained analysis by applying our method on top of two state-of-the-art methods: *Network Slimming (SLIM)* [18] and *Dependency Graph (DEPG)* [5].

Specifically, given an upper bound pruning ratio for one network (*e.g.*, 70%), we enumerate all pruning configurations below the upper bound with a specific pruning ratio step size (*e.g.*, 5%). For each pruning ratio, we run the baseline

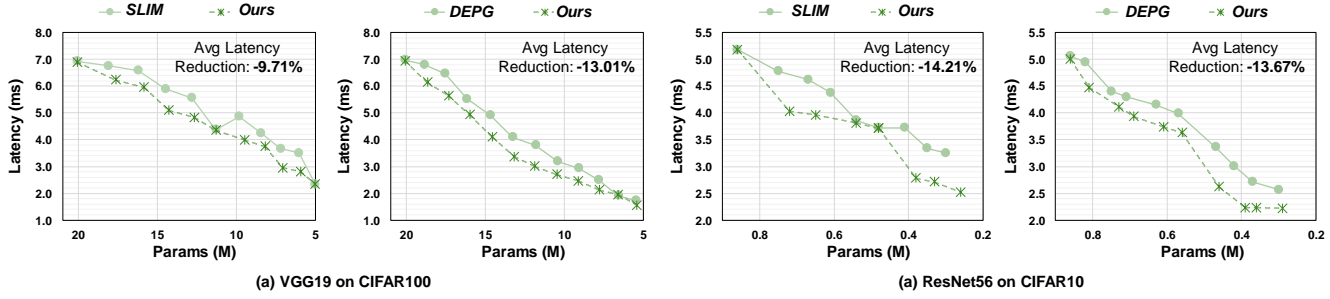


Fig. 10. We applied our fine-tuning algorithm (*Ours*) onto each pruning configuration generated by *SLIM* [18] and *DEPG* [5]. Experiments show our optimization achieves consistent latency reduction (on average 10% to 14%) on top of the baseline. Latency measured on 3080Ti with batch size = 128.

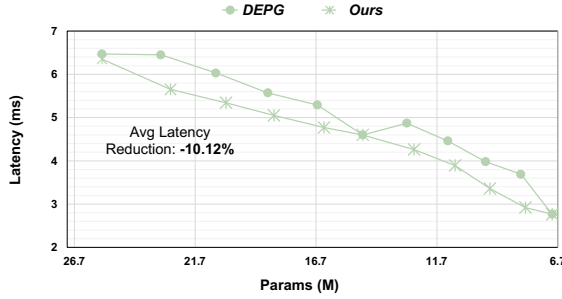


Fig. 11. We applied our fine-tuning algorithm (*Ours*) onto each pruning configuration generated by *DEPG* [5] (ResNet50 backbone on ImageNet). Experiments show our pruning optimization achieves consistent latency reduction (on average 10.12%) on top of the baseline methods. Latency measured on 3080Ti with batch size = 8.

algorithms (*SLIM*, and *DEPG*) to find the accuracy-optimized pruning configurations. Then our layer pruning and grafting Algorithm 1 is applied on top to finetune the baseline pruning configuration to remove the GPU tail effect, and thus generated a both accuracy- and latency-optimized configuration. We then compare the latency vs parameter amount plot to compare each of the baseline and our optimized results.

Figure 10 shows the detailed latency curve comparison. Latency are measured on 3080Ti GPU with batch size 128 for both CIFAR10 and CIFAR100. As we can see, most pruning configurations generated by tail-unaware baselines (*SLIM*, and *DEPG*) run with longer latencies, indicating they are far from optimal in terms of actual GPU runtime efficiency. By contrast, by slightly optimizing each layer's configuration to remove the tail effect, our fine-tuned model architecture could easily save 10%-14% average latency for nearly all pruning configurations.

To further demonstrate that our pruning and grafting algorithm only has minimal accuracy impact, we compare the model re-training performance between the baselines and our optimized model configurations. Here for simplicity, we only compare the model pruning settings that are reported in the official codebase benchmarks⁵. The results are shown in Table. IV. As we can see, ours algorithms yields minimal accuracy variation (-0.15% to +0.02%) but achieves consistently 10% to 17.7% latency reduction.

The above experiments demonstrate the major benefits of our algorithm: *orthogonal design*, that is, our pruning and grafting algorithm could be combined with various of different SOTA pruning algorithm logics (e.g., local-based ranking [14],

TABLE IV
ACCURACY / LATENCY COMPARISON WITH SOTA BENCHMARKS.

		Org. Params	Params	Accuracy	Latency (ns)
VGG19 (CIFAR100)	SLIM [18]	20.09M	1.27M	67.75%	1.03E+06
	SLIM+Ours	20.09M	1.25M	67.64%	9.2E+05 (-10.7%)
	DEPG [5]	20.09M	1.18M	70.39%	1.01E+06
	DEPG+Ours	20.09M	1.23M	70.24%	9.1E+05 (-9.9%)
ResNet56 (CIFAR10)	SLIM [18]	0.86M	0.41M	93.29%	3.72E+06
	SLIM+Ours	0.86M	0.41M	93.31%	3.46E+06 (-6.99%)
	DEPG [5]	0.86M	0.37M	93.64%	2.71E+06
	DEPG+Ours	0.86M	0.36M	93.57%	2.23E+06 (-17.7%)
ResNet50 (IMGNET)	DEPG [5]	25.56M	12.74M	75.83%	4.87E+06
	DEPG+Ours	25.56M	12.53M	75.78%	4.23E+06 (-13.1%)

TABLE V
ACCURACY OPTIMIZATION ON EFFICIENTNETS.

Method	Acc.%	GPU:Titan-V		GPU:P6000	
		Time(ms)	FLOP/s	Time(ms)	FLOP/s
EfficientNet (ImageNet)	B0	77.52	12.6	61.9G	13.8
	Ours	81.49 (+3.97)	12.7	414.2G	14.1
EfficientNet (ImageNet)	B1	79.38	17.8	78.7G	19.8
	Ours	82.08 (+2.7)	18.0	442.7G	19.9
EfficientNet (ImageNet)	B2	80.18	18.2	109.9G	19.9
	Ours	82.72 (+2.54)	18.4	547.3G	20.4

[8], [18] or global-based ranking mechanisms [5], different l1-norm or other importance criteria, etc). When the SOTA algorithms generate an accuracy-oriented optimal configuration, further applying our algorithm on top will then generate a both accuracy-optimal and GPU efficiency-optimal configuration, thus achieving consistently better accuracy-latency trade-offs.

B. Evaluation on the NAS Approach

In this part, we apply our optimization method to further optimize the NAS network's performance on GPUs. Specifically, we optimize the EfficientNet [34] series of model structures to achieve better accuracy latency trade-offs. The evaluation results are shown in Table V. The latency evaluation is conducted on Titan-V and P6000 GPUs with batch size = 1 to simulate the real-time performance.

Accuracy Maximization: As Table V shows, with the similar runtime latency, our optimized EfficientNet model could achieve much better accuracy (+3.97%, +2.7%, +2.54%, respectively) on the challenging ImageNet dataset.

Better Latency Accuracy Trade-offs: We composedly compare the model accuracy-latency performance with original Efficient-

⁵<https://github.com/VainF/Torch-Pruning/tree/master/benchmarks>

Nets [34], and several major types of DNN structures including ResNets, ResNeXts, InceptionNet, and Dense Net. The results are shown in Fig. 12. From the *accuracy* perspective, our optimized B0 to B3 models achieve the highest accuracies than most baselines under the same latency. From the *latency* perspective, our model B2's latency is $1.5\times$ less than the models which achieve the same accuracy level, *e.g.*, EfficientNet-B3 and B4. Therefore, both perspectives demonstrate the effectiveness of our method in achieving better accuracy latency trade-offs on GPUs.

Optimization Explanation: Here we explain our optimization mechanisms for the EfficientNet series of networks. When conducting EfficientNet inference on GPUs, we observe that most of layers in these models consume less than one GPU wave of workload, *i.e.*, highly under-utilizing the GPUs. The GPU throughput is only 61.9 - 109.9 GFLOP/s for B0 to B2 as shown in Table V. Such under-utilized GPU capacity allows us to conduct layer growing to reach one full wave for each layer without latency increment, thus bringing higher model capacity and model accuracy.

For implementation, EfficientNet series of networks have three scaling dimensions in their initial design: layer width (w), input resolution (r) and network depth (d) [34]. Out of the three, we balanced scale up both (w) and (r) to increase the network capacity. The depth (d) dimension, however, is kept same with the baseline model, as network layers are run sequentially and depth scaling would inevitably incur latency increase. By width and resolution growing, the optimized models have higher accuracy as well as improved GPU throughput, *e.g.*, 414.2 - 547.3 GFLOP/s, while maintaining the similar latency with baselines, as shown in Table V.

C. Generalizability across GPUs

We further extend the generalizability evaluation to different high-end GPU, P6000 and embedded GPU, Jetson Nano. The detailed GPU specifications can be found in Table II.

The overall results are shown in Table VI and Table VII. On the P6000 GPU (Table VI), our method could achieve **9.0%** to **27.2%** latency reduction than SOTA methods while maintaining

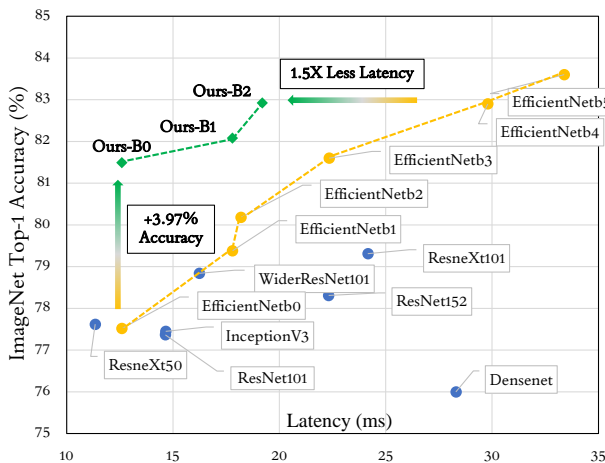


Fig. 12. ImageNet accuracy and latency comparisons. Compared to EfficientNets (yellow) and other models, our GPU-aware optimized models achieve better accuracy latency trade-offs. Evaluated platform: NVIDIA Titan-V GPU.

TABLE VI
GENERALIZABILITY EVALUATION ON PASCAL P6000 GPU.

	Method	Params	FLOP/s	Acc.%	Time (ns)
VGG16	HRank	1.90M	1.68T	93.1	4.15E6
	Ours	2.04M	1.86T	92.9	3.02E6 (-27.2%)
ResNet56	HRank	0.48M	0.84T	93.6	5.84E6
	Ours	0.50M	1.00T	93.7	5.32E6 (-9.0%)

TABLE VII
GENERALIZABILITY EVALUATION ON JETSON NANO GPU.

	Method	Params	FLOP/s	Acc.%	Time (ms)
VGG16	HRank	1.90M	35.5G	93.1	60.5
	Ours	2.04M	39.4G	92.9	48.1 (-20.5%)
ResNet56	HRank-1	0.48M	25.6G	93.6	82.1
	Ours	0.50M	28.8G	93.7	68.0 (-17.1%)
	HRank-2	0.24M	16.7G	92.3	67.2
	Ours	0.39M	24.3G	92.5	58.1 (-13.3%)

similar accuracy for VGG16 and ResNet56, demonstrating the generality across different high-end GPU architectures. For Jetson Nano GPU (Table VII), although it has relatively smaller computing capacity, our method still achieves **13.3%** to **20.5%** latency reduction, demonstrating the generalizability for both high-end and embedded GPU platforms. The above results demonstrate the effectiveness and generalizability of our DNN optimization algorithm. Without any GPU-specific assumptions, our profiling-guided design could be applied to a spectrum of GPUs to enhance the DNNs' accuracy-latency trade-offs.

D. Generalizability in DNN Architectures

In this part, we conduct generalizability evaluation of DNN latency staircase in terms of different DNN hyper-parameters, including batch sizes, input resolutions and filter shapes.

Batch Size Variation: Fig. 13 shows the latency *w.r.t.* varied batch size from 1 to 256 for two CONV layers of VGG16 on ImageNet resolution. The latency staircase consistently exists for all layers with different batch sizes. Specifically, layers with larger batch size incurs a longer processing cycle for a wave, leading to a higher latency staircase.

Number of Filters Variation: Fig. 13 also shows that CONV-5 has more levels of staircase compared to CONV-3. The reason is that CONV-5 has more filters (128) than CONV-3 (64). Since the number of filters determines the number of thread blocks, more sequential GPU waves are needed with a larger number of filters, leading to more levels of staircase.

Input Resolution Variation: Similar latency evaluation is also conducted *w.r.t.* varied input resolution from 128^2 to 1024^2 for CONV-3 and CONV-5 in Fig. 14. The latency staircase still consistently shows in both low to high resolutions.

Filter Shape Variation: In addition, we evaluate layers with different filter shapes from 1×1 , 3×3 , 5×5 dense filters to 3×3 depthwise filter, while keep the maximum layer workload for

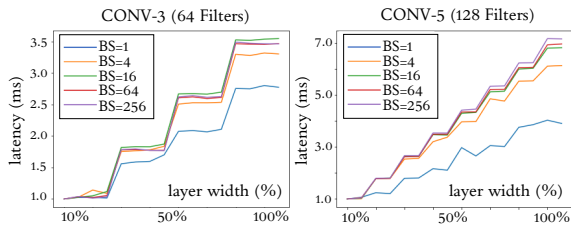


Fig. 13. Latency staircase *w.r.t* varied batch size (1 to 256 in sub-figure) and number of filters (64 to 128).

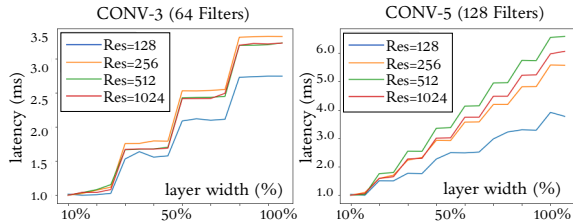


Fig. 14. Latency Staircase *w.r.t* varied input resolution.

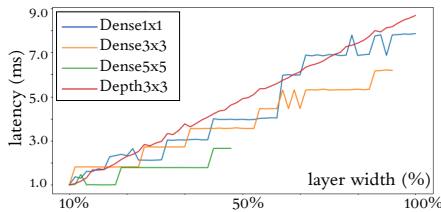


Fig. 15. Latency staircase *w.r.t* varied filter shape.

each type to be the same⁶. Fig. 15 shows the generalizability of latency staircase *w.r.t*. 1x1, 3x3 and 5x5 dense filters. One exception here is the depthwise 3x3 filter, whose latency results demonstrate a relatively linear pattern. The reason is that the depthwise filter is more light-weight compared to dense filters. Each GPU processing cycle for one wave of depthwise workload is thus smaller, leading to no dramatic upward edge in the latency curve. However, as depthwise filters are usually combined with 1x1 dense filters to compose depth-width layers, *e.g.*, in MobileNets [31], MnasNets [33], the latency staircase still exists for such models.

VII. LIMITATIONS AND FUTURE WORKS

A. Future Works for Concurrent Kernel Analysis

Our current work adopts a *single-model* execution assumption as stated in NN-meter [40], *i.e.*, the assumption that kernels run sequentially on each device, even for those ones without dataflow dependency. This is true for most edge GPUs with limited capacity [40]. Meanwhile, we have observed the same sequential execution patterns hold for our tested architectures (*e.g.*, VGGs, ResNets, EfficientNets) and tested server-level GPUs (*e.g.*, Ampere 3080Ti, Volta Titan-V, Pascal P6000).

However, with the continuous CUDA feature upgrading, we do see recently some new CUDA features (such as multi-stream,

⁶The difference between dense and depthwise filter is the dense kernel is of shape $n \times n \times 512$, while the depth-wise kernel is only $n \times n \times 1$. The latter one is mainly used in light-weight models like MobileNets [31].

multi-process service MPS [27], multi-instance MIG [26]) enable *multiple models* and *concurrent kernel execution* on the same GPU since the Ampere architecture. This will lead to complex cross-effects between several kernels such that when multiple kernels share the same resources, the tail effect's impact can be different from current single-kernel execution. However, in the current stage, a practical limitation that we have been facing is that, up-to-now, the available CUDA profiling toolsets (NVIDIA NSight/Compute [24]) still cannot force two kernels to accurately align to co-run, thus it can be hard to analyze their concurrency states, nor profile two arbitrary kernels' concurrent execution time accurately. Multi-round end-to-end multi-model co-running and profiling can be a potential way to remediate such limitations but also require more statistical analysis.

B. Limitations for Vision Transformers

Our current work mainly focuses on convolutional neural networks including traditional VGG and ResNets, as well as the NAS-based networks like EfficientNets, which cover most of the basic convolution components for modern CNNs. Recently, vision transformer (ViT) architectures are introduced [17], [4]. ViT architecture features with a new meta-layer type, *i.e.*, multi-head self-attention (MHSA) layer, and a new type of transformer blocks composed MHSA layers and feed-forward fully connected (FC) layers.

With the specific transformer block structure and MHSA layers, the CNN-oriented channel pruning mechanisms for CNNs are no longer suitable. Instead, ViT pruning method design and the latency analysis faces multiple major challenges that are still awaiting research and analysis: (1) ViT pruning can happen within more dimensions, *e.g.*, image patch embedding size (E), number of head (H), query & query size (QK), value size (V) and FC layer hidden dimension (M). (2) Pruning different ViT dimensions will require corresponding dimension matching across inter-dependent layer shapes [37], [29], for example, image patch embedding size (E) will also impact all query sizes and FC layer's output sizes. With compound dimension pruning challenges and different attention computation mechanisms, our current tail effect analysis is not directly applicable onto the ViT-based architecture pruning, as shown in Figure 16. However, the fundamental parallel computing mechanism and the wave-based granularity of GPU

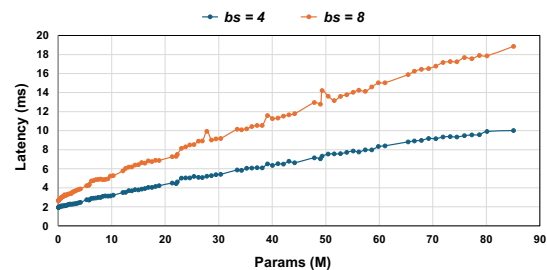


Fig. 16. We applied fine-grained pruning on ViT-Base. As we can observe, the model latency tends to be linearly correlates to the ratio of pruned num of heads, indicating the limitation of the current tail effect analysis on MHSA regarding the multi-step multi-head computation patterns, correlated FFA computation changes, etc. Latency measured on 3080Ti with batch size = 8/4.

architecture still remains. Therefore, we leave the compound tail-effect analysis on ViTs as future work.

VIII. CONCLUSION

In this work, we revisit the GPU tail effect, a classic parallel system issue, and shows it can cause significant GPU under-utilization and lower the GPU throughput for DNN inference. Detailed DNN workload characterization demonstrates the prevalence of GPU tail effect across different DNN architectures is due to the unique deep structure and the light-weight layer workload, which exacerbates the tail effect in DNN inference. We then demonstrate two user cases with simple yet effective solutions to eliminate the tail effect and enhance the DNN run-time efficiency. With effectiveness, feasibility, and generalizability well proved, our optimization methods show outstanding DNN latency accuracy trade-offs, e.g., 11%–27% latency reduction over state-of-the-art (SOTA) DNN pruning and NAS approaches.

REFERENCES

- [1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.
- [2] Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on computer vision and pattern recognition*, pages 11398–11407, 2019.
- [3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.
- [4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
- [5] Gongfan Fang, Xinyin Ma, Mingli Song, Michael Bi Mi, and Xinchao Wang. Depgraph: Towards any structural pruning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16091–16101, 2023.
- [6] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.
- [7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
- [8] Yang He, Guoliang Kang, Xuanyi Dong, Yanwei Fu, and Yi Yang. Soft filter pruning for accelerating deep convolutional neural networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, pages 2234–2240, 2018.
- [9] Yang He, Ping Liu, Ziwei Wang, Zhilan Hu, and Yi Yang. Filter pruning via geometric median for deep convolutional neural networks acceleration. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4340–4349, 2019.
- [10] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.
- [11] Yihui He, Xiangyu Zhang, and Jian Sun. Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1389–1397, 2017.
- [12] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
- [13] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.
- [14] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrank: Filter pruning using high-rank feature map. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1529–1538, 2020.
- [15] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, and Ling Shao. Hrankplus github repo, 2020. <https://github.com/lmbxmu/HRankPlus>.
- [16] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.
- [17] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 10012–10022, 2021.
- [18] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. In *Proceedings of the IEEE international conference on computer vision*, pages 2736–2744, 2017.
- [19] microsoft. Deep learning inference service at microsoft, 2020. <https://www.usenix.org/system/files/opml19papers-soifer.pdf>.
- [20] NVIDIA. Cuda pro tips (page 13), 2020. <https://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf>.
- [21] NVIDIA. Cudnn kernel invoking logic, 2020. https://docs.nvidia.com/deeplearning/cudnn/api/index.html-cudnnGetConvolutionForwardAlgorithm_v7.
- [22] NVIDIA. Dnn compiling stack and kernel base, 2020. <https://developer.download.nvidia.com/video/gputechconf/gtc/2020/s21685-cuDNN-v8-New-Advances-in-Deep-Learning-Acceleration-APIs-Optimizations-and-How-to-Tackle-the-Future-Challenges-in-Hardware-and-Software.pdf>.
- [23] NVIDIA. Jetson devices, 2020. <https://www.nvidia.com/en-us/autonomous-machines/jetson-store/>.
- [24] Nvidia. Nsight compute — nvidia, 2020. <https://developer.nvidia.com/nsight-compute>.
- [25] NVIDIA. Nvidia cudnn documentation — kernel selection heuristics, 2020. <https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.htmltroubleshooting>.
- [26] NVIDIA. Nvidia multi instance gpu (mig), 2020. <https://docs.nvidia.com/datacenter/tesla/mig-user-guide/>.
- [27] NVIDIA. Nvidia multi process service (mps), 2020. https://docs.nvidia.com/deploy/pdf/cuda_multi_process_service_overview.pdf.
- [28] NVIDIA. Nvidia titan v — volta architecture, 2020. <https://www.nvidia.com/en-us/titan/titan-v/>.
- [29] Lorenzo Papa, Paolo Russo, Irene Amerini, and Luping Zhou. A survey on efficient vision transformers: algorithms, techniques, and performance benchmarking. *arXiv preprint arXiv:2309.02031*, 2023.
- [30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.
- [31] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.
- [32] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [33] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828, 2019.
- [34] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.
- [35] Han Vanholder. Efficient inference with tensorrt, 2016.
- [36] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 10734–10742, 2019.

- [37] Huanrui Yang, Hongxu Yin, Maying Shen, Pavlo Molchanov, Hai Li, and Jan Kautz. Global vision transformer pruning with hessian-aware saliency. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 18547–18557, 2023.
- [38] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 285–300, 2018.
- [39] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-grained gpu sharing primitives for deep learning applications. *arXiv preprint arXiv:1902.04610*, 2019.
- [40] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 81–93, 2021.



Fuxun Yu received the B.S. degree from Harbin Institute of Technology, Harbin, China, in 2017. He received his Ph.D. degree in the Department of Electrical and Computer Engineering at George Mason University under the supervision of Prof. Xiang Chen. His current research interests include high performance deep neural network computing, full-stack DNN computing optimization, deep learning security, interpretability and explainability of deep learning.



Zirui Xu received the B.S. and M.S. degree from Beijing Jiaotong University, Beijing, China, in 2014 and in 2017, respectively. He received his Ph.D. degree with the Department of Electrical and Computer Engineering at George Mason University under the supervision of Prof. Xiang Chen. His current research interests include high performance mobile computing system, neural network model optimization, and mobile intelligent application robustness and security.



Longfei Shangguan is an assistant professor in the Computer Science Department. Before joining Pitt, he was a senior researcher at Microsoft, Redmond. Dr. Shangguan's research interests are in all aspects of IoT systems: from building novel IoT applications, solving security issues, all the way down to optimizing the network stack and designing low-power IoT hardware. His research has been recognized by a Google Research Scholar Award, MobiCom Best Paper Runner-up Award, Ubicomp Distinguished Paper Award, and an AIOtsys Young Scientist Award.



Di Wang is currently a Principal Research Manager with Microsoft. He earned his B.E. in Computer Science and Technology from Zhejiang University (2005), M.S. in Computer Systems Engineering from Technical University of Denmark (2008), and Ph.D. in Computer Science and Engineering from The Pennsylvania State University (2014). His research spans the areas of artificial intelligence, computer systems, computer architecture, and energy efficient system design and management. He has authored more than 40 peer-reviewed papers. He received 5 best paper awards and 2 best paper nominations.



Dimitrios Stamoulis is currently a Principal Research Manager at Microsoft AI, working on Neural Architecture Search (NAS) for object detection applications. He received his PhD in Electrical and Computer Engineering from Carnegie Mellon University under the supervision of Diana Marculescu. Previously, He received a MEng in ECE from McGill University, Montreal, and a Diploma in ECE from NTUA, Athens.



Rishi Madhok is currently a Senior Applied Science Manager at Microsoft and leads high-impact engineering and applied AI projects. He received his M.S. in Computer Science from Carnegie Mellon University. Before that, he received his B.S. degree from Delhi Technological University. His current research interests lie in object detection, semantic segmentation, and tracking models from aerial/satellite imagery.



Nikolaos Karianakis is currently a Principal Research Manager in Mixed Reality at Microsoft and leads high-impact engineering and applied AI projects. He received his Ph.D. in Computer Science from UCLA at the Vision Lab under the supervision of Stefano Soatto. His research spans the broader areas of Computer Vision and Machine Learning. His current focus is Computer Vision and Large Language Models for Aerial and Horizontal Imaging.



Ang Li is a tenure-track assistant professor in the Department of Electrical and Computer Engineering at University of Maryland College Park. He obtained Ph.D. from Duke University under the supervision of Professor Yiran Chen. His research interests lie in the intersection of machine learning and edge computing, with a focus on building large-scale networked and trustworthy intelligent systems to solve practical problems in a collaborative, scalable, secure, and ubiquitous manner.



ChenChen Liu received the M.S. degree from Peking University, Beijing, China, in 2013, and the Ph.D. degree from the ECE Department at the University of Pittsburgh, Pittsburgh, USA, in 2017. She is currently an Assistant Professor in the Department of Computer Science and Electrical Engineering at University of Maryland, Baltimore County. Her current research interests include brain-inspired computing system, machine learning, and emerging nonvolatile memory technologies.



Yiran Chen received B.S. (1998) and M.S. (2001) from Tsinghua University and Ph.D. (2005) from Purdue University. After five years in the industry, he joined the University of Pittsburgh in 2010 as Assistant Professor and was promoted to Associate Professor with tenure in 2014, holding Bicentennial Alumni Faculty Fellow. He is now the John Cocke Distinguished Professor of Electrical and Computer Engineering at Duke University and serving as the director of the NSF AI Institute for Edge Computing Leveraging the Next-generation Networks (Athena), the NSF Industry-University Cooperative Research Center (IUCRC) for Alternative Sustainable and Intelligent Computing (ASIC), and the co-director of Duke Center for Computational Evolutionary Intelligence (DCEI).



Xiang Chen is a Tenure-Track Associated Professor of the Department of Computer Science and Technology, the School of Computer Science, Peking University. He achieved his MS. and Ph.D. degrees with a major in Computer Engineering under the supervision of Dr. Yiran Chen at the University of Pittsburgh and finished the bachelor's degree at Northeastern University in China in 2010. After his Ph.D. graduation in 2016, he directly joined George Mason University, where he led 7 National Science Foundation (NSF) projects and achieved the NSF CAREER Award. In 2023, he joined Peking University. His research works focus on mobile and distributed computing systems, high performance intelligence computing, and related Edge, IoT, and CPS applications.