VELTAIR: Towards High-Performance Multi-tenant Deep Learning Services via Adaptive Compilation and Scheduling

Zihan Liu
[1]Shanghai Jiao Tong University
[2]Shanghai Qi Zhi Institute
Shanghai, China
altair.liu@sjtu.edu.cn

Jingwen Leng*
[1]Shanghai Jiao Tong University
[2]Shanghai Qi Zhi Institute
Shanghai, China
leng-jw@sjtu.edu.cn

Zhihui Zhang
[1]Shanghai Jiao Tong University
[2]Shanghai Qi Zhi Institute
Shanghai, China
zhihui.zhang@sjtu.edu.cn

Quan Chen
[1]Shanghai Jiao Tong University
[2]Shanghai Qi Zhi Institute
Shanghai, China
chen-quan@cs.sjtu.edu.cn

Chao Li
[1]Shanghai Jiao Tong University
[2]Shanghai Qi Zhi Institute
Shanghai, China
lichao@cs.sjtu.edu.cn

Minyi Guo*
[1]Shanghai Jiao Tong University
[2]Shanghai Qi Zhi Institute
Shanghai, China
guo-my@cs.sjtu.edu.cn

ABSTRACT

Deep learning (DL) models have achieved great success in many application domains. As such, many industrial companies such as Google and Facebook have acknowledged the importance of multi-tenant DL services. Although the multi-tenant service has been studied in conventional workloads, it is not been deeply studied on deep learning service, especially on general-purpose hardware.

In this work, we systematically analyze the opportunities and challenges of providing multi-tenant deep learning services on the general-purpose CPU architecture from the aspects of scheduling granularity and code generation. We propose an adaptive granularity scheduling scheme to both guarantee resource usage efficiency and reduce the scheduling conflict rate. We also propose an adaptive compilation strategy, by which we can dynamically and intelligently pick a program with proper exclusive and shared resource usage to reduce overall interference-induced performance loss. Compared to the existing works, our design can serve more requests under the same QoS target in various scenarios (e.g., +71%, +62%, +45% for light, medium, and heavy workloads, respectively), and reduce the averaged query latency by 50%.

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; *Cloud computing*; • **Computing methodologies** → Concurrent algorithms.

KEYWORDS

Multi-tenant, Deep Learning Service, Compiling, Scheduling

*Jingwen Leng and Minyi Guo are the corresponding authors of this paper.

ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland

ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507752

ACM Reference Format:
Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: Towards High-Performance Multi-tenant Deep Learning Services via Adaptive Compilation and Scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3503222.3507752

1 INTRODUCTION

Deep learning (DL) models have achieved great success in the various domains including vision [29, 36, 49, 50, 52, 53], natural language processing [15, 24], and even graph learning [62, 67]. To meet the need of rising computation power of DL models, computer architects have proposed various hardware designs including general-purpose hardware [45] and domain-specific architectures [7, 10, 18, 19, 26, 27, 32, 58, 61, 68] for accelerating deep learning models for their superior energy efficiency.

Different from the computation-heavy training process, it is difficult for the inference of a single deep learning model to fully use the hardware, which typically runs with a small batch size [2]. As such, sharing multiple DL models on a single hardware, i.e., multi-tenant deep learning serving, has become increasingly important [12, 21]. Compared to the single-tenant serving, multi-tenancy brings several challenges, including resource management and allocation, shared resource competition [40, 60], tasks scheduling [39, 51], etc. For conventional multi-tenant workloads, researchers have proposed various solutions based on resource partition [6], hardware isolation [34], and so on. Similarly, researchers have proposed various architectural support for multi-tenant DL serving [2, 12, 21] that leverages temporal and spatial multitasking.

However, the multi-tenant DL serving has its unique challenges, which are overlooked by previous multi-tenant DL serving works. We first find that owing to the complex inner-structure of the DL models [11], the scheduling granularity has a profound impact on the multi-model serving throughput. Meanwhile, we demonstrate that the performance of DL models is very sensitive to code generation strategies [8, 56, 65, 66]. In specific, those current DL compilers mainly focus on optimizing the performance of a single model or

even a single layer by various code transformations under the assumption of singe-tenancy. Our experimental results show that the performance of generated code degrades rapidly under multi-tenant scenarios due to the shared resource contention.

In this work, we propose Veltair, a software solution that provides the high-throughput and low-interference multi-tenant deep learning serving. We systematically analyze the resource allocation conflict and inter-layer interference on the CPU platform, which closely represents the industrial practice [28]. Our analysis indicates that the fixed scheduling granularity adopted by previous works [12, 21] is sub-optimal when the system load changes. Meanwhile, we perform a naive extension to the TVM's auto-scheduler [8], which lets us identify the best-performing code version under different interference levels. We show that the performance of the best code version under a specific interference level degrades quickly under a different interference level. These insights call for both adaptive scheduling and adaptive compilation for achieving the high-performance multi-tenant DL serving.

For the adaptive scheduling, we find that the sub-optimal performance of the fixed model-wise scheduling scheme is caused by the inefficient CPU resource utilization, while the fixed layer-wise scheduling scheme is caused by the frequent resource conflict. To reduce the resource conflict with CPU resource usage efficiency guaranteed under different situations, we propose a layer-block granularity scheduling strategy, which is finer than the model-wise scheduling but coarser than layer-wise scheduling. By setting a dynamic threshold, we can achieve both low conflict possibility and high CPU resource usage efficiency.

For the adaptive compilation, we analyze the relationship between the interference-prone code version and the interference-tolerant code version for a set of deep learning layers. We find that those different versions essentially lie in the Pareto frontier of trade-off space between parallelism and locality. Given this insight, we propose a single pass compiling strategy based on the existing auto-scheduler. The extended auto-scheduler is able to compile multiple versions of implementations that are suitable for different system interference pressure levels.

To evaluate our design, we choose various workloads from the industry-level MLPerf [48] benchmark ranging from light to heavy workload and compare with the existing multi-tenant DL serving solution, Planaria [21]. Compared to the existing work, our design serves more requests under the same QoS target in various scenarios (e.g., +71%, +62%, +45% for light, medium, and heavy workloads, respectively), and reduce the averaged query latency by 50%.

To summarize, we make the following contributions in this work.

- We analyze and identify the performance-critical optimization knobs for multi-tenant DL services, including the adaptive scheduling and the adaptive compilation (Sec. 3).
- We propose a static multi-version compiler that extends the existing TVM's compilation framework and can identify different optimal code versions under different interference levels. The key novelty in our compiler is a multi-version search algorithm in a single pass (Sec. 4.1).
- We propose a runtime scheduler design that dynamically forms a layer-block as the scheduling unit. The scheduler uses a dynamic threshold-based layer-block formation algorithm to balance the resource usage efficiency and scheduling conflict rate (Sec. 4.2).
- We evaluate and compare the proposed ideas in Veltair, where the combined adaptive compilation and scheduler can improve the system by 45% - 71% in different workload mixes. We also show that the query execution latency in our design is within 10% gap of the isolated execution case, meaning Veltair is close to the performance upper bound on the studied hardware platform (Sec. 5).

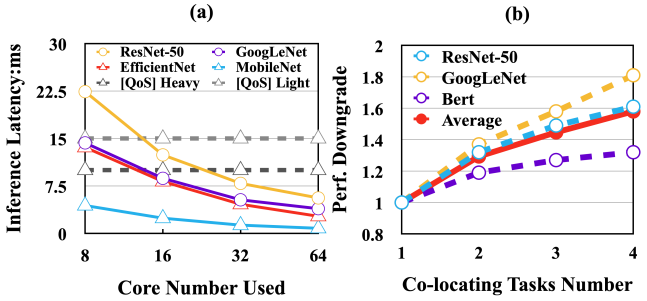

Figure 1: (a) All models in MLPerf vision can meet the QoS target by using a few cores. (b) Performance slowdown when simply co-locating multiple tasks together.

2 MOTIVATION AND CHALLENGES FOR MULTI-TENANT DNNS

In this section, we explain why the CPU architecture is suitable for providing the multi-tenant DL services. In specific, we show that the existing high-performance CPU is more than enough to serve multiple deep learning inference tasks under their QoS target. We then show that the quality of code generation is the key to fully unleashing the potential of the underlying hardware.

2.1 DNN Execution Characterization on CPU

With the tremendous improvement of hardware architecture design and manufacturing process, the performance of single computing hardware is increasing rapidly, making training and inference of deep neural networks easier and faster. Some companies even use CPUs as their deep learning back-end. These hardware mostly use multi-degree parallelism to increase overall throughput. However, when providing deep learning inference services with small batch sizes, these hardware will suffer from severe under-utilization since the deep learning inference service is not intense enough to fill the hardware resource.

As illustrated in Fig. 1a, a high-performance CPU (AMD Threadripper 3990X [1]) is more than enough to provide deep learning inference tasks. When serving vision tasks in MLPerf [48], the CPU platform can reach around 300 Query per Second by simply using all CPU cores for a task. So, to fully utilize the hardware and increase the energy efficiency, increasing deep learning service providers begin to introduce the task-level parallelism by sharing one computing hardware among multiple customers/requests which is called **multi-tenant deep learning service**, by either

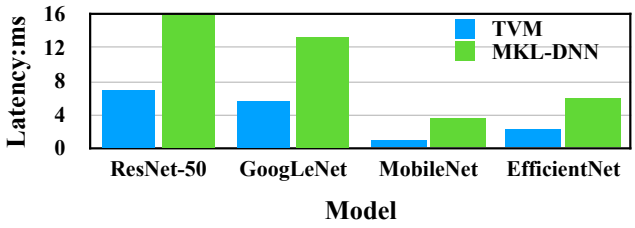

Figure 2: Performance comparison between vendor-supplied MKL-DNN library and TVM compiler.

temporal multiplexing (e.g. PREMA [12], AI-MT [2]) or spatial multiplexing (e.g. NVIDIA Multi-Process Service [44], NVIDIA Ampere Multi-Instance GPU [47]). By leveraging task-level parallelism, multiple customers/requests can fully occupy the throughput of the hardware, thus increasing the overall efficiency.

On the other hand, some deep learning tasks also consist of multiple sub-tasks. For example, auto-piloting on a smart vehicle consists of multiple direction object sensing and tracking tasks, SLAM tasks, decision-making tasks, etc.; personal voice assistant service on a home device consists of voice recognition, voice synthesis, etc. Those sub-tasks can but also should be launched in a parallel way for real-time interaction, and thus may share the resources on a single computing hardware. Currently, few deep learning systems are designed to face the multi-tenant serving scenarios. So in this work, we propose to explore the optimization opportunities at both compiling and runtime for co-locating and scheduling multiple tasks on a single hardware. Specifically, we focus on the problem of co-locating multiple latency-critical DL tasks on a multi-core architecture hardware, and the objective is to serve as many DL tasks as possible (i.e., maximize the metric of query per second) under the task latency constraints (i.e., ensuring that it finishes within a time limit). However, our design can be easily extended to support the co-location of DL tasks and best-effort tasks.

To co-locate multiple deep learning tasks on a single computing back-end, one naive approach is to simply dump all the candidate tasks onto the hardware and fill the empty slot once a task is complete. However, the most important challenge is how to manage the limited hardware resources like physical cores for the CPU architecture, streaming multiprocessors (SMs) for the GPU architecture, or even sub-arrays of a systolic architecture.

In addition to exclusive resources, various shared resources on the computing back-end are critical to the performance of a task, including cache bandwidth, cache capacity, memory bandwidth, etc. Naively scheduling all candidate tasks to the hardware would result in severe interference and performance loss due to the competition of these resources. We conduct a simple experiment that co-locates multiple ResNet-50, GoogLeNet, and SSD inference tasks on a single CPU. As illustrated in Fig. 1b, the task suffers from up to 1.8× latency under heavy workload pressure. As such, the inference can severely impact the QoS, but is considered by current DL serving systems. In contrast, our work considers both compilation and scheduling strategies to handle the interference.

Table 1: Optimization strategies in VELTAIR and prior works.

Multiplexing	Granularity	Compilation	Work
Temporal	Static (Model)	Static	PREMA [12]
	Static (Layer)		AI-MT [2]
Spatial	Static (Model)	Static	Planaria [21]
	Static (Model/Layer)		Parties [6]
	Static (Model/Layer)	Adaptive	Protean [35]
	Adaptive (Layer Block)	Adaptive	VELTAIR (ours)

2.2 DNN Compilation on CPU

Although vendor-provided libraries can offer optimized DNN computation with convenient APIs, an increasing number of researchers and developers have begun to use automatic high-performance code generators for even higher performance. Among deep learning compilers, TVM gains great success for its convenience, high quality of generated code, and cross-platform capability. Moreover, things are getting more convenient once the TVM Auto-Scheduler (i.e., Ansor [65]) is introduced. Now, researchers can simply define the computation logic they want, run the auto-scheduling procedure, and TVM would return the code with similar or even better performance compared to vendor-provided libraries, such as MKL-DNN [30] and MLAS [41] on CPU, cuDNN [46] on GPU.

The other advantage of using the DNN compiler is that the generated code is user-visible while vendor-supplied libraries are usually closed-sourced. Given those reasons, we choose the TVM compiler to generate the codes for running DNN models in this work. We also conduct a performance comparison experiment between the Intel MKL-DNN [30] and TVM. As Fig. 2 shows, the TVM generally outperforms the vendor-supplied library.

For compiling DNN layers or models on the CPU, we mainly consider the nested loop transformation and some CPU-specific annotation or pragma including parallelization and unrolling. The compiling procedure is actually a trade-off between the parallelism and locality of the program, which we will discuss later in the paper.

3 OPTIMIZATION SPACE ANALYSIS

In this section, we first identify the optimization space that is critical for achieving high-performance multi-tenant DL services. In specific, we study the two optimization knobs, namely the scheduling granularity and compilation strategy.

We characterize the impact of those two knobs on the performance measured by QoS satisfaction rate [4, 6, 51, 57] representing how many requests are finished within the QoS target of multi-tenant deep learning services on the CPU.

Our main finding is two-fold. First, a fixed scheduling granularity, such as the entire model [12] or the sub-layer block [2, 21], leads to the sub-optimal performance, owing to the diversity of DNN models and their distinctive inner characteristics. Second, the performance of the existing compilation strategies, which aim to maximize the code performance under the solo-run case, degrades significantly when multiple DNN models run together and interfere with each other. Such two findings motivate the design of the *adaptive scheduling* and *adaptive compilation* in VELTAIR.

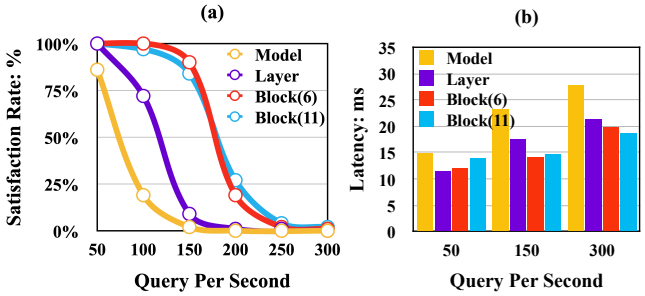

Figure 3: Performance comparison of different scheduling granularities under different query arrival rate. (a) QoS satisfaction rate. (b) Average query latency.

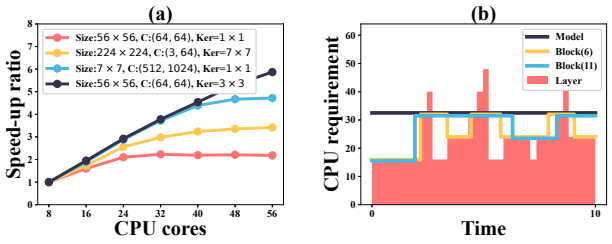

Figure 4: (a) Speedup trend of increasing core number for 4 different conv layers selected from the ResNet-50 model. (b) Core number allocation comparison for different scheduling granularities under the same QoS target. The layer-wise scheduling approach (red shadowed area) represents the minimum required core number for meeting the QoS target.

3.1 Optimization Space Definition

We first explain and define the optimization space of multi-tenant DL services. Conventional workloads such as Silo [55] and Moses [33] choose the entire query as the scheduling unit because the query has no internal structures. In contrast, DNNs are layer-based, for which the scheduling unit can range from one layer to the entire model. Meanwhile, DNNs are also computation-intensive, and their performances are sensitive to the code quality as shown in Sec. 2.2. In this work, we consider these two knobs jointly, i.e., scheduling granularity and compilation strategy, for achieving high-performance multi-tenant deep learning services.

Scheduling granularity refers to the size of the entity for allocating resources and scheduling on the hardware. For example, in the conventional online services, prior works typically choose the entire query as the scheduling unit [6, 34, 35, 39]. However, we have more choices on the scheduling granularity in deep learning services because DNN models have a complex inner organization consisting of *layers* or *operators*, such as `conv` (i.e., convolution), `relu` and `pooling`. As such, we can either choose an entire model (i.e., coarse-grained) or a single layer (i.e., fine-grained) as the scheduling unit. To achieve higher resource usage efficiency and reduce the resource usage conflict, we consider a new scheduling granularity of multiple layers as a unit, which we call *layer block*.

Compilation strategy refers to the code generation options for a DNN model or a DNN layer. For example, we have described the different code generation options (i.e., nested loop transformation) in Sec. 2.2. As we will show later, the optimal code generation option for minimizing the execution latency depends on the interference levels caused by co-running DNN models. To the best of our knowledge, we are the first to consider the compilation as an optimization knob in the multi-tenant DNN serving scenario.

Tbl. 1 compares the choices of those two optimization knobs in prior state-of-the-art solutions against Veltair. Specifically, our work is *adaptive* in both the scheduling granularity and compilation strategy. Previous work AI-MT [2] and Planaria [21] decompose a layer into multiple smaller parts, or sub-layers, for more flexible scheduling. However, the improvement is limited as we will show that the layer-wise scheduling unit is already inferior to our adaptive block scheduling in Sec. 3.2. In other words, sub-layer scheduling is overly fine-grained. Other work Protean [35] and Parties [6] mainly target conventional interactive services, so both of them use static scheduling. The static scheduling can either choose the layer or the entire model in the case of multi-tenant DL services. Note that Protean [35] also uses an adaptive compilation strategy, only targeting non-DL workloads. As such, it can not be applied to compile DNN models because they have a different set of compiler optimization options as we will show later. In the following parts of this section, we will justify the choice of our optimization knobs through detailed experimental results.

3.2 Scheduling Granularity Analysis

We first compare the performance of multi-tenant DL services under different scheduling granularities, including layer-wise, model-wise, and layer-block scheduling. We then explain why these static schedulings fail to fully utilize the hardware resources, which leads to the need for an adaptive scheduling granularity.

Experimental Setup. For the model-wise scheduling, we implement a simple First Come First Serve (FCFS) strategy used in prior work [25, 51]. In other words, the tasks will be served immediately if there are available resources while waiting otherwise. For the layer-wise scheduling, we implement an algorithm similar to Planaria [21] that allocates the resource to every layer and allow tile-wise preemption if the requested resources exceed the available number. For the layer block scheduling, we simply set the layer block-size to 6 and 11 respectively to study the impact of the block-size. We compare the performance of those scheduling schemes under different query arrival rates (i.e., query per second, QPS). We report the QoS satisfaction ratio in Fig. 3a and averaged model execution latency in Fig. 3b as evaluation metrics. For the fair comparison of different scheduling strategies, we run a total number of 30, 000 ResNet-50 models with identical uniform arriving times to eliminate the instability caused by the randomness.

Results. As shown in Fig. 3a, the performance of both model-wise and layer-wise scheduling degrade much faster than the block-wise scheduling. Meanwhile, the best block-size for optimal performance varies with the query arrival rate. For example, the block-size of 6 layers performs best at 150 QPS, while the block-size of 11 is better at 200 QPS. We have the same observations in Fig. 3b for the averaged query execution latency. These results confirm the criticality of scheduling granularity for multi-tenant DL services.

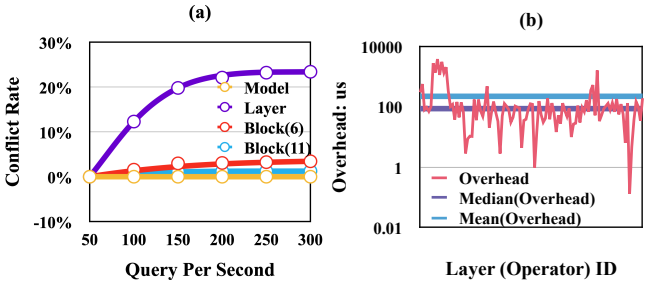

Figure 5: (a) Scheduling conflict rate comparison for different scheduling granularities under different query arrival rates. (b) Measured per-layer conflict scheduling overhead.

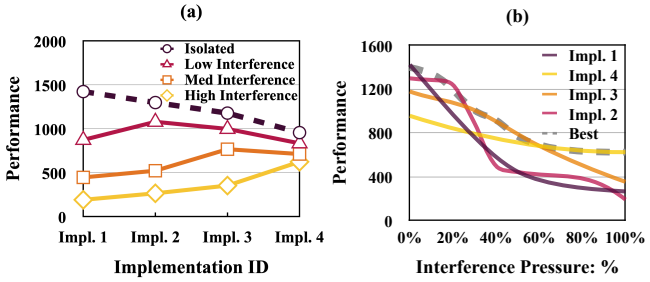

Figure 6: (a) Performance of four versions of the same layer under different interference levels. (b) The best performance (dotted line) is achieved by combining all four versions.

Model-Wise Inefficiency. We find that the distinctive computation resource requirement across DNN layers is the root cause for why the model-wise scheduling is sub-optimal. Fig. 4a plots the speedup for different ResNet layers under different CPU core numbers, which shows that different layers have different scalability trends when allocated core number increases. However, the model-wise scheduling evenly assigns a fixed number of cores to all layers in the model, which results in the CPU core resource wastage because many layers only require a small number of cores. Fig. 4b compares the core number allocation between the model-wise scheduling and layer-wise scheduling. Intuitively, the layer-wise scheduling scheme represents the minimum core allocation for satisfying the model's QoS target. We find that the model-wise scheme allocation (black line) is far from the optimal core allocation (red shadowed area). As a result, the QoS satisfaction ratio drops dramatically once the query arrival rate exceeds 50 QPS in Fig. 3a.

Layer-Wise Inefficiency. We find that the layer-wise scheduling is sub-optimal owing to the frequent *scheduling conflict* when the query arrival rate is high. For example, there are layers in Fig. 4b that require large core numbers (e.g., more than 48 out of 64 cores). The scheduling conflict occurs when a layer requests more cores than currently available cores. Fig. 5a compares the conflict rate among different scheduling granularities, where the layer-wise scheduling is highest (e.g., 23.8% conflict rate with 300 QPS).

For a layer that experiences scheduling conflict, we implement a technique to increase the resource utilization. In specific, we first let the layer use all the available cores and increase its core usage once more cores become available. However, using more cores needs to spawn more threads, whose overhead is non-negligible and worsens the model's overall latency. To illustrate this point, we quantify this overhead for each layer in ResNet-50 by measuring a layer's latency with and without scheduling conflict. Fig. 5b shows the results, with the mean of 220 μs and median of 100 μs.

The scheduling conflict overhead measured above explains the overall latency of the layer-wise scheduling for ResNet-50 at the 300 QPS. The execution latency without scheduling conflict is 18.54 *ms*. But with the conflict rate of 23.8%, the total conflict overhead is estimated to be $23.8\% \times 55 \times 220\ \mu s = 2.86\ ms$, with the 55 layers (53 conv and 2 GEMM) in ResNet-50. As such, the estimated overall latency is $2.86 + 18.54 = 21.4ms$, which matches the measured latency for the layer-wise scheduling at 300 QPS in Fig. 3b.

Summary. The above results show that the optimal scheduling scheme should strike a good balance between the averaged resource usage and the scheduling conflict. The model-wise scheduling generates the smooth resource usage pattern and hence low conflict, but uses unnecessarily more resources to meet the QoS target. In contrast, layer-wise scheduling uses the minimum CPU resources, but the per-layer characteristics lead to a substantial scheduling conflict overhead. The layer block scheduling combines the advantages of both model-wise and layer-wise scheduling. Furthermore, the optimal performance cannot be achieved by simply setting a fixed layer block size as demonstrated in Fig. 3. In other words, the optimal block organization depends on the model characteristics and query arrival rate. As such, we propose to use adaptive layer block-size, and will explain it with greater details in later sections.

3.3 Compilation Strategy Analysis

We now perform a set of experiments to study the impact of compilation strategies on multi-tenant deep learning services. The key insight in our experiments is that the optimal compilation strategy changes under different interference levels. As such, the adaptive compilation is needed to achieve high-performance multi-tenant DL services. Furthermore, we propose to use multi-version static compilation to avoid the overhead of just-in-time (JIT) compilation.

Extending TVM Auto-Scheduler. Recall that in Sec. 2.2, the current TVM compilation strategy uses an auto-scheduler [65] to search for the implementation that achieves the best or the lowest

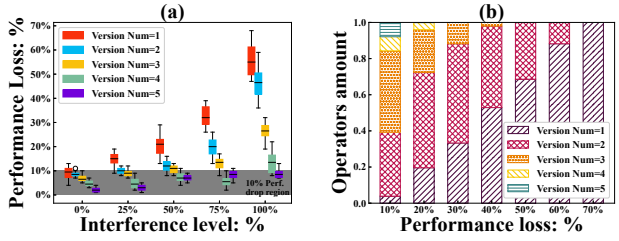

Figure 7: (a) Performance loss of retaining different numbers of versions compared against retaining all ten versions under different interference levels. (b) Distribution of code version count to maintain various performance loss.

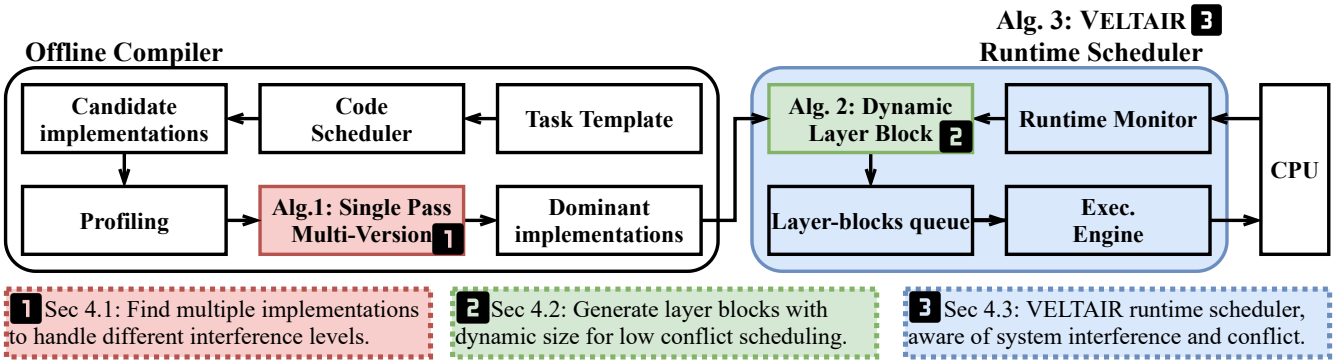

Figure 8: Overview of VELTAIR, which comprises of the offline static compiler and the online runtime scheduler for adaptive compiling and scheduling. The static compiler and runtime scheduler leverage the single-pass multi-version search and dynamic threshold-based layer-block formation algorithm, respectively. By monitoring the system load and interference pressures, the scheduler adaptively selects the optimal code version and scheduling granularities.

latency. This compilation strategy does not consider the existence of interference when multiple DNN models run together, which can lead to a significant performance slowdown as shown in Fig. 1b.

To mitigate the impact of interference, we propose a naive extension for the TVM's existing auto-scheduler [65]. To identify the best code implementation for the target layer at a given interference level, we launch a background layer that produces the desired level of interference and run the TVM's auto-scheduler with long enough iterations (e.g., 1024 iterations). As such, the returned schedule can be regarded as the optimal version under this interference level. In this experiment, we use a frequently occurred ResNet `conv` layer with the feature map size of 14×14, kernel size of 3×3, input and output channel size of 256, and study the performance of different compilation strategies under different interference levels.

Results. Fig. 6 compares the performance of four different implementations under different interference levels. In specific, the four implementations correspond to the optimal ones searched with zero, low, medium, and high interference levels, respectively. As Fig. 6a shows, the `impl.-1`, which is also the default choice of TVM auto-scheduler, achieves the best performance when no interference exists. However, its performance also degrades rapidly, which can be up to 7× at the high inference level. In contrast, the `impl.-4` has the lowest performance when no inference exists but achieves the highest performance under the high interference. These results show that the optimal code implementations vary according to the interference levels, and our simple extension to the TVM auto-scheduler can effectively find these optimal implementations.

Since a model may experience all ranges of interferences in the multi-tenant DL services at one run, a static code version cannot achieve the best performance. Fig. 6b further quantifies the performance trend of the above four versions against different interference levels, where each version outperforms others only within a narrow interference interval. As such, we have to combine all the four versions across all the interference levels to achieve the best performance, which is the dotted grey line in Fig. 6.

General Cases. We further profile the rest of the ResNet-50 layers under different interference levels to fully understand the impact of the compilation strategy. Specifically, we choose ten interference levels and identify the best-performing version at each level, which leads to a total number of ten implementation versions for each layer. Fig. 7a compares the performance loss of using a various number of versions against using all the ten versions. If we use only one implementation, the performance loss increases as the interference level increases. In contrast, using five versions out of the ten versions can maintain the performance loss within 10%.

Multi-Version Static Compilation. One naive way to exploit the above insights for multi-tenant DL services is to perform a Just-in-Time (JIT) compilation according to the interference level. However, the JIT compilation overhead can offset the benefit of adaptive compilation. Instead, we propose to use the static multi-version compilation to achieve the same benefit of the adaptive JIT compilation. Fig. 7b plots the ratio of code version count to maintain various performance losses compared to the case of using all the ten versions. Although the above results have shown that it requires five code versions to stay within 10% performance loss, the majority (i.e., over 80%) of layers only require three code versions.

4 DETAILED DESIGN OF VELTAIR

In this work, we propose VELTAIR, a software solution for high-performance multi-tenant deep learning model serving. Based on the previous insights, VELTAIR performs adaptive compiling and scheduling. Fig. 8 shows its overview that has two main components, i.e., the offline static compiler and the online runtime scheduler.

Instead of performing dynamic compilation whose overhead can account for the model serving latency, we propose to use a static multi-version compiler that extends the existing TVM's compilation framework and can identify different optimal code versioned under different interference levels. The key novelty in our compiler is a single-pass multi-version search algorithm (described in Sec. 4.1).

The VELTAIR runtime scheduler dynamically forms the layer block as the scheduling unit, which balances the core usage efficiency and scheduling conflict rate. The key in the scheduler is a dynamic threshold-based layer block formation algorithm that we describe in Sec. 4.2. The runtime scheduler exploits a performance

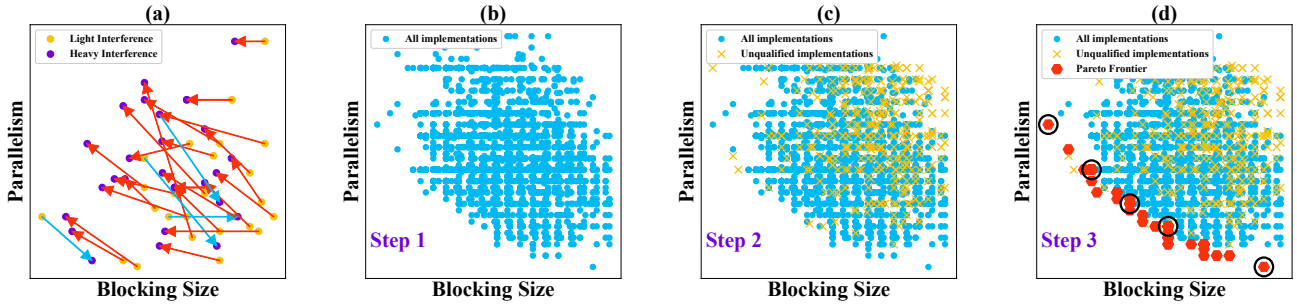

Figure 9: (a) The heavy-inference-optimal version generally prefers a large parallelism and a small blocking size (low locality), while the light-inference-optimal version prefers the opposite. (b-d) Steps to find optimal code version under different interference levels. We use an exemplary conv layer of $H_{in} = W_{in} = 7$, $C_{in} = 832$, $C_{out} = 384$, $H_K = W_K = 1$, $H_{out} = W_{out} = 7$.

counter-based interference proxy. By monitoring the system load and interference pressures, it adaptively selects the optimal code version and scheduling granularities, which are detailed in Sec. 4.3.

4.1 Single-Pass Static Multi-Version Compiler

In Sec. 3.3, we have described the benefits of adaptive compilation for handling the interference in the multi-tenant DL services. We have proposed a naive extension for the TVM auto-scheduler to search for the best code version at a given interference level. The extended auto-scheduler launches an additional background layer that can generate the desired interference level during the search process. This approach is effective for identifying the best code version at different interference levels but is time-consuming as it requires multiple passes of the TVM auto-scheduler. A single pass for a layer is typically 20 minutes on our high-end CPU, which means searching for five versions would take close to two hours.

To facilitate the multi-version compilation process, we propose a single-pass search algorithm that adds almost no overhead to the original TVM auto-scheduler. Our key insight is that we can use the well-known computer architecture tradeoff between parallelism and locality to explain why certain versions are extremely sensitive to interference while others are much less sensitive. Built upon this insight, we can explore the parallelism-locality tradeoff space in a single search pass, from which we then pick the desired versions.

Parallelism-Locality Tradeoff. We first use experimental results to illustrate that the finding of different optimal versions under different interference levels is essentially a tradeoff between program parallelism and locality. In this experiment, we use the straightforward extension described in Sec. 3.3 to search for the two optimal code versions, one under the light interference level and the other one under the heavy interference level. We then record the corresponding complication flags for these two versions. Based on the recorded flags, we compute a parallelism metric by simply multiplying the loop unrolling factor and parallelization factor. We compute a locality metric by directly using the tiling/blocking size.

In Fig. 9a, we compare the above two metrics of the two code versions that achieve the best performance under the light inference and heavy inference, respectively. We observe that the heavy-inference-optimal version generally prefers high parallelism and a small blocking size, while the light-inference-optimal version prefers the opposite. We then derive the following insight: generated codes with a higher locality (a larger blocking size) perform better under the light interference (***interference-vulnerable***), while generated codes with a higher parallelism perform better under the heavy interference (***interference-tolerant***).

The above insight reflects the well-understood parallelism-locality tradeoff. To exploit the large locality, a layer needs to use more on-chip memory like the LLC in CPU, which are shared resources among multiple CPU cores. However, the performance of the layer quickly degrades when there is contention on the shared resources (L3 cache and the corresponding bandwidth according to our observation). To mitigate the impact of contention, the layer can limit its locality and use more parallelism to remedy its performance loss.

Single-Pass Compilation. We now use the examples in Fig. 9b-d to walk through our single-pass compilation algorithm, which has three steps. The details of the algorithm are provided in Agl. 1.

Algorithm 1 Static multi-version compilation in a single pass.

Input: $layers[N]$, qos
Output: $candidate\_impls[N][V]$, $dominant\_impls[N][]$

function FindingImpl($layers$, qos)
  for $\_l$ **in** $layers[N]$ **do**
    $\_l.qos \leftarrow qos \times \frac{\_l.op\_count}{\sum_{x \in layers[1:N]}(x.op\_count)}$
    $impls[] \leftarrow Ansor(\_l, 1024)$
    $impls[] \leftarrow [x.time \leq \_l.qos \ for \ x \ in \ impls]$
    $d\_impl[] \leftarrow ExtractDominant(impls)$
    $d\_impl[].sort(key = x.block\_size)$
    for $\_i$ **in** $d\_impl, step \leftarrow \frac{d\_impl.length}{V}$ **do**
      $c\_impl.push\_back(\_i)$
    end for
    $candidate\_impls.push\_back(c\_impl)$
    $dominant\_impls.push\_back(d\_impl)$
  end for
  return $candidate\_impls$, $dominant\_impls$
end function

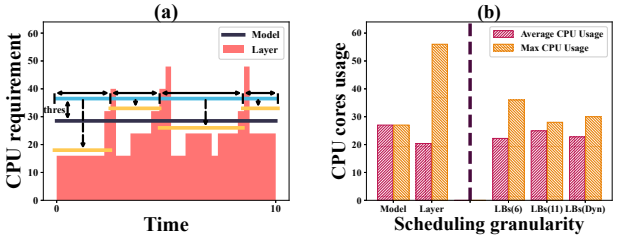

Figure 10: (a) Forming the layer-blocks by a threshold and minimize the layer-blocks' CPU usage. (b) Average and maximal CPU usage of various scheduling granularity.

The first step (Line 2 - 4 in Agl. 1) directly leverages the TVM's auto-scheduler to collect candidate implementations. In this step, we enable the operator fusion optimization in the auto-scheduler, which includes common fusion patterns like convolution followed by ReLU (`conv-relu`) and convolution followed by batch normalization and ReLU (`conv-batchnorm-relu`). Instead of searching for the best-performing implementation, we record as many samples as possible and calculate their parallelism and locality metrics as Fig. 9b shows. In the second step (Line 5), we then filter out samples whose performance can not satisfy this layer's QoS target as Fig. 9c shows. We set the layer's performance as the minimal floating-point operation per second (i.e., FLOPS) that the corresponding model needs to achieve to meet the model's latency target.

In the third step (Line 6 to 7 and Line 14 to 29), we select the *dominant* implementations via *ExtractDominant* function, where there are no other implementations with both smaller blocking size and parallelism than each chosen one. In other words, these dominant implementations form the Pareto frontier (red markers in Fig. 9d), which is an optimal solution to the multi-objective optimization problem. In the last step (Line 8 to 12), we uniformly choose five versions from the Pareto frontier (circled ones in Fig. 9d). Since not all layers require five versions to maintain the close performance to the optimal, we test the performance of the selected five versions under different interference levels and remove the ones whose performance is within 90% of the full five versions. This optimization leads to the reduced storage overhead of code multi-versioning.

4.2 Dynamic Threshold Based Layer-Block Formation

As previously explained in Sec. 3.2, the layer-block-based scheduling outperforms the layer-wise and model-wise scheduling through balancing the minimal average core usage and scheduling conflict rate. However, a fixed-sized layer-block is not efficient because the optimal block size varies with the system load and the interference from other co-executed models. As such, we propose a *dynamic-sized* layer-block approach to achieve the high core efficiency and low conflict rate according to the system load and interference level.

To reduce the scheduling conflict rate via the layer-block scheduling, we first identify the layers that are most likely to trigger conflicts. To identify these conflict-prone layers, we calculate the required CPU core number for each layer to complete within its QoS target, from which we can compute the model's averaged core number. We compare the layer-wise core number against the model-wise average value, and identify the layers with a much higher CPU core number requirement than the averaged value as conflict-prone layers. For each conflict-prone layer, we form a layer-block that can reduce its core usage by increasing the core usage for other layers in the block while still satisfying the QoS target.

We walk through the ResNet example in Fig. 10a to illustrate the intuition of our method. We first form the layer-wise (red shadowed area) and model-wise (black horizontal line) scheduling plan in Fig. 10a. We denote the average core count in the mode-wise scheduling as *Avg_C*. We then use a runtime-decided threshold *thres* that ranges from zero to maximal core count. We iterate over all layers and identify the conflict-prone ones whose core requirement exceeds *Avg_C+thres*, i.e., the blue line in Fig. 10a. We refer to each conflict-prone layer to the splitting pivot, which is essentially the beginning layer for a block. As a result, there are four blocks marked by arrows. For each formed block, we calculate its QoS target by summing up all its layers. We then recalculate the core requirement (yellow line) of each block to satisfy its QoS target.

Agl. 2 formally describes the above dynamic threshold-based layer-block formation algorithm. where the threshold is determined at runtime according to the system load and co-executed models' characteristics. Sec. 4.3 will provide the details of how we adjust the threshold. With Agl. 2, we can generate proper layer-blocks using no more than *Avg_C+thres* CPU cores under different system loads. The basic idea is that when the system load is low, we use a high threshold since the conflict possibility is low, which means each layer can use as many cores as possible for maximizing the CPU resource usage efficiency. When the system load is high, we use

Algorithm 2 Dynamic threshold based layer-block formation algorithm.

Input: $layers[N]$, $impls$, $thres$
Output: $Layer\_Block$

```
1:  function FindingFirstPivot(layers, impls, thres)
2:      splitting_pivot ← 0
3:      Avg_C ← Core_@ Model Granularity(layers)
4:      for _l in layers[1 : N] do
5:          if Core(impls[_l]) ≥ thres + Avg_C then
6:              splitting_pivot ← _l
7:              break
8:          end if
9:      end for
10:     return splitting_pivot
11: end function
12: function FormingLBs(layers)
13:     Layer_Block ← [], begin ← 0
14:     while layers.length() ≠ 0 do
15:         sp ← Finding1stPivot(layers, impls, thres)
16:         Layer_Block.push_back(layers[begin : sp])
17:         layers ← layers[sp + 1 :]
18:         begin ← sp + 1
19:     end while
20:     return Layer_Blocks
21: end function
```

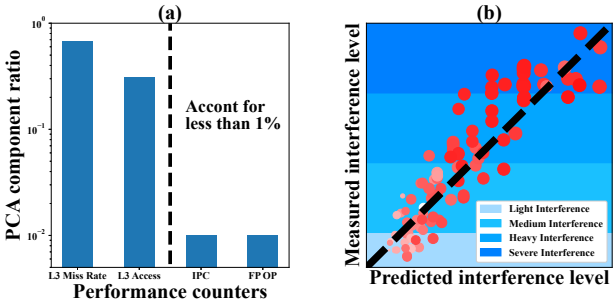

Figure 11: (a) Dominant components of performance counters according to PCA. (b) Accuracy validation of the interference pressure proxy using L3 miss rate and access counters.

a low threshold which means each layer should have core counts close to the average value for reducing the scheduling conflict rate.

The essence of Agl. 2 is to reduce the high core usage of error-prone layers and remedy its latency loss by increasing the core usage of other layers. This can increase average core usage compared to the most efficient layer-wise scheduling. However, according to our evaluation, the gap between optimal CPU resource usage is smaller than 10%, which is acceptable. Fig. 10b compares the average core usage and maximum core usage of different scheduling granularities when co-locating two ResNet-50 models. Our algorithm is effective at reducing the gap between optimal CPU cores usage (i.e., achieving high resource efficiency) and maximum core usage (i.e., reducing the scheduling conflict rate).

4.3 VELTAIR Runtime Scheduler

In this subsection, we describe the details of the runtime scheduler in VELTAIR. Our scheduler monitors the current CPU inference level and dynamically chooses the code version derived from Sec. 4.1 and scheduling granularity using the algorithm described in Sec. 4.2. In specific, we explain how we derive the system interference pressure level and how we determine the dynamic threshold for Agl. 2.

Interference Proxy. We first build the proxy for monitoring the system interference pressure level by using hardware performance counters. According to previous studies [35, 63, 64], performance counters have a strong relation with interference pressure level. We define the interference pressure level of the system as the average performance slowdown ratio of layers running on the system. To figure out what performance counters decide the interference level, we conduct a principal component analysis (PCA) [17] on collected performance counters, including L3 cache miss Rate, L3 access, instruction per cycle (IPC), float-point operations, etc. It turns out that L3 cache-related counters account for over 99% of the data variance, as shown in Fig. 11a. As such, we choose the L3 miss rate and L3 access to construct a simple linear interference model. As shown in Fig. 11b, the predicted interference level matches the measured interference level well. Using this simple linear model, we can derive the interference pressure level with low cost at runtime.

Dynamic Scheduling Threshold. The threshold used by the layer-block formation in Sec. 4.2 indicates the additional core counts that each layer block can use beyond the model's averaged requirement. When a model runs exclusively, it can use as many cores as it desires. However, when multiple models run concurrently, each model should try to reduce its core usage to avoid scheduling conflicts. As such, we use a simple heuristic that determines the threshold by subtracting the total core number by the sum of all models' average core count and distributing the remaining cores according to each model's average core count. For example, three models A, B, C use 12, 12, 24 CPU cores on average respectively. On average, $64 - (12 + 12 + 24) = 16$ cores are idle, and we assign 4, 4, 8 as threshold to A, B, C respectively. In our study, we observe that a model with a high average core usage typically has a high peak core usage. Thus, dividing the idle cores by the model's average core usage can better fit each model's computation demand.

Putting All Together. We now describe the runtime scheduler in VELTAIR that exploits the aforementioned algorithms. Agl. 3 illustrates the pseudo-code of the scheduler, where the task dispatcher simply sends tasks to a worker if there are enough idle cores. The code implementation search is done offline at the compiling stage as Agl. 1 details. At runtime, VELTAIR collects the performance counters once a layer block is finished, and the scheduler will form the next layer block from the remaining layers according to the current system load with different implementations according to the current interference pressure level. As mentioned before, codef implementations with different interference tolerance levels have significant differences in parallelism and locality, which means the same layer in a model will have different CPU requirements under different interference pressure levels leading to different layer blocks. Note that when selecting the interference tolerance level of the next layer-block, we will ignore the ongoing but soon-to-finish layer-blocks, since they will have little influence on system interference from now on. To determine the soon-to-finish layer-block, we examine whether its remaining execution latency is within a

Algorithm 3 The details of VELTAIR scheduler.

```
function VeltairTaskDispatcher
    Dispatch tasks following Poisson distribution
end function
function VeltairWorker
    while true do
        if worker is busy then
            Wait for last task to finish
        end if
        t ← fetch_task(), begin ← 0
        while t.finished ≠ True do
            i ← system interference
            thres ← #C_Total − Σ_{t_active}(#C_Model Granularity(t))
            pivot ← Finding1stPivot(t, impls_i, thres)
            t[begin : pivot].Execute()
            t ← t[pivot + 1 :]
            begin ← pivot + 1
        end while
    end while
end function
```

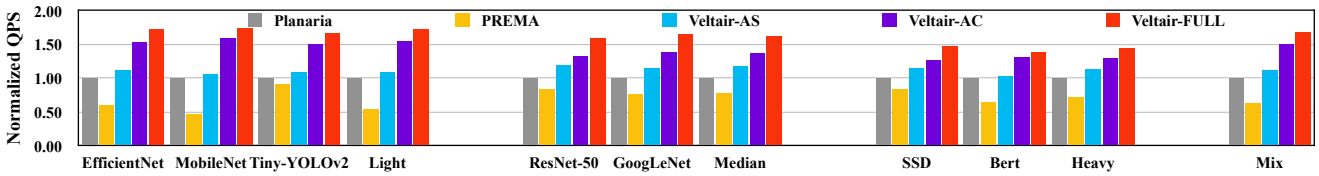

Figure 12: Query per second (QPS) with 95% tasks QoS satisfied for various workloads and scheduling strategies.

threshold (e.g., 10%) according to our offline profile-based latency model and interference proxy model.

5 EVALUATION

We now demonstrate the effectiveness of the adaptive compiling and scheduling in Veltair. We first describe our evaluation setup, baseline, and metrics. We compare the performance of Veltair against other work and show that the combination of our adaptive compiling and scheduling is essential for achieving the hight-performance in multi-tenant DL services.

5.1 Experimental Setup

Multi-Tenant Deep Learning Models. To simulate the realistic situation of deep learning services, we use deep learning models from MLPerf (Server) [48] as listed in Tbl. 2. The evaluated models include image classification, object detection, and neural machine translation (NMT) tasks. We categorize the workload of the models from light, medium to heavy, and set the QoS target for them according to the guidance of MLPerf.

Workload Generation. We also follow the MLPerf guidance to generate random queries with Poisson distribution, where the λ parameter of the distribution stands for the QPS (query per second) of the workload. We evaluate our design under *Light*, *Medium*, *Heavy*, and *Mix* workload. For the mixed workload, the frequency of every task is set to be inversely proportional to QoS requirements [42].

Hardware and Software. For all experiments, we use a machine equipped with a high-end server-level CPU Ryzen Threadripper 3990X [1] and 256 GB DDR4 RAM at 3200 MHz. The CPU has 64 physical cores and 256MB L3 cache capacity, and works at 2.9 GHz with AVX-2 enabled. To obtain stable experimental results, we turn off certain features such as simultaneous multi-threading (SMT) and dynamic voltage and frequency scaling (DVFS). We believe that turning off these features do not change our insights owing to the following reasons. The SMT mainly enhances the sharing of L1 cache, while we identify LLC as the main contentious resource. However, SMT leads to a significant latency fluctuation because of the possibility that two logical threads of different tasks are assigned to the same physical core. The DVFS also leads to latency fluctuation that increases the conflict rate. We implement the static multi-version compiler by extending the TVM v0.8 [8]. For the runtime scheduler implementation, we use MPICH 3.3.2 [16], which serves the multi-tenant DNN models via multi-processing. The OS is Ubuntu 20.04 on Windows Subsystem for Linux (WSL 2).

Table 2: Evaluated multi-tenant DL models.

Category	Workload	Name	QoS (ms)
Image Classification	Medium	ResNet-50 [29]	15
	Medium	GoogLeNet [52]	15
	Light	EfficientNet [53]	10
	Light	MobileNet-V2 [50]	10
Object Detection	Heavy	SSD [36]	100
	Light	Tiny-YOLOV2 [49]	10
NMT	Heavy	Bert-Large [15]	130

Evaluation Metrics. We use QPS with 95% tasks QoS satisfied, average latency, and CPU usage efficiency as our evaluation metrics.

- QPS with 95% Tasks QoS Satisfied: this metric represents how many requests the system can serve per second with almost all the query requests (95%) finish within the QoS target.
- Average Latency: This metric measures the average execution latency of all the queries.
- CPU Usage Efficiency: This metric measures the average CPU usage of the tasks by dividing the total execution time by the sum of multiplying of the core usage and execution time of each layer.

Baseline Choice. Since we co-locate multiple DNN models and let them spatially share the hardware, we choose Planaria [21] as the baseline in our evaluation. It should be noted that Planaria is based on the hardware-software co-design, while we port the software scheduling part to the CPU platform. To justify why we only consider the spatial multitasking scenario, we also implement another baseline scheduling method PREMA [12], which is a temporal multitasking algorithm and lets tasks with high priority preempt.

Evaluation Plan. We study the effectiveness of different components by evaluating the following configurations of Veltair.

- Veltair-AS: with only adaptive scheduling.
- Veltair-AC: with only adaptive compilation.
- Veltair-FULL: with both adaptive scheduling and adaptive compilation enabled.

5.2 Query per Second (QPS) Improvement

Fig. 12 demonstrates the QPS improvement of Veltair against the baseline Planaria [21] for studied models in different levels of workloads. Veltair-FULL achieves an average of 71%, 62%, 44%

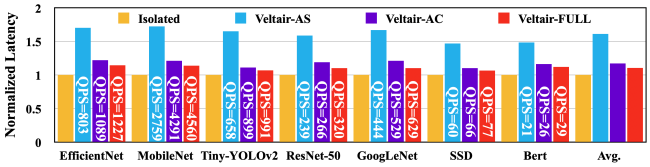

Figure 13: Average query execution latency comparison between solo-run (isolated) and various VELTAIR configurations (VELTAIR-AS, VELTAIR-AC, VELTAIR-FULL).

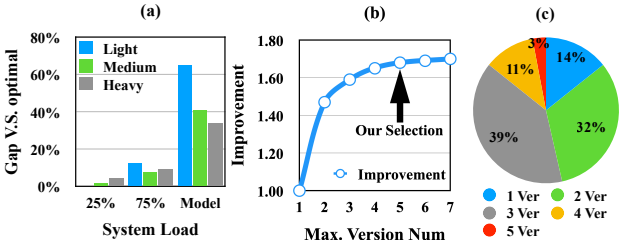

Figure 14: (a) Gap between optimal core usage of layer block-based scheduling strategy under different system load, comparing to the fine-grained layer-wise scheduling. (b) Improvement under different version number. (c) The ratio of number of used versions for all the layers in seven DNN models.

improvement in the light, medium, heavy workloads respectively, and an average of 68% improvement in the mix workloads.

We also observe that the adaptive compilation (VELTAIR-AC) achieves better improvements than the adaptive scheduling (VELTAIR-AS). However, these two techniques are synergistic, and both are critical components for fulfilling the performance improvement of the full version of our design (VELTAIR-FULL). Without the adaptive scheduling, VELTAIR-AC only achieves 50% QPS improvement in contrast to the 68% improvement of VELTAIR-FULL in the mix workloads. The reason is that without adaptive scheduling, many layers will choose implementations with lower locality but higher parallelism to handle interference, leading to increased CPU requirement and thus increased conflict possibility. In contrast, the dynamic layer block formation in adaptive scheduling can mitigate these conflicts.

In Fig. 12, we also observe that the temporal multitasking-based multi-DNN serving scheme (PREMA [12]) generally performs worse than the spatial multitasking-based multi-DNN serving. This observation justifies the choice of spatial multitasking of our work.

5.3 Query Execution Latency Result

We compare the average query latency of various VELTAIR configurations against the solo-run case in Fig. 13. For each model, the latency is measured at the QPS where 95% of queries can meet their QoS target (i.e., same to the QPS metric in Fig. 12). Since the solo-run latency is the shortest latency each model can achieve on the studied CPU platform, this comparison lets us identify how much additional room there is for further optimization in VELTAIR.

Fig. 13 shows that the inference latency of VELTAIR-AS is 1.6× of the isolated solo-run execution, which means the adaptive scheduling cannot reduce the execution latency. On the other hand, the latency of VELTAIR-AC is 1.17× of the isolated execution, confirming its ability for reducing latency under interference. With both adaptive scheduling and compilation, the average latency is only 1.1× of the isolated execution, which means that VELTAIR-FULL is close enough to the optimal serving result on the studied platform.

5.4 Result of CPU Efficiency

In VELTAIR, the layer-block-based scheduling leads to smoother CPU core usage with reduced conflict rate but potentially uses more cores. We quantify the gap of average core usage between VELTAIR scheduling and the layer-wise scheduling. Recall that the fine-grained layer-wise scheduling indicates the minimal core usage. Fig. 14 shows that even under 75% system load, the core usage gap of VELTAIR is less than 10% compared to the minimal core usage of the layer-wise scheduling. In contrast, the model-wise has a much larger gap of 47%. These results confirm that our layer-block-based scheduling strikes a balance between reducing the scheduling conflict rate and maintaining the high resource usage.

5.5 Sensitivity and Overhead Analysis

Sensitivity. In VELTAIR, we empirically set the maximal version number V to 5. We now study the performance improvement under different V in Fig. 14b, which shows the improvement saturates after four versions. Fig. 14c plots the version count distribution for different layers, which shows that only 3% layers require five versions. These results justify the choice of using five versions.

Scheduling Overhead. The scheduling overhead of VELTAIR mainly consists of two parts. The first part is the runtime layer block formation procedure, which scans the layers only once and has the complexity of $O(N)$. The second part comes from the linear-model-based interference proxy. Owing to the low complexity of the scheduling algorithm and the proxy model, we find that their overall overhead is less than 0.1 *ms* for serving each DNN model.

6 RELATED WORKS

We compare and contrast VELTAIR with previous works in the following three aspects, including the task co-location, multi-tenant deep learning services, and deep learning compiler.

6.1 Task Co-location in Datacenter

The rapid improvement of hardware computation power makes it possible to share the hardware among multiple tasks for higher throughput. Many works have studied how to co-locate a latency-critical (LC) task with multiple best-effort tasks [4, 5, 39]. Parties [6] proposes a resource-partitioning technique to co-locate multiple LC services. For more intelligent resource partition and management, CuttleSys [34] proposes a sampling-reconstruction-prediction-based strategy with reconfigurable architecture. Bubble-series [40, 60] proposed an online contention measurement and control system to relax the performance loss caused by contention and is free from the online compiler, execution checkpoint, code variants rerouting, etc. While previous works mainly focus on resource partition, isolation, and management, Protean [35] and other

works [54] extend the optimization space by introducing runtime code transformation for lower L3 cache pollution.

6.2 Multi-Tenant Deep Learning Service

Different from conventional workload, deep learning services are computation-intensive with complex inner structures and should be specifically treated when co-locating them. DART [59] proposes a pipeline-based method to co-locate multiple DNN workloads on multiple heterogeneous computation nodes. While in this work, we mainly consider co-locating multiple DNN tasks on one homogenous hardware. PREMA [12] and AI-MT [2] propose temporal multiplexing architectures with preemption-based strategy and computation-memory overlapping-based strategy, respectively. In contrast, Planaria [21] proposes a spatially decomposable systolic architecture to co-locate tasks with proper computation and memory resources. This work aims at relaxing the problem from compiling and scheduling aspect with less constraint on the back-end hardware as long as it is programmable. In addition to hardware-software co-design, DyNet [43] mainly handles the problem of scheduling RNNs. LazyBatch [11] proposes a batch-based approach to handle multiple DNN requests. Ebird [13] also proposes a batch-based approach to enable concurrent execution of DNNs with high data transfer-compute overlapping. Abacus [14] proposes an operator overlapping strategy based on precise latency prediction. Besides the multi-DNN serving scenario, emerging microservice-based workloads also have complex inner structures similar to DNN models [20], to which our design may also be applied.

6.3 Deep Learning Compiler

For the better flexibility and performance of DNN model execution, recent researchers propose various DL compilers including TVM [8], TensorComprehensions [56], Tiramisu [3], TensorFlow-XLA [22]. These DL compilers are often integrated with front-end optimizers like TASO [31] or Grappler [23]. Meanwhile, they also introduce domain-specific language to make it convenient for users to define their own computation. For the code generation optimization with a huge search space, researchers apply both machine-learning-based methods including AutoTVM [9], Ansor [65], FlexTensor [66] and heuristic based methods including DLFusion [37] and Paleozoic [38]. These compilers target general-purpose hardware or DL accelerators, and generally outperform vendors-provided libraries. However, these works mainly focus on optimizing the performance of the stand-alone execution of DNN operators. In contrast, we explore compilation optimization for co-locating multiple deep learning tasks, for which we show the interference-aware compilation is critical.

7 CONCLUSION

In this work, we proposed Veltair, a compiler-scheduler system for high performance multi-tenant deep learning service. By leveraging multi-version compiling and layer-block scheduling, we achieve 1.7× system maximal QPS and reduce 50% of the computation latency with little overhead. We first evaluate the proper scheduling granularity in deep learning tasks, and we propose a layer-block scheduling strategy with dynamically adjustable size to reduce the resource conflict. Then we study the compilation options and propose a single-pass multi-version compilation to handle the performance loss of interference caused by shared resource competition in multiple neural networks co-locating. We demonstrate the advantages of Veltair in the aspects of improvement in QPS, QoS satisfaction rate, computation latency, and resource usage efficiency using the standard MLPerf Server test suite.

ACKNOWLEDGEMENT

This work was supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (NSFC) grant (U21B2017, 62072297, 61832006). We thank the anonymous reviewers and our shepherd Prof. Xipeng Shen for their constructive feedback for improving the work. We also thank Zhanda Zhu, Zihan Liu, Yijia Diao, and Vega Jiang for the beneficial discussion and continuous support.

REFERENCES

[1] AMD. 2020. Ryzen Threadripper 3990X Processor. https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3990x.

[2] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. 2020. A Multi-Neural Network Acceleration Architecture. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 940–953. https://doi.org/10.1109/ISCA45697.2020.00081

[3] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. https://doi.org/10.1109/CGO.2019.8661197

[4] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. 2017. Prophet: Precise QoS Prediction on Non-Preemptive Accelerators to Improve Utilization in Warehouse-Scale Computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 17–32. https://doi.org/10.1145/3037697.3037700

[5] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. 2016. Baymax: QoS Awareness and Increased Utilization for Non-Preemptive Accelerators in Warehouse Scale Computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*. ACM, 681–696. https://doi.org/10.1145/2872362.2872368

[6] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3297858.3304005

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*. ACM, 269–284. https://doi.org/10.1145/2541940.2541967

[8] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. 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 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 578–594. https://doi.org/10.5555/3291168.3291211

[9] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31*. 3393–3404. https://doi.org/10.5555/3327144.3327258

[10] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, June 18-22, 2016*. IEEE Computer Society, 367–379. https://doi.org/10.1109/ISCA.2016.40

[11] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. 2021. Lazy Batching: An SLA-aware Batching System for Cloud Machine Learning Inference. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021*. IEEE, 493–506. https://doi.org/10.1109/HPCA51647.2021.00049

[12] Yujeong Choi and Minsoo Rhu. 2020. PREMA: A Predictive Multi-Task Scheduling Algorithm For Preemptible Neural Processing Units. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2020, San Diego, CA, USA, February 22-26, 2020*. IEEE, 220–233. https://doi.org/10.1109/HPCA47549.2020.00027

[13] Weihao Cui, Mengze Wei, Quan Chen, Xiaoxin Tang, Jingwen Leng, Li Li, and Mingyi Guo. 2019. Ebird: Elastic Batch for Improving Responsiveness and Throughput of Deep Learning Services. In *37th IEEE International Conference on Computer Design, ICCD 2019, Abu Dhabi, United Arab Emirates, November 17-20, 2019*. IEEE, 497–505. https://doi.org/10.1109/ICCD46524.2019.00075

[14] Weihao Cui, Han Zhao, Quan Chen, Ningxin Zheng, Jingwen Leng, Jieru Zhao, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2021. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. https://doi.org/10.1145/3458817.3476143

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. https://doi.org/10.18653/v1/n19-1423

[16] Message Passing Interface Forum. 1994. MPI: A message - passing interface standard.

[17] Karl Pearson F.R.S. 1901. On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, 11 (1901), 559–572. https://doi.org/10.1080/14786440109462720

[18] Yiming Gan, Yuxian Qiu, Lele Chen, Jingwen Leng, and Yuhao Zhu. 2020. Low-Latency Proactive Continuous Vision. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*. https://doi.org/10.1145/3410463.3414650

[19] Yiming Gan, Yuxian Qiu, Jingwen Leng, Minyi Guo, and Yuhao Zhu. 2020. Ptolemy: Architecture Support for Robust Deep Learning. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 241–255. https://doi.org/10.1109/MICRO50266.2020.00031

[20] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, et al. [n.d.]. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 3–18. https://doi.org/10.1145/3297858.3304013

[21] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. 2020. Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*. IEEE, 681–697. https://doi.org/10.1109/MICRO50266.2020.00062

[22] Google. 2020. XLA: Optimizing Compiler for TensorFlow. https://www.tensorflow.org/xla.

[23] Google. 2021. TensorFlow graph optimization with Grappler. https://www.tensorflow.org/guide/graph_optimization.

[24] Yue Guan, Jingwen Leng, Chao Li, Quan Chen, and Minyi Guo. 2020. How Far Does BERT Look At: Distance-based Clustering and Analysis of BERT's Attention. In *Proceedings of the 28th International Conference on Computational Linguistics (COLING)*. International Committee on Computational Linguistics, 3853–3860. https://doi.org/10.18653/v1/2020.coling-main.342

[25] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 443–462. https://doi.org/10.5555/3488766.3488791

[26] Cong Guo, Bo Yang Hsueh, Jingwen Leng, Yuxian Qiu, Yue Guan, Zehuan Wang, Xiaoying Jia, Xipeng Li, Minyi Guo, and Yuhao Zhu. 2020. Accelerating sparse DNN models without hardware-support via tile-wise sparsity. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE/ACM. https://doi.org/10.1109/SC41405.2020.00020

[27] Cong Guo, Yangjie Zhou, Jingwen Leng, Yuhao Zhu, Zidong Du, Quan Chen, Chao Li, Bin Yao, and Minyi Guo. 2020. Balancing Efficiency and Flexibility for DNN Acceleration via Temporal GPU-Systolic Array Integration. In *57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020*. IEEE, 1–6. https://doi.org/10.1109/DAC18072.2020.9218732

[28] Kim M. Hazelwood, Sarah Bird, David M. Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 620–629. https://doi.org/10.1109/HPCA.2018.00059

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[30] Intel. 2019. Math Kernel Library for Deep Neural Networks. https://github.com/rsdubtso/mkl-dnn.

[31] Zhihao Jia, Oded Padon, James J. Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 47–62. https://doi.org/10.1145/3341301.3359630

[32] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12. https://doi.org/10.1145/3079856.3080246

[33] Harshad Kasture and Daniel Sánchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization, IISWC 2016, Providence, RI, USA, September 25-27, 2016*. IEEE Computer Society, 3–12. https://doi.org/10.1109/IISWC.2016.7581261

[34] Neeraj Kulkarni, Gonzalo Gonzalez-Pumariega, Amulya Khurana, Christine A. Shoemaker, Christina Delimitrou, and David H. Albonesi. 2020. CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO50266.2020.00060

[35] Michael A. Laurenzano, Yunqi Zhang, Lingjia Tang, and Jason Mars. 2014. Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers. In *47th Annual IEEE/ACM International Symposium on Microarchitecture, (MICRO)*. IEEE Computer Society, 558–570. https://doi.org/10.1109/MICRO.2014.21

[36] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single Shot MultiBox Detector. In *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9905)*. Springer, 21–37. https://doi.org/10.1007/978-3-319-46448-0_2

[37] Zihan Liu, Jingwen Leng, Quan Chen, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2020. DLFusion: An Auto-Tuning Compiler for Layer Fusion on Deep Neural Network Accelerator. In *IEEE International Conference on Parallel & Distributed Processing with Applications (ISPA)*. IEEE, 118–127. https://doi.org/10.1109/ISPA-BDCloud-SocialCom-SustainCom51426.2020.00041

[38] Zihan Liu, Jingwen Leng, Guandong Lu, Chenhui Wang, Quan Chen, and Minyi Guo. 2020. Survey and design of paleozoic: a high-performance compiler tool chain for deep learning inference accelerator. *CCF Trans. High Perform. Comput.* 2, 4 (2020), 332–347. https://doi.org/10.1007/s42514-020-00044-7

[39] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/2749469.2749475

[40] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/2155620.2155650

[41] Microsoft. 2021. Optimize and Accelerate Machine Learning Inferencing and Training. https://onnxruntime.ai/.

[42] Pascale Minet, Eric Renault, Ines Khoufi, and Selma Boumerdassi. 2018. Analyzing Traces from a Google Data Center. In *14th International Wireless Communications & Mobile Computing Conference, (IWCMC)*. IEEE, 1167–1172. https://doi.org/10.1109/IWCMC.2018.8450304

[43] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. *arXiv preprint arXiv:1701.03980* (2017).

[44] NVIDIA. [n.d.]. *Multi-Process Service*. NVIDIA.

[45] NVIDIA. 2021. NVIDIA A100 Tensor Core GPU. https://www.nvidia.com/en-us/data-center/a100/.

[46] NVIDIA. 2021. NVIDIA cuDNN. https://developer.nvidia.com/cudnn.

[47] NVIDIA. 2021. NVIDIA MULTI-INSTANCE GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/.

[48] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, et al. 2020. MLPerf Inference Benchmark. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, (ISCA)*. IEEE, 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[49] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 6517–6525. https:

//doi.org/10.1109/CVPR.2017.690

[50] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. IEEE Computer Society, 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[51] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. ACM. https://doi.org/10.1145/3341301.3359658

[52] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*. IEEE Computer Society, 1–9. https://doi.org/10.1109/CVPR.2015.7298594

[53] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*. PMLR, 6105–6114.

[54] Lingjia Tang, Jason Mars, and Mary Lou Soffa. 2012. Compiling for niceness: mitigating contention for QoS in warehouse scale computers. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012, San Jose, CA, USA, March 31 - April 04, 2012*. ACM, 1–12. https://doi.org/10.1145/2259016.2259018

[55] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[56] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018).

[57] Wei-Jen Wang, Yue-Shan Chang, Win-Tsung Lo, and Yi-Kang Lee. 2013. Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments. *J. Supercomput.* 66, 2 (2013), 783–811. https://doi.org/10.1007/s11227-013-0890-2

[58] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-side Sparse Tensor Core. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*. IEEE, 1083–1095. https://doi.org/10.1109/ISCA52012.2021.00088

[59] Yecheng Xiang and Hyoseung Kim. 2019. Pipelined Data-Parallel CPU/GPU Scheduling for Multi-DNN Real-Time Inference. In *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*. IEEE, 392–405. https://doi.org/10.1109/RTSS46320.2019.00042

[60] Hailong Yang, Alex D. Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *The 40th Annual International Symposium on Computer Architecture (ISCA)*. https://doi.org/10.1145/2485922.2485974

[61] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. IEEE Computer Society, 20:1–20:12. https://doi.org/10.1109/MICRO.2016.7783723

[62] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. 2020. Architectural Implications of Graph Neural Networks. *IEEE Computer Architecture Letter* (2020). https://doi.org/10.1109/LCA.2020.2988991

[63] Wenyi Zhao, Quan Chen, Hao Lin, Jianfeng Zhang, Jingwen Leng, Chao Li, Wenli Zheng, Li Li, and Minyi Guo. 2019. Themis: Predicting and Reining in Application-Level Slowdown on Spatial Multitasking GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium, (IPDPS)*. IEEE, 653–663. https://doi.org/10.1109/IPDPS.2019.00074

[64] Xia Zhao, Magnus Jahre, and Lieven Eeckhout. 2020. HSM: A Hybrid Slowdown Model for Multitasking GPUs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 1371–1385. https://doi.org/10.1145/3373376.3378457

[65] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. https://doi.org/10.5555/3488766.3488815

[66] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Architectural Support for Programming Languages and Operating Systems, Lausanne (ASPLOS)*. https://doi.org/10.1145/3373376.3378508

[67] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2021. Graph Neural Networks: A Review of Methods and Applications. arXiv:1812.08434 [cs.LG]

[68] Yangjie Zhou, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. 2021. Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. https://doi.org/10.1109/IISWC53511.2021.00029