GACER: Granularity-Aware ConcurrEncy Regulation for Multi-Tenant Deep Learning

Yongbo Yu[1], Fuxun Yu[2], Zhi Tian[1], Xiang Chen[3]
[1]George Mason University, [2]Microsoft, [3]Peking University
yyu25@gmu.edu,fuxunyu@microsoft.com,ztian1@gmu.edu,xiang.chen@pku.edu.cn

Abstract

As deep learning continues to advance and is applied to increasingly complex scenarios, the demand for concurrent deployment of multiple neural network models has arisen. This demand, commonly referred to as multi-tenant computing, is becoming more and more important. However, even the most mature GPU-based computing systems struggle to adequately address the significant heterogeneity and complexity among concurrent models in terms of resource allocation and runtime scheduling. And this usually results in considerable resource utilization and throughput issues. To tackle these issues, this work proposes a set of optimization techniques that advance the granularity of computing management from both the spatial and temporal perspectives, specifically tailored to heterogeneous model compositions for deep learning inference and training. These techniques are further integrated as *GACER* — an automated optimization framework that provides high-utilization, high-throughput, and low-latency multi-tenant computing support. And our experiments demonstrate that *GACER* significantly improves the overall resource utilization and consistently achieves outstanding speedups compared to native GPU computing frameworks and existing state-of-the-art optimization works.

ACM Reference Format:
Yongbo Yu[1], Fuxun Yu[2], Zhi Tian[1], Xiang Chen[3]. 2024. GACER: Granularity-Aware ConcurrEncy Regulation for Multi-Tenant Deep Learning . In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676718

*Corresponding author: Xiang Chen (xiang.chen@pku.edu.cn)

1 Introduction

The explosive success of deep learning techniques in various cognitive tasks, such as image classification and speech recognition, has made neural network models the hot spot of computing systems research. One of the primary drivers behind this trend is the support provided by GPUs, which offer excellent computing capacity and parallelism capability [1, 2]. Despite the active emergence of various accelerators or systems based on ASIC and FPGA, GPUs remain the most widely adopted platform in practice, accounting for >85% market share in both cloud and edge applications [3].

However, for a long time in the past, due to the substantial parameters of neural network models, many research works tended to adopt a generic setting: one GPU instance could only host a single model. And even many recent outstanding efforts have not stepped out of this rut (e.g., MetaFlow [4], IOS [5]). However, cutting-edge developments have gradually revolutionized this setting: With the miniaturization of neural networks and the massification of GPUs, it is now possible for a single GPU to host multiple models simultaneously [6, 7]. Furthermore, the need for concurrent model processing also scales up with multi-task or multi-modality intelligence integration, such as in autonomous driving [8–10]. Along with this trend, GPU manufacturers like NVIDIA actively promote related software and hardware techniques [11–13]. Thus, "**multi-tenant**" deep learning computing becomes more and more prominent, especially for GPU-based systems.

However, multi-tenant deep learning presents a greater challenge than conventional single-tenant deployment regarding computing management. This is due to the increased heterogeneity and complexity of the concurrency of multiple neural network models, which can vary in operator composition, structural design, and model scale [14, 15]. Unfortunately, these issues are not well addressed in current GPU computing, even with emerging supporting techniques from manufacturers: (1) When it comes to resource allocation, current techniques for issuing concurrent models into the GPU either result in using fixed hardware resource budgets or competing models for resources in a greedy manner. These approaches at the model level can lead to the wastage of hardware capabilities or resource contention and the corresponding overhead [16–18]. (2) When it comes to runtime scheduling, though many recent works have dived into the operator level, most of them either cannot handle multi-tenant scenarios or have overlooked the multi-tenant coordination overhead, leading to ineffective and unscalable runtime management [5, 19]. (3) When rising to the system level for comprehensive optimization, current approaches can only consider a certain compute stack, or a single granularity optimization from spatial or temporal. Some of these single-dimensional optimizations bring huge overheads, while others are too narrow to deal with complex multi-tenant scenarios. This hinders the flexibility of multi-tenant computing deployments and ignores and obscures many optimization opportunities.

Given these observations, the primary motivation for multi-tenant deep learning optimization is to develop a fine-grained and feasible coordination computing framework to resolve resource contention and enhance complementary resource utilization across different model tenants with minimal regulation overhead. Therefore, several optimization expectations for multi-tenant deep learning can be derived: (1) From a spatial perspective, the resource allocation scheme requires an operator-level granularity of regulation to dynamically control the resource consumption of operators, thus adapting to the resource consumption dynamics of intra-model operators and therefore complement intra-model resource requirements [19, 20] (2) From a temporal perspective, the granularity of multi-tenant runtime scheduling should not only deepen to the operator level, but also improve the manageability to regulate the runtime overhead and balance the overall performance given different deployment complexities [15, 19] (3)From the perspective of a cross-computing-stack system optimization, co-optimization across computing stacks should be used to circumvent the limitations of a single stack so as to achieve complementary strengths of different optimization domains. These optimization expectations from both the spatial and temporal perspectives point to the same optimization focus of this work:

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10.
https://doi.org/10.1145/3676536.3676718

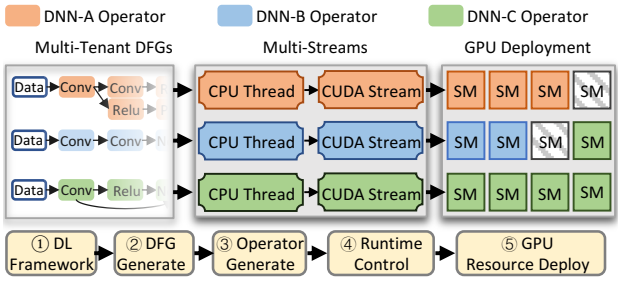

Figure 1. Multi-Tenant Deep Learning with GPUs

To optimize the computing performance across multiple computing stacks and advance the current multi-tenant concurrency regulation granularity from the both spatial and temporal domains.

Centering on this research focus, our work extensively examines the entire GPU-based system stacks and conducts a comprehensive analysis of current multi-tenant computing issues with state-of-the-art techniques. Subsequently, we propose a set of optimization techniques that advance the granularity of computing management in both the spatial and temporal management domains, significantly improving runtime performance for both deep learning inference and training. The contributions of this work are as follows:

- We reveal computing issues of multi-Tenant deep learning on both front-end graph computing and back-end CPU-GPU coordination stages and proposed advanced spatial and temporal optimization schemes methods to solve these issues.
- In the spatial domain, we proposed a dynamic compiling method by rewriting the model graph structure and therefore to enhance the operator split and fusion manageability for resolving the model contention and exploits resource utilization.
- In the temporal domain, we innovatively escalate the CPU-GPU synchronization mechanisms by modifying the `CUDA Streams` and introduce fine-grained synchronization pointers to further coordinate run-time graph scheduling at the operator level.
- The proposed techniques are integrated into an automated optimization framework — *GACER*, which leverages a low-cost search method to identify the particular spatial and temporal deployment configuration for both offline and online multi-tenant deep learning scenarios. This framework is lightweight and flexible and can be modified for any models.

GACER consistently provides high-utilization and high-throughput computing support to multi-tenant deep learning on GPUs. Compared to conventional computing framework without specific multi-tenant support (e.g., TVM), *GACER* could consistently achieve almost ~70% speeds up. And compared to state-of-the-art multi-tenant optimization works, it could also achieve ~30% acceleration with ~40% resource utilization enhancement, and demonstrate outstanding capabilities for even more complex scenarios.

2 Background and Motivation

2.1 Multi-Tenant Deep Learning Deployment

This work focuses on a generic multi-tenant deep learning deployment setting on GPUs [26]. This process can be summarized as five execution stacks at the bottom of Fig. 1. The stacks ①, ②, ③ are computational front-end, and the stacks ④, ⑤ are back-end. When multiple heterogeneous DNN models are deployed on a single GPU, and each model is defined by the DL frameworks (①) and the model compiler compiles the model structure into a data flow graph (② DFG) that defines the computing sequence of a series of different operators by layers (e.g., Conv, ReLu, etc.). And these generated operators (③) have a particular computational pattern and resource requirements, including computing resources measured by the occupancy of streaming multi-processors (SM), and memory resources measured by memory bandwidth utilization. The DFG compilation is generally performed on the CPU side, and the DFG of each DNN tenant is wrapped into one or several threads for later GPU deployment. It is worth noting that the simultaneous multi-threads wrapping process from multi-DFG has less overhead than initializing an operator at a time. On the GPU side, the runtime control stack (④) uses several GPU computing streams [12] to control when and how operators are dispatched to the physical GPU resource pool (⑤).

2.2 Multi-Tenant Computing Support

Emerging GPU Techniques: There are some new features proposed by NVIDIA, such MIG [13] and MPS [11], which enable GPUs to divide the resource pool (⑤) into multiple concurrent portions for multi-tenant deployment. Although these methods provide GPUs with spatial resource management capabilities, they often suffer from reconfiguration overhead and a lack of certain runtime flexibility. Unlike MIG and MPS, Multi-Stream (MS) [12] differs from the hardware resource partition mechanism; it could dynamically share resources when running multiple tenants, facilitating resource budget adjustment for multi-tenant models in runtime control stack (④). This is usually done by the native black-box GPU scheduler, often resulting in considerable resource contention and inappropriate scheduling cases [19], but we could leverage certain APIs to adjust the dispatching results.

State-of-the-Art Optimization Works: Based on these emerging techniques, many front-end and the back-end optimization works have been proposed, Table 1 gives the reviews of recent works for multi-tenant computing. Some studies adjust the batch size on the framework stack (①) and resource adaption with MIG or MPS to implement software-hardware co-optimization for enhancing multi-tenant resource sharing [18, 21]. KRISP [22] is a optimization method on operator generation stack (③) and use the AMD's CU masking mechanism to adjust the number of CU occupied by each operator when deployed with multiple models. Meanwhile, very recent works focus on DFG (②) and runtime stack (④) and expand the computing granularity of MS into the operator level [5, 19, 23, 27]. These works are breaking the granularity of the runtime control, thus enabling more flexible concurrent deployment of GPUs. For instance, AutoMT [19] utilizes MS and implemented inter-operator scheduling with DFG for optimizing multi-model by balancing the workload to modify greedy resource contentions. In summary, current approaches are optimizing how multi-tenants share GPU resources across both spatial and temporal domains.

In addition to reviewing the basic mechanisms, pros and cons of different techniques, Table 1 also provides a comparison of the optimization granularity. While these works enable multi-tenant deep learning on GPUs, most are still limited to a coarse granularity (esp. model level) and focus on a singular optimization domain.

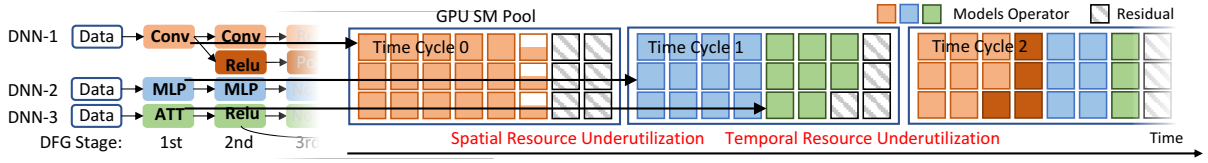

Figure 2. GPU Resource Utilization Analysis from Spatial and Temporal Perspectives

3 Design Motivation and Challenges

This work focuses on advancing the manageability and efficiency of multi-tenant deep learning at the fine-grained operator-level and expanding the spatial-temporal optimization space for comprehensive performance escalation from front-end and back-end stacks. Here we refer to the unused portion of the GPU resources as "residue", and a GPU SM pool-oriented example is shown in Fig. 2, which is further analyzed from both spatial and temporal perspectives.

Persistent Resource Utilization Issues Fig. 2 highlights common multi-tenant computing issues that arise when deploying heterogeneous DNN models on a GPU with MS techniques and the ideal operator-level resource allocation. In this process, each operator is issued from different model DFGs into CUDA streams to share the computing resource at each cycle. Despite the utilization of the best available multi-tenant deployment settings above, resource under-utilization issues are still inevitable. In most deployment scenarios, even with operator-level resource allocation in place, the total resource requirements from concurrent tenants of a DFG stage will not exactly match the available GPU resource volume. In Fig. 2, when the three operators Conv, MLP and ATT in the 1st stage are simultaneously sent to the streams for deployment, only the first Conv from DNN-1 is deployed in time cycle T_0, whereas the MLP from DNN-2, ideally processed concurrently with the orange layer, is postponed to time cycle T_1 for activation due to a lack of available resources. This inability to deploy results in parallel conflict and thus spatial resource under-utilization. On the other hand, the unpredictable order of operator execution makes some operators which could been concurrently executed, but to be delayed, such as the Relu of DNN-1 in Fig. 2. This operator can be deployed at time cycle T_1 without computational dependency errors. This situation leads to some temporal resource under-utilization. To illustrate this persistent issue, the fig. 3 displays an empirical example using the NVIDIA Nsight [28]. As shown there: In some GPU cycles, concurrent operator deployment cannot be established for parallel processing, resulting in even worse resource under-utilization. In other cycles, the concurrently deployed operators are unable to utilize the SM pool fully.

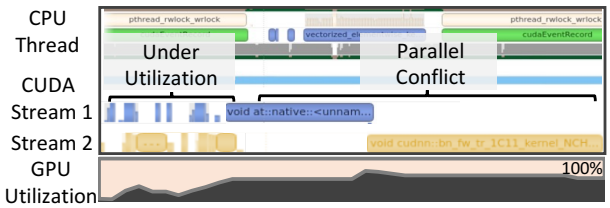

Figure 3. Issues Shown in NVIDIA Nsight

Spatial Residue Optimization by Operator Transformation is well known to adjust the operator workload by compiling stack optimizations like split into multiple smaller operators. However, when it comes to the multi-tenant scenario, the operator transformation becomes more challenges in terms of complexity and granularity. Therefore, a more ***dynamic compiling mechanism*** is urgently needed to adjust the operator workload and general the DFG. However, certain challenges arise: (1) Current workload adjustment is applied to all operators through batch size direction in the DL frameworks [18, 21]. These adjustments are model-level coarse-grained, which can leave significant fine-grain under-utilization. Meanwhile, these works reliance on model-scoped partitioning techniques, existing spatially partitioned inference servers incur high reconfiguration overheads (in the 10's of seconds). Even though some work [22] try to adjust the size of each operator to solve the above problem, such tuned operators cannot take advantage of optimized operator acceleration libraries (e.g., CuDNN), leading to deployments that are not always optimal. Existing frameworks need to be modified at the library-level so that the model structure can be dynamically compiled into different DFGs at the front-end, thus enabling each operator to be split into multiple workloads for deployment. And this process should be flexible, maximizing the effectiveness of existing optimization systems. (2) Meanwhile, operator transformation introduces split and fusion to achieve operator-level workload partition. While the reconfiguration overhead of these operations is only microseconds, the optimization involves multiple operators in multi-tenant scenarios. Therefore, the optimization complexity also scales up to determine the appropriate operator for workload adjustment or split, taking into account the transformation overhead.

Temporal Residue Optimization by Operator Reordering is also applied to adjust operator computing sequences by manipulating the DFG. When it comes to the multi-tenant scenario, an

Table 1. Granularity and Performance of Recent Works on Multi-Tenant DL Computing Optimization

	Major Stack	Techniques	Approaches	Granularity	Remaining Issues
Spatial	• ① Framework	Gslice[18], Salus[21]	• Multi-Tenant with MPS • Batching Configuration	• Model Level	• Coarse-Grained Optimization • Reconfiguration Overhead
	• ③ Operator	KRISP[22]	• Operator Resizing • Resource Partitioning	• Operator Level	• Only Support AMD GPU • Incompatibility with Acceleration Libraries
Temporal	• ② DFG • ④ Runtime	AutoMT[19] POS[23]	• DFG-based Scheduling with MS • Resource Contention Optimization	• Operator Level	• Unregulated Resource Allocation • Runtime Regulation Overhead
	• ④ Runtime	EdgeBatch [24] Gpipe [25]	• Time-Sliced Sharing • Runtime Singe-Tenant Switch	• Sub-Model/ Model Level	• Tenant Switch Overhead • Not Really concurrent

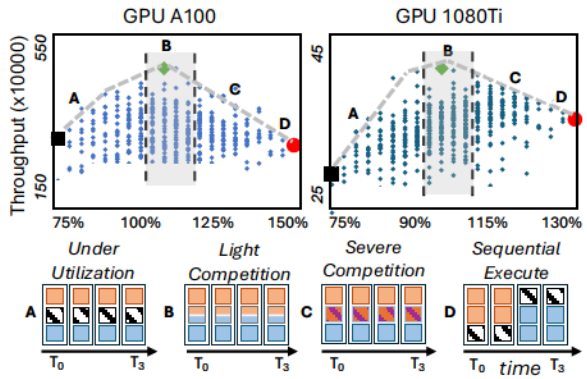

Figure 4. Operator Deployment Analysis

escalated CPU-GPU coordination mechanisms is needed to segment multi-DFGs into several operator stages and introduce fine-grained synchronization at back-end. In addition to conventional scheduling issues, such as layer dependency, new challenges are posed: (1) Most of current runtime scheduling work focus on single DFG and cannot handle multi-tenant scenarios [5, 23]. Multi-tenant runtime control involves multiple operators across various DFG settings, resulting in unprecedented dynamic complexity. Although the CPU-GPU synchronization provided by MS could achieve model reordering, specific library-level optimization is required to modify the `CUDA Stream` to introduce fine-grained synchronization. (2) Even, some work consider the complexity of multi-DFGs, they overlook the multi-tenant coordination overhead (~microseconds), which is caused by CPU and multi-stream synchronization, thus leading to ineffective and unscalable runtime management. Therefore, a significant DFG reconfiguration delay would be introduced, requiring an effective runtime overhead regulation scheme.

Spatial and Temporal Co-optimization: It is important to note that, both spatial and temporal optimization involves more than just minimizing residue and thus improve utilization; it also requires identifying the diverse "sweet-zones" of the GPU resource pool and make the total resource utilization falls in this "sweet-zone". We conducted a straightforward experiment. By simultaneously running two identical operators (e.g., Convolution or MaxPooling) and adjusting their batch sizes, we varied each operator's SM occupancy. We then examined the correlation between throughput (calculated as the total batch size divided by the combined runtime of both operators) and the overall SM occupancy. Each data point in our study represents a unique instance of concurrent operation, as exemplified in Fig. 4.

Our experiments on two distinct GPUs, NVIDIA A100 and 1080Ti, yield insightful results. On the A100, when total resource utilization is below capacity, the throughput is comparatively lower, as indicated in A. Intriguingly, the peak throughput doesn't occur at 100% resource utilization, but rather at around 115%, as indicated in B. At this point, there is competition sharing in some of the SMs, but this actually improves the utilization of resources. Beyond this point, increased competition between operators leads to a reduction in throughput (C). Further enlarging the size of the operators results in a fallback to sequential execution due to deployment constraints (D), as in the T_0 and T_1 in Fig. 2. A similar pattern is observed on the 1080Ti, though it emerges sooner, attributable to the 1080Ti's finer granularity in SM deployment and lower SM sharing power.

Hence, the objective of GPU optimization is to break the deployment and scheduling granularity by operator transformation and reordering to change the size of the residue. This is a full-stack optimization that ultimately aims to make GPU resource utilization from A, C and D to B. Consequently, the subsequent section will concentrate on strategies for optimizing residue.

4 Granularity-Aware Multi-Tenant Regulation

Fig. 5 shows the architecture of *GACER*. We describe the key components of *GACER* and their roles below.

4.1 Problem Formulation

Multi-tenant DNN consists of n models: $M_1, M_2, ..., M_n$. And each model M can be represented by a DFG with a series of operators O. So we have the model operator list $M_n = [O_{n,1}, O_{n,2}, ..., O_{n,i}]$, and each $O_{n,i}$ has two attributes $U_{n,i}, T_{n,i}$, U is resource utilization (SM occupancy), and T is execution time.

Optimization Objective: We abstract the total GPU SM resource as resource utilization U_{GPU}. The residual in time cycles S_t could be formulated as

$$R_{S_t} : [U, T] = R_{S_t} : [U_{GPU} - U_t, T_t]. \tag{1}$$

We sum R_{S_t} across all cycles to compute the total residue R.

$$R = \sum_{S_0 \to S_t} (U_{GPU} - U_t) * T_t. \tag{2}$$

Our objective is to minimize R, and in the equation 2, R has two variables: the operator size $U_{n,i}$ and computing sequence $S_0 \to S_t$, and the former is determined by $BS_{n,i}$. Therefore, we could optimize two sub-objectives, finding the appropriate $BS_{n,i}$ and $S_0 \to S_t$ to minimize the R.

$$\tau\ (BS_{n,i},\ S_0 \to S_t) \to Min\ R, \tag{3}$$

where τ is an approximation approach.

4.2 Spatial Utilization Optimization

We design a novel dynamic compiling method by rewriting the model graph structure to optimize the first sub-objective.

DFG Dynamic Compiling allows a model's DFG to be deployed at a finer granularity. *(1)Finer-grained operator splitting:* An original operator is split into multiple copies of the operator using the split operation, and each operator has a sub-batch as input for separate computation. The result of the computation is then fused back to the original operator (Fig. 6 (ii)). *(2)Operator workload adaptation:* The batch size for each sub-operator relates to the size of the workload for each operator, and adjusting the appropriate size can effectively reduce the spatial residual in Fig. 6 (i). *(3)Transformation*

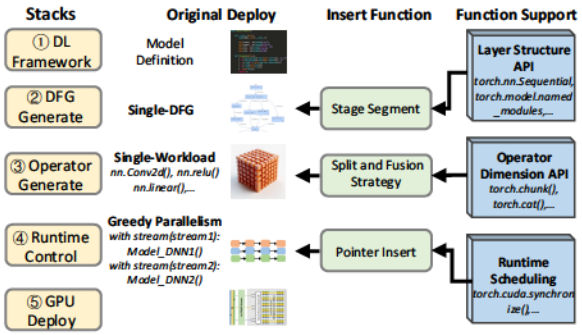

Figure 5. GACER Architecture Design

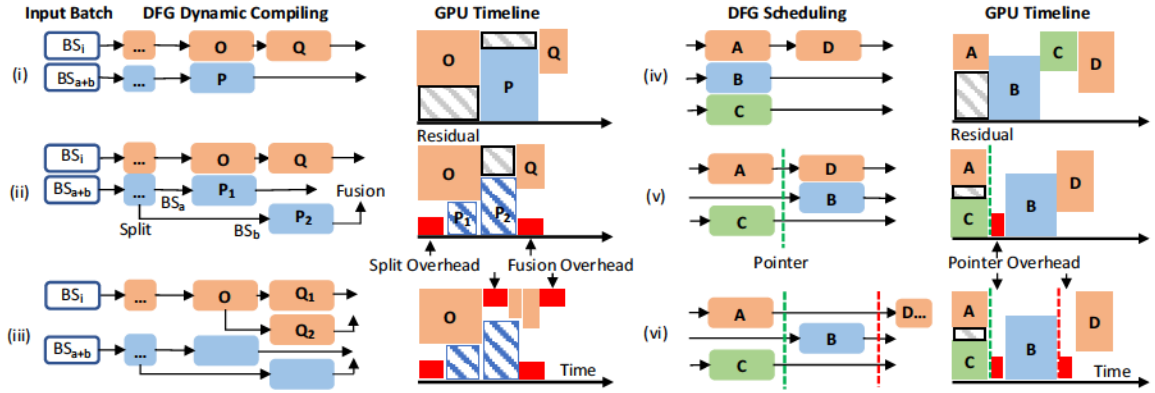

Figure 6. Operator Transformation and Reordering

overhead invoking optimization: The operator transformation regulation must introduce additional split and fusion operations along the batch size dimensional, which also brings additional overhead (Fig. 6 (ii)). The size of this overhead is related to the hardware, for example, on NVIDIA 1080Ti GPU, this time is about 15us. Therefore, some layers with GPU computation time smaller than this overhead will not be able to gain after the transformation, e.g., split Q brings more overhead than the original residual in Fig. 6 (iii).

Dynamic Compiling Support: The current DNN model definition structure employs the same batch size for all layers, resulting in a model-level granularity that fails to consider varying workloads in each layer. Therefore, we extend the layer structure of the model in the DL framework, which enables our framework to split the batch size of each operator. We rewrite the API model definition using PyTorch's some operator dimension API "`torch.chunk()`" and "`torch.cat()`". The "chunk()" can decompose the operator's workload in the batch direction. Thus the operator splits into multiple copies for separate calculations. Then we can perform a "cat()" operation on the result to restore the input feature map size. By decomposing heavy workload operators into smaller ones and concatenating the resulting micro-batches, we can effectively control runtime resource consumption. This case does not affect the feature map of each input, and thus the result of the computation.

Operator Split and Fusion: To find which operator should be split, we need to determine the size of the current time cycle residual R_{S_t} and the resource usage corresponding to the different batch sizes of the next operator. We need to make sure that the split batch size can fit into the current residual.

The operator size is determined by the layer structure LS, the input size IS, and the batch size BS. We analyze DNN operators with different batch settings using the GPU analysis tool, NVIDIA Nsight Computing [29], and set up offline lookup tables:

$$O[U] = f_U(O, BS) \text{ and } O[T] = f_T(O, BS). \tag{4}$$

We use the IS, LS as the feature of O. With the above lookup tables, we can also simply set up two reverse lookup tables to lookup BS based on U and T.

$$BS = f_{BS}(O, O[U]) \text{ or } BS = f_{BS}(O, O[T]). \tag{5}$$

Split in stage: We calculate the R_{S_t} of time cycle S_t within some operators *stage* according to Eq. 1. We then give two conditions Eq. 6 and 7 to determine the appropriate batch size. The conditions show that this residue can fit at least one batch of the potential operator $O_{n,i}$ from resource utilization and execution time, respectively.

$$R_{S_t}[U] > f_U(O_{n,i}, BS = 1), \text{where } O_{n,i} \in S_{t+1} \text{ and } \forall O_n \notin S_t \tag{6}$$

$$R_{S_t}[T] > f_T(O_{n,i}, BS = 1) + Transformation\ Overhead,, \quad \text{where } O_{n,i} \in S_{t+1} \text{ and } \forall O_n \notin S_t \tag{7}$$

If there is $O_{n,i}$ satisfying the above condition, we then calculate the feasible batch using Eq. 8, otherwise skip S_t into S_{t+1}.

$$Min(f_{BS}(O_{n,i}, R_{S_t}[U]), f_{BS}(O_{n,i}, R_{S_t}[T])) \tag{8}$$

We compute feature maps from different batch sizes and fuse the feature maps as the output. Then, we go to the S_{t+1} and repeat the above calculations until *stage* last period. At this point all residues within the *stage* are eliminated, leaving only the last residue.

4.3 Temporal Utilization Optimization

In the temporal domain, we break through multiple DFGs into fine-grained operator segments, and therefore operator scheduling across multiple different models is implemented.

DFG Graph Scheduling: *(1)Finer-grained stage segmentation*: For reordering the operator execution sequence, we introduce specific pointers into the DFG. The pointer is a CPU/GPU synchronization mechanism that ensures operators continue to be deployed only after previously issued operators have been completed. Through these pointers, we could divide the DFGs into several different stages (Fig. 6 (v)). Such stage splittings ensure the operators only share the assigned resources in the same stage, thus supporting the stage-level concurrency control. *(2)Stage-level concurrency control*: By adjusting where the pointers are inserted, we could control how many operators are assigned in each stage. This enables us to reduce the temporal residual, such as the residual in Fig. 6 (iv). *(3)Pointer overhead invoking optimization:* However, the reordering method uses synchronization operation which is the runtime control, so this inevitably introduces scheduling overhead, as shown in Fig. 6 (vi). The GPU waits until the CPU finishes synchronizing the pointer and then sends a new operator to the GPU. This situation can lead to a lot of GPU wait time, resulting in additional residual resources. Adding a large number of pointers leads to frequent GPU waits, which can seriously slow down the overall efficiency of the GPU.

Escalated CPU/GPU Coordination: To support this method, we also re-design the DNN API at the library level. This requires simultaneous efforts in both stacks ② and ④: stage segment and pointer insert, as shown in Fig. 5. For the stage segment, we use some layer structure API splitting layers into different stages without changing the computational dependencies between layers. First, we

use the "`torch.model.named_modules()`" method to get all the model layer objects and then use "`torch.nn.Sequential`" to refine all the objects into several stages of DFG objects. For pointer insert between different stages, we use the runtime control API "`torch.cuda.stream.wait_event()`" to insert synchronization pointers to different stages. At each time, the CPU will only send operators within the same stage from different DFGs to different streams of the GPU for deployment, so that we could only co-run the layers in the same stages. Then we insert the synchronize API "`torch.cuda.synchronize()`" at end of the stages. This situation forces the CPU to wait for all GPU streams to finish computation and achieve CPU/GPU synchronization before proceeding to the next stage for parallel computation. Based on the above operations, we can implement operator reordering and thus fine-grained scheduling of operators.

Run-Time Graph Scheduling: The runtime deployment case in Fig. 6 (v) can be abstracted as a series of operators concurrently occupying the GPU resource pool. We divide this process into multiple periods S_t according to the different resource occupancy, and the resource occupancy remains the same within the same S_t for several time cycles. Therefore, S_t is a binary array, the first index is the aggregate SM occupancy U_t of several concurrent operators and the second is the the number of sustained time cycles T_t. Therefore, the period in Fig. 2 can be abstracted as following:

$$\begin{aligned} &S_0 : [U_A + U_C, T_A],\ S_1 : [U_B, T_B], S_2 : [U_D, T_D], ..., \\ &S_t : [U_t, T_t], where\ U_t = \sum_{U_{n,i}\ in\ S_t} U_{n,i},\ and\ U_t <= U_{GPU} \end{aligned} \tag{9}$$

Thus, S_t is possible to react to the operators' computation sequence.

There are two considerations, the number of pointers (the number of stages - 1) and the location of the pointers. For the number of pointers, we adaptively determine it based on the computation time of the DFGs. Since each model may have a different computation time, we first align the different DFGs. We pair the computation sequences of multiple models so that the operators of different models can be evenly distributed to multiple stages. This method is crucial for maintaining a balanced workload during the execution of multiple models, thus avoiding the overall under-utilization of GPUs in the tail of some models with too long computation time.

Stage Initialization: We set the number of stages based on the minimum operator count $Min(|M_n|)$ in any model for even distribution. Then, we divide other models into equal stages, considering each model's total execution time:$\sum_{T\in O, O\in M_j} T_{j,i}/min(|M_n|)$. Operators on the boundary of two neighboring stages are assigned to the side of the stage occupied by the operator according to the occupancy ratio. Therefore, we divide the operators of all models into $Min(|M_n|)$ stages and also get an initialized pointer location.

Scheduling Cross Stages: The purpose of this step is to adjust the position and number of pointers to reduce residual. We fill in the last period residue in $stage_k$ using the operator within the next $stage_{k+1}$. We need to determine whether this residue is greater than the computation time required for potential O.

$$\begin{aligned} R_{S_t}[T] > &O_{n,i}[T]\ +\ Pointer\ Overhead, \\ &where\ O_{n,i} \in S_{t+1}\ and\ \forall O_n \notin S_t \end{aligned} \tag{10}$$

If $R_{S_t}[T]$ is less than that, then the pointer here will bring less overhead than gain, in which case the pointer will be deleted. If $R_{S_t}[T]$ is greater than that, then continue to determine whether the utilization is sufficient:

$$R_{S_t}[U] > O_{n,i},\ where\ O_{n,i}\ \in\ S_{t+1}\ and\ \forall O_n \notin S_t \tag{11}$$

If the Eq. 11 holds, then the operator $O_{n,i}$ can be moved in. After moving in the new operator, recalculate the residue of the tail period and go back to the start of Scheduling Cross Stages to determine if the next operator can be moved.

Delete Pointer: After the above synchronization overhead analysis, we check if the pointer can be removed. There are two ways to delete a pointer besides the way in Eq. 10. First, if the time duration of $stage_k$ < synchronization overhead, then delete pointer and merge $stage_k$ and $stage_{k+1}$; Second, if $stage_k$ has only one operator, then delete pointer and merge $stage_k$ and $stage_{k+1}$. After deleting, the algorithm goes back to Step 2, until finishing all stages.

4.4 Spatial and Temporal Joint Optimization

Algorithm 1 Joint Optimization Algorithm for DFGs

Require: N DFGs
Ensure: The optimization strategy
Step 1: Stage Initialization
for U_{GPU} = 90%; $U_{GPU} \le$ 125%; U_{GPU} += 5% **do**
  Step 2: Split in stage
  for $stage_i$ from 0 to k **do**
    for each S_t in $stage_i$ **do**
      Calculate the residual R_{S_t} according to Eq. 1.
      if Eq. 6 and 7 are true **then**
        Compute the feasible batch using Eq. 8.
      else Skip S_t to the next period S_{t+1}.
    Step 3: Reordering Cross Stages
    for last R_{S_t} in $stage_i$ **do**
      if Eq. 11 is true **then**
        if Eq. 10 is true **then**
          Move pointer; Go back to step 3.
        else Go to Step 4.
      else Delete the pointer;i + +; Go to Step 2.
  Step 4: Delete Pointer
  Check if the pointer here can be removed.
  Step 5: Strategy Profiling
  Run the current optimization strategy and record the latency.
  Go to Step 1.
Step 6: Compare and Output the Final Result
Compare the latency under different U_{GPU} and take the shortest latency as the final result.

Based on the formulation and regulation design, we propose our spatial and temporal co-optimization framework to minimize R by finding the optimal batch size for each operator and multi-DFGs' computing sequence. We propose a 6-step optimization method, as shown in Algorithm 1, which is a combination of heuristics and search which allows it to find effective results quickly.

5 Performance Evaluation

In this section, we evaluate the proposed system performance, including end-to-end speedup and GPU utilization enhancement targeting multi-tenant batched-job tasks [30].

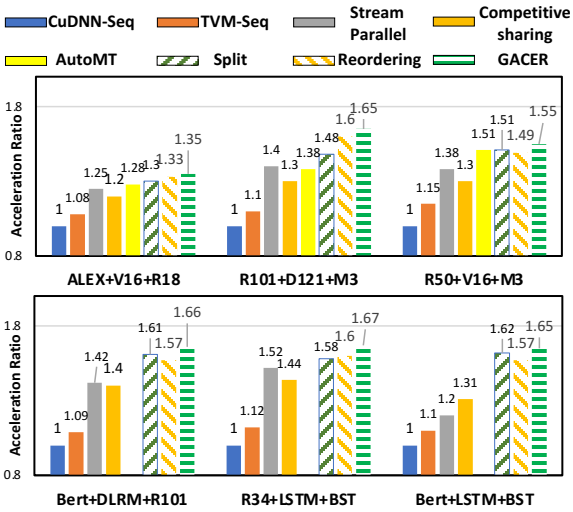

Figure 7. Runtime Performance of *GACER* (with A100)

5.1 Multi-Tenant Setting

Model Type: In this work, we selected three types of DNN models for multi-tenant combinations, namely, *convolutional-based vision CNN models* (including AlexNet (Alex), VGG16 (V16), ResNet18 (R18), ResNet34 (R34), ResNet50 (R50), ResNet101 (R101), MobileNetV3 (M3) and DenseNet (D121)); *language model* (including, BERT-base [31], LSTM [32]), and *recommendation model* (including DLRM [33], BST [34]).

Model Workload: In this work, we manipulate the workload mainly with the computing batch size, which is the most fundamental workload regulation unit, that could also demonstrate other potential workload factors such as input sizes. Specifically, for vision models, we select a batch size range of 4 to 128 with an image scale of 224x224x3, which covers most of the intervals in the inference. For the language model, we use the emotion classification dataset ML2020spring with batch size 128. In the case of recommendation models, we chose batch size 64 on the Amazon dataset *Book* [35].

Hardware Variation: We take into consideration three different GPUs, namely, 1080Ti, P6000, and A100.

State-of-the-Arts: Given different hardware platforms, we adopted several state-of-the-art works for comparative evaluation. Regarding particular resource allocation optimization and runtime scheduling strategies. *CuDNN-Seq* [36] is the most fundamental method, which only relies on the default sequential graph compiling of Pytorch and basic CuDNN operator optimization. *TVM-Seq* [37] is an operator optimization method that adopts the TVM library to search for the optimal kernel for each operator. However, it is limited to sequential execution of these kernels. *Stream-Parallel* [12] is the concurrent execution strategy from native GPU multi-stream support. *Competitive-Sharing* [14] is a spatial technique based on MPS and allocates the resources to each model based on the models' FLOPS. This method is used to verify the efficiency of a static resource allocation method. *AutoMT* [19] coordinates DNN computing on the runtime level and maintain a continuously balanced resource utilization across the entire inference process. However, this method only supports convolutional networks.

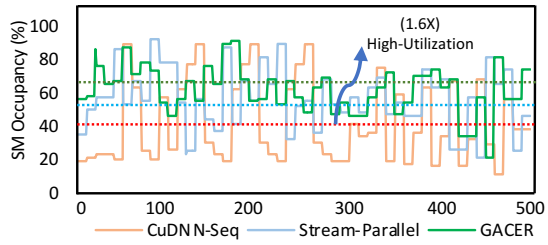

Figure 8. Analysis on GPU Utilization Enhancement

5.2 Overall *GACER* Performance

Before analyzing the four factors, we first show the *GACER* general performance by comparing the general latency of the baselines and our methods *GACER*. To demonstrate the effectiveness of each design component, we decompose and evaluate our method step by step, i.e., using spatial granularity regulation (*Split*), the method only using temporal granularity regulation (*Reordering*), and the combined optimization (*GACER*). The results are shown in Fig. 7. All latency is normalized by the CuDNN-Seq baseline to show the relative acceleration ratio.

Based on the results, it can be observed that our framework *GACER* could consistently yield 1.35×~1.67× speed-up compared to the sequential baselines across all six model combinations. The Stream-Parallel solution also yields a certain speed-up than CuDNN-Seq, but the acceleration ratio is much less, usually 1.24×~1.51×. Also, the Competitive-Sharing acceleration effect is very unstable, specifically due to the fact that fixed resource allocation cannot satisfy many particularly unbalanced model workload scenarios. AutoMT only supports the CNN model, so it does not yield results on some model combinations. However, even on the CNN model combination, it performs poorly in combinations with simpler models, such as only 1.25× acceleration on the first combination.

To delve deeper into the performance of model combinations under parallelization, we analyze GPU runtime statistics to assess overall GPU utilization in various scenarios. We use the achieved SM Occupancy from NVIDIA NSight Profiler as an indicator metric of GPU utilization information. Fig. 8 presents a comprehensive comparison of utilization statistics among CuDNN-Seq, Stream-Parallel, and *GACER*, on the R101+D121+M3. As observed, our method significantly improves utilization, achieving approximately 60% higher utilization than the sequence method and almost 40% more than the Stream-Parallel method. This enhancement aligns well with our observed speed-up performance.

5.3 *GACER* with Model Type Combinations

To delve deeper into *GACER* performance with various model combinations, we quantify the utilization of resizing and reordering techniques in each combination. This helps to discern which model types are more amenable to these methods. We examine four distinct model combinations, with the findings illustrated in Fig. 9. The

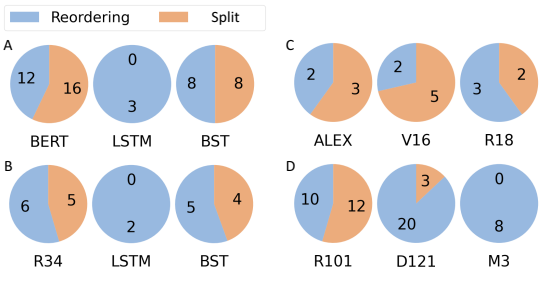

Figure 9. Analysis of Split and Reordering Usage

Table 2. Scalability Evaluation (GPU: A100, Latency: ms)

Models	Names	CuDNN-Seq	TVM-Seq	Stream-Parallel	Com-Sharing	*GACER*
2× models	V16 + LSTM	4.32	4.2 (1.03x)	3.81 (1.13x)	3.89 (1.11x)	3.30 (**1.31x**)
	R18 + R34	5.12	4.08 (1.25x)	3.91 (1.31x)	3.93 (1.3x)	3.46 (**1.48x**)
	R34 + R50	7.52	6.59 (1.14x)	6.10 (1.23x)	5.99 (1.26x)	5.01 (**1.50x**)
	R50 + R101	13.73	13.39 (1.03x)	11.54 (1.19x)	11.52 (1.19x)	9.80 (**1.40x**)
4×	V16 + R18 + M3 + LSTM	9.42	9.23 (1.02x)	7.54 (1.25x)	8.02 (1.17x)	6.51 (**1.48x**)
	M3 + R34 + R50 + BST	18.36	15.35 (1.2x)	13.30 (1.38x)	14.01 (1.31x)	11.27 (**1.63x**)
5×	VGG + R18 + R34 + BERT + DLRM	37.86	32.24 (1.17x)	25.31 (1.5x)	27.22 (1.39x)	22.14 (**1.71x**)

Table 3. *GACER* Optimization Cost (s)

#Batch Size	4	8	16	32	64	128
R34+V16+D121	0.90	0.92	0.98	1.04	1.08	1.10
R50+V16+M3	1.10	1.13	1.19	1.22	1.25	1.26
R34+LSTM+BST	0.84	0.85	0.88	0.90	0.93	0.94

Table 4. GPU Generality Evaluation (ms). C is CuDNN-Seq and S is Stream-Parallel, P is on GPU P6000 and T is on GPU 1080Ti.

Models	C-P	C-T	S-P	S-T	*GACER*-P	*GACER*-T
ALEX+V16+R18	18.74	19.56	14.99	15.28	13.48(**1.39x**)	14.81(**1.32x**)
D121+V16+LSTM	17.83	18.02	14.73	15.27	12.92(**1.38x**)	13.54(**1.33x**)
R50+V16+M3	28.54	32.88	20.88	23.48	19.02(**1.50x**)	21.07(**1.56x**)
R101+D121+M3	40.51	44.89	29.35	32.06	25.63(**1.58x**)	27.37(**1.64x**)
R34+LSTM+BST	12.35	14.50	8.23	10.13	7.97(**1.55x**)	8.51(**1.70x**)

optimization strategies applied to different combinations of model types can vary dramatically, Spatial granularity adjustment is particularly beneficial for models with extensive operator workloads. In combination A, both BERT and BST, are transformer-based models, in particular, the BERT model stacks a structure of 12 encoders with resource-intensive multi-head attention and feed-forward operations. In such scenarios, coarse deployment granularity can lead to significant resource contention, necessitating using resizing to mitigate deployment granularity. Therefore, as many as 28 reordering and split are applied to BERT and 16 to BST. On the other hand, temporal granularity regulation significantly boosts the efficiency of model combinations with a higher layer number. An exemplary case is the R101+D121+M3, due to the large number of operators, reordering is used up to 38 times.

5.4 *GACER* with Model Number Variation

Another key factor impacting multi-tenant DNN scenarios is the number of concurrent models. To analyze this, we thoroughly evaluate the scalability of *GACER* under various model inference loads on a single GPU. Our tests include configurations of 2×, 4×, and 5× models, encompassing several multi-tenant combinations.

Our findings, as detailed in Table 2, demonstrate *GACER* robust scalability with varying numbers of tenants. Notably, the framework consistently achieves acceleration ranging from 1.31× to 1.71× compared to the sequential baseline across all evaluated benchmarks. A key factor behind this is *GACER* ability to rapidly adjust the number of models by setting the initial number of GPU streams. Furthermore, our heuristic optimization algorithm efficiently computes residuals based on the current set of concurrent operators.

5.5 *GACER* Optimization Overhead with Workload Variation

The efficiency of optimization is a crucial metric in evaluating an algorithmic framework. A key factor influencing this efficiency is the change in workload, which can vary significantly in different applications and use cases. Our comprehensive examination, as detailed in Table 3, looks at the optimization times required by *GACER* across various batch sizes and model combinations. The data indicates that, regardless of the workload's complexity or the batch size, the optimization time required by *GACER* is consistently brief, remaining within a few seconds. *GACER* is flexible for both online and offline settings, capable of pre-optimization in static or infrequently changed models to reduce runtime overhead, and also suitable for online tasks that can tolerate minor delays.

5.6 *GACER* with Hardware Variation

We then evaluate the generality of our method with different GPU platforms. We conduct extensive tests on five multi-tenant setups using two distinct GPUs: the NVIDIA P6000 (P) and the NVIDIA 1080Ti (T). The results are detailed in Table 4, which compares the performance of CuDNN-Seq (C) and Stream-Parallel (S). We standard the batch sizes 8 for each vision model. In these tests, *GACER* demonstrates a significant performance improvement. On the P6000, we observe an acceleration ranging from 1.38× to 1.58×, while on the 1080Ti, the acceleration varied from 1.32× to 1.70×.

Interestingly, the performance gains exhibited by *GACER* are more pronounced on the 1080Ti. This suggests that *GACER* optimization mechanisms are particularly effective in environments where resource constraints are more stringent, thereby offering greater relative improvements in such scenarios. When platform resources are constrained, their ability to be concurrent and deployed is reduced. Without granularity optimization, its resource utilization drops severely, especially when faced with large models.

6 Conclusion

In this work, we focus on multi-tenant deep learning for GPU-based systems and reveal several computing issues. We find that the granularity of computing management in both the spatial and temporal management domains is extremely important. Based on the optimization of granularity, we proposed *GACER*, an automated optimization framework that provides high-utilization, high-throughput, and low-latency multi-tenant deep learning computing support. This framework is a revolutionary approach to multi-tenant deployment and can provide a solid foundation for the future development of multi-tenant deep learning.

References

[1] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[2] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE micro*, 31(5): 7–17, 2011.

[3] Market Reports. Global data center accelerator market size, status and forecast 2020-2025, 2021. https://www.mynewsdesk.com/brandessence/pressreleases/data-center-accelerator-market-size-2021-cagr-38-dot-7-percent-3112488.

[4] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems*, 1:27–39, 2019.

[5] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems*, 3:167–180, 2021.

[6] Shaohui Lin, Rongrong Ji, Chenqian Yan, Baochang Zhang, Liujuan Cao, Qixiang Ye, Feiyue Huang, and David Doermann. Towards optimal structured cnn pruning via generative adversarial learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2790–2799, 2019.

[7] 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.

[8] Shaoshan Liu, Liangkai Liu, Jie Tang, Bo Yu, Yifan Wang, and Weisong Shi. Edge computing for autonomous driving: Opportunities and challenges. *Proceedings of the IEEE*, 107(8):1697–1716, 2019.

[9] Magdalini Eirinaki, Jerry Gao, Iraklis Varlamis, and Konstantinos Tserpes. Recommender systems for large-scale social networks: A review of challenges and solutions, 2018.

[10] Stylianos Mystakidis. Metaverse. *Encyclopedia*, 2(1):486–497, 2022.

[11] NVIDIA. Multi-Process Service, 2021. URL https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[12] NVIDIA. Multi-Stream, 2020. URL https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[13] NVIDIA. Nvidia multi instance gpu (mig). 2020.

[14] Yongbo Yu, Fuxun Yu, Zirui Xu, Mingjia Zhang Di Wang, Ang Li, Shawn Bray, Chenchen Liu, and Xiang Chen. Powering multi-task federated learning with competitive gpu resource sharing. 2022.

[15] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. Lazy batching: An sla-aware batching system for cloud machine learning inference. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 493–506. IEEE, 2021.

[16] Oscar Koller, Necati Cihan Camgoz, Hermann Ney, and Richard Bowden. Weakly supervised learning with multi-stream cnn-lstm-hmms to discover sequential parallelism in sign language videos. *IEEE transactions on pattern analysis and machine intelligence*, 42(9):2306–2320, 2019.

[17] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. Nimble: Lightweight and parallel gpu task scheduling for deep learning. *Advances in Neural Information Processing Systems*, 33:8343–8354, 2020.

[18] Aditya Dhakal, Sameer G Kulkarni, and KK Ramakrishnan. Gslice: Controlled spatial sharing of gpus for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 492–506, 2020.

[19] Fuxun Yu, Shawn Bray, Di Wang, Longfei Shangguan, Xulong Tang, Chenchen Liu, and Xiang Chen. Automated runtime-aware scheduling for multi-tenant dnn inference on gpu. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[20] Qiumin Xu, Hyeran Jeon, Keunsoo Kim, Won Woo Ro, and Murali Annavaram. Warped-slicer: Efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 230–242. IEEE, 2016.

[21] Peifeng Yu and Mosharaf Chowdhury. Fine-grained gpu sharing primitives for deep learning applications. *Proceedings of Machine Learning and Systems*, 2: 98–111, 2020.

[22] Marcus Chow, Ali Jahanshahi, and Daniel Wong. Krisp: Enabling kernel-wise right-sizing for spatial partitioned gpu inference servers. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 624–637. IEEE, 2023.

[23] Ziyang Zhang, Huan Li, Yang Zhao, Changyao Lin, and Jie Liu. Pos: An operator scheduling framework for multi-model inference on edge intelligent computing. In *Proceedings of the 22nd International Conference on Information Processing in Sensor Networks*, pages 1–1, 2023.

[24] Daniel Zhang, Nathan Vance, Yang Zhang, Md Tahmid Rashid, and Dong Wang. Edgebatch: Towards ai-empowered optimal task batching in intelligent edge systems. In *2019 IEEE Real-Time Systems Symposium (RTSS)*, pages 366–379. IEEE, 2019.

[25] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[26] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Multi-model machine learning inference serving with gpu spatial partitioning. *arXiv preprint arXiv:2109.01611*, 2021.

[27] Aodong Chen, Fei Xu, Li Han, Yuan Dong, Li Chen, Zhi Zhou, and Fangming Liu. Opara: Exploiting operator parallelism for expediting dnn inference on gpus. *arXiv preprint arXiv:2312.10351*, 2023.

[28] NVIDIA. NVIDIA Nsight Systems, 2020. URL https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[29] NVIDIA. NVIDIA Nsight Computing, 2020. URL https://developer.nvidia.com/nsight-compute.

[30] Sheng-Chun Kao and Tushar Krishna. Magma: An optimization framework for mapping multiple dnns on multiple accelerator cores. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 814–830. IEEE, 2022.

[31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[32] Yong Yu, Xiaosheng Si, Changhua Hu, and Jianxun Zhang. A review of recurrent neural networks: Lstm cells and network architectures. *Neural Computation*, 31 (7):1235–1270, 2019.

[33] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[34] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. Behavior sequence transformer for e-commerce recommendation in alibaba. In *Proceedings of the 1st International Workshop on Deep Learning Practice for High-Dimensional Sparse Data*, pages 1–4, 2019.

[35] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep interest network for click-through rate prediction. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 1059–1068, 2018.

[36] NVIDIA. Nvidia CuDNN Documentation,, 2020. URL https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html.

[37] 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.