Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU

Fuxun Yu[1], Shawn Bray[2], Di Wang[3], Longfei Shangguan[3], Xulong Tang[4], Chenchen Liu[2] and Xiang Chen[1]
[1]*George Mason University, Fairfax, VA, USA*
[2]*University of Maryland, Baltimore County, Baltimore, MD, USA*
[3]*Microsoft, Redmond, WA, USA*
[4]*University of Pittsburgh, Pittsburgh, PA, USA*
[1]{*fyu2, xchen26*}*@gmu.edu,* [2]{*shawnb2, ccliu*}*@umbc.edu*
[3]{*wangdi, longfei.shangguan*}*@microsoft.com,* [4]*tax6@pitt.edu*

2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD) | 978-1-6654-4507-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCAD51958.2021.9643501

***Abstract*—With the fast development of deep neural networks (DNNs), many real-world applications are adopting multiple models to conduct compound tasks, such as co-running classification, detection, and segmentation models on autonomous vehicles. Such multi-tenant DNN inference cases greatly exacerbate the computational complexity and call for comprehensive collaboration for graph-level operator scheduling, runtime-level resource awareness, as well as hardware scheduler support. However, the current scheduling support for such multi-tenant inference is still relatively backward. In this work, we propose a resource-aware scheduling framework for efficient multi-tenant DNN inference on GPU, which automatically coordinates DNN computing in different execution levels. Leveraging the unified scheduling intermediate representation and the automated ML-based searching algorithm, optimal schedules could be generated to wisely adjust model concurrency and interleave DNN model operators, maintaining a continuously balanced resource utilization across the entire inference process, and eventually improving the runtime efficiency. Experiments show that we could consistently achieve 1.3×∼1.7× speed-up, comparing to regular DNN runtime libraries (e.g., CuDNN, TVM) and particular concurrent scheduling methods (e.g., NVIDIA Multi-Stream).**

I. Introduction

As deep neural networks (DNNs) have demonstrated superior performance in vast cognitive tasks [1–3], the expectations for DNN-powered intelligence have grown rapidly over the past few years. In addition to the real-time needs of DNN optimization regarding its deep structures and heavy workloads [4–6], recent real-world applications further require *multi-tenant DNN computation* for even compound tasks [7–9]. For example, it is critical for an autonomous driving system to inference multiple DNN models simultaneously on the same hardware for segmentation [10], detection [11], and classification [12]. And for larger-scale cases, such multi-tenant computing necessity also emerges in cloud computing clusters and industrial-level data centers for resource utilization improvement, drawing significant attention from intelligence services providers, such as Microsoft and NVIDIA [13–15].

The ***multi-tenant DNN inference*** exacerbates the computational complexity on top of existing DNN problems. However, the corresponding computing support is still relatively backward. As the major platform for the multi-tenant inference — current GPUs' computing strategies are still limited to traditional approaches of *sequential execution* (e.g., MPI-processing [16]) and *parallel/concurrent execution*[1] (e.g., NVIDIA multi-Stream execution [17]).

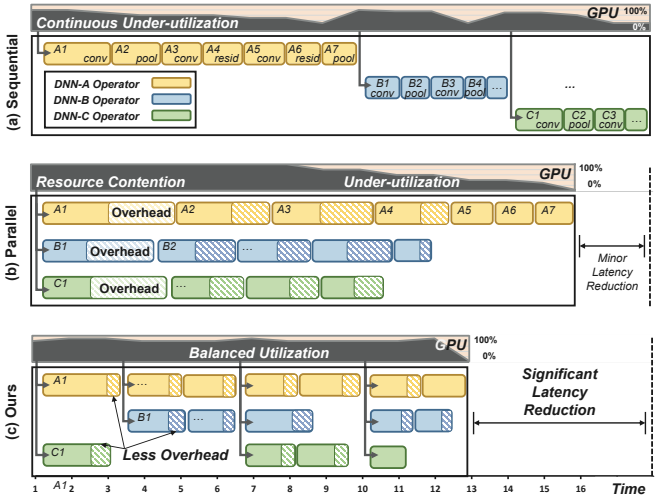


Fig. 1. Scheduling for Multi-tenant DNN Inference on a Single GPU

As demonstrated in Fig. 1, these limited strategies cannot achieve satisfying performance for multi-tenant DNN inference: **(a)** Although the sequential execution dedicates the entire GPU's resource to each model and achieves the shortest per-model inference latency as shown in Fig. 1 (a), continuous ***resource under-utilization*** is inevitable due to the single-operator execution (e.g., *conv*, *pooling*), not to mention the cumulative overall ***runtime latency***. **(b)** For the concurrent execution in Fig. 1 (b), though indispensable parallelism for multiple models earns latency optimization to a certain degree, it hasn't touched the particular computing complexity in multi-tenant DNN inference. Taking the first round of convolution from the three DNN models (i.e., *A1*, *B1*, *C1*) as an example, simple parallelism would introduce considerable ***contention overhead*** as operators can compete for computing resources simultaneously. While looking into later stages of this concur-

This work is partially supported by NSF CNS-2003211 and CNS-1939380.

[1]We treat "parallel" and "concurrent" as the same meaning in our work, according to the definition of "concurrency" in the NVIDIA document [17].

978-1-6654-4507-8/21/$31.00 ©2021 IEEE

rent execution, GPU under-utilization strikes back due to the ***unbalanced scheduling*** for different model depths.

Thus, to strive for optimal runtime latency and resource utilization, the multi-tenant DNN inference raises particular GPU scheduling requirements not only for analyzing and relieving *local* operator contention, but also for managing *global* model concurrency balance as per model structure divergence. Bringing this "*local-global*" need into the existing DNN execution stack as shown in Fig. 2, we can see that, it calls for comprehensive collaboration from ***the graph-level operator scheduling***, ***the runtime-level resource awareness***, as well as ***the hardware scheduler support***. However, most existing DNN scheduling methods are limited in a single-level optimization scope. For example, many works are proposed singularly for low-level intra-operator optimization, such as loop tiling and unrolling [18–20]; Similarly, many graph-based scheduling works focus only on high-level inter-operator fusion/substitution optimization [21–23]. As a result, neglecting one or the other, these methods fail to meet the cross-level scheduling requirement by the multi-tenant DNN inference.

In this work, we propose a *runtime-aware scheduling framework for efficient multi-tenant DNN inference on GPU*, which *automatically* coordinates concurrent DNN computing in different execution levels. As shown in Fig. 1 (c), the proposed method could take both the local operator contention and the global model structural divergence into consideration. The final scheduling method wisely adjusts model concurrency by interleaving operators for less contention overhead, maintaining a continuously balanced resource utilization across the entire inference process, and eventually improving the runtime efficiency. To achieve such a scheduling target, we make the following contributions:

- We first *abstract the multi-tenant DNN inference scheduling* as a fine-grained concurrency control problem. Incorporating the GPU multi-stream and synchronization mechanisms, multiple concurrency control levels are identified in the GPU inference flow to provide the fundamental support for the scheduling optimization;
- Based on the problem abstraction, *a unified scheduling Intermediate Representation (IR)* is specified to formulate the scheduling factors by taking both graph-level and the runtime-level execution mechanism into consideration, and eventually build a structural search space for the final scheduling optimization;
- In the established scheduling search space, we transform multi-tenant scheduling into an optimization problem and propose an automated ML-based searching algorithm to find the optimal scheduling strategy on GPU. Specifically, the GPU runtime resource is profiled and adopted as the searching cost, granting the whole solution with expected runtime awareness.

We conduct extensive experiments across a wide range of multi-tenant inference scenarios. The results show that our method could consistently achieve 1.3×∼1.7× acceleration than the common deep learning runtime libraries (e.g., CuDNN, TVM) and other concurrent scheduling methods (e.g., NVIDIA Multi-Stream). Meanwhile, benefited from the end-to-end search method design, our method could be easily applied onto 10s of multi-tenant combinations and GPU platforms with short search time (∼2mins), demonstrating the great scalability of our automated scheduling framework.

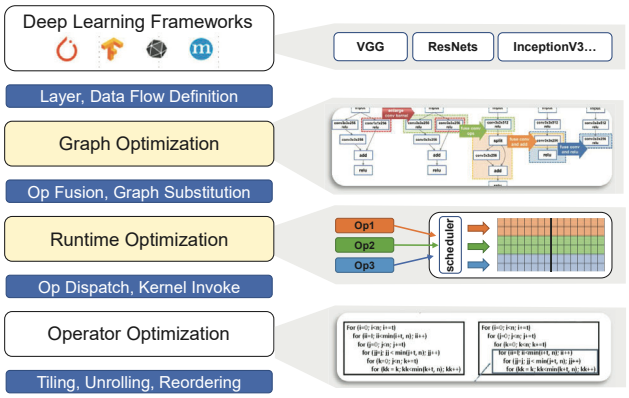


Fig. 2. DNN execution stack. Our work proposes a graph- and runtime-level cross-layer scheduling framework for multi-tenant inference optimization.

II. Backgrounds and Motivation

A. Cross-Level Scheduling through DNN Execution Stack

We first expand the backgrounds of DNN execution stack as shown in Fig. 2, that is composed of multiple architecture levels [18]: **(a)** The top *framework level* includes different deep learning development frameworks, such as TensorFlow and PyTorch, that define various DNN model structures. **(b)** The *graph level* untangles model structures to abstract individual operators and identify the data processing flow as directed acyclic graphs (DAG) [21]. Graph-based optimization is thus introduced into this level to achieve operator fusion and sub-graph substitution, and therefore reduce memory access/operator invoking overheads, etc. **(c)** Down to the *runtime level*, it controls when and how operators are dispatched onto physical computing units and is critical in our balanced resource utilization. This is usually done by the native black-box GPU scheduler, but we could leverage certain APIs to adjust the dispatching results. In our work, we use the "stream" [17] and "synchronization" [24] APIs to achieve fine-grained operator concurrency control as we will show later. **(d)** The *operator level* is the bottom level that conduct per-operator execution, such as tiling, unrolling, reordering, etc., to improve the computing efficiency. Such intra-operator optimization has a distinct scope and is orthogonal to the concurrent operator scheduling, and thus is not considered in this work.

Motivation: As existing single-level works (e.g., graph-alone, operator-alone) can hardly offer comprehensive solution, we aim to bridge different levels in this work and build a cross-level scheduling framework to improve the multi-tenant inference performance from graph to runtime.

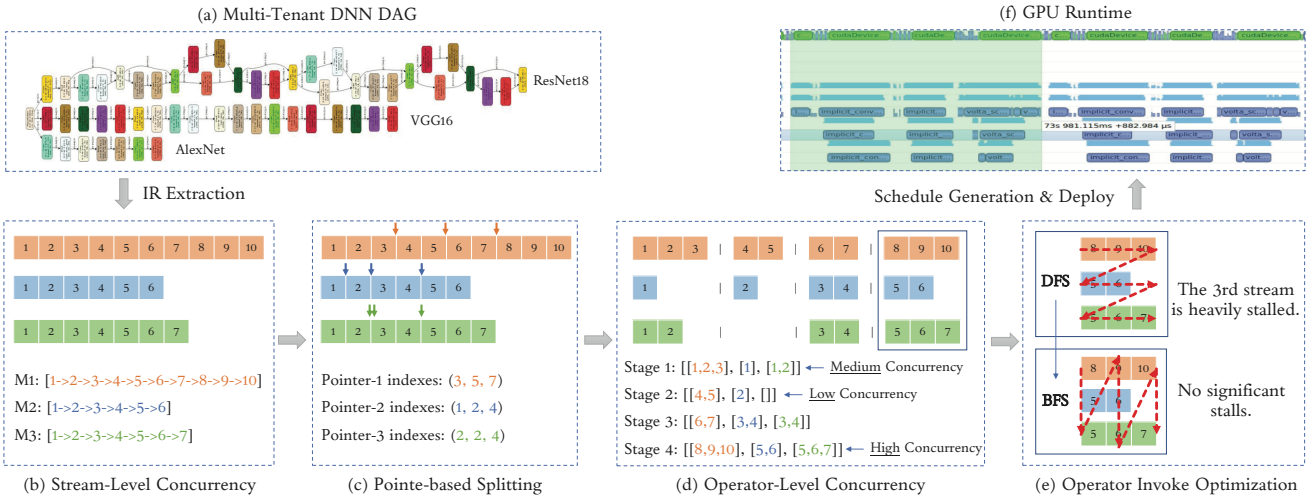


Fig. 3. Overview of Our Proposed Automated Scheduling Strategy Search Framework.

B. Resource Contention in Multi-Tenant DNN Inference

Resource contention is a specific challenge emerging in the multi-tenant inference. Different from single-model inference which usually faces under-utilization issues, resource contention reflects the operator competition for limited hardware resources. Specifically, there are two types of contention that are commonly known: *computing contention* and *memory contention*. Generally, convolutional operators tend to be compute-bound as it involves mostly FLOPs-intensive computing, while the operators like pooling/residual connections are usually memory-bound. When executed concurrently, same type of operators can saturate the corresponding resources, and cause slower execution speed due to the limited size of shared memory and L1 cache, memory bandwidth, etc.

Motivation: Thus, multi-tenant scheduling is a finer concurrency control problem, which not only concerns under-utilization but also contention issues.

C. Scheduling Complexity in Multi-Tenant DNN Inference

Moreover, multi-tenant inference optimization have a much larger complexity: **(a)** From a spatial perspective for inference parallelism, on each single execution stage, the overall amount of operators from different models is much more significant than a single-model scheduling scope. **(b)** From a temporal perspective, through the whole execution process, different model structures and depths also raise considerable scheduling challenges to maintain consistent and balanced resource utilization and improve the runtime latency. Therefore the design/search space of the scheduling strategy for multi-tenant DNN inference becomes ever complex, and conventional manual tuning or heuristic-based methods are hard to scale and reach satisfying performance.

Motivation: In addition to other design motivations, we will eventually solve this problem by proposing an efficient search space representation and leverage automated ML-based methodologies to coordinate massive operators for optimal resource utilization and runtime latency.

III. The Scheduling Framework

A. Fine-grained Scheduling Problem Abstraction

This work targets at efficient multi-tenant DNN inference on GPUs. Considering the applications such as autonomous driving systems, we specify it as a compound task consisting of N independent DNN models sharing the same input for different inference sub-tasks. Demonstrated as Fig. 3 (a), each DNN inference sub-task consists of a series of operators, such as *conv*, *bn*, *relu*, *pooling*, etc, which must be performed in certain order according to the data flow dependency. While across DNN models, operators are independent and thus could be flexibly scheduled with certain degrees of concurrency. Our optimization objective is to minimize the overall latency of N inference sub-tasks, which is the overall time from the earliest starting time of the tasks to the latest ending time.

The key of multi-tenant scheduling is to manage the concurrency for consistent and balanced resource utilization. Therefore, we abstract the multi-tenant DNN inference scheduling as a fine-grained concurrency control problem through the following steps: **(a)** *Achieving the* ***stream-level*** *concurrency*: We allocate one *GPU processing stream* for each model to achieve the concurrency (Fig. 3 (b)). However, even with certain concurrency, native GPU stream-based scheduling dispatches operators without dedicated scheduling management. **(b)** *Finer-grained* ***stage splitting***: To achieve finer-grained operator-level concurrency control, we insert synchronization barriers, namely ***pointers***, to split each stream's operator sequence into several shorter stages (Fig. 3 (c)). Such stage splittings ensure the operators to only share the assigned resources in the same stage, thus supporting the stage-level concurrency control. **(c)** ***Stage-level*** *concurrency control*: By adjusting where the pointers are inserted, we could control how many operators are assigned in each stage. This enable us to reduce or increase the concurrency in a fine-grained manner to manage the resource utilization (Fig. 3 (d)). **(d)** ***Intra-stage*** *operator invoking optimization*: After deciding the scheduling strategy, our final step is the scheduling deployment. During

this implementation, we also optimize the operator invoking logic to prevent the invoking overhead of early streams from stalling later ones (3-e), as we will introduce later.

B. Unified Intermediate Representation Design

As a multi-tenant DNN inference task consists of N parallelable models: $M_1, M_2, ..., M_N$. We represent each DNN model by one stand-alone operator sequence[2]:

$$\begin{aligned} \mathrm{M}_1 &: [1,\ 2,\ ...,\ a], \\ \mathrm{M}_2 &: [1,\ 2,\ ...,\ b], \\ \mathrm{M}_N &: [1,\ 2,\ ...,\ c], \end{aligned} \tag{1}$$

where M indicates a DNN model, each number in one list indicates one operator's index, and a (or b, c) is the largest index of the DNN's operators.

Stream: To satisfy the sequential dependency per model, we assign each model to one stand-alone *GPU processing stream*:

$$S_i \leftarrow \mathrm{M}_i, \quad i \in (1, 2, ..., N), \tag{2}$$

where S_i indicates the i-th stream. An example with three streams is shown in Fig. 3 (b). Operators in one stream can only be launched sequentially, while operators in different streams could be executed concurrently.

The multi-stream mechanism enables the maximum concurrency of DNN inference streams. However, as aforementioned, scheduling by streams alone can only have *stream-level concurrency control*, which is still coarse-grained and does not suffice to manage each operator's associated concurrency during its life span. To control the concurrency in a finer granularity, we then use synchronization barriers to split each stream's full sequence into several shorter stages.

Pointer: We use *pointers* to annotate the appropriate positions where we insert synchronization barriers. An illustrated pointer-based stage splitting is shown in Fig. 3 (c)(d). Taking the first stream as an example, a pointer set with three pointers divides the first stream sequence into four shorter ones:

$$\begin{aligned} \rho_1 : (\mathbf{3,\ 5,\ 7}) \ +\ \mathrm{S}_1 : [1, 2, 3, ..., 9, 10] = \\ \mathrm{S}_1^{'} : [1, 2, 3], [4, 5], [6, 7], [8, 9, 10], \end{aligned} \tag{3}$$

where ρ_1 is the pointer indexes, S_1 is the original operator sequence, and $S_1^{'}$ is the split sequence with synchronization barriers inserted. Each pointer set splits one stream sequence into several shorter ones, thus enabling a finer-grained concurrency scheduling.

Stage: Between each two pointers, the launched operators form a *stage*. Due to the sync barriers, all operators in the same stage must all finish so as to step into the next stage. Thus, by controlling how many operators are launched in each stage, we could precisely manage the concurrency in the most fine-grained operator level. An example is given in Fig. 3 (d). By inserting the first synchronization barrier, we could enable six operators to concurrently execute in the first stage:

$$\text{Stage 1} : [S_1(1, 2, 3),\ \ S_2(1),\ \ S_3(1, 2)]. \tag{4}$$

By contrast, we could also reduce the concurrency in the second stage by assigning no operators in the third stream:

$$\text{Stage 2} : [S_1(4, 5),\ \ S_2(2),\ \ S_3(\text{None})]. \tag{5}$$

Similarly, all stages can be generated with a desired concurrency, thus enabling *operator-level concurrency control*.

Schedule: The final scheduling strategy is composed of multiple stages in the synchronization barriers' ordering, which is represented as a multi-stage nested list:

$$\text{Schedule } \tau : [\text{Stage 1},\ \text{Stage 2},\ \text{Stage 3}, ...], \tag{6}$$

where τ indicate the composed scheduling strategy, which can have multiple stages, depending on the number of synchronization barriers (i.e., pointers) we used to split each stream sequence. Fig. 3 (d) shows an example that uses three sync pointers for four stages. More synchronization enables finer-grained concurrency control, but at the price of potentially higher synchronization overhead.

C. Automated Scheduling Search

The IR design explicitly defines the scheduling factor and the corresponding strategy for multi-tenant GPU inference. However, it is still challenging to identify the particular scheduling controls given various compound tasks with uncertain DNN structures. As aforementioned, considering the complexity, manual schedule tuning can take considerable efforts and also cannot scale with more complicated models' combination and varied GPU platforms. Therefore, we propose to use an ML-based search approach to solve the scheduling problem in an automated manner.

Formulation: Formally, our primary search target is to find an optimal scheduling strategy that yields the lowest latency:

$$\tau^* = \arg\min_{\tau}\ f(\tau), \quad \text{for } \tau \in D_\tau, \tag{7}$$

where τ^* is the optimal scheduling strategy, f is the cost model that evaluates the latency of the current schedule τ, and D_τ is the search space of all potential schedules. Specifically, to solve this search problem, three basic components need to be clarified, namely, the search space, the cost model and the searching algorithm.

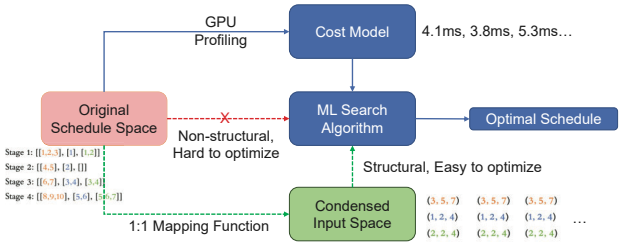


Fig. 4. The automated scheduling search framework overview.

[2]For multi-branch models like ResNets, we also serialize the operators into one sequential sequence as their *intra-model concurrency* is limited. Such a representation enables us to better optimize the *inter-model concurrency* in the multi-tenant inference scenario.

Search Space is supposed to enumerate all possible scheduling strategy candidates. To represent such a search space, we adopt the scheduling factors from the proposed IR design (i.e., streams, pointers and stages).

As defined in Eq. 6, τ can be formulated as a nested list and can be treated as a graph-level scheduling strategy. Although such a nested list is easy to understand and facilitates the deployment process onto GPU, the list-based search space D_τ is non-structural with varied list lengths and can be hard to directly optimize. To solve this problem, we leverage the one-to-one mapping property between pointer indexes ρ and the schedule lists τ, and shrink the search space to a lower-dimensional pointer index matrix by building an 1:1 schedule mapping function, as shown in Fig. 4:

$$\begin{aligned} &\rho^* = \arg\min_{\rho} \ f(\tau), \\ &\text{s.t.} \ \ \tau = T(G, \rho), \quad \text{for } \rho \in D_\rho. \end{aligned} \tag{8}$$

Here the scheduling generation function $T(\cdot)$ generates one schedule τ based on two inputs: the graph G and the pointer matrix ρ. As G is usually fixed in a given task, the schedule generation function $T(\cdot)$ maps each pointer matrix to one schedule. Thus, searching schedule could be transformed to searching the pointer index matrix, the latter of which has a much more structural input space. By such transformation, we could thus greatly reduce the optimization difficulty.

Cost Model: With the search space defined, we then require a cost model $f(\tau)$ to evaluate the performance of each schedule candidate. There are two major ways to construct the cost model: modeling-based or profile-based. The modeling-based method [22] builds hardware-specific modeling to estimate the real runtime performance, which is efficient but can be inaccurate in complex scenarios. The profiling-based method [18] is more accurate but requires physical hardware execution, which can be more time-consuming if the search space is very large.

Algorithm 1 Coordinate Descent Search Algorithm.

Input: The IR of N models $M[N]$, the number of pointers in each model P, the rounds of search R.
Output: The optimal pointer matrix ρ $[N, P]$.
Initialize a dictionary $D\{\text{schedule:cost}\}$ to store records.
for rounds $r = 1$ **to** R **do**
  for model $i = 1$ **to** N **do**
    Sample M candidates $\rho_{1:M}[i]$ for the i-th row.
    for the m-th candidate $\rho_m[i]$ **do**
      Profile the latency lat_m by multiple runs.
      Append $\{\rho_m : lat_m\}$ to the records D.
    end for
    Update the i-th row $\rho[i]$ of pointer matrix to the one with the lowest latency by $\arg\min(lat_m)$.
  end for
end for
Sort the global records D by the profiled latency.
Return the schedule ρ with the globally lowest latency.

In this work, we use the profiling-based cost model since our empirical case study shows that, our searching time can be maintained at small scale ($\sim$mins) benefited from our dedicated search space abstraction. Therefore, the profiling-based model can give accurate runtime-aware performance cost and lead to better search performance in our case. For the cost model implementation, we leverage our built infrastructure, which could efficiently generate and deploy each candidate schedule onto the target GPU and obtained the profiled latency during multiple averaged runs. The averaged latency is then used as the cost of each candidate schedule.

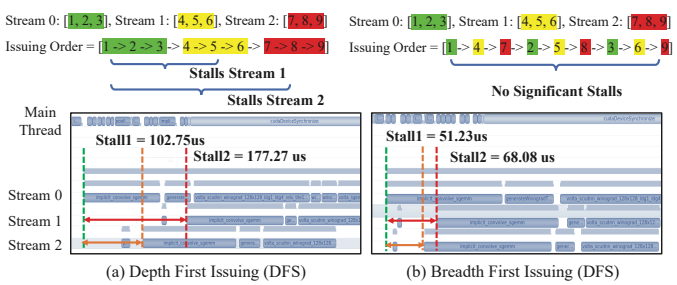
Search Algorithm: With the input space and cost model defined, we could then use ML-based methods to search for the optimal schedule with the minimal latency.

In this work, we mainly implement two search algorithms, the random search and the coordinate descent search. The random search method samples scheduling solutions (different pointer matrices) randomly from the search space and profiles their latency as the cost. A memory module will record all schedules and costs, and after certain rounds of search, the algorithm will return the schedule with the lowest latency. As we will show later, though the random search algorithm is simple, it could greatly reduce the multi-tenant runtime latency by large margins, highlighting the advantages of our problem abstraction and the search framework design.

Based on a similar process, the coordinate descent search algorithm improves the sampling efficiency by adopting a coordinate-alternated search philosophy. The overview of the coordinate descent search algorithm is shown in Algorithm 1. It treats different streams' pointer index vectors (rows in the pointer matrix) as different coordinates. Then it alternatively finds the optimal pointer index vector for each coordinate, during when other coordinates' solution are kept as the previous optimal one. The optimal pointer index vectors for all streams are updated for each round, and after certain rounds, the algorithm returns the optimal schedule from all previously searched schedules. Generally, the coordinate descent search algorithm could yield slightly performance than random search algorithm. But both methods could yield near optimal schedule solutions within short time, as we will evaluate later.

D. Implementation Optimization

After determining the optimal schedule, we can deploy the schedule onto the GPUs. This is done by invoking the



(a) Depth First Issuing (DFS) (b) Breadth First Issuing (DFS)

Fig. 5. Long-sequence operator invoking by DFS can significantly stall other streams. We optimize the scheduler logic to BFS issuing to reduce such stall.

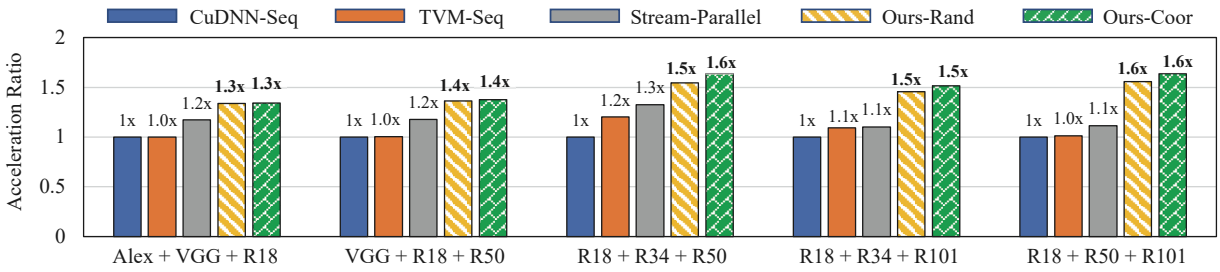


Fig. 6. Runtime Performance of the Proposed Automated Scheduling Framework. We mainly compare the acceleration ratio with three baselines: CuDNN-based sequential execution (CuDNN-Seq), TVM-based sequential execution (TVM-Seq), Stream-based parallel execution (Stream-parallel). Here Ours-R and Ours-G denotes the performance of our framework with random search and coordinate descent search. Test Platform: Titan V GPU.

GPU kernels according to each stage's IR. In multi-stream execution, the operator invoking is controlled by a main thread and invoking each operator takes a small duration of time. Although individually small, an inappropriate invoking sequence can also influence the latency, especially in high-concurrency stages with many operators.

Fig. 5 (a) showcase one example of operator invoking-caused stall. The default scheduler utilizes a depth-first (DFS) issue logic that issues all operator sequentially in one stream to ensure the operator dependency is maintained, and then iterates overall all streams. However, when there are multiple operators in the beginning streams, operators in the later streams can be significantly stalled due to the accumulated operator invoking overhead. To relieve such stall, we optimize the default DFS logic into a breadth-first (BFS) strategy. Fig. 5 (b) illustrates the BFS logic, which issues one operator from each stream interleavingly, and then iterates until all operators are issued. In such cases, all streams get similar invoking priority, and the operator dependency is also maintained in each stream. As a result, we could greatly reduce the operator invoking overhead for the later streams. Fig. 5 shows an example which we could reduce the stall from $102.75 \rightarrow 51.23$us and $177.27 \rightarrow 51.23$us for *Stream-1* and *Stream-2*.

IV. Experimental Evaluation

A. Experiment Setup

Model Zoo for Multi-Tenant Combination: We construct various multi-tenant scenarios by leveraging the following neural network models: AlexNet ($Alex$), VGG16 (VGG), ResNet18 ($R18$), ResNet34 ($R34$), ResNet50 ($R50$) and ResNet101 ($R101$). These models have distinctive model depths with operator numbers varying from $7 \sim 20$ to $86 \sim 216$. In addition, operators from different models also have particular computing and memory requirements. For example, the convolution operators have a wide range of computing complexity, e.g., from $32 \sim 64$ filters per layer to $256 \sim 512$ filters per layer. Therefore, each different multi-tenant combination based on the above models will pose its unique resource utilization imbalance challenges and has distinctive optimal scheduling strategies, mimicking the varied and complex multi-tenant scenarios of real-world applications.

Evaluation and Comparison Baselines: Three popular baseline scheduling strategies are considered.

- CuDNN-Seq [25]: The default strategy supported by the NVIDIA CuDNN library, which runs the multi-tenant inference sequentially;
- TVM-Seq [18]: A operator-level optimization method that adopts the TVM library [18] to search for the optimal kernel for each operator. However, without runtime support, it can only run these kernels sequentially;
- Stream-Parallel [17]: The concurrent execution strategy from native GPU multi-stream support [17]. It assigns models to different streams and leverages the default GPU scheduler to schedule the execution sequence.

Inference Setup: We conduct neural network inference on ImageNet [1] that has an image scale of 224x224x3 with single batch size to mimic the inference in practical applications such as autonomous driving. Two NVIDIA GPU platforms are utilized: Titan V of Volta architecture, P6000 of Pascal architecture. For all latency measurement, we record the averaged latency (ms) by profiling the same number of runs for our method and the baselines.

B. Speed-Up Evaluation

We first compare the inference latency of the baselines and our methods. The results are shown in Fig. 6. All methods' latency is normalized by the CuDNN-Seq baseline to show the relative acceleration ratio. Five multi-tenant settings, which cover a wide range of multi-tenant combinations are built up. For example, $Alex + VGG + R18$ which is a relatively simple ones (10$\sim$30 operators), and $R18+R50+R101$ whose operator numbers can over 200 is the most complex one, *etc*. For our method, we show both search algorithms' performance in our framework – the random search (*Ours-Random*) and coordinate descent search (*Ours-Coor*).

Overall Speed-up: It can be observed that our scheduling framework could consistently yield $1.3\times \sim 1.6\times$ speed-up compared to the sequential baselines across all five model combinations. Although the Stream-Parallel solution also yields a certain speed-up than CuDNN-Seq, its acceleration ratio is only $1.1\times \sim 1.3\times$, which is much less than ours.

Higher Speed-up in Highly Non-balanced Scenarios: It is worth noting that our method achieves the highest acceleration ratio, i.e., $1.5\times$ and $1.6\times$, on the two most challenging scenarios $R18 + R34 + R101$ and $R18 + R50 + R101$.

TABLE I
SCALABILITY EVALUATION (BS=1, 224X224, GPU: TITAN-V W/ VOLTA ARCH, LATENCY: MS)

#Models	Names	CuDNN-Seq	TVM-Seq	Stream-Parallel	Ours-R	Ours-C
2× models	VGG + R18	3.989	3.898	3.638	3.096 (**1.29**×)	2.912 (**1.37**×)
	R18 - R34	4.673	3.453	3.743	3.382 (**1.38**×)	3.128 (**1.49**×)
	R34 + R50	6.688	5.785	5.449	4.725 (**1.41**×)	4.478 (**1.49**×)
	R50 + R101	10.75	10.435	8.588	8.385 (**1.28**×)	8.203 (**1.31**×)
3×	VGG + R18 + R50	7.674	7.637	6.522	5.639 (**1.36**×)	5.587 (**1.37**×)
	R18 + R34 +R50	8.344	6.949	6.301	5.404 (**1.54**×)	5.096 (**1.63**×)
5×	VGG + R18 + R34 +R50 + R101	17.962	16.742	12.848	10.91 (**1.65**×)	10.42 (**1.72**×)

However, the Stream-Parallel performs poorly (only $1.1\times$) in these two settings. The reason is that such two multi-tenant combinations introduce extremely distinctive model lengths from 29 operators (ResNet18) to 200 operators (ResNet101), which brings significant resource imbalance between early and later stages across the entire processing. The native hardware scheduler in Stream-Parallel cannot take this into consideration and push all operators into the beginning stages, and thus cannot balance the resource utilization effectively. Therefore, it can only reach limited acceleration ratio.

Illustration of our Resource Balance Mechanism: In contrast to the hardware scheduler in Stream-Parallel, our method could effectively find a better scheduling solution via our pointer-based barrier insertion and automated search algorithm and hence achieves higher speed-up in the highly unbalanced scenarios. We visualize the kernel invoke timeline of our scheduling strategy on the $R18 + R50 + R101$ scenario, as is depicted in Fig. 7, to reveal the mechanism. The number of operators issued in each stage is symbolically denoted by the length of each colored block. The results show that our searched scheduling could effectively reduce the number of operators in the early stages to avoid potential resource contention and leave more operators into the later stages to enhance resource utilization. As such, our scheduling enables optimal resource utilization and finally achieves significantly lower latency performance than the Stream-Parallel solution.

Analysis on GPU Utilization Enhancement: We further profiled and checked the GPU runtime statistics to analyze and compare the overall GPU utilization with different scheduling strategies. Fig. 8 demonstrates the utilization statistics comparison between CuDNN-Sequential, Stream-Parallel, and Our scheduling strategy. We use the number of active warps per second as an indicator of GPU utilization information [26]. As is observed, our scheduling strategy averagely obtains $1.5\times$ utilization enhancement than the sequential schedule, which is consistent with our speed-up performance.

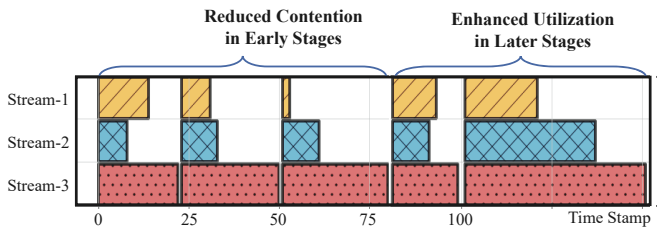


Fig. 7. Illustration of our Resource Balance Mechanism: Our method could find a balanced schedule to avoid both contention and under-utilization, thus achieving better performance than sequential and native parallel solutions.

C. Scalability and Generality Performance

In this section, the scalability and generality of our automated scheduling framework are evaluated.

Scalability with Varied Number of Tenants: We evaluate the scalability of our scheduling framework with varied number of model inference on one single GPU. Specifically, we test on three settings: $2\times$ models, $3\times$ models, and $5\times$ models with seven multi-tenant combinations in total.

The overall latency is shown in Table I, which reveals that our framework could scale well with the different number of tenants. Our framework could consistently obtain $1.3\times$ to $1.7\times$ acceleration than the sequential baseline across all benchmarks. Especially, in the five-model combination setting, we achieve the lowest runtime latency 10.42 ms, which is 7.5 ms lower than CuDNN-Seq (17.96 ms), and 2.4 ms lower than Stream-Parallel (12.85 ms), demonstrating the huge potential of our framework in accelerating practical applications.

Generality with Different GPUs: We then evaluate the generality of our scheduling framework with different GPU platforms. We test five multi-tenant settings on a different GPU: NVIDIA P6000 of Pascal architecture. The P6000 GPU is the last version before Titan-V and has slightly lower peak computing performance (12.6 vs. 14.9 TFLOPS). As the overall performance in Table II shows, our scheduling framework also yield significant performance gain ($1.25\times$ to $1.47\times$ acceleration) on the different GPU platform.

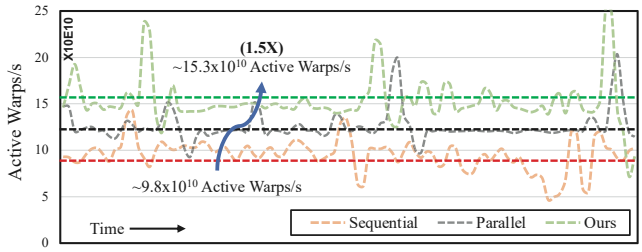


Fig. 8. Enhanced GPU Utilization Statistics. The number of active warps per second shows that our schedule could yield continuously better SM utilization.

TABLE II
GENERALITY EVALUATION (BS=1, 224X224, GPU: P6000 W/ PASCAL ARCH, LATENCY: MS)

Models	CuDNN-Seq	TVM-Seq	Stream-Parallel	Ours-Rand	Ours-Coor
Alex + VGG + R18	5.754	5.523	4.694	4.225 (**1.36**×)	4.126 (**1.39**×)
VGG + R18 + R50	9.687	8.978	8.524	7.739 (**1.25**×)	7.425 (**1.30**×)
R18 + R34 +R50	9.884	9.352	7.714	7.031 (**1.41**×)	6.727 (**1.47**×)
R18 + R34 + R101	14.278	13.256	11.833	11.08 (**1.29**×)	10.463 (**1.36**×)
R18 + R50 + R101	15.785	14.631	12.32	11.246 (**1.40**×)	10.711 (**1.47**×)

Advantage of Automated Searching: The above evaluations demonstrate that our framework could produce an optimal scheduling with better resource utilization and higher runtime speed. In addition, the experiments results also reflect one of the most promising advantage of our framework – ***easy-to-scale***. With the automated search algorithm design, our framework could automatically find the optimal scheduling strategies for varied number of tenants, distinct multi-model combinations, and different GPU platforms, significantly relieving the scheduling complexity and manual tuning efforts.

D. Search Algorithm Comparison and Overhead Analysis

In this section, we compare the search algorithms and analyze their introduced off-line running cost.

Search Algorithm Comparison: Fig. 9 compares two search algorithms' performance through their searching latency. The blue line (*Naive-Parallel*) illustrates the native stream-based scheduling performance. The green line (*Ours-Coor*) denotes the scheduling latency with coordinate descent search while the red line (*Ours-Rand*) shows the random search results. The same search rounds are conducted in the evaluations. The results indicate that the coordinate search generally has better performance than random search in the four multi-tenant conditions. Especially, in complex scenarios like Fig. 9 (d), random search may generate infeasible solutions that are filtered out and leave only few solutions, and thus have slightly worse performance than coordinate descent search.

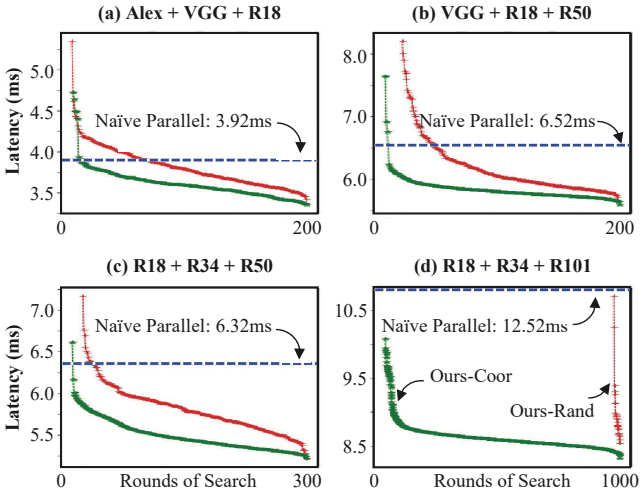


Fig. 9. The Search Algorithm Comparison.

TABLE III
THE FRAMEWORK RUNNING OVERHEAD (TITAN-V).

#Search Rounds	100	300	500	1000
Alex + VGG + R18	∼9.8s	∼28.9s	∼51.4s	∼1min35s
VGG + R18 + R50	∼10.3s	∼27.1s	∼48.9s	∼1min28s
R18 + R50 + R101	∼16.2s	∼45.3s	∼1min32s	∼2min42s

Nevertheless, both our search methods outperform the stream-based parallel solution by a large margin across all cases.

Framework Overhead Analysis: Our framework could usually yield near optimal schedule solutions within short search time. The framework's running time is demonstrated in Table III. We profile the coordinate descent search with different search rounds from 100 to 1000, which are general settings for most aforementioned multi-tenant scenarios. As the results show, our framework's running overhead maintains in the range of ten seconds to several minutes at most. Meanwhile, as such automated schedule can be pre-conducted offline given a defined multi-tenant scenario, we consider such offline tuning overhead is highly acceptable.

V. CONCLUSION

In this work, we tackle the multi-tenant inference optimization problem on GPU. Differently from single-model inference optimization, multi-tenant computation brings significantly higher compute complexity. To solve such compute complexity, we build an automated scheduling framework for multi-tenant inference optimization. Specifically, We first abstract the multi-tenant DNN inference scheduling as a fine-grained concurrency control problem, and implement the concurrency control by utilizing stream and synchronization based mechanisms. Based on the problem abstraction, we then formulate the DNN compute graphs and the scheduling factors with a unified IR design. Based on that, we formally define the scheduling search space. In the established scheduling search space, we transform multi-tenant scheduling into an optimization problem and propose an automated ML-based searching algorithm to find the optimal scheduling strategy. Experiments demonstrate our method could yield near optimal performance within short time, and meanwhile surpass previous scheduling method by $1.3\times \sim 1.7\times$ acceleration.

REFERENCES

[1] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[2] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *European conference on computer vision*. Springer, 2014, pp. 740–755.

[3] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.

[4] T.-J. Yang, A. Howard, B. Chen, X. Zhang, A. Go, M. Sandler, V. Sze, and H. Adam, "Netadapt: Platform-aware neural network adaptation for mobile applications," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 285–300.

[5] X. Li, Y. Zhou, Z. Pan, and J. Feng, "Partial order pruning: for best speed/accuracy trade-off in neural architecture search," in *Proceedings of the IEEE Conference on computer vision and pattern recognition*, 2019, pp. 9145–9153.

[6] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2019, pp. 10 734–10 742.

[7] X. Chang, H. Pan, W. Sun, and H. Gao, "Yoltrack: Multitask learning based real-time multiobject tracking and segmentation for autonomous vehicles," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[8] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 2636–2645.

[9] S. Chowdhuri, T. Pankaj, and K. Zipser, "Multinet: Multi-modal multi-task learning for autonomous driving," in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 2019, pp. 1496–1504.

[10] C. Yu, J. Wang, C. Peng, C. Gao, G. Yu, and N. Sang, "Bisenet: Bilateral segmentation network for real-time semantic segmentation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018.

[11] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *arXiv preprint arXiv:1506.01497*, 2015.

[12] Z. Kim, "Robust lane detection and tracking in challenging scenarios," *IEEE Transactions on Intelligent Transportation Systems*, vol. 9, no. 1, pp. 16–26, 2008.

[13] Microsoft, "Deep learning inference service at microsoft," 2020, https://www.usenix.org/system/files/opml19papers-soifer.pdf.

[14] NVIDIA, "Nvidia triron inference server," 2020, https://developer.nvidia.com/nvidia-triton-inference-server.

[15] "Nvidia a100 whitepaper," 2020, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[16] "Mpi for python," 2020, https://mpi4py.readthedocs.io/en/stable/.

[17] NVIDIA, "Cuda streams," 2020, https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf.

[18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. 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)*, 2018, pp. 578–594.

[19] TensorFlow, "Tensorflow xla (accelerated linear algebra)," 2020, https://www.tensorflow.org/xla.

[20] NVIDIA, "Nvidia tensorrt," 2020, https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[21] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, and A. Aiken, "Taso: optimizing deep learning computation with automatic generation of graph substitutions," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 47–62.

[22] Y. Yang, P. M. Phothilimtha, Y. R. Wang, M. Willsey, S. Roy, and J. Pienaar, "Equality saturation for tensor graph superoptimization," *arXiv preprint arXiv:2101.01332*, 2021.

[23] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, M. P. Phothilimtha, S. Wang, A. Goldie *et al.*, "Transferable graph optimizers for ml compilers," *arXiv preprint arXiv:2010.12438*, 2020.

[24] "Cuda stream sync," 2020, https://docs.nvidia.com/cuda/cuda-runtime-api/.

[25] NVIDIA, "Nvidia cudnn documentation," 2020, https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html.

[26] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, "Ios: Inter-operator scheduler for cnn acceleration," *arXiv preprint arXiv:2011.01302*, 2020.