Opara: Exploiting Operator Parallelism for Expediting DNN Inference on GPUs

Aodong Chen, Fei Xu, *Member, IEEE*, Li Han, *Member, IEEE*, Yuan Dong, Li Chen, *Member, IEEE*, Zhi Zhou, *Member, IEEE*, Fangming Liu, *Senior Member, IEEE*

Abstract—GPUs have become the *defacto* hardware devices for accelerating Deep Neural Network (DNN) inference workloads. However, the conventional *sequential execution mode of DNN operators* in mainstream deep learning frameworks cannot fully utilize GPU resources, even with the operator fusion enabled, due to the increasing complexity of model structures and a greater diversity of operators. Moreover, the *inadequate operator launch order* in parallelized execution scenarios can lead to GPU resource wastage and unexpected performance interference among operators. In this paper, we propose *Opara*, a resource- and interference-aware DNN Operator parallel scheduling framework to accelerate DNN inference on GPUs. Specifically, *Opara* first employs `CUDA Streams` and `CUDA Graph` to *parallelize* the execution of multiple operators automatically. To further expedite DNN inference, *Opara* leverages the resource demands of operators to judiciously adjust the operator launch order on GPUs, overlapping the execution of compute-intensive and memory-intensive operators. We implement and open source a prototype of *Opara* based on PyTorch in a *non-intrusive* manner. Extensive prototype experiments with representative DNN and Transformer-based models demonstrate that *Opara* outperforms the default sequential `CUDA Graph` in PyTorch and the state-of-the-art operator parallelism systems by up to $1.68\times$ and $1.29\times$, respectively, yet with acceptable runtime overhead.

Index Terms—DNN inference, DNN operator parallelism, scheduling, GPU resource utilization

1 Introduction

Deep Neural Networks (DNNs) have gained notable success in various business fields such as image processing, speech recognition, and virtual reality [1]. In general, DNN inference tasks are exceptionally latency-sensitive. For instance, latency requirements in autonomous driving scenarios are non-negotiable (*e.g.*, within 100 milliseconds) due to safety considerations [2]. Accordingly, increasing attention from both academia and industry has been paid to efficient model serving. To meet such performance requirements, modern cloud datacenters are hosting thousands of GPUs to accelerate DNN inference for users. For instance, Alibaba Cloud houses more than $6,000$ GPUs, many of which are tasked with managing a substantial volume of inference requests [3].

Cloud-based GPUs are equipped with an increasing amount of computational power, which typically exceeds the resource demands of individual inference tasks, leading to under-utilization and wastage of hardware resources [4]. To achieve the objective model accuracy with fewer computations, several recent works (*e.g.*, Szegedy et al. [5]) focus on substituting large operators with several smaller and multiple-branch operators in DNN models, which further exacerbates the issue of GPU under-utilization. While batching requests [6] or co-locating model inference tasks [7] on a GPU can mitigate such GPU under-utilization, it inevitably prolongs the model inference due to batching latency and performance interference [8]. Moreover, the operator fusion [9] cannot fully utilize the GPU resources (as discussed in Sec. 2.2) due to the limited scope of pre-defined fusion rules. Fortunately, as DNN models can typically be represented by a Directed Acyclic Graph (DAG) with *parallel operators*, it provides us an opportunity to *exploit inter-operator parallelism for accelerating DNN inference on GPUs while improving the GPU utilization.*

However, it is *nontrivial* to efficiently parallelize the execution of DNN operators for a DNN inference task due to the following two facts. *First*, the model DAG typically exhibits considerable complexity, often incorporating hundreds of operators with complex inter-operator dependencies. For simplicity, existing deep learning (DL) frameworks execute DNN operators one by one in topological sorting order [10]. To achieve operator parallelism, a recent work (*i.e.*, Nimble [11]) relies on a reduction transformation of the DNN computation graph, which inevitably brings heavy computation overhead. *Second*, inadequate operator parallel scheduling can adversely impact the DNN inference performance. As evidenced by motivation experiments in Sec. 2.3, the inadequate operator launch order in mainstream DL frameworks (*e.g.*, PyTorch) can prolong the DNN inference latency by up to 29%, due to the GPU blocking caused by the non-preemption feature of `CUDA` kernels [12] and per-

- *Aodong Chen, Fei Xu, and Yuan Dong are with the Shanghai Key Laboratory of Multidimensional Information Processing, School of Computer Science and Technology, East China Normal University, 3663 N. Zhongshan Road, Shanghai 200062, China. Email:* `fxu@cs.ecnu.edu.cn`.
- *Li Han is with the School of Software Engineering, East China Normal University, 3663 N. Zhongshan Road, Shanghai 200062, China. E-mail:* `hanli@sei.ecnu.edu.cn`.
- *Li Chen is with the School of Computing and Informatics, University of Louisiana at Lafayette, 301 East Lewis Street, Lafayette, LA 70504, USA. E-mail:* `li.chen@louisiana.edu`.
- *Zhi Zhou is with the Guangdong Key Laboratory of Big Data Analysis and Processing, School of Computer Science and Engineering, Sun Yat-sen University, 132 E. Waihuan Road, Guangzhou 510006, China. E-mail:* `zhouzhi9@mail.sysu.edu.cn`.
- *Fangming Liu is with Peng Cheng Laboratory, and Huazhong University of Science and Technology, China. E-mail:* `fangminghk@gmail.com`.

Manuscript received August XX, 2023; revised May XX, 2024.

formance interference among parallelizable operators [8]. In addition, several existing works (*e.g.*, IOS [13]) fail to consider the operator launch and function call overhead due to excessive CPU-GPU interactions when parallelizing DNN operators in the DL framework.

To address the challenges above, in this paper, we design *Opara*, a resource- and interference-aware DNN Operator parallel scheduling framework, with the aim of expediting the execution of DNN inference while improving the GPU utilization. We make the following contributions as below.

▷ We propose a lightweight stream allocation algorithm without any modifications or transformations of the computation graph. It greedily allocates operators without dependencies to multiple `CUDA Streams` to maximize operator parallelism. Meanwhile, operators with data dependencies are allocated to the same `CUDA Stream` without impacting parallel executions of operators, thereby reducing the number of time-consuming synchronization operations.

▷ We devise a resource- and interference-aware operator launch algorithm to judiciously prioritize launching operators with a small amount of GPU resource demands, so as to effectively mitigate GPU resource fragmentation and performance interference while reducing DNN inference latency. Such resource demands of operators can be obtained by lightweight inference profiling in practice.

▷ We have implemented a prototype of *Opara* (https://github.com/icloud-ecnu/Opara) as a plug-in module of PyTorch 2.0 to parallelize the executions of DNN operators. It can generate a parallelized `CUDA Graph` by capturing the stream allocation plan and optimized operator launch order to mitigate the operator launch and function call overhead. Our prototype experiments with 6 representative DNN and Transformer-based models demonstrate that *Opara* outperforms the default sequential `CUDA Graph` in PyTorch and the state-of-the-art DNN operator parallelism systems by up to 1.68× and 1.29×, respectively.

2 Background and Motivation

In this section, we first introduce how DNN operators are executed in mainstream DL frameworks, and identify the key factors that cause the low GPU utilization when serving DNN inference on GPUs. We then conduct motivation experiments to show how to judiciously parallelize the operator executions on GPUs.

2.1 DNN Operator Executions on NVIDIA GPUs

After being scheduled on GPUs, a DNN operator is actually recognized as a kernel. In general, a kernel comprises multiple thread blocks, which are the smallest scheduling granularity in `CUDA`. A thread block is scheduled to a Streaming Multiprocessor (SM) once the SM has sufficient resources to meet its resource demands [14]. In particular, an SM can concurrently execute multiple thread blocks, and each SM is constrained by a limited number of threads, shared memory, and registers.

To enable parallel executions of operators, we launch operators on multiple `CUDA Streams`. Each stream is actually a task *queue* that executes tasks sequentially. The execution order of kernels in different `CUDA Streams` is determined by their arrival order at the stream head. In general, the kernel execution time is considerably short as the batch size is typically small (*i.e.*, ranging from 1 to 16) in latency-critical inference scenarios [9], [15]. Accordingly, the kernel launch overhead constitutes the primary time cost for DNN inference, which negatively impacts the performance gains achieved by the parallel executions of kernels in multiple `CUDA Streams`. To reduce such overhead, `CUDA Graph` is a key feature introduced from `CUDA 10` that allows scheduling multiple DNN operators on a GPU device at a time.

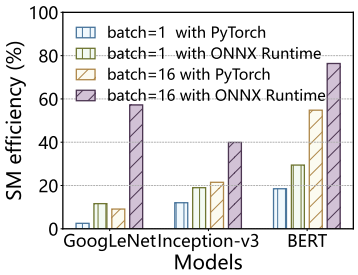


Fig. 1: Average SM efficiency of an NVIDIA A100-PCIE-40GB GPU when running GoogLeNet, Inception-v3, and BERT models in PyTorch and ONNX Runtime.

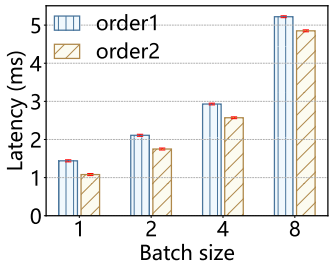


Fig. 2: Inference latency of GoogLeNet running on an NVIDIA RTX 2080 Super GPU with different operator launch orders and batch sizes.

2.2 Low GPU Utilization Due to Sequential Execution of DNN Operators

Mainstream DL frameworks execute DNN operators *sequentially* in topological sorting order, which cannot fully utilize GPU resources. To illustrate that, we conduct motivation experiments using the stock PyTorch 2.0 and ONNX Runtime 1.12[1] with the operator fusion [9] enabled. We serve three typical DNN inference models including GoogLeNet[2], Inception-v3 [5], and BERT[3] on both an NVIDIA A100-PCIE-40GB GPU and an NVIDIA RTX 2080 SUPER GPU. In particular, we adopt the SM efficiency measured using NVIDIA Nsight Compute CLI[4] to evaluate the GPU utilization.

As shown in Fig. 1, DNN inference on the mainstream DL frameworks achieves relatively low to medium GPU utilization even with the operator fusion technique enabled. Specifically, the SM efficiency of GoogLeNet, Inception-v3, and BERT with the batch size as 1 is merely 2.53%, 12.04%, and 18.5%, respectively, achieved in the stock PyTorch on an A100 GPU. Even when the batch size is increased to 16 and serving DNN inference in the ONNX Runtime, the SM efficiency of the three workloads is moderately increased to 57.21%, 39.9%, and 76.37%, respectively. Moreover, we repeat our experiments on a less powerful GPU (*i.e.*, RTX 2080 SUPER), and the SM efficiency of the three workloads ranges from 10.47% to 82.98%. Such experiment results above indicate that (1) the *sequential* execution of DNN operators is the root cause of low GPU utilization for serving DNN inference; (2) The operator fusion technique can only combine a certain number of parallelizable operators based on the pre-defined fusion rules [9], resulting in moderate

1. https://onnxruntime.ai/
2. https://pytorch.org/hub/pytorch_vision_googlenet/
3. https://huggingface.co/google-bert
4. https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

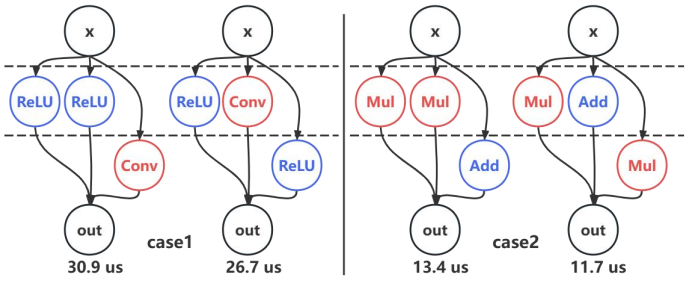


Fig. 3: Overlapping the execution of compute-intensive and memory-intensive operators denoted as red circles and blue circles, respectively.

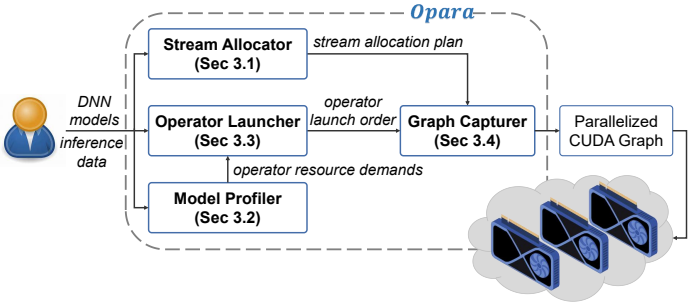


Fig. 4: System overview of *Opara*.

GPU utilization. Accordingly, there still exists enough room to exploit the *operator parallelism* for improving the GPU utilization of DNN inference on GPUs.

2.3 Performance Impacts of Operator Launch Order

Apart from the sequential execution of DNN operators, the *inadequate operator launch order* (*i.e.*, the topological sorting order of the model DAG) in mainstream DL frameworks can also lead to idle GPU resource usage and performance interference, thereby prolonging the inference latency. We conduct two motivation experiments to identify why the operator launch order can impact the inference latency.

GPU blocking. As each operator has a different number of blocks requiring three types of resources for execution, *i.e.*, threads, shared memory, and registers, a *resource-unaware* operator launch order can easily *block* the execution of operators until enough resources become available on the GPU. Such *GPU blocking* can severely waste the available GPU resources. As shown in Fig. 2, changing the operator launch order from order 1 (*i.e.*, depth-first topological sorting) to order 2 (*i.e.*, *Opara* designed in Sec. 3) for GoogLeNet can reduce the inference latency by up to 29% with different batch sizes. Furthermore, we repeat such an experiment on the A100 GPU, and the experiment results show around 10.3% of performance improvement by optimizing the operator launch order for GoogLeNet.

Performance interference. As the performance interference among operators can prolong the inference latency [8], we further conduct another experiment on A100 to illustrate the effectiveness of *overlapping the execution of compute-intensive and memory-intensive operators in mitigating the inference*. As depicted in Fig. 3 (case 1), prioritizing the parallel execution of `ReLU` and `Conv` operators can cause less severe interference, compared with parallelizing two `ReLU` operators, leading to a 13.6% reduction in the inference latency. Similarly, prioritizing the launch order of `Add` operator in case 2 can increase the inference performance by 12.7%, simply because the execution of compute-intensive and memory-intensive operators is overlapped.

Summary. Low GPU utilization of DNN inference is mainly caused by two factors: *First*, the *sequential* execution of DNN operators cannot fully utilize the GPU resources, even with the operator fusion enabled. *Second*, the default topological sorting order of operator launch is commonly *resource-* and *interference-unaware*. Accordingly, judiciously parallelizing DNN operators with an adequate operator launch order is compelling for accelerating DNN inference on GPUs while improving the GPU utilization.

3 System Design

In this section, we design *Opara* illustrated in Fig. 4, an operator parallel scheduling framework to reduce DNN inference latency while improving the GPU resource utilization. Specifically, *Opara* takes DNN models and input tensors (*i.e.*, inference data) from users. According to the operator dependencies in the model DAG, the Stream Allocator first employs a stream allocation algorithm to determine which stream the operators should be allocated to. The Model Profiler then gathers the resource demands of each operator using the model profiling. With such resource demands of operators, the Operator Launcher further employs a resource- and interference-aware operator launch algorithm to optimize the operator launch order on GPUs. Finally, the Graph Capturer generates a parallelized `CUDA Graph` by combing the stream allocation plan and operator launch order, thereby enabling efficient DNN inference on GPUs.

3.1 Stream Allocator

To parallelize the execution of operators in `CUDA Streams`, we leverage the computation graph (*i.e.*, DAG) of DNN models to determine *how many streams to launch and how to allocate operators to the streams*.

Definition of a model DAG. DNN computation graph can be represented as a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of operators in the model, and $\mathcal{E}$ denotes the operator dependencies. Each vertex $v \in \mathcal{V}$ denotes a DNN operator (*e.g.*, `Conv`, `MaxPool`). Each edge $\langle u, v \rangle \in \mathcal{E}$ denotes the operator dependency, where u is a predecessor of v and v is a successor of u. The set of all predecessors of an operator v are denoted as $\mathcal{N}_{pred}$. The set of all successors of an operator v are denoted as $\mathcal{N}_{succ}$.

Problem formulation and analysis. As a maximum of $|\mathcal{V}|$ streams can be launched for a DAG, we simply use a matrix $\mathcal{A}$ of size $|\mathcal{V}| \times |\mathcal{V}|$ to represent the stream allocation plan, which determines how the DNN operators are parallelized and synchronized. Each element $a_{ij} \in \mathcal{A}$ is a boolean value, indicating whether the i-th operator is executed in the j-th stream. We formulate the inference latency T_{inf} as

$$T_{inf} = T_{para} + T_{overhead}, \tag{1}$$

where T_{para} denotes the parallelized execution time of a DNN model and $T_{overhead}$ denotes the operator synchronization overhead given a stream allocation plan $\mathcal{A}$. In more detail, as the plan $\mathcal{A}$ can parallelize the execution of DNN operators, T_{para} can further be formulated as

$$T_{para} = h(\mathcal{A}) \times T_{seq}, \tag{2}$$

where T_{seq} denotes the inference latency of *sequential execution* of DNN operators and $h(\mathcal{A}) \in (0, 1]$ denotes the inference acceleration factor achieved by operator parallelism with the plan $\mathcal{A}$. To ensure the parallelized execution of the model, a certain number of synchronization operators [16] require to be inserted into the model DAG. Such a process introduces significant operator synchronization overhead $T_{overhead}$, which can be formulated as

$$T_{overhead} = g(\mathcal{A}) \times t_{overhead}, \tag{3}$$

where $t_{overhead}$ is the time overhead caused by one synchronization operator and $g(\mathcal{A})$ is *positively correlated* with the number of synchronization operators in approximation.

By substituting Eq. (3) and Eq. (2) into Eq. (1), we aim to minimize the inference latency T_{inf} by identifying an optimal stream allocation plan $\mathcal{A}$. Accordingly, we formulate the stream allocation optimization problem as

$$\min_{\mathcal{A}} \quad T_{inf} = h(\mathcal{A}) \times T_{seq} + g(\mathcal{A}) \times t_{overhead} \tag{4}$$

$$\text{s.t.} \quad \sum_{j=1}^{|\mathcal{V}|} a_{ij} = 1, \qquad \forall i \leq |\mathcal{V}| \tag{5}$$

where Constraint (5) mandates that each operator must be allocated to and only to one stream. $t_{overhead}$ and T_{seq} can be considered as constant values given a DNN model. Actually, our optimization problem in Eq. (4) (*i.e.,* minimizing $h(\mathcal{A})$ and $g(\mathcal{A})$) can be considered as scheduling DAGs with dependency constraints (*i.e.,* adjusting the matrix $\mathcal{A}$) to minimize the makespan, which has been proven to be an NP-hard problem [17]. Accordingly, we turn to devising a heuristic algorithm to acquire an appropriate (*i.e.,* sub-optimal) solution to our stream allocation problem.

Algorithm 1: Stream allocation algorithm in *Opara*.

Input : DNN computation graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: Set of streams to be launched $\mathcal{S}$

1 **Initialize:** $\mathcal{S} \leftarrow \emptyset$, `SYNC` $flag \leftarrow$ `False` for each operator $v \in \mathcal{V}$, and sort $\mathcal{V}$ in topological sorting order;
2 **for** *each operator* $v \in \mathcal{V}$ **do**
3 **for** *each predecessor* $p \in \mathcal{N}_{pred}$ *of* v **do**
4 **if** $flag$ *of* p *is* `False` **then**
5 *stream* of $v \leftarrow$ *stream* of p; `// put` v `and` p `in the same stream`
6 $flag$ of $p \leftarrow$ `True`;
7 **break** out of the loop;
8 **if** *stream of* v *is null* **then**
9 *stream* of $v \leftarrow$ launching a stream; `// put` v `in a newly launched stream`
10 $\mathcal{S} \leftarrow \mathcal{S} \cup \{$*stream* of $v\}$;
11 **return** $\mathcal{S}$;

Stream allocation algorithm. The *key idea* of Alg. 1 is to allocate parallelizable operators to multiple `CUDA Streams` as much as possible (*i.e.,* minimizing the value of $h(\mathcal{A})$). Moreover, we greedily put non-root nodes (*i.e.,* operators) in the same `CUDA stream` as one of their predecessor operators, so as to avoid excessive synchronization operators (*i.e.,* minimizing the value of $g(\mathcal{A})$). Specifically, given a computation graph $\mathcal{G}$, *Opara* first initializes a set of streams to be launched $\mathcal{S}$ and the `SYNC` $flag$ of each $v \in \mathcal{V}$. It then enumerates operators in $\mathcal{V}$ in topological sorting order (lines 1-2). For each operator $v \in \mathcal{V}$, it iterates over all of its predecessors $p \in \mathcal{N}_{pred}$ (line 3). If the current predecessor p has not yet contributed to reducing the synchronization overhead (*i.e.,* $flag$ is `False`), it allocates v to the same stream of p, and set the $flag$ of p as `True` (lines 4-7). If v does not find a predecessor that satisfies such a condition above, we allocate the operator v to a newly launched stream (lines 8-10). In particular, the parallelized execution of streams does not impact each other as long as operators are not executed on GPUs. To ease the understanding of Alg. 1, we present an illustrative example in Appendix A.

3.2 Model Profiler

As discussed in Sec. 2.3, the blocks in an operator execute the same instructions even with different data, which indicates that the GPU resources required by the blocks in an operator are the same. Accordingly, we obtain the resource demands of each operator by simply profiling the resource consumption (*i.e.,* the amount of shared memory, the number of registers and GPU threads) of a block in an operator. Such resource demands of operators will be used by Operator Launcher to determine an adequate operator launch order. In particular, we implement our Model Profiler utilizing the `torch.profiler.profile()` API, and it requires profiling each DNN inference *only once* to acquire the resource demands information for each operator, thereby bringing acceptable profiling overhead. We will examine the inference profiling overhead of *Opara* in Sec. 5.3.

3.3 Operator Launcher

Problem analysis. As illustrated in Sec. 2.3, inadequate operator launch orders can significantly affect the DNN inference latency. To identify an optimal launch order, a naive solution is iterating through *all possible* topological sorting orders of a model DAG and choosing the order with the lowest inference latency. However, such a method involves selecting nodes with zero indegree and deleting the corresponding vertices and their connected edges. By assuming n operators exist in a model DAG, the time complexity of traversing all topological sorting orders is $\mathcal{O}(n!)$, which is also an NP-hard problem [17]. As a result, we turn to designing a heuristic operator launch algorithm to solve such a complex problem.

Resource- and interference-aware operator launch algorithm. Launching operators with heavy resource demands first to the GPU is likely to cause resource fragmentation, hindering the GPU executions of subsequent operators. Moreover, the GPU can thus be blocked due to the non-preemptive feature of kernel execution [12]. To maximize the GPU utilization, the *key idea* of Alg. 2 is to greedily prioritize launching the operators with the least amount of GPU resource demands, aiming at maximizing the parallel executions of multiple operators within a model. To further mitigate the performance interference among operators [8], we simply *overlap* the execution of compute-intensive operators and memory-intensive operators, as classified by our offline-collected operator table.

Specifically, it first initializes and maintains a priority queue $\mathcal{Q}$ of operators in resource- and interference-aware

Algorithm 2: Operator launch algorithm in *Opara*.

Input : DNN computation graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Output: Operator queue $\mathcal{Q}$ in resource- and interference-aware operator launch order

1 **Initialize:** List of operators to be launched $\mathcal{L} \leftarrow \emptyset$, $\mathcal{L}_{mem} \leftarrow \emptyset$, $\mathcal{L}_{comp} \leftarrow \emptyset$, and $\mathcal{Q} \leftarrow \emptyset$;
2 Add the operators $v \in \mathcal{V}$ with an *indegree* of 0 that are *memory-intensive* and *compute-intensive* to $\mathcal{L}_{mem}$ and $\mathcal{L}_{comp}$, respectively;
3 **while** $\mathcal{L}_{mem}$ *or* $\mathcal{L}_{comp}$ *is not empty* **do**
4 $\mathcal{L} \leftarrow$ alternately choose a non-empty list from $\{\mathcal{L}_{mem}, \mathcal{L}_{comp}\}$;
5 $v_{min} \leftarrow$ the operator that requires the least amount of GPU resources in $\mathcal{L}$;
6 $\mathcal{L}.remove(v_{min})$, and $\mathcal{Q}.append(v_{min})$; // launch the operator v_{min}
7 **for** *each successor* $s \in \mathcal{N}_{succ}$ *of* v_{min} **do**
8 Update *indegree* of $s \leftarrow$ *indegree* of $s - 1$;
9 **if** *indegree of* $s == 0$ **then**
10 **if** *operator* s *is* memory-intensive **then**
11 $\mathcal{L}_{mem}$.append(s); // add s to $\mathcal{L}_{mem}$
12 **else**
13 $\mathcal{L}_{comp}$.append(s); // add s to $\mathcal{L}_{comp}$
14 **return** $\mathcal{Q}$;

operator launch order (line 1). It then retrieves all operators to be launched with an indegree of 0 in a list $\mathcal{L}$, which alternates between the non-empty lists for memory-intensive operators $\mathcal{L}_{mem}$ and for compute-intensive operators $\mathcal{L}_{comp}$ (lines 2-4). Each time the operator requiring the least amount of GPU resources (*e.g.*, shared memory, threads, registers) is chosen from $\mathcal{L}$ and then put into the queue $\mathcal{Q}$ (lines 5-6). In particular, the potential GPU blocking issue faced by the remaining large operators is *noncritical* in our scenario, as $\mathcal{L}$ is dynamic and can be compensated for the upcoming small operators to be launched. Finally, $\mathcal{L}_{mem}$ and $\mathcal{L}_{comp}$ are continuously updated by adding new operators with an indegree of 0 (lines 7-13).

3.4 Graph Capturer

To eliminate the overhead caused by kernel launches and function calls, the Graph Capturer first sets the `CUDA Streams` obtained from the Stream Allocator to the capture mode, and then it launches the operators of the DNN model to these streams according to the operator launch order specified by the Operator Launcher. To ensure the dependencies among operators, the Graph Capturer also launches the necessary synchronization operators to the streams. Consequently, a `CUDA Graph` is generated to enable operator parallelization while improving the GPU utilization. Such a graph capture process is lightweight and non-intrusive to PyTorch, as it has been exposed as a high-level API in PyTorch officially. We simply use the PyTorch API to capture and then generate the `CUDA Graph`.

4 Implementation of *Opara*

We implement a prototype of *Opara* with around $1,000$ lines of Python codes, which have been integrated into PyTorch 2.0 as a plug-in module. The source codes are currently publicly available on GitHub (https://github.com/icloud-ecnu/Opara). Specifically, we employ `torch.fx.Graph` as the computation graph for DNN models in *Opara*. Its Intermediate Representation (IR) allows us to schedule DNN operators directly in Python. In more detail, we leverage the `torch.cuda.set_stream()` API in PyTorch to launch operators on the `CUDA Streams`. To particularly guarantee the operator dependency in parallelized executions of streams, we add the appropriate synchronization operators to the model graph using the `event.record()` and `stream.wait_event(event)` APIs. Finally, we use `torch.cuda.graph(g)` to generate a `CUDA Graph` that can execute DNN operators in parallel based on the `CUDA Streams`. In summary, we build our prototype of *Opara* only using the high-level APIs of PyTorch in a lightweight and non-intrusive manner, rather than modifying the computation graph construction module as in Nimble [11].

5 Performance Evaluation

In this section, we carry out prototype experiments to demonstrate the efficacy and runtime overhead of *Opara* in comparison to the stock PyTorch and state-of-the-art operator parallelism frameworks.

5.1 Experimental Setup

Hardware configuration and workloads. We conduct our experiments on an NVIDIA A100-PCIe-40GB GPU and an NVIDIA GeForce RTX 2080 SUPER-8GB GPU. We implement *Opara* based on CUDA 11.7, cuDNN 8.5.0, and as a plug-in module of PyTorch 2.0. Our experiments employ 6 representative DNN models, including Inception-v3 [5], GoogLeNet, DeepFM [18], NASNet[5], BERT, and T5[6]. The first three models are executed on the RTX 2080 GPU, while the latter three are run on the A100 GPU.

Baselines and metrics. We compare DNN inference performance of *Opara* with that of the stock PyTorch (with `CUDA Graph` disabled), default sequential `CUDA Graph`, ONNX Runtime (with operator fusion enabled), Rammer [9], and Nimble [11]. Specifically, we implement Rammer's BFS-based operator scheduling algorithm (named Wavefront) into PyTorch 2.0. Nimble transforms the computation graph into a bipartite graph and identifies its maximum matching to determine an appropriate stream for each operator. In particular, we focus on 4 key metrics including DNN inference latency, SM efficiency, and GPU memory consumption, as well as DNN inference throughput. All the experiment results are averaged over $1,000$ runs.

5.2 Effectiveness of *Opara*

End-to-end inference latency. As shown in Fig. 5(a), *Opara* consistently outperforms the five baselines for six representative DNN models. Specifically, *Opara* can achieve $1.80\times$ to $10.97\times$ speedup compared to the stock PyTorch. This is because *Opara* utilizes `CUDA Graph` to eliminate the operator launch and function call overhead. *Opara* surpasses the default `CUDA Graph` by up to $1.68\times$, simply because of the

5. https://huggingface.co/timm/nasnetalarge.tf_in1k
6. https://huggingface.co/google-t5

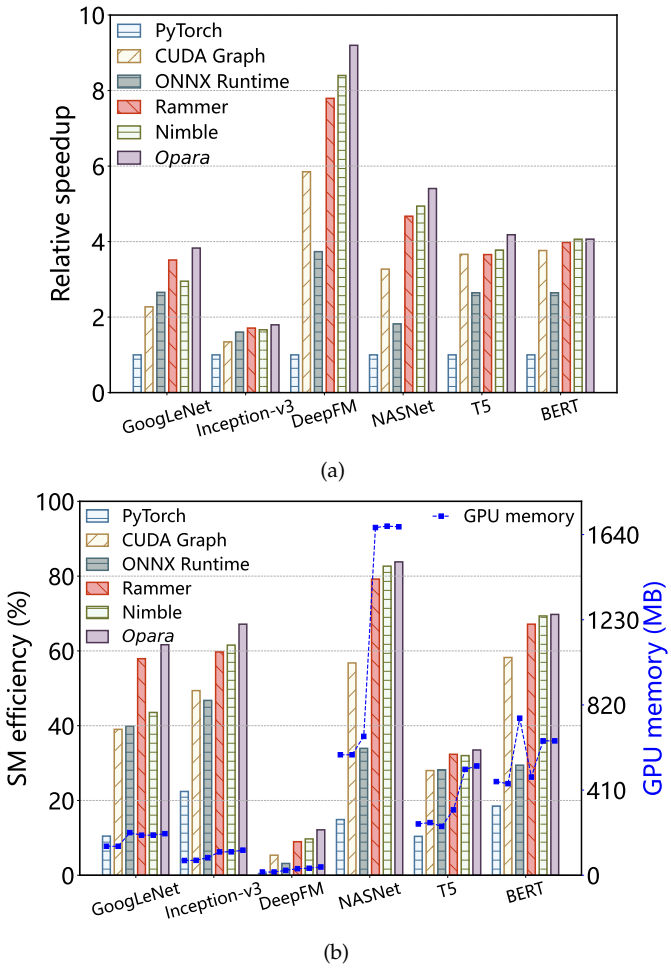


Fig. 5: (a) Relative speedup, (b) SM efficiency and peak memory consumption of GPUs running representative DNN models with batch size set as 1 achieved by PyTorch, `CUDA Graph`, ONNX Runtime, Rammer, Nimble, and *Opara* operator scheduling mechanisms.

parallel execution of DNN operators in *Opara*. Though the operator fusion technique outperforms the stock PyTorch, *Opara* achieves a higher speedup by up to 2.97× and 1.18×, compared with ONNX Runtime and Rammer, respectively. Such performance improvements above are mainly due to two facts: *First,* the operator fusion cannot combine all parallelizable operators based on the pre-defined fusion rules, which cannot fully utilize the GPU resources. *Second, Opara* accelerates model inference through operator parallelization, while the Wavefront scheduling algorithm in Rammer introduces additional synchronization overhead (*i.e.,* the unnecessary operator waiting time during wave executions). Furthermore, *Opara* outperforms Nimble by up to 1.29× because it judiciously alternates the scheduling of different types of operators with the lowest GPU resource consumption for each kernel launch time. Moreover, *Opara* initiates enough streams to increase parallelism (*e.g.,* 28 streams with *Opara* versus 4 streams with Nimble for GoogLeNet), thereby maximizing the operator parallelism.

GPU utilization and memory consumption. To unveil the performance gains of *Opara*, we proceed to look into the GPU utilization (*i.e.,* SM efficiency) and memory consumption during the model inference. As shown in Fig. 5(b), *Opara* exhibits a similar improvement in GPU utilization compared to the five baselines as in Fig. 5(a). Specifically,

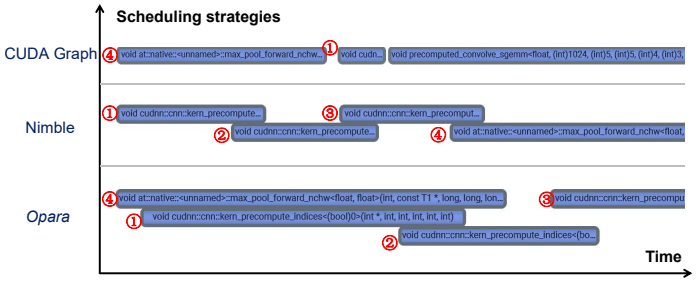


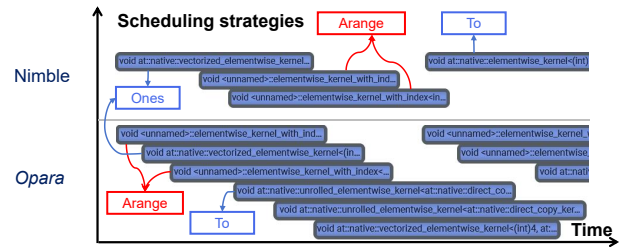
Fig. 6: Timeline of operator executions during a segment of inference process of GoogLeNet achieved by `CUDA Graph`, Nimble, and *Opara*.

Opara significantly improves the GPU utilization compared to the stock PyTorch, because *Opara* mitigates the scheduling overhead of the stock PyTorch. When compared with the default `CUDA Graph`, *Opara* increases the GPU utilization of Inception-v3, GoogLeNet, DeepFM, NASNet, BERT, and T5 by 36%, 58%, 126%, 48%, 20%, and 19%, respectively. Such performance gains mainly come from the parallelized execution of operators. When compared to ONNX Runtime, Rammer, and Nimble, *Opara* boosts the GPU utilization by up to 3.86×, 1.36× and 1.42× mainly because (1) maximizing stream allocations in *Opara* can increase operator parallelism opportunities, and (2) optimizing the operator launch order in *Opara* further minimizes the GPU idle time. Furthermore, the parallel execution of operators requires an increased amount of data to reside in the GPU memory simultaneously, thereby leading to a higher peak GPU memory consumption of *Opara* than that of sequential executions.

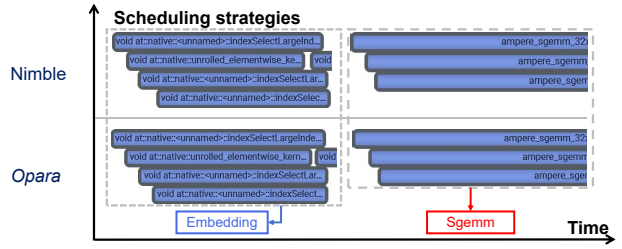
Timeline of operator executions. We further illustrate the operator execution timeline by taking a segment of inference process of GoogLeNet as an example. In particular, we leverage NVIDIA Nsight System CLI[7] to track the timeline of operator executions. As depicted in Fig. 6, we observe that `CUDA Graph` executes the 4 operators sequentially in a stream, and only operators 4 and 1 appear within the time window. The remaining operators 2 and 3 are forced to queue up, which leads to a long inference time. Though Nimble can parallelize operators in the order of 1, 2, 3, and 4, it only schedules 2 operators on two streams, causing a long GPU idle time. In contrast, *Opara* prioritizes operator 4 and initiates more streams than Nimble, so that operators 4, 1, and 2 can be executed in parallel to maximize the operator parallelism. Accordingly, *Opara* can achieve the shortest inference latency by exploring operator parallelism compared with `CUDA Graph` and Nimble.

Effectiveness of *Opara* on Transformer-based models. We conduct experiments with T5 and BERT model, and *Opara* outperforms Nimble by 9.3% for the T5 model as shown in Fig. 5(a). This is because *Opara* optimizes the launch order of operators in T5 and schedules them into 6 streams compared with 3 streams in Nimble. Moreover, the *operator diversity* in T5 offers *Opara* overlap the compute-intensive `Arange` operators and the memory-intensive `To` and `Ones` operators, as shown in Fig. 7(a). For BERT, however, *Opara* achieves a similar operator launch order and the same number of streams as Nimble as depicted in Fig. 7(b). This is because the parallelizable operators of BERT are always the `Embedding` operators or the `Sgemm`

7. https://docs.nvidia.com/nsight-systems/UserGuide/index.html



(a) T5



(b) BERT

Fig. 7: Timeline of Transformer-based models achieved by Nimble and *Opara* operator scheduling mechanisms. Red rectangles and blue rectangles represent compute-intensive operators and memory-intensive operators, respectively.

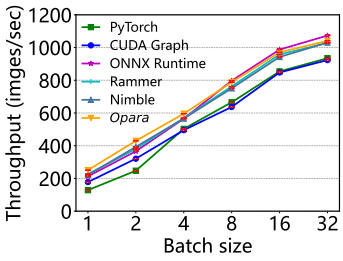


Fig. 8: Inference throughput of Inception-v3 with *Opara* and the five baselines by varying the batch size from 1 to 32 on an RTX 2080 GPU.

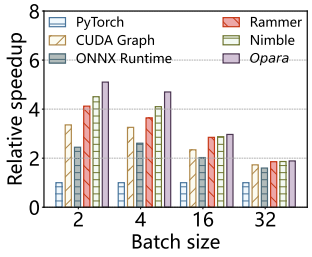


Fig. 9: Relative speedup of Inception-v3 with *Opara* and the five baselines by varying the batch size from 2 to 32 on an A100 GPU.

operators, which reduces the opportunity for operator overlapping and launch order optimization. Accordingly, *Opara* achieves marginal performance gains for BERT compared with Nimble, yet 1.08× to 4.06× speedup compared to the stock PyTorch and `CUDA Graph` as shown in Fig. 5(a).

Throughput under different batch sizes. As depicted in Fig. 8, we observe that *Opara* consistently surpasses the five baselines except for ONNX Runtime by varying the batch size from 1 to 32. Nevertheless, the performance gains of *Opara* gradually diminish as the batch size increases. As an example, *Opara* outperforms the default `CUDA Graph` by 1.41× and 1.09× when the batch size is 1 and 32, respectively. This is because the amount of GPU resources occupied by a single operator increases when dealing with larger batch sizes, resulting in fewer GPU resources available for the execution of parallelized operators. This also explains why *Opara* exhibits marginal throughput improvement for large batch sizes of 16 and 32. The results above also show that maximizing the operator parallelism can also improve the inference throughput.

Effectiveness of *Opara* on high-end GPUs with sufficient resources. We repeat the inference experiment of Inception-v3 on a high-end GPU (*i.e.*, A100). As shown in Fig. 9, we observe that *Opara* consistently outperforms

TABLE 1: Computation time (in milliseconds) of the stream allocation algorithm in *Opara* (*i.e.*, Alg. 1) and Nimble [11] for various models.

	BERT	GoogLeNet	NASNet	Inception-v3	T5
Opara	0.58	0.27	1.75	0.50	2.8
Nimble	20.8	5.80	257.83	14.40	161.4

the five baselines by varying the batch size from 2 to 32, mainly because operator parallelism works well for high-end GPUs with sufficient resources. In more detail, *Opara* achieves an inference speedup by up to 2.08×, 1.29×, and 1.15× compared to ONNX Runtime, Rammer, and Nimble, respectively. In particular, *Opara* achieves speedups of 1.47× and 1.18× relative to ONNX Runtime for batch sizes of 16 and 32, which is larger than the results achieved on the RTX 2080 GPU. This is because the A100 GPU provides sufficient resources, which allows the operator parallelism to achieve more performance gains than the operator fusion.

5.3 Runtime Overhead of *Opara*

We evaluate the runtime overhead of *Opara* in terms of algorithm computation time and inference profiling overhead. As listed in Table 1, *Opara* can reduce the computation time of the stream allocation algorithm by up to two orders of magnitude compared with Nimble [11]. This is because Nimble requires a graph transformation together with an exhaustive search in the bipartite graph. Such a process is time-consuming with a complexity in the order of $\mathcal{O}(n^3)$, where n is the number of operators in a model DAG. In contrast, the time complexity of *Opara* can be reduced to the order of $\mathcal{O}(n)$, simply because the inner loop of Alg. 1 (lines 3-10) in *Opara* only depends on the maximum width (*i.e.*, typically below 20) of the computation graph. Accordingly, as DNN models become increasingly complex [19], the number of operators n gets even larger, while the algorithm computation overhead of *Opara* can still be well contained. In addition, as the Model Profiler needs to run the DNN inference only once, *Opara* requires several (*i.e.*, 4.25) milliseconds of profiling overhead in our experiment. In sum, the runtime overhead of *Opara* is practically acceptable.

6 Related Work

Inter-operator parallelism within a single model. To parallelize the execution of DNN operators, Rammer [9] proposes fine-grained operator scheduling based on the Wavefront algorithm and enables operator fusion on a GPU device. To increase the operator parallelism, Cocktailer [20] further co-schedules control flow and data flow operators based on Rammer. The two prior works above operate at the *compilation* level, which requires significant compilation overhead and manual customization of operators. Orthogonal to them, *Opara* focuses on the *runtime operator scheduling* optimization of the stream allocation and the operator launch order. A recent work Nimble [11] leverages the bipartite graph algorithm to schedule operators on `CUDA streams` adequately. IOS [13] deploys operator fusion and dynamic programming to determine operator parallelization plans. However, Nimble and IOS require a lengthy search process and neglect the optimization space of operator launch order. In contrast, *Opara* utilizes the `CUDA Graph` to eliminate

such performance overhead. It also employs a *lightweight* stream allocation algorithm to achieve inter-operator parallelism. To reduce the GPU idle time and interference, *Opara* determines a feasible operator launch order according to operator resource demands.

Inter-operator parallelism among different models. To improve GPU utilization, several works parallelize operators from multiple models co-located on a GPU device. For example, S^{3}DNN [21] and Abacus [22] optimize the co-location of operators from different models and schedule them to the corresponding steams. To minimize the model co-location interference, *iGniter* [8] and Orion [15] focus on optimizing the GPU resource allocation and operator scheduling on multiple prioritized streams, respectively. Paella [16] dispatches the optimal kernel from multiple models by jointly considering the remaining time and model fairness. Different from optimizing the inference co-location, *Opara* minimizes the inference latency while increasing the GPU utilization by parallelizing operators within a single model. Moreover, it achieves inter-operator parallelism as a plug-in module of PyTorch 2.0 without developing a new DL runtime or framework.

Intra-operator parallelism. Existing DL frameworks, such as PyTorch and TensorFlow, employ expert-optimized operator libraries to accelerate individual operators. TVM [23] uses machine learning methods to automatically search for efficient operators, which is time-consuming and requires the specified parameter space manually. To achieve automated code generation, Ansor [24] implements an automatic search space construction. As a single DNN operator cannot fully utilize GPU resources in general, *Opara* can work with the intra-operator parallelism methods above to further improve GPU resource utilization.

7 Conclusion and Future Work

This paper presents the design and implementation of *Opara*, a lightweight operator scheduling framework to speed up DNN inference on GPUs. By reducing the synchronization overhead among operators, *Opara* designs a stream allocation algorithm to automatically allocate operators without dependencies to different `CUDA streams`, thereby achieving operator parallelism effectively. Furthermore, *Opara* leverages non-intrusive inference profiling to judiciously select an appropriate operator launch order to mitigate interference and maximize the GPU utilization. Extensive prototype experiments show that *Opara* can improve the performance of DNN inference by up to 29%, as compared to the state-of-the-art operator parallelism systems.

We plan to extend *Opara* in the following directions: (1) constructing an analytical model to analyze the performance interference caused by inter-operator parallelism, and (2) examining the effectiveness of *Opara* for accelerating more large models (*e.g.*, GPT-3, LLaMA).

References

[1] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. P. Reyes, M.-L. Shyu, S.-C. Chen, and S. S. Iyengar, "A Survey on Deep Learning: Algorithms, Techniques, and Applications," *ACM Computing Surveys*, vol. 51, no. 5, pp. 1–36, 2018.

[2] D. Mendoza, F. Romero, and C. Trippel, "Model Selection for Latency-Critical Inference Serving," in *Proc. of ACM Eurosys*, Apr. 2024, pp. 1016–1038.

[3] Q. Weng, W. Xiao, Y. Yu, W. Wang, C. Wang, J. He, Y. Li, L. Zhang, W. Lin, and Y. Ding, "MLaaS in the Wild: Workload Analysis and Scheduling in Large-Scale Heterogeneous GPU Clusters," in *Proc. of USENIX NSDI*, Apr. 2022, pp. 945–960.

[4] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "MISO: Exploiting Multi-Instance GPU Capability on Multi-Tenant GPU Clusters," in *Proc. of ACM SOCC*, Nov. 2022, pp. 173–189.

[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the Inception Architecture for Computer Vision," in *Proc. of IEEE CVPR*, Jun. 2016, pp. 2818–2826.

[6] W. Cui, H. Zhao, Q. Chen, H. Wei, Z. Li, D. Zeng, C. Li, and M. Guo, "DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs," in *Proc. of USENIX ATC*, Jul. 2022, pp. 183–198.

[7] A. Dhakal, S. G. Kulkarni, and K. Ramakrishnan, "GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform," in *Proc. of ACM SOCC*, Oct. 2020, pp. 492–506.

[8] F. Xu, J. Xu, J. Chen, L. Chen, R. Shang, Z. Zhou, and F. Liu, "iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in the Cloud," *IEEE Transactions on Parallel & Distributed Systems*, vol. 34, no. 03, pp. 812–827, 2023.

[9] L. Ma, Z. Xie, Z. Yang, J. Xue, Y. Miao, W. Cui, W. Hu, F. Yang, L. Zhang, and L. Zhou, "Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks," in *Proc. of USENIX OSDI*, Nov. 2020, pp. 881–897.

[10] Y. Zhao, Q. Sun, Z. He, Y. Bai, and B. Yu, "AutoGraph: Optimizing DNN Computation Graph for Parallel GPU Kernel Execution," in *Proc. of AAAI*, Feb. 2023, pp. 1–9.

[11] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning," *Advances in Neural Information Processing Systems*, vol. 33, pp. 8343–8354, 2020.

[12] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith, "GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed," in *Proc. of IEEE RTSS*, Dec. 2017, pp. 104–115.

[13] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, "IOS: Inter-Operator Scheduler for CNN Acceleration," in *Proc of MLSys*, Apr. 2021, pp. 167–180.

[14] G. Gilman and R. J. Walls, "Characterizing Concurrency Mechanisms for NVIDIA GPUs under Deep Learning Workloads," *ACM SIGMETRICS Performance Evaluation Review*, vol. 49, no. 3, pp. 32–34, 2022.

[15] F. Strati, X. Ma, and A. Klimovic, "Orion: Interference-aware, Fine-grained GPU Sharing for ML Applications," in *Proc. of ACM Eurosys*, Apr. 2024, pp. 1075–1092.

[16] K. K. Ng, H. M. Demoulin, and V. Liu, "Paella: Low-latency Model Serving with Software-defined GPU Scheduling," in *Proc. of ACM SOSP*, Oct. 2023, pp. 595–610.

[17] J. D. Ullman, "NP-complete Scheduling Problems," *Journal of Computer and System Sciences*, vol. 10, no. 3, pp. 384–393, 1975.

[18] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "DeepFM: A Factorization-Machine based Neural Network for CTR Prediction," in *Proc. of IJCAI*, Aug. 2017, pp. 1725–1731.

[19] Y. Shi, Z. Yang, J. Xue, L. Ma, Y. Xia, Z. Miao, Y. Guo, F. Yang, and L. Zhou, "Welder: Scheduling Deep Learning Memory Access via Tile-graph," in *Proc. of USENIX OSDI*, Jul. 2023, pp. 701–718.

[20] C. Zhang, L. Ma, J. Xue, Y. Shi, Z. Miao, F. Yang, J. Zhai, Z. Yang, and M. Yang, "Cocktailer: Analyzing and Optimizing Dynamic Control Flow in Deep Learning," in *Proc. of USENIX OSDI*, Jul. 2023, pp. 681–699.

[21] H. Zhou, S. Bateni, and C. Liu, "S^{3}DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads," in *Proc. of IEEE RTAS*, Apr. 2018, pp. 190–201.

[22] W. Cui, H. Zhao, Q. Chen, N. Zheng, J. Leng, J. Zhao, Z. Song, T. Ma, Y. Yang, C. Li *et al.*, "Enable Simultaneous DNN Services Based on Deterministic Operator Overlap and Precise Latency Prediction," in *Proc. of ACM SC*, Nov. 2021, pp. 1–15.

[23] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in *Proc. of USENIX OSDI*, Oct. 2018, pp. 579–594.

[24] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen *et al.*, "Ansor: Generating High-Performance Tensor Programs for Deep Learning," in *Proc. of USENIX OSDI*, Nov. 2020, pp. 863–879.

APPENDIX

.1 An Illustrative Example of Alg. 1

As an illustrative example in Fig. 10, Alg. 1 first traverses the operators in the computation graph in topological sorting order. It then categorizes the operators into four types based on their predecessor relationships as follows.

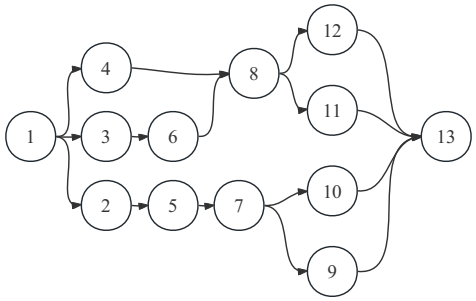


Fig. 10: A stream allocation example with a typical DNN model.

- For an operator i without a predecessor, like operator 1 in the example, we place the operator i in a newly-created stream.
- For an operator i with only one predecessor and its predecessor has only one successor, like operators 5, 6, and 7 in the example, we place the operator i in the same stream as its predecessor.
- For an operator i with a unique predecessor that has multiple successors, if the operator i is the first successor of its predecessor, like operators 2, 9, and 11 in the example, we place the operator i in the same stream as the predecessor; otherwise, like operators 3, 4, 10, and 12 in the example, we place the operator i in a newly created stream. This is because its predecessor has already contributed to reducing the synchronization overhead for the first successor (*i.e.*, the `SYNC` $flag$ of their predecessors has been set as `True`).
- For an operator i with multiple predecessors, like operators 8 and 13 in the example, we place the operator i in the same stream as the first predecessor with the `SYNC` $flag$ set as `False`, by traversing all its predecessors.