Opara: Exploring Operator Parallelism for Expediting DNN Inference on GPUs

Abstract

GPUs have become the *defacto* hardware devices to accelerate Deep Neural Network (DNN) inference in deep learning (DL) frameworks. However, the conventional *sequential execution mode of DNN operators* in mainstream DL frameworks cannot fully utilize GPU resources, due to the increasing complexity of DNN model structures and the progressively smaller computational sizes of DNN operators. Moreover, the *inadequate operator launch order* in parallelized execution scenarios can lead to GPU resource wastage. To address such performance issues above, we propose *Opara*, a lightweight and resource-aware DNN Operator parallel scheduling framework to accelerate the execution of DNN inference on GPUs. Specifically, *Opara* first employs `CUDA Graph` and `CUDA Streams` to automatically *parallelize* the execution of multiple DNN operators. It further leverages the resource requirements of DNN operators to judiciously adjust the operator launch order on GPUs to expedite DNN inference. We implement a prototype of *Opara* based on PyTorch in a *non-intrusive* manner. Our prototype experiments with representative DNN 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 1.61× and 1.24×, respectively, yet with acceptable runtime overhead.

Keywords: DNN inference, DNN operator parallelism, Scheduling, GPU 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. To enhance computational efficiency, several recent works (*e.g.,* Szegedy et al. [4]) focus on substituting large operators with small and multiple-branch operators in DNN models, which exacerbates the under-utilization of GPU resources. While batching requests or concurrent processing of multiple model inference tasks can mitigate such GPU under-utilization [5], it adversely results in increased latency for model inference due to batching latency and interference. As a result, it is compelling to improve the GPU utilization without impacting the DNN inference latency. As DNN models can typically be represented by a Directed Acyclic Graph (DAG) with *parallel operators*, it provides us an opportunity to *explore operator parallelism for accelerating DNN inference on GPUs while improving the GPU utilization.*

Unfortunately, it is *nontrivial* to efficiently parallelize the execution of DNN operators for a DNN inference task due to the following two facts. *First,* the DAG of a DNN model typically exhibits considerable complexity, often incorporating hundreds of DNN operators with complex inter-operator dependencies. For simplicity, existing deep learning (DL) frameworks execute DNN operators one by one in topological sorting order, which *overlooks the parallelization opportunities among operators.* To achieve operator parallelism, a recent work (*i.e.,* Nimble [6]) relies on a reduction transformation of the DNN computation graph, which inevitably brings heavy computation overhead. *Second,* inadequate operator parallel scheduling can negatively impact the DNN inference performance. As evidenced by our motivation experiment in Sec. 2.3, the default and inadequate operator launch order in mainstream DL frameworks can prolong the DNN inference latency by up to 20%, due to the GPU blocking caused by the non-preemption feature of CUDA kernels [7]. In addition, several existing works (*e.g.,* IOS [8]) fail to consider the operator launch overhead 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-aware and lightweight 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-aware operator launch algorithm to judiciously prioritize launching operators with a small amount of GPU resource requirements, so as to effectively mitigate GPU resource fragmentation while reducing DNN inference latency. Such resource requirements of operators can be obtained by a lightweight inference profiling in practice.
- We have implemented a prototype of *Opara* (https://github.com/OparaSys/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 overhead and function call overhead. Our prototype experiments with representative DNN 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 1.61× and 1.24×, respectively.

The rest of the paper is organized as follows. Sec. 2 presents the background and motivation of this paper. Sec. 3 elaborates the design of four key components in *Opara*. Sec. 4 implements our *Opara* prototype based on PyTorch. Sec. 5 evaluates the effectiveness and runtime overhead of *Opara*. Sec. 6 discusses related work and Sec. 7 concludes this paper.

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 present an illustrative example to show how to judiciously parallelize the operator[1] 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 requirements [9]. 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` [10]. Each `CUDA 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 DNN inference scenarios. Accordingly, the kernel launch overhead constitutes the primary time cost for DNN inference, which offsets the performance gains achieved by the parallel executions of kernels in multiple `CUDA Streams`. To reduce such kernel launch overhead, `CUDA Graph` [10] is a key feature introduced from `CUDA 10` that allows scheduling multiple DNN operators on GPUs at a time.

2.2 Low GPU Utilization Due to Sequential Execution of DNN Operators

Mainstream DL frameworks generally do not support inter-operator parallel execution for DNN inference, due to the complexity of parallel programming. Instead, they execute DNN operators *sequentially* in topological sorting order, which overlooks the

[1] Operators are commonly referred to as kernels once submitted to the GPU.

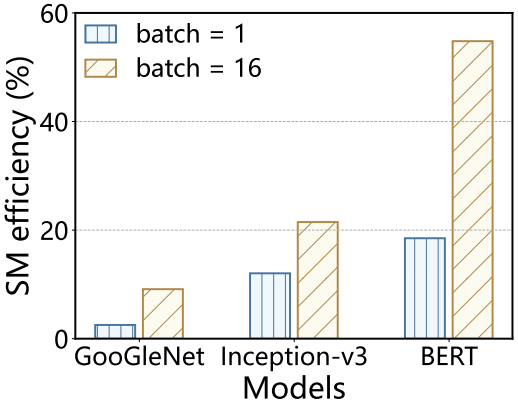

Fig. 1 Average SM efficiency of an A100-PCIE-40GB GPU when running GoogLeNet, Inception-v3, and BERT models with different batch sizes.

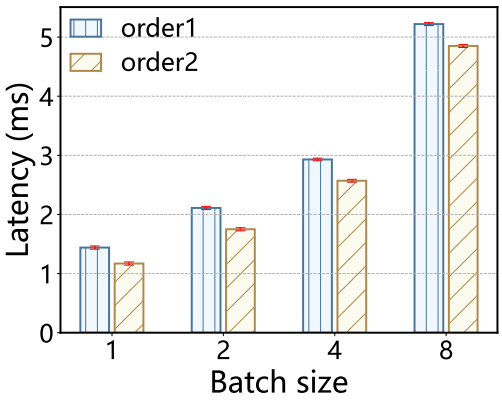

Fig. 2 Inference latency of GooGleNet running on an RTX 2080 Super GPU with different operator launch orders and batch sizes.

operator parallelism opportunity and cannot fully utilize the GPU resource. To illustrate that, we conduct motivation experiments with GoogLeNet [11], Inception-v3 [4], and BERT [12] on an NVIDIA A100-PCIE-40GB GPU and an NVIDIA RTX 2080 SUPER GPU. The experiment setup is the same as in Sec. 5.1. In particular, we adopt the SM efficiency measured using NVIDIA Nsight Compute CLI[2] to evaluate the GPU utilization.

As shown in Fig. 1, DNN inference on the mainstream DL framework (*i.e.,* the latest PyTorch 2.0) leads to relatively low GPU utilization on A100. Specifically, the SM efficiency of GoogLeNet with the batch size set as 1 is a mere 2.53%, while that of Inception-v3 is 12.04%. Even when the batch size increases to 16, the SM efficiency of GoogLeNet is still 9.12% and that of Inception-v3 reaches at 21.48%. Similarly, larger models such as BERT cannot fully utilize the GPU resource. By setting the sequence length as 32, the SM efficiency of BERT with the batch size set as 1 and 16 is 18.5% and 54.8%, respectively. Meanwhile, we repeat our experiments on a less powerful GPU (*i.e.,* RTX 2080 SUPER), and the SM efficiency of the three workloads is ranging from 10.47% to 82.98%. Our experiment results indicate that the *sequential* execution of DNN operators is the root cause of low GPU utilization for running DNN inference, and exploring the operator parallelism can expedite the DNN inference on GPUs while improving GPU utilization.

2.3 An Illustrative Example: Impacts of Operator Launch Order on Inference Latency

Apart from the sequential execution of DNN operators, the *inadequate operator launch order* can also lead to idle GPU resource usage, thereby prolonging DNN inference latency. To illustrate that, we present an illustrative example of scheduling three operators A, B, and C for execution on GPUs. Each operator has a different number of blocks requiring three types of resources for execution, *i.e.,* threads, shared memory,

[2]Nsight Compute CLI: https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html

and registers. Block resource requirements vary among operators but they are identical within the same operator.

We only consider the shared memory resource for simplicity. We suppose that a block of the operator A, B, and C requires 50%, 60%, and 40% of shared memory, and a default operator launch order 1 in the DL framework is $B \rightarrow A \rightarrow C$. In that case, after operator B has been scheduled for execution on the GPU, operator A cannot be scheduled due to insufficient (*i.e.,* 40% remaining) shared memory resources. Operators A and C will be blocked until enough resources become available on the GPU. In contrast, a better operator launch order 2 can be $B \rightarrow C \rightarrow A$. In that case, after scheduling blocks of kernel B on the GPU, blocks of kernel C can be scheduled on GPUs immediately, followed by blocks of kernel A waiting for available GPU resources. Accordingly, the operator launch order 2 can significantly reduce the GPU idle time and improve the GPU utilization.

Based on the above, the default operator launch order in the DL framework is the topological sorting order of the DAG of DNN models. Such an operator launch order is *resource-unaware* and leads to GPU blocking, thereby wasting the available GPU resources. As shown in Fig. 2, changing the operator launch order from order 1 to order 2 for GooGleNet can reduce the inference latency by up to 20% with different batch sizes. Furthermore, we repeat such an experiment on the A100 GPU, and the experiment results show around 9.2% of performance improvement by optimizing the operator launch order for GooGleNet. As a result, determining a *resource-aware* operator launch order can improve the GPU utilization and reduce the DNN inference latency.

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. *Second*, the default topological sorting order of operator launch is commonly inadequate and *resource-unaware*. Accordingly, judiciously parallelizing the DNN operators with an adequate operator launch order can accelerate DNN inference on GPUs while improving the GPU utilization.

3 System Design

In this section, we design *Opara*, an operator parallel scheduling framework to reduce DNN inference latency while improving the GPU resource utilization. *Opara* comprises four components including Model Profiler, Operator Launcher, Stream Allocator, and Graph Capturer.

As illustrated in Fig. 3, *Opara* takes DNN models and input tensors (*i.e.,* inference data) from users. According to the operator dependencies in the DAG of DNN models, 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 requirements of each operator during the model profiling process. With such resource requirements of operators, the Operator Launcher further employs a resource-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.

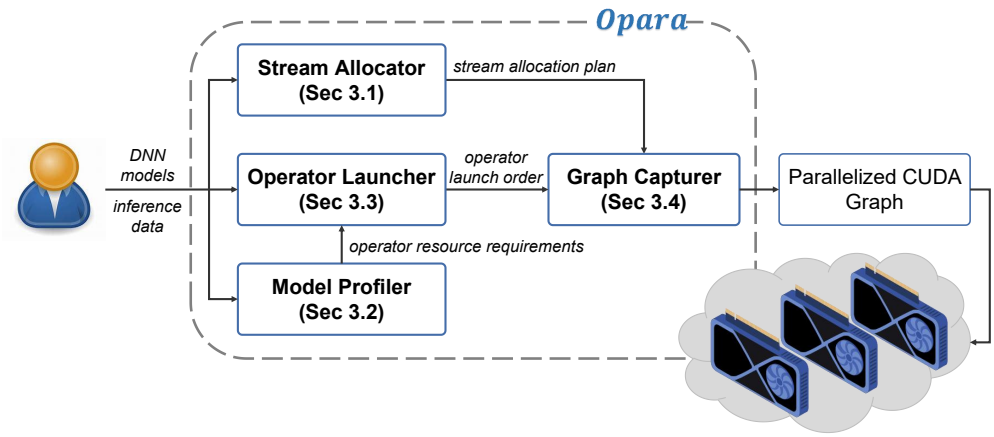

Fig. 3 System overview of *Opara*.

3.1 Stream Allocator

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

Definition of Computation Graph. 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.,* a linear algebra operation like convolution. 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 Analysis. As the number of launched streams does not impact the DNN inference latency, a maximum of $|\mathcal{V}|$ streams can be launched on GPUs. In that case, we need to allocate the $|\mathcal{V}|$ operators to the launched streams, with each operator having $|\mathcal{V}|$ choices. Accordingly, the time complexity of the brute-force search method is in the order of $\mathcal{O}(|\mathcal{V}|^{|\mathcal{V}|})$, and it turns out to be an NP problem. Moreover, operator synchronization is required to ensure the operator dependency $\langle u, v \rangle$, which in turn introduces non-negligible delays to DNN inference latency.

Stream Allocation Algorithm. To solve such a complex stream allocation problem in polynomial time, we design a heuristic algorithm in Alg. 1. The key idea of our algorithm is to allocate parallelizable operators to multiple `CUDA Streams` as much as possible. To avoid excessive synchronization operations, we aim to greedily put non-root nodes (*i.e.,* operators) in the same `CUDA stream` as one of their predecessor operators. Specifically, given a computation graph $\mathcal{G}$, *Opara* first initializes a set of streams to be launched $\mathcal{S}$ and then enumerates operators in $\mathcal{V}$ in topological sorting order (line 1-2). For each operator $v \in \mathcal{V}$, it iterates over all of its predecessors $p \in \mathcal{N}_{pred}$ (line 3). If v is the first successor of the current predecessor p, it allocates v to the same stream of p; otherwise, it moves on to the next predecessor (line 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 9-11). In particular, the streams consume GPU resources only when operators are running, which indicates that performance interference among streams does not occur when operators are not executed on GPUs.

Algorithm 1: Stream allocation algorithm in *Opara*.

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

1: **Initialize:** $\mathcal{S} \leftarrow \emptyset$;
2: **for** each operator $v \in \mathcal{V}$ **do**
3: **for** each predecessor $p \in \mathcal{N}_{pred}$ of v **do**
4: **if** v is the first successor of p **then**
5: stream of $v \leftarrow$ stream of p; `// put` v `and` p `in the same stream`
6: **break** out of the loop;
7: **end if**
8: **end for**
9: **if** *stream* of v **is** *null* **then**
10: *stream* of $v \leftarrow$ launching a stream; `// put` v `in a newly launched stream`
11: $\mathcal{S} \leftarrow \mathcal{S} \cup \{stream \text{ of } v\}$;
12: **end if**
13: **end for**
14: **return** $\mathcal{S}$.

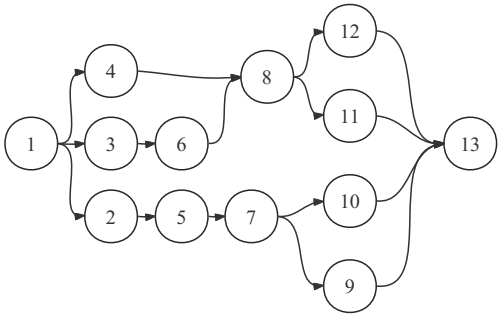

Fig. 4 An example of a computation graph in a DNN model.

As an illustrative example in Fig. 4, we show how Alg. 1 works to allocate the DNN operators to streams. Specifically, as operator 1 has no predecessor, we allocate it to a newly launched stream. For operator 2, which is the first successor of operator 1, we allocate it in the same stream of operator 1 to mitigate operator synchronization overhead. Similarly, operators 5, 7, 9, and 13 are also allocated to the same stream of operator 2. Meanwhile, operators 3, 6, 8, and 11 are allocated to the same stream. Finally, operators 4, 10, and 12 are allocated to three different streams, respectively.

3.2 Model Profiler

As discussed in Sec. 2.3, the blocks in an operator execute the same instructions even with different data they process, which indicates that the GPU resources required by the blocks in an operator are the same. Accordingly, we obtain the resource requirements 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 requirements of operators will be used by Operator Launcher to determine an adequate operator launch order. In particular, our Model Profiler requires profiling each DNN inference *only once* to acquire the resource requirements 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. Therefore, it is essential to determine an optimal operator launch order. A naive solution is to iterate through all possible topological sorting orders and choose 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. We assume that there are n operators in the computation graph, then the time complexity of traversing all topological sorting orders is $\mathcal{O}(n!)$, which is also an NP problem. As a result, we turn to design a heuristic operator launch algorithm to solve such a complex problem.

Resource-aware Operator Launch Algorithm. As illustrated in Sec. 2.3, launching operators with heavy resource requirements 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 [7]. To mitigate such a problem and maximize the GPU utilization, we design a heuristic algorithm in Alg. 2 to greedily prioritize launching the operators with the least amount of GPU resource requirements. Accordingly, Alg. 2 can maximize the parallel executions of multiple operators within a single model. In particular, the potential starvation issue faced by larger operators is noncritical in our scenario, as our algorithm objective is to minimize the DNN inference latency. Specifically, it first initializes and maintains a priority queue $\mathcal{Q}$ of operators in resource-aware operator launch order (line 1). It then retrieves all operators to be launched with an indegree of 0 in a list $\mathcal{L}$ (line 2-6). 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 $\mathcal{Q}$ (line 7-9). Moreover, the operator list $\mathcal{L}$ is continuously updated by adding new operators with an indegree of 0 (line 10-15).

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

Algorithm 2: Operator launch algorithm in *Opara*.

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

Initialize: List of operators to be launched $\mathcal{L} \leftarrow \emptyset$, and $\mathcal{Q} \leftarrow \emptyset$;
for each operator $v \in \mathcal{V}$ **do**
  if *indegree* of $v == 0$ **then**
    $\mathcal{L}.append(v)$;
  end if
end for
while $\mathcal{L}$ is not empty **do**
  $v_{min} \leftarrow$ the operator that requires the least amount of GPU resources in $\mathcal{L}$;
  $\mathcal{L}.remove(v_{min})$, and $\mathcal{Q}.append(v_{min})$; `// launch the operator` v_{min}
  for each successor $s \in \mathcal{N}_{succ}$ of v_{min} **do**
    indegree of $s \leftarrow$ *indegree* of $s - 1$; `// update the` *indegree* `of` s
    if *indegree* of $s == 0$ **then**
      $\mathcal{L}.append(s)$; `// update the operator list to be launched` $\mathcal{L}$
    end if
  end for
end while
return $\mathcal{Q}$.

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. This graph capture process is lightweight and non-intrusive to the DL framework.

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/OparaSys/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 the Python environment. In more detail, we leverage the `torch.cuda.set_stream()` API in PyTorch to launch operators on the `CUDA Stream`. In particular, as operators are executed asynchronously on multiple streams for parallelized execution, appropriate operator synchronizations are required to guarantee that the parallelized computation conforms to operator dependencies. We implement the operator synchronizations 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 manner, rather than modifying the computation graph construction module as in Nimble [6]. Accordingly,

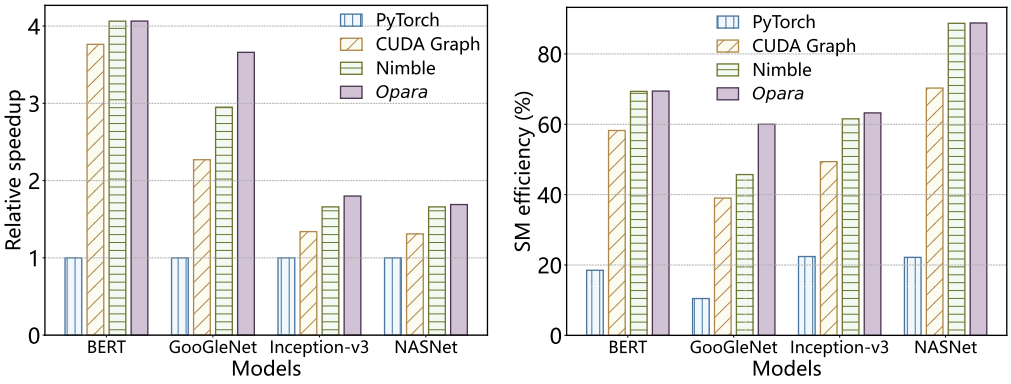

Fig. 5 (a) Relative speedup and (b) SM efficiency of GPUs running representative DNN models with batch size set as 1 achieved by PyTorch, `CUDA Graph`, Nimble, and *Opara* operator scheduling mechanisms.

Opara is non-intrusive to the DL framework, and it can stably work as long as the framework APIs are not updated.

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

We conduct our experiments on an NVIDIA A100-PCIe-40GB GPU with Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz, and an NVIDIA GeForce RTX 2080 SUPER-8GB GPU with an Intel(R) Core(TM) i9-10920X CPU @ 3.50GHz. We implement *Opara* using PyTorch 2.0, CUDA 11.7, and cuDNN 8.5.0, as a plug-in module of PyTorch, as discussed in Sec. 4. We compare DNN inference performance achieved by *Opara* with that achieved by the stock PyTorch (with `CUDA Graph` disabled), default sequential `CUDA Graph`, and Nimble [6]. Our experiments employ representative DNN models, including BERT [12], GooGleNet [11], Inception-v3 [4], and NASNet [13]. All the experiment results are averaged over $1,000$ runs.

5.2 Effectiveness of *Opara*

End-to-end Inference Latency. We first examine whether *Opara* can accelerate the DNN inference. As shown in Fig. 5(a), *Opara* consistently outperforms the other three baselines with four representative DNN models[3]. Specifically, *Opara* can achieve 1.69× to 4.06× speedup compared to the stock PyTorch. This is because *Opara* utilizes

[3] As BERT is GPU memory intensive, we execute it on the A100, while the other three models are executed on the RTX 2080 SUPER.

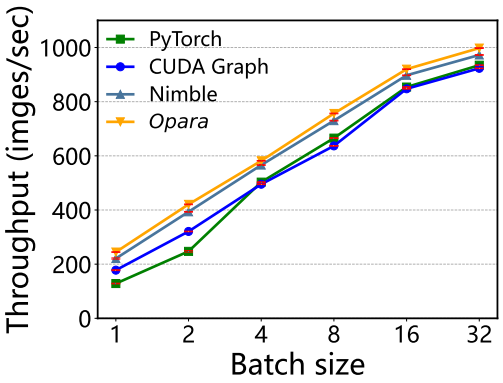

Fig. 6 Inference throughput of Inception-v3 achieved by PyTorch, `CUDA Graph`, Nimble, and *Opara* by varying the batch size from 1 to 32.

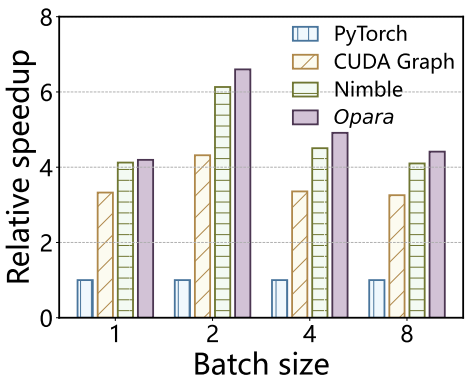

Fig. 7 Relative speedup of Inception-v3 achieved by PyTorch, `CUDA Graph`, Nimble, and *Opara* by varying the batch size from 1 to 8.

`CUDA Graph` to eliminate the function call overhead and operator launch overhead. *Opara* surpasses the default `CUDA Graph` by up to 1.61×, simply because of the parallel execution of DNN operators in *Opara*. Interestingly, *Opara* achieves a small speedup of 1.08× compared to CUDA Graph for BERT, and its speedup relative to Nimble for BERT and NASNet is negligible (1.01×). This is because many memory-intensive operators exist in BERT and NASNet, which limits the performance gains of operator parallelism, due to increased resource competition among parallelized operators. Furthermore, *Opara* outperforms Nimble by up to 1.24× because it judiciously schedules the operator with the lowest GPU resource consumption for each kernel launch time, thereby reducing GPU idle time and maximizing the operator parallelism.

GPU Utilization. To unveil the performance gains of *Opara*, we proceed to look into the GPU utilization (*i.e.,* SM efficiency of GPUs) during the execution of the four models. As shown in Fig. 5(b), *Opara* exhibits a similar improvement in GPU utilization compared to the three baselines as in Fig. 5(a). Specifically, *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 BERT, GooGleNet, Inception-v3, and NASNet by 19%, 55%, 34%, and 26%, respectively. Such performance gains come from the parallelized execution of operators. When compared to Nimble, *Opara* boosts the GPU utilization by 1.01× to 1.35× mainly due to our adequate operator launch order in *Opara* to minimize GPU idle time.

Throughput under Different Batch Sizes. We further examine the improvement of *Opara* in terms of DNN inference throughput across varying batch sizes. As depicted in Fig. 6, we observe that *Opara* consistently surpasses the other three baselines 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 1.38× 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, which results in less amount of GPU resources available for the execution of parallelized operators. The experiment

Table 1 Computation overhead (in milliseconds) of the stream allocation algorithm in *Opara* and Nimble with different DNN models.

	BERT	GooGleNet	Inception-v3	NASNet
Opara	0.58	0.27	0.50	1.75
Nimble [6]	20.8	5.80	14.40	257.83

results above also demonstrate that maximizing the parallelism opportunities among DNN operators can expedite DNN inference while improving the GPU utilization.

Effectiveness of *Opara* on GPUs with Sufficient Resources. To validate the efficacy of *Opara* on GPUs equipped with powerful computing resources, we repeat experiments by comparing performance gains with the three baselines on the A100. As shown in Fig. 7, we observe that *Opara* consistently outperforms the three baselines by varying the batch size from 1 to 8, mainly due to the better performance of operator parallelism on the A100. In more detail, both *Opara* and Nimble can benefit from the operator parallelism compared to the stock PyTorch and default `CUDA Graph`. Moreover, *Opara* can achieve 1.02× to 1.09× speedup compared to Nimble on the A100, which are slightly smaller performance gains compared to that on the RTX 2080 SUPER. This is because sufficient resources are available on the A100, resulting in GPU under-utilization of DNN inference and limited optimization opportunities for the operator launch order. As the batch size increases to 8, a higher GPU resource requirements of the operator enlarge the optimization space for adjusting the operator launch order, leading to higher performance gains of *Opara* compared to Nimble.

5.3 Runtime Overhead of *Opara*

We evaluate the runtime overhead of *Opara* in terms of algorithm computation time and inference profiling overhead. We first compare the computation overhead of the stream allocation algorithm in *Opara* and Nimble [6]. As listed in Table 1, *Opara* incurs stream allocation computation time of 0.58 ms, 0.27 ms, 0.5 ms, and 1.75 ms for BERT, GooGleNet, Inception-v3, and NASNet, respectively. In contrast, such algorithm computation time in Nimble is 20.8 ms, 5.80 ms, 14.40 ms, and 257.83 ms for the four models, respectively, which is at least a magnitude larger than that of *Opara*. This is because Nimble requires transforming the computation graph into a bipartite graph, together with an exhaustive search in the bipartite graph. Such a process is time-consuming, with an algorithmic time complexity in the order of $\mathcal{O}(n^3)$, where n is the number of operators in a DNN DAG. In contrast, the time complexity of *Opara* can be reduced to the order of $\mathcal{O}(n)$. This is because the inner loop of the stream allocation algorithm in *Opara* only depends on the maximum width of the computation graph, which is typically quite small (*i.e.,* below 20).

As DNN models become increasingly complex and network depth increases in the future [14], the number of operators will also grow exponentially. Such an algorithm overhead in Nimble becomes unacceptable when the number of operators is large enough. 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, Graphi [15] proposes a profiler-based DNN operator scheduling system to minimize resource contention in a multi-core CPU platform. Rammer [16] proposes the concept of "rtask", which allows finer-grained computing scheduling on the device. Such an approach enables computing task combinations in different operators executed on SMs. To maximize the operator parallelism, Nimble [6] leverages the bipartite graph algorithm to adequately launch operators on `CUDA streams`. Such an algorithm requires a lengthy search process and neglects the potential GPU resource wastage due to inadequate operator launch orders. A recent work named IOS [8] deploys operator fusion and dynamic programming to determine operator parallelization strategies. Such a complex method takes hours of searching and overlooks the operator launch overhead. Different from the prior works above, *Opara* utilizes `CUDA Graph` to eliminate such a performance overhead. It also employs a *lightweight* stream allocation algorithm to parallelize the execution of operators. To minimize the GPU idle time, *Opara* determines an efficient operator launch order according to operator resource requirements.

Inter-operator Parallelism among Different Models. A number of works aim to improve GPU utilization by co-locating multiple models so that operators from different models can be parallelized. For example, S^3DNN [17] designs a heuristic parallelism algorithm to schedule each operator of different models to corresponding streams. HiveMind [18] leverages model batches and operator fusion techniques to construct a large computation graph that is comprised of multiple DNN models. Both *iGniter* [19] and GSLICE [5] employ the NVIDIA Multi-Process Service (MPS) to co-locate multiple DNN inference on GPUs to improve GPU utilization. Yu et al. [20] partition each model into multiple phases to balance the GPU load and improve GPU utilization in a multi-model co-location scenario. Abacus [21] selects the optimal combination of operator co-locations between different models. Different from optimizing the inference model co-location, *Opara* minimizes model inference latency while increasing the GPU utilization by parallelizing operators within a single model. Moreover, it achieves the inter-operator parallelism as a plug-in module of PyTorch 2.0 without developing a new DL runtime or framework.

Intra-operator Parallelism. The performance of an individual operator running on GPUs has been well studied. Existing DL frameworks, such as PyTorch, TensorFlow [22], and TensorRT [23], employ expert-optimized operator libraries. To reduce human interactions, TVM [24] uses machine learning methods to automatically search for efficient codes according to the artificially specified parameter space. Such a search process is time-consuming and it still requires manual definition of the search space. To achieve fully automated code generation, FlexTensor [25] and Ansor [26] implement an automatic search space construction. As the number of parallel units in commodity GPUs increases, an individual DNN operator cannot fully utilize the GPU resources.

Orthogonal to prior works above, *Opara* focuses on designing lightweight heuristic algorithms to achieve inter-operator parallelism within a single model. It can work with the intra-operator parallel optimization methods to significantly improve GPU resource utilization.

7 Conclusion and Future Work

This paper presents the design and implementation of *Opara*, a lightweight DNN operator scheduling framework to speed up DNN inference on GPUs. By reducing the synchronization overhead among operators, *Opara* designs a lightweight 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 optimal operator launch order to maximize the GPU utilization. Extensive prototype experiments show that *Opara* can improve the performance of DNN inference by up to 24%, in comparison 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 inference latency and the performance interference among operators caused by inter-operator parallelism, as well as (2) implementing *Opara* on top of other DL frameworks, *e.g.,* TensorFlow.

References

[1] Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M.P., Shyu, M.-L., Chen, S.-C., Iyengar, S.S.: A Survey on Deep Learning: Algorithms, Techniques, and Applications. ACM Computing Surveys **51**(5), 1–36 (2018)

[2] Liu, L., Wang, Y., Shi, W.: Understanding Time Variations of DNN Inference in Autonomous Driving. arXiv preprint arXiv:2209.05487 (2022)

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

[4] Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the Inception Architecture for Computer Vision. In: Proc. of IEEE CVPR, pp. 2818–2826 (2016)

[5] Dhakal, A., Kulkarni, S.G., Ramakrishnan, K.: GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform. In: Proc. of ACM SOCC, pp. 492–506 (2020)

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

[7] Amert, T., Otterness, N., Yang, M., Anderson, J.H., Smith, F.D.: GPU Scheduling on the NVIDIA TX2: Hidden Details Revealed. In: Proc. of IEEE RTSS, pp. 104–115 (2017)

[8] Ding, Y., Zhu, L., Jia, Z., Pekhimenko, G., Han, S.: IOS: Inter-Operator Scheduler for CNN Acceleration. In: Proc of MLSys, pp. 167–180 (2021)

[9] Gilman, G., Walls, R.J.: Characterizing Concurrency Mechanisms for NVIDIA GPUs under Deep Learning Workloads. ACM SIGMETRICS Performance Evaluation Review **49**(3), 32–34 (2022)

[10] NVIDIA: CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide

[11] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A.: Going Deeper with Convolutions. In: Proc. of IEEE CVPR, pp. 1–9 (2015)

[12] Devlin, J., Chang, M.-W., Lee, K., Toutanova, K.: BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv preprint arXiv:1810.04805 (2018)

[13] Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning Transferable Architectures for Scalable Image Recognition. In: Proc. of IEEE CVPR, pp. 8697–8710 (2018)

[14] Bianchini, M., Scarselli, F.: On the Complexity of Neural Network Classifiers: A Comparison between Shallow and Deep Architectures. IEEE Transactions on Neural Networks and Learning Systems **25**(8), 1553–1565 (2014)

[15] Tang, L., Wang, Y., Willke, T.L., Li, K.: Scheduling Computation Graphs of Deep Learning Models on Manycore CPUs. arXiv preprint arXiv:1807.09667 (2018)

[16] Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., Zhou, L.: Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In: Proc. of USENIX OSDI, pp. 881–897 (2020)

[17] Zhou, H., Bateni, S., Liu, C.: S^3DNN: Supervised Streaming and Scheduling for GPU-Accelerated Real-Time DNN Workloads. In: Proc. of IEEE RTAS, pp. 190–201 (2018)

[18] Narayanan, D., Santhanam, K., Phanishayee, A., Zaharia, M.: Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. In: Proc. of NeurIPS Workshop on Systems for Machine Learning, pp. 1–8 (2018)

[19] Xu, F., Xu, J., Chen, J., Chen, L., Shang, R., Zhou, Z., Liu, F.: iGniter: Interference-Aware GPU Resource Provisioning for Predictable DNN Inference in

the Cloud. IEEE Transactions on Parallel & Distributed Systems **34**(03), 812–827 (2023)

[20] Yu, F., Bray, S., Wang, D., Shangguan, L., Tang, X., Liu, C., Chen, X.: Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU. In: Proc. of IEEE/ACM ICCAD, pp. 1–9 (2021)

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

[22] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., *et al.*: TensorFlow: A System for Large-Scale Machine Learning. In: Proc. of USENIX OSDI, pp. 265–283 (2016)

[23] Vanholder, H.: Efficient Inference with TensorRT. In: Proc. of GPU Technology Conference, pp. 2–2 (2016)

[24] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., et al.: TVM: An Automated End-to-End Optimizing Compiler for Deep Learning, 579–594 (2018)

[25] Zheng, S., Liang, Y., Wang, S., Chen, R., Sheng, K.: FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In: Proc. of ACM ASPLOS, pp. 859–873 (2020)

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