HIOS: Hierarchical Inter-Operator Scheduler for Real-Time Inference of DAG-Structured Deep Learning Models on Multiple GPUs

Turja Kundu
Department of Computer Science and Engineering
University of North Texas
Denton, TX, USA
turjakundu@my.unt.edu

Tong Shu*
Department of Computer Science and Engineering
University of North Texas, Denton, TX, USA
tong.shu@unt.edu
ORCID: 0000-0001-8617-1772

***Abstract*—Neural-network-enabled data analysis in real-time scientific applications imposes stringent requirements on inference latency. Meanwhile, recent deep learning (DL) model design trends to replace a single branch with multiple branches for high prediction accuracy and robustness, which makes inter-operator parallelization become an effective approach to improve inference latency. However, existing inter-operator parallelization techniques for inference acceleration are mainly focused on utilization optimization in a single GPU. With the data size of an input sample and the scale of a DL model ever-growing, the limited resource of a single GPU is insufficient to support the parallel execution of large operators. In order to break this limitation, we study hybrid inter-operator parallelism both among multiple GPUs and in each GPU. In this paper, we design and implement a hierarchical inter-operator scheduler (HIOS) to automatically distribute large operators onto different GPUs and group small operators in the same GPU for parallel execution. Particularly, we propose a novel scheduling algorithm, named HIOS-LP, which consists of inter-GPU operator parallelization through iterative longest-path (LP) mapping and intra-GPU operator parallelization based on a sliding window. In addition to extensive simulation results, experiments with modern convolutional neural network benchmarks demonstrate that our HIOS-LP outperforms the state-of-the-art inter-operator scheduling algorithm IOS by up to 17% in real systems.**

***Index Terms*—inter-operator parallelism, deep learning inference, multi-GPU environment**

I. Introduction

While real-time deep learning (DL) applications powered by robust multi-branch neural architectures have become a driving force for in-situ data analysis in vital scientific domains, many scientific applications advance explicit demands on inference latency. For example, in energy fusion, the fast-evolving dynamics of the plasma control systems, which manage the reactor diagnostics, impose stringent latency limitations on DL inference in the millisecond scale [1], [2].

In order to improve inference latency, some research efforts have been devoted. On one hand, inter-operator parallelism has been explored within a single GPU to optimize hardware utilization [3], [4]. However, existing inter-operator scheduling techniques are only fit for small operators, because the parallel execution of large operators in a single GPU causes unnecessary hardware resource contention. On the other hand, the CPU is also utilized to accelerate a DL inference by collaborating with GPU [5], [6]. Unfortunately, such a technique leads to prediction accuracy loss, because data compression and approximate computing have to be incorporated to avoid high data transfer overhead via the low-bandwidth PCIe bus between the CPU and the GPU.

Recently, real-time big-data processing faces the demands of huge volume and high velocity. In a convolutional neural network (CNN) inference, high-resolution input images usually make both the computation task of each operator and intermediate data between operators extremely large. As a result, a single GPU does not have enough computational power to support such inter-operator parallelism.

At present, various servers and edge computers can be equipped with multiple homogeneous GPUs connected via high-speed NVLink to enhance total computational power. For instance, Dell Precision 5820 Tower Workstation and Dell PowerEdge R750XA Rack Server can support two and four powerful GPUs, respectively. Also, supercomputers and clusters have high-speed network interconnect among GPU compute nodes. Unfortunately, although these platforms provide new opportunities for inter-operator parallelism on multiple GPUs, this problem is still largely underexplored. To meet ever-growing demands on the low latency of DL inference with large operators by fully utilizing the richer computing resource of multiple GPUs, this paper studies inter-operator scheduling for a DL inference on multiple GPUs by integrating inter-GPU and intra-GPU operator parallelization.

Inter-operator parallelism among multiple GPUs and within each GPU provides more computational resources and a richer scheduling space to reduce inference latency indeed, but the inter-operator scheduling problem on multiple GPUs is much more challenging to be solved. 1) Spatially, mapping operators onto GPUs does not only entail allocating independent operators onto different GPUs to improve the degree of parallelism, but also involves assigning dependent operators onto the same GPU to minimize data transfer time because communication time between GPUs compared to operator

This research work is done in the Smart High-performance and Ubiquitous Systems (SHU'S) lab at the University of North Texas.

* Corresponding author: Tong Shu, Department of Computer Science and Engineering, University of North Texas, Denton, TX 76207, USA

latency is not negligible. 2) Temporally, assigning operators in stages should not only avoid hardware under-utilization and resource contention among operators in a single GPU, but also needs to account for explicit and implicit dependencies between operators on different GPUs. 3) The interaction between spatial and temporal optimization makes the joint two-dimensional optimization problem very intractable.

To address these challenges, we propose a Hierarchical Inter-Operator Scheduler based on the Longest-Path (HIOS-LP). Due to the extremely high computational complexity of the hybrid inter-GPU and intra-GPU operator scheduling problem, we decouple this two-dimensional problem into two loosely coupled one-dimensional subproblems as follows: i) The spatial operator-to-GPU mapping subproblem is onto which GPU each operator should be mapped, and ii) the temporal operator-to-stage allocation subproblem is at what time each operator should start in their respective GPUs. For the spatial operator-to-GPU mapping, we partition the computational graph of a DL model into groups of dependent operators with the least dependencies between groups to increase the degree of parallelism across GPUs. Then, we assign all the operators in each group onto the same GPU to minimize data transfer overheads. Specifically, we iteratively choose the longest path, whose middle operators have no dependencies with selected operators, from unselected operators in the computational graph and greedily map the entire path onto the most suitable GPU. For the temporal operator-to-stage allocation, we dynamically adjust the degree of parallelism in stages within each GPU using a sliding window to avoid under-utilization and resource contention. Also, we adopt topological sorting in the original computational graph and loop detection in the scheduled computational graph, which includes groups of concurrent operators (i.e. stages), to address explicit and implicit operator dependencies, respectively.

In this paper, our contributions are summarized as follows:

- We point out a substantial inference latency issue for real-time DL applications: when a large operator can saturate hardware's high parallelism, inter-operator parallelism within a single GPU would degrade inference efficiency. Therefore, inter-operator parallelism across multiple GPUs is crucial to accelerate the inference of multi-branch DL models. **As far as we know, this paper is the first work on hybrid inter-GPU and intra-GPU inter-operator parallelization to improve the latency of a single inference.**
- We design and implement a hierarchical inter-operator scheduler (HIOS)[1] based on CUDA-aware MPI and cuDNN, incorporating a novel HIOS-LP algorithm that we designed. The proposed HIOS-LP can identify a near-optimal schedule for hybrid inter-GPU and intra-GPU inter-operator parallelization. This technique can serve as a platform-agnostic general technique to speed up the inference latency for popular frameworks such as PyTorch [7] and TensorFlow [8] due to its orthogonality.
- We apply HIOS-LP to two real-life CNN models with varying input image sizes and generate customized operator schedules. As a result, the extensive experimental results verify the superiority of HIOS-LP over the state-of-the-art inter-operator scheduler IOS by up to 16.5% and over HIOS-MR (our other algorithm for inter-operator scheduling on multiple GPUs) by up to 16.8%.

[1]The source code is available at https://github.com/SHUs-Lab/HIOS

II. Background and Motivation

High-resolution data often need to be processed by DL models in real-time applications, such as biomedical analysis and remote sensing [9]–[11]. Using high-resolution images as direct inputs for DL models poses a challenge in inference latency, because the operator size (i.e., the computational workload of an operator) is rocketing while the data size of an input sample is rapidly growing. However, simply resizing the images to a lower resolution significantly decrease prediction accuracy. For example, in irrigation type mapping research [12], very-high-resolution satellite imagery needs to be analyzed through CNN inference, and an original image has the resolution of 5000×5000 pixels. For acceptable inference efficiency, geospatial scientists have to downsize the original image to 500×500 pixels or below in preprocessing. Unfortunately, such approximation suppresses fine details and loses a lot of valuable information. Therefore, it is crucial to accelerate the inference of a DL model composed of large operators.

A common practice to improve DL inference efficiency is parallelization [4]. Intra-operator parallelism can split a large operator into smaller ones through data partitioning, but often involves additional computational overhead, further degrading the inference efficiency and overall system throughput. Thus, such a technique is only used when the memory size of a single GPU is insufficient to support the execution of an entire operator. Without the limit from memory size, inter-operator parallelism is usually put into practice to improve inference efficiency [3], [4], [13].

A. Resource Contention v.s. Under-Utilization

A DL model consists of various operators with different compute workloads, varying with the input data size. A small operator running alone on a GPU can only occupy a portion of streaming multiprocessors (SMs), leading to resource underutilization. Therefore, multiple small operators executing in parallel on one GPU can increase resource utilization and improve inference efficiency. However, a large operator running alone on a GPU may fully utilize all the SMs and exhaust the majority of registers and shared memory in each SM. In this case, two or more large operators executing in parallel on one GPU causes serious resource contention and context switch overheads. To validate this, we conduct the following experiment and take an example of a convolution operator with the kernel size of (5, 5) and the stride of (1, 1), whose input consists of 48 image channels with exponentially increasing size from 8×8 to 1024×1024. For each input image size, we measure the execution time of two such convolution

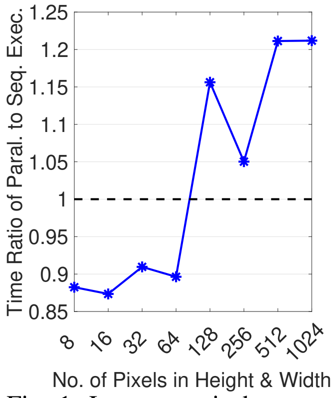


Fig. 1: Latency ratio between parallel and sequential execution of two identical convolutional operators with varying input image sizes

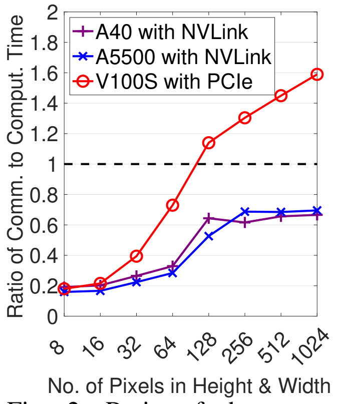


Fig. 2: Ratio of the transfer time of input data to the computation time of a convolutional operator for different input image sizes

operators running separately in sequential and parallel on a single Nvidia Ampere A40 GPU. The ratios between their sequential execution times and parallel execution times are plotted in Fig. 1, which shows that the parallel execution time is shorter than the sequential execution time for input image sizes of no more than 64×64 (i.e., low workloads), but is longer for input image sizes of 128×128 and beyond (i.e., high workloads). Therefore, inter-GPU operator parallelization becomes critical for the inference acceleration of DL models with large operators.

B. *Computational Workload v.s. Communication Overhead*

Multi-GPU platforms, such as a multi-GPU server/workstation and a GPU cluster, break through the capacity limit of a single GPU and provide more computing resources to support inter-operator parallelism among multiple GPUs. Also, NVLink and NVSwitch techniques [14] sharply reduce the data communication time between GPUs and decrease the overheads of inter-GPU operator parallelization in comparison to the traditional PCIe bus. For example, we measure the execution time of the convolution operator aforementioned in §II-A and the data transfer time of its input image between two homogeneous GPUs over varying input image sizes on three dual-GPU platforms with different GPUs and interconnections between GPUs. Then, we plot the ratio of the communication time to the computation time in Fig. 2, which shows that communication overheads are not negligible. However, dual Nvidia Ampere A40 or RTX A5500 GPUs connected via an NVLink bridge have a lower time ratio of data communication to operator computation than two Nvidia Tesla V100S GPUs connected via a PCIe Gen3 interface and thus are a more suitable platform to support inter-GPU inter-operator parallelization. When the communication overhead is much higher than the tiny computation tasks of operators, parallel execution of operators within a single GPU is more efficient than simultaneously running operators over multiple GPUs. In contrast, when the communication overhead is relatively low in comparison to the computation task of an

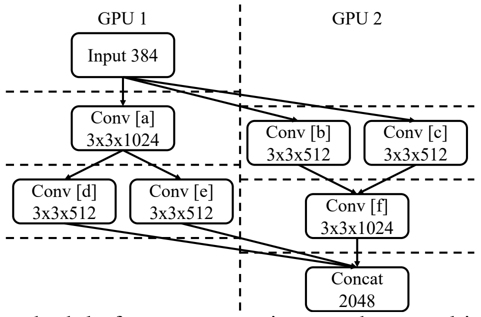


Fig. 3: A schedule for a computation graph on multiple GPUs

operator, the communication latency can often be hidden by parallelizing operator computation and data communication. Because the workloads of various operators are very different in a DL model, hybrid inter-operator parallelism both between GPUs and within each GPU can flexibly exploit the resource of multi-GPU platforms to minimize inference latency for real-time DL applications.

III. Problem Formulation

In this section, we formulate an inter-operator scheduling problem to minimize DL inference latency on multiple GPUs.

A. *Cost Models*

Computation Graph. The structure of a DL model, such as a CNN, can be defined by a directed acyclic graph (DAG) $G = (V, E)$, where each vertex $v \in V$ represents an operator, such as convolution, pooling, and linear operators, and each edge $e = (u, v) \in E$ indicates an operator dependency, i.e., a tensor that is the output of operator u and the input of operator v. Each vertex v has a weight for the execution time $t(v)$ of this operator running alone on a GPU, and each edge e also has a weight for the data transfer time $t(e) = t(u, v)$ between operators u and v on two different GPUs.

Graph Partition. A workstation or server is equipped with M homogeneous GPUs, directly connected through high-speed NVLink/NVSwitch (where M is an even number in practice). The computation graph is partitioned into m ($\leq M$) computation subgraphs. An MPI program instance has m processes, one of which launches all the operators in one subgraph to execute on the same GPU. In contrast, operators in different subgraphs run on different GPUs. Fig. 3 shows that the computation graph is partitioned into two computation subgraphs, which are assigned to different GPUs.

Parallelization Strategy. Hierarchical inter-operator parallelism is applied, including two levels: 1) parallel execution of independent operators in different computation subgraphs on their respective GPUs, and 2) concurrent execution of independent operators in the same computation subgraph on the assigned GPU. Specifically, to concurrently execute multiple operators within the same GPU, these operators need to be put into different CUDA streams. Operators (i.e., CUDA kernels) in the same CUDA stream are executed sequentially; operators in different CUDA streams will be executed concurrently or in parallel if there are enough SM resources. The maximum number of CUDA streams can be preset to be L.

Stage. To both take advantage of intra-GPU inter-operator parallelism and prevent excessive resource contention among concurrent operators within each GPU, each computation subgraph is partitioned into multiple stages, each of which consists of a set of independent operators. Stages are executed sequentially and operators in the same stage are executed concurrently. After the input data of all the operators in a stage S are ready, the total time of concurrent execution of these operators with the same start time is denoted by $t(S)$. In fact, if a part of these operators has ready input data, they may execute earlier in a practical system. However, we assume that they start simultaneously because this assumption does not only make the concurrent execution time of a set of independent operators easy to measure, but can also guarantee a tight upper bound of inference latency. Stages of different computation subgraphs are not synchronized. Fig. 3 shows a possible schedule: each of the two computation subgraphs is partitioned into two stages. In the first computation subgraph, the first stage contains operator a, and the second stage contains operators d and e; in the second computation subgraph, the first stage contains operators b and c, and the second stage contains operator f.

Operator Synchronization. Since there exists data dependency between operators allocated onto different GPUs, an operator can only start to run when all its input data are ready on its GPU. As a result, when an operator in the current stage waits for the input data to be transferred from another GPU, its current GPU might become idle. For this precedence constraint, if there exists a directed edge (i.e., execution dependency) from an operator in the jth stage $S_{i,j}$ to an operator in the j'th stage $S_{i,j'}$ on the ith GPU, the finish time $t^F(S_{i,j})$ of stage $S_{i,j}$ must not be later than the start time $t^S(S_{i,j'})$ of stage $S_{i,j'}$. Furthermore, if there exists a directed edge e from an operator in stage $S_{i,j}$ on the ith GPU to an operator in stage $S_{i',j'}$ on the i'th GPU, the finish time $t^F(S_{i,j})$ of stage $S_{i,j}$ plus the data transfer time $t(e)$ must not be later than the start time $t^S(S_{i',j'})$ of stage $S_{i',j'}$.

Schedule. We define a computation graph G's schedule Q that maps all the operators onto no more than M GPUs as $Q = \{Q_i | 1 \le i \le M\}$, where $Q_i = \{S_{i,j} | 1 \le j \le K_i\}$. Here, $S_{i,j}$ is the set of concurrent operators in the jth stage on the ith GPU, and the total number K_i of stages on the ith GPU is 0 if no operator runs on the GPU. For example, the schedule for Fig. 3 is $Q = \{Q_1, Q_2\}$, where $Q_1 = \{\{a, d\}, \{e\}\}$ and $Q_2 = \{\{b, c\}, \{f\}\}$. The schedule Q_i executes the computation subgraph from the first stage $S_{i,1}$ to the last stage S_{i,K_i} sequentially. $S_{i,j}$ may contain only one operator (e.g., a very large operator that saturates the entire GPU).

B. Problem Definition

Given the computation graph of an arbitrary DAG-structured neural network $G = (V, E)$, M homogeneous GPUs in a symmetric multiprocessing (SMP) system, the execution time $t(S)$ of each set S of no more than K independent operators concurrently running on a single GPU, and the communication time $t(u, v)$ of data transfer between operators u and v running on different GPUs.

Question is to find a feasible schedule Q to minimize the inference latency for the given computation graph G running on at most M GPUs.

Subject to the precedence constraint

$$t^S(S_{i',j'}) - t^F(S_{i,j}) \ge \begin{cases} t(u,v), & \text{if } i \ne i' \\ 0, & \text{if } i = i' \end{cases}, \quad \forall (u,v) \in E, u \in S_{i,j}, v \in S_{i',j'}$$

Our problem is NP-hard, because it is provable to find an optimal schedule for a set of jobs is NP-complete [15], and this job scheduling problem can be reduced to our problem in polynomial time.

IV. Algorithm Design

This section first elaborates the design of our HIOS-LP algorithm, including inter-operator parallelization among multiple GPUs in §IV-A and within each GPU in §IV-B. Then, we analyze the time complexity of HIOS-LP in §IV-C.

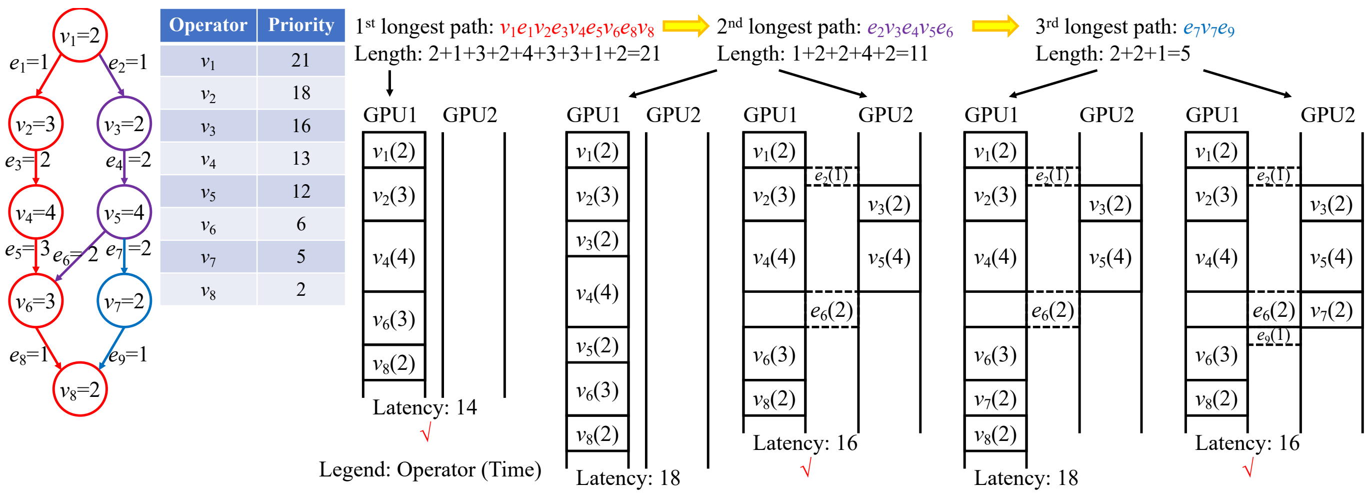
A. Inter-GPU Inter-Operator Parallelization

In inter-operator scheduling over multiple GPUs, two questions need to be answered for each operator, i.e., on which GPU and at what time to execute each operator. On one hand, if we determine each operator's GPU and start time one by one, it is difficult to consider the relationship between operators for global optimization, such as reducing unnecessary data transfer between different GPUs and hiding the data communication time by leveraging operators' computation time. On the other hand, if we consider the spatial and temporal scheduling of all operators at one time, the problem is too complicated to deal with. To make a tradeoff between optimity and feasibility, we partition all the operators into multiple groups and incrementally choose a group for scheduling. In each group, operators are closely related to each other and critical to the end-to-end inference latency.

Based on this idea, the algorithm for inter-GPU operator parallelization is described in Lines 1–16 of Alg. 1. Initially, the unscheduled computation subgraph G' is the whole computation graph G, and the computation subgraph G_i allocated onto the ith GPU, which has no operators (in Lines 1–3).

Spatial Operator Mapping onto GPUs (in Lines 5–16): The schedule of the longest path (LP) in the original computation graph G is viewed as the performance bottleneck for end-to-end latency optimization. Subsequently, the longest path in the unscheduled computation subgraph G' also plays the most important role in the inference latency. Therefore, we iteratively choose the longest path from G' to form a group of operators for their GPU allocation as a whole every time. When all the operators on the longest path P are allocated onto the same GPU, the data transfer time along P disappears.

It is worth noting that because the longest path is identified prior to the GPU allocation of its operators, we pay attention to the upper bound of path latency (where any two adjacent operators along any path are supposed to be on different GPUs at worst), and thus count both vertex and edge weights (i.e.,



Operator	Priority
v_1	21
v_2	18
v_3	16
v_4	13
v_5	12
v_6	6
v_7	5
v_8	2

Fig. 4: An example to illustrate how Alg. 1 works.

Algorithm 1 Longest-Path-Based Operator Scheduling

Inputs: computation graph $\boldsymbol{G}$, number $\boldsymbol{M}$ of GPUs, execution time $\boldsymbol{t(S)}$ of each set S of independent operators on one GPU, and data transfer time $\boldsymbol{t(u,v)}$ between each pair of operators u and v on different GPUs.

Output: schedule $\boldsymbol{Q}$ with minimal inference latency L.

```
1: $G' = G$;
2: for $i \in \{1, \cdots, M\}$ do
3:     $G_i = \emptyset$;
4: while $G' \neq \emptyset$ do
5:     From $G'$, find the longest path $P$ whose intermediate vertices
       (excluding the first and last vertices in $P \cap G'$) have no edges
       from/to any vertex in the scheduled computation subgraph $G$-$G'$;
6:     $G' = G' - P$;
7:     $L = +\infty$;
8:     for $i \in \{1, 2 \cdots, M\}$ do
9:         $G_i' = G_i \cup P$;
10:        Sort all vertices in $G$-$G'$ according to the descending
           order of their priority indicators, as $v_1, v_2, \ldots, v_n$;
11:        for $j \in \{1, 2, \ldots, n\}$ do
12:            Schedule $v_j$ at the earliest available start time on its
               GPU;
13:        Obtain the corresponding schedule $Q'$ and its latency $L'$;
14:        if $L' < L$ then
15:            $L = L'$; $I = i$; $Q = Q'$;
16:    $G_I = G_I \cup P$;
17: ($Q$, $L$) = parallelize($G$, $Q$);    // See Alg. 2.
18: return ($Q$, $L$).
```

operator computation time and data transfer time between GPUs) towards the path length in finding the longest path.

While exploring path candidates for the longest path in G', it is necessary to enforce a constraint that all the intermediate vertices (except the first and last vertices) on a path candidate have no data dependency with any mapped operator (in Line 5). The reason is as follows. Without loss of generality, we assume that an intermediate vertex v on a path candidate P has an incoming edge from a vertex u in the scheduled computation subgraph, so operator v cannot start before receiving input data from operator u. In this case, even if v and its preceding operator on P are on different GPUs, the data transfer time between them might be hidden by awaiting data from operator u. Due to this potential time constraint relaxing, the previous vertices before v on P might have a better mapping option for latency reduction.

After we find the longest path P under the aforementioned constraint from the unscheduled computation subgraph (called the longest valid path), all the vertices and edges on P is removed from the unscheduled computation subgraph G' (in Line 6). (Note that an edge is removed from G' only if its two vertices are both removed.) Then, we try to temporarily map all the operators of the path P onto one GPU at one time and exhausted all the GPUs (in Lines 8 and 9). In each try, we intend to schedule operators on P and all the previously mapped operators in their respective GPUs according to the temporal operator scheduling method (described in the next paragraph). Next, we search for the best GPU to map the path P onto so that the scheduled computation subgraph G-G' has the minimum end-to-end latency (in Lines 14 and 15), and eventually allocate all the operators of P onto the best GPU (in Line 16).

Temporal Operator Scheduling for Data Dependency (in Lines 10–13): We first define the priority indicator $p(v)$ of operator v as the length (including the sum of all the vertex and edge weights) of the longest path from v to the last operator in the original computation graph G (equivalent to the opposite number of v's latest start time in G). Let g_j denote the GPU that operator v_j is mapped to. Then, we sort all the vertices in the mapped computation subgraph G-G' according to the descending order of their priority indicators and label them as $v_1, v_2, \ldots, v_n$ (in Line 10). This order is a topological sort. Next, following this order, we schedule each operator v_j one by one at their earliest available start time on its GPU g_j (in Lines 11–12). As a result, we can obtain the corresponding schedule and measure its inference latency (in Line 13).

Example: Fig. 4 demonstrates how HIOS-LP identifies the optimal schedule of inter-GPU operator parallelization for a given computation graph of eight operators on two GPUs. The execution times of each operator and the transfer time of each data dependency are listed on the left of Fig. 4. Based on the original computation graph, the priority indicators of all the operators are calculated and listed in the table in Fig. 4. First, we find the longest path $P_1 = \{v_1, e_1, v_2, e_3, v_4, e_5, v_6, e_8, v_8\}$ in red, and initially map all its operators $\{v_1, v_2, v_4, v_6, v_8\}$ onto GPU 1 due to the homogeneity of GPUs. Then, we find the second longest valid path $P_2 = \{e_2, v_3, e_4, v_5, e_6\}$ in purple from the unmapped subgraph. Note that the start and end edges $\{e_2, e_6\}$ are included in P_2. Here, we do not select the longer path of $\{e_2, v_3, e_4, v_5, e_7, v_7, e_9\}$, because its intermediate operator v_5 has an outgoing edge to operator v_6 that has been mapped to GPU 1 and thus this path is not a valid candidate. Next, we try to separately map P_2 onto GPU 1 and 2, and obtain two schedules with the minimum latency of 18 and 16, respectively. Therefore, we decide to map P_2 onto GPU 2. In the following, we find the next longest valid path $P_3 = \{e_7, v_7, e_9\}$ in blue from the unmapped subgraph, and try to map it onto each GPU. Through the temporal operator scheduling method, we obtain a schedule with a latency of 18 if P_3 is on GPU 1 and another schedule with a latency of 16 if P_3 is on GPU 2. Eventually, P_3 is mapped onto GPU 2, and the inter-GPU operator scheduling algorithm finds the optimal schedule with the inference latency of 16.

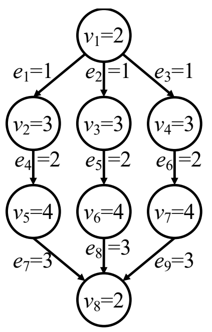
B. *Intra-GPU Inter-Operator Parallelization*

Given a computation graph G and its schedule Q with inter-operator parallelism on multiple GPUs and sequential operator execution on each GPU, we further explore opportunities for inter-operator parallelization on each GPU for inference latency reduction. However, the algorithm for an inter-operator scheduler (IOS) in [4] cannot be applied here for the following reasons. First, IOS is an exact algorithm with exponential time complexity, and thus it is unaffordable to apply the IOS algorithm to operator scheduling on each GPU due to its unscalability for large-scale DL models. Second, IOS only schedules operators on a single GPU, so do not consider the significant effect of data dependency/synchronization between operators across GPUs, leading to the suboptimality of operator scheduling within each GPU.

To meet the requirements on the low time complexity of intra-GPU operator scheduling and its compatibility with data dependency across GPUs, we propose an efficient intra-GPU inter-operator parallelization method, described in Alg. 2. First, we sort all the vertices in the given computation graph G according to the descending order of their priority indicators and label them as $v_1, v_2, \dots, v_n$ (in Line 2). Along this order, we iteratively slide an operator window W to make its beginning align with each operator in turn, where the window masks no more than w continuous operators on one GPU (in Lines 3–6). For a window of size p ($2 \leq p \leq w$) starting at an operator, if all the operators masked by the window are independent of each other, we will try to group them for parallel execution in the same stage (in Lines 7–9), and check whether a cycle is involved after these operators

Algorithm 2 Intra-GPU Inter-Operator Parallelization
Function Name: parallelize()
Inputs: computation graph $\boldsymbol{G}$, its schedule $\boldsymbol{Q}$ with inter-operator parallelism among GPUs and sequential operater execution on each GPU, and maximum window size $\boldsymbol{w}$.
Output: schedule $\boldsymbol{Q}$ of minimal inference latency $\boldsymbol{L}$ with inter-operator parallelism among GPUs and within each GPU.

```
1: Let $L$ be the inference latency of $Q$;
2: Sort all vertices in $G$ according to the descending order of their
   priority indicators, as $v_1, v_2, \dots, v_n$;
3: for $i$ from 1 to $n$-1 do
4:     Let $g_i$ denote the GPU that operator $v_i$ is mapped to;
5:     for $p \in \{1, \cdots, w-1\}$ do
6:         $W = \{v_i$ and its $p$ succeeding operators on $g_i$ according
           to schedule $Q\}$;
7:         if operators in $W$ are independent with each other then
8:             $G' = G$;
9:             Group all operators $v \in W$ for parallel execution in
               the same stage and merge them as a single node in $G'$;
10:            if there exists no cycle in $G'$ then
11:                $G = G'$;
12:                Reschedule all the operators/grouped operators in
                   $G$ at their earliest available start time without changing their
                   execution order on each GPU, as schedule $Q'$;
13:                Let $L'$ be the inference latency of $Q'$;
14:                if $L' < L$ then
15:                    $L = L'$; $Q = Q'$;
16: return $(Q, L)$.
```



Parallel Operators	Execution Time	Parallel Operators	Execution Time
(v_2, v_3)	5	(v_3, v_7)	6
(v_2, v_4)	**5**	(v_4, v_5)	6
(v_3, v_4)	5	(v_4, v_6)	6
(v_2, v_6)	6	(v_5, v_6)	7
(v_2, v_7)	6	**(v_5, v_7)**	**7**
(v_3, v_5)	6	(v_6, v_7)	7

Maximum window size: 2

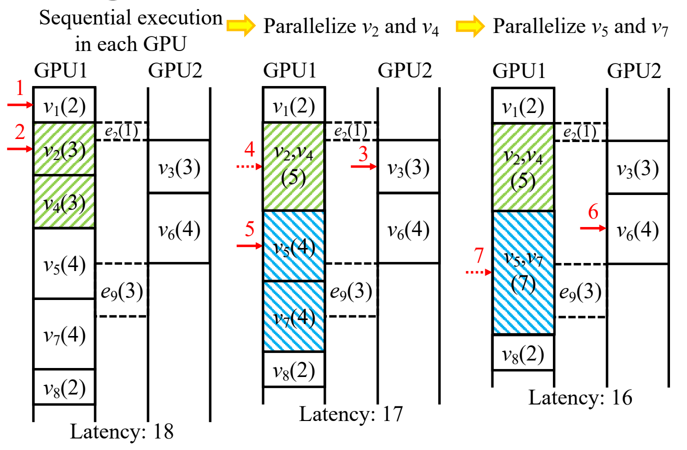


Fig. 5: An example to illustrate how Alg. 2 works.

are grouped into a single vertex in the computation graph. If not, we temporarily reschedule all the operators/grouped operators at their earliest available start time without changing their execution order on each GPU, generating a schedule candidate Q' (in Lines 10–12). For each operator, we try different window sizes from 2 to w (in Line 5) and update the schedule Q if the schedule candidate with the lowest inference latency performs better (in Lines 13–15).

Example: Fig. 5 demonstrates how HIOS-LP explores the opportunities of intra-GPU inter-operator parallelization for a given computation graph along with its inter-GPU operator schedule. The execution time of each operator and the transfer time of each data dependency are listed on the top left of Fig. 5. Since the maximum window size is preset to be 2, the execution time of each pair of independent operators running in parallel on a GPU is listed on the top right of Fig. 5. In the given computation graph, all the operators have been labeled according to the descending order of their priority indicators. We traverse operators from v_1 to v_7 as follows: 1) v_1 must finish before v_2 on GPU 1, so is skipped. 2) v_2 is followed by v_3 closely on GPU 1 and they can be parallelized for latency reduction, so they are grouped. 3) v_3 must finish before v_6 on GPU 2, so is skipped. 4) v_4 has been grouped with v_2 for parallel execution on GPU 1, so is skipped. 5) v_5 is followed by v_7 closely on GPU 1 and they can be parallelized for latency reduction, so they are grouped. 6) v_6 is followed by nothing on GPU 2, so is skipped. 7) v_7 has been grouped with v_5 for parallel execution on GPU 1, so is skipped. Finally, the inference latency is improved from 18 to 16 due to inter-operator parallelism exploration on each GPU.

C. Time Complexity

The time complexity of inter-GPU inter-operator parallelization is $O(M\cdot|V|^3\cdot|E|)$, because finding the longest path under our constraint has the time complexity of $O(|V|^2\cdot|E|)$ and is invoked $O(|V|)$ times as well as the temporal operator scheduling method has the time complexity of $O(|E|)$ and is invoked $O(M\cdot|V|)$ times. Here, M is the number of GPUs. The time complexity of intra-GPU inter-operator parallelization is $O(w^2\cdot|V|\cdot|E|^3)$. Therefore, HIOS-LP has the time complexity of $O(|V|\cdot|E|(M\cdot|V|^2+w^2\cdot|E|^2))$, where w is the preset maximum window size.

V. Simulation

In this section, we conducted extensive simulations to evaluate the inference latency of different inter-operator scheduling algorithms in various scenarios.

A. Simulation Settings

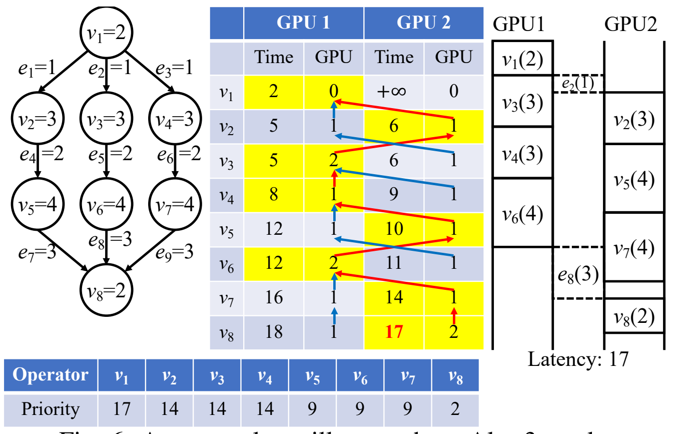
We generate a series of random DL model structures, in each of which the number of operators and the number of layers are preset to 200 and 14, respectively. The number of inter-operator dependencies is preset to 2 times the number of operators. The number of homogeneous GPUs is preset to 4. The execution time of an operator is randomly selected between 0.1 and 4 milliseconds; the transfer time between GPUs for the output data of an operator is a maximum of 0.1 milliseconds and p of the execution time of this operator, where p is preset to 80%. By default, each data point denotes the average of 30 randomly generated instances with standard deviations.

Algorithm 3 Mapping-Recording-Based Operator Scheduling

Inputs: computation graph $\boldsymbol{G}$, number $\boldsymbol{M}$ of GPUs, execution time $\boldsymbol{t(S)}$ of each set S of independent operators on one GPU, and data transfer time $\boldsymbol{t(u,v)}$ between each pair of operators u and v on different GPUs.

Output: schedule $\boldsymbol{Q}$ with minimal inference latency $\boldsymbol{L}$.

```
1: Sort all operators $v \in G$ according to their priority indicators in
   descending order as $v_1, v_2, \ldots, v_n$;
2: for $i$ from 1 to $n$ do
3:     for $j \in \{1, \cdots, M\}$ do
4:         $t_{i,j} = +\infty$; $g_{i,j} = 0$;   // Initialize a $n \times M$ 2D table.
5: $t_{1,1} = t(v_1)$;
6: for $i$ from 2 to $n$ do
7:     for $j \in \{1, \cdots, \min(M, i)\}$ do
8:         for $k \in \{1, \cdots, \min(M, i-1)\}$ do
9:             /* Obtain previous operators' schedule, i.e., the GPU
   index $g(v)$ and finish time $t_F(v)$ for each $v \in \{v_{i-1}, \ldots, 1\}$ in
   turn from the 2D table. */
10:            $m = k$;
11:            for $l$ from $i-1$ to 1 do
12:                $t_F(v_l) = t_{l,m}$; $g(v_l) = m$; $m = g_{l,m}$;
13:            /* Compute the earliest finish time of $v_i$ on the $j$th
   GPU if $v_{i-1}$ is mapped onto the $k$th GPU. */
14:            $t_k = \max\{t_F(v_l) | g(v_l) = j, 1 \le l \le i-1\}$;
15:            for $u \in$ {preceding vertices of $v_i$ in $G$} do
16:                if $u$ is on the $j$th GPU then
17:                    $t_k = \max\{t_F(u), t_k\}$;
18:                else
19:                    $t_k = \max\{t_F(u) + t(u, v_i), t_k\}$;
20:            if $t_{i,j} > t_k + t(v_i)$ then
21:                $t_{i,j} = t_k + t(v_i)$; $g_{i,j} = k$;
22: $m = J = \arg\min_j t_{n,j}$;
23: $Q_j = \emptyset$ for $\forall 1 \le j \le M$;
24: for $i$ from $n$ to 1 do
25:     $Q_m = \{\{v_i\}, Q_m\}$; $m = g_{i,m}$;
26: $Q = \{Q_j | 1 \le j \le M\}$;
27: ($Q$, $L$) = parallelize($G$, $Q$);   // See Alg. 2.
28: return ($Q$, $L$).
```



	GPU 1		GPU 2	
	Time	GPU	Time	GPU
v_1	2	0	$+\infty$	0
v_2	5	1	6	1
v_3	5	2	6	1
v_4	8	1	9	1
v_5	12	1	10	1
v_6	12	2	11	1
v_7	16	1	14	1
v_8	18	1	17	2

Operator	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
Priority	17	14	14	14	9	9	9	2

Fig. 6: An example to illustrate how Alg. 3 works.

B. Algorithms for Comparison

We compared the performance of different scheduling algorithms:

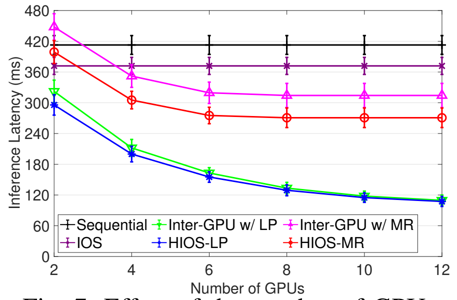


Fig. 7: Effect of the number of GPUs

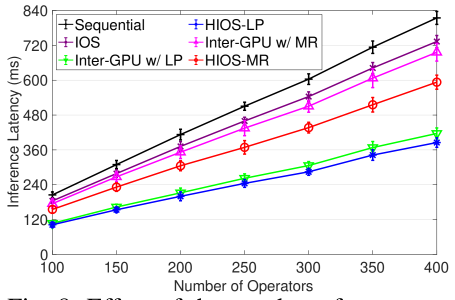


Fig. 8: Effect of the number of operators

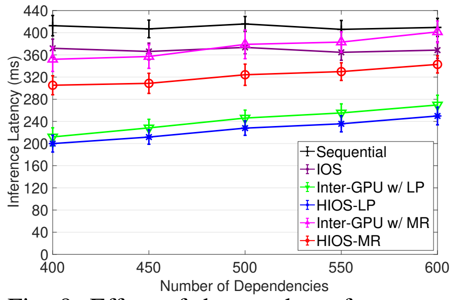


Fig. 9: Effect of the number of operator dependencies

- **Sequential scheduling**: Sequential scheduling executes the operator one by one according to a certain topological order in a single GPU.
- **IOS** [4]: IOS schedules operators only in a single GPU, and is an exponential-time algorithm based on dynamic programming, which is combined with schedule pruning for exploration space reduction.
- **HIOS based on the longest path (HIOS-LP)**: Our HIOS-LP schedules operators on multiple GPUs and is a polynomial-time algorithm based on iterative longest-path mapping, including **LP-based inter-GPU operator scheduling (inter-GPU w/ LP)** and intra-GPU inter-operator parallelization.
- **HIOS based on mapping recording (HIOS-MR)**: Our HIOS-MR also schedules operators on multiple GPUs and is a polynomial-time algorithm based on recording important attempts of mapping previous operators, described as follows and shown in Alg. 3.

HIOS-MR includes two parts: **MR-based inter-GPU operator scheduling (inter-GPU w/ MR)** illustrated in Fig. 6 and intra-GPU inter-operator parallelization (the same as Alg. 2). For the former, we first generate a $n \times M$ two-dimensional table, where the ith row corresponds to operator v_i, and the jth column corresponds to the jth GPU. In this table, each pair $(t_{i,j},\ g_{i,j})$ is used to record the earliest finish time of v_i that is mapped onto the jth GPU, and correspondingly the index of the GPU that v_{i-1} is mapped onto when v_i on the jth GPU finishes by $t_{i,j}$ (in Lines 2–4). Then, we map operators one by one according to their topological sorting (i.e. descending priority indicators). Initially, v_1 is mapped onto GPU 1 (in Line 5). For each subsequent operator v_i, we try to map it onto the first to the $\min(M, i)$th GPU (in Lines 6 and 7). When we suppose that v_i is mapped to GPU j, we search its earliest finish time by exploring $\min(M, i-1)$ recorded schedules of $v_1, v_2, \ldots, v_{i-1}$, in each of which v_{i-1} is mapped to a different GPU j and has the earliest finish time $t_{i-1,j}$ based on the previous search (in Lines 8–21). Finally, based on this complete table, we identify the earliest finish time $t_{n,J}$ of v_n over all the GPUs, and iteratively go along the direction pointed at by $g_{i,m}$ from $g_{n,J}$ to obtain the optimized schedule of $v_1, v_2, \ldots, v_n$ (in Lines 22–26).

C. Number of GPUs

We first compare HIOS-LP in terms of inference latency with sequential scheduling, IOS, HIOS-MR, LP-based inter-GPU operator scheduling, and MR-based inter-GPU operator scheduling over different numbers of GPUs and observe the scalability of HIOS-LP with the ascending number of GPUs. We plot the average and standard deviation of inference latency of these randomly generated DL models in Fig. 7. From the figure, it is witnessed that with the number of GPUs increasing from 2 to 12, the speedup of HIOS-MR is no more than 1.5 over sequential scheduling and no more than 1.4 over IOS, but the speedup of HIOS-LP is 1.4 to 3.8 over sequential scheduling and 1.3 to 3.5 over IOS. Therefore, HIOS-LP overperforms HIOS-MR in terms of inference latency and the scalability of HIOS-LP with the number of GPUs is also better than HIOS-MR.

D. Number of Operators

Then, we evaluate the performance of these six scheduling algorithms in terms of inference latency for DL models with the number of operators from 100 to 400 at an interval of 50. We plot the inference latency in Fig. 8, where we observe that the speedup of HIOS-LP is 2.01 to 2.12 over sequential scheduling, 1.81 to 1.91 over IOS, 1.51 to 1.54 over HIOS-MR, and 1.05 to 1.08 over LP-based inter-GPU operator scheduling. This exhibits a satisfactory scalability property of HIOS-LP with respect to the DL model size. In addition, intra-GPU inter-operator parallelization reduces the inference latency of LP-based inter-GPU operator scheduling by 5.7% to 7.7% and that of MR-based inter-GPU operator scheduling by 13.3% to 14.6%. This means that LP-based inter-GPU operator scheduling maps more independent operators onto different GPUs, leading to a smaller degree of parallelism within each GPU that can be explored by intra-GPU inter-operator parallelization.

E. Number of Inter-Operator Dependencies

We also examine the inference latency achieved by these six scheduling algorithms for DL models consisting of 200 operators with the number of inter-operator dependencies from 400 to 600 at an interval of 50 and plot the measurements in Fig. 9. These measurements show that with the number of inter-operator dependencies growing, the speedup of HIOS-LP decreases from 2.06 to 1.64 over sequential scheduling and from 1.86 to 1.48 over IOS, while the speedup of HIOS-MR also declines from 1.35 to 1.19 over sequential scheduling and from 1.22 to 1.08 over IOS. The performance optimization of HIOS-LP (and HIOS-MR) is limited by less independent operators, which is caused by the increasing number of inter-operator dependencies.

F. Degree of Parallelism in Deep Learning Models

We run these six scheduling algorithms for DL models with different degrees of parallelism by increasing the number of operator layers from 6 to 22 at an interval of 4. The inference latency optimized by these six scheduling algorithms is plotted in Fig. 10, which shows that the inference latency by sequential scheduling, IOS, and HIOS-MR is around 411 ms, 371 ms, and 305 ms, respectively, and remains unchanged over the different average number of operators per layer. It is worth noting that the inference latency by HIOS-LP is reduced from 233 ms to 174 ms with the number of operator layers decreasing from 22 to 6 (i.e., the increasing degree of parallelism in a DL model). Therefore, HIOS-LP is self-adaptive to the degree of parallelism in DL models.

G. Communication Overheads among GPUs

We execute these six scheduling algorithms for different platforms where the time ratio p of data transfer to operator computation varies from 0.4 to 1.2 at an interval of 0.2. The inference latency achieved by these six scheduling algorithms is plotted in Fig. 11. From this figure, we see that with communication overheads increasing, the inference latency by HIOS-LP declines from 2.23 to 1.78 of that by sequential scheduling and from 2.01 to 1.60 of that by IOS, and the inference latency by HIOS-MR also decreases from 1.52 to 1.10 of that by sequential scheduling and from 1.37 to 0.99 of that by IOS. Therefore, HIOS-LP is suitable for platforms with multiple GPUs connected via the high-speed NVLink/NVSwitch, where p is typically less than one.

VI. Experiments

In this section, we implemented a hierarchical inter-operator scheduler (HIOS) and incorporated inter-operator scheduling algorithms mentioned in §V-B into our HIOS to evaluate their performance in the real system.

A. Experimental Setup

We implemented the proposed hierarchical inter-operator scheduler (HIOS) on multiple GPUs by extending the cuDNN-based C++ Engine in IOS and incorporating CUDA-aware MPI for inter-GPU data communication. Also, we implemented the proposed inter-operator scheduling algorithm in Python and generate schedules in JSON for executing inference on multiple GPUs.

Then, we conducted experiments on the following multi-GPU platform. The Dell PowerEdge R750XA Rack Server has two 26-core/52-thread 2.2GHz Intel Xeon Gold 5320 CPU processors, 192GB RDIMM main memory, and two Nvidia Ampere A40 GPUs, connected via one Nvidia NVLink bridge with a bidirectional bandwidth of 112.5 GB/s. Each GPU has 10,752 CUDA cores, 48GB GDDR6 GPU memory, a memory bandwidth of 696 GB/s, and compute capability of 8.6.

We ran each application with exclusive access to the computing resource. Throughout the experiments, cuDNN 8.8.1, CUDA 12.1, NVIDIA driver 530.30.02, and Open MPI 4.1.4 were used. By default, each data point in experiments denotes the average of measurements on 36 runs as [4].

B. Benchmarks

We used two real-life CNN models from [4] as benchmarks in our experiments. The batch size of CNN inference is set to one for the fastest response.

- **Inception-v3** [16] is the third edition of Google's Inception CNN, which has 48 layers, and starts as a module for GoogLeNet. It is used for assisting in image analysis and object detection. The default input size is 299×299 pixels at least. Here, Inception-v3 has 119 operators and 153 inter-operator dependenices.
- **NASNet** [17] is a type of CNN discovered through neural architecture search and a family of models that were designed automatically by learning the model architectures directly on the dataset of interest. The building blocks consist of normal and reduction cells. The default input size is 331×331 pixels at least. Here, NASNet has 374 operators and 576 inter-operator dependencies.

C. Evaluation Metrics

We used two metrics to evaluate inter-operator scheduling algorithms.

- **Inference latency** (in milliseconds): Inference latency is an actual measurement in machine learning to determine the system performance in the prediction of various models for a specific application. The actual inference latency refers to the time taken to predict one data sample provided only one data sample is processed at a time (i.e., with batch size one).
- **Time cost of scheduling optimization** (in minutes): The time cost of scheduling optimization denotes the execution time of a scheduler that takes inputs to produce an optimized schedule. It is affected by a scheduling algorithm's time complexity and scalability.

D. Inference Latency of Different Benchmarks

We measure the actual inference latency of two CNN models with varying input image sizes and plot the measurements based on sequential scheduling, IOS, HIOS-LP, and HIOS-MR in Fig. 12. In order to explore the effect of operator workloads on the performance of operator scheduling algorithms, we test CNN inference over different input image sizes from the default size to the largest size of $2^K \times 2^K$ pixels. From Fig. 12, we see that HIOS-LP reduces the inference latency by 6.1% to 19.7% for Inception-v3 and by up to 14.5% for NASNet in comparison with sequential scheduling, and decreases the inference latency by 3.3% to 16.5% for Inception-v3 and by up to 11.1% for NASNet in comparison with IOS. This

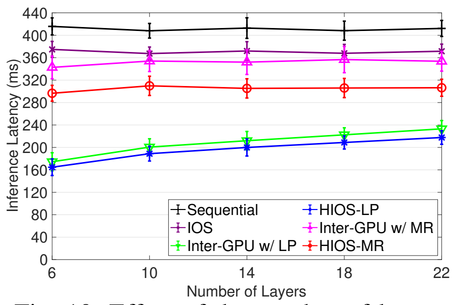


Fig. 10: Effect of the number of layers

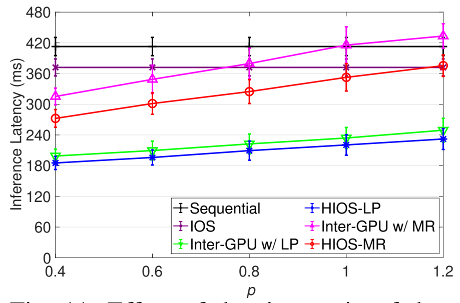
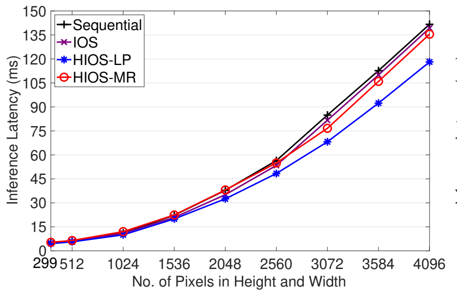
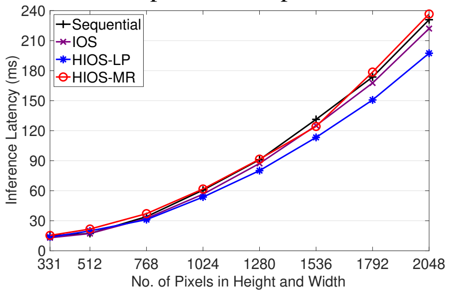


Fig. 11: Effect of the time ratio of data transfer to operator computation

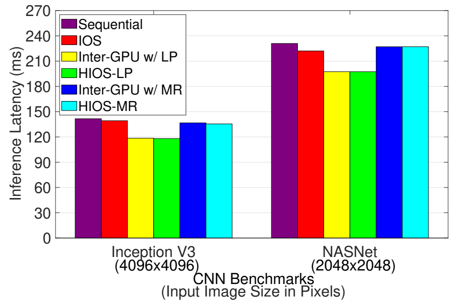


(a) *Inception V3*

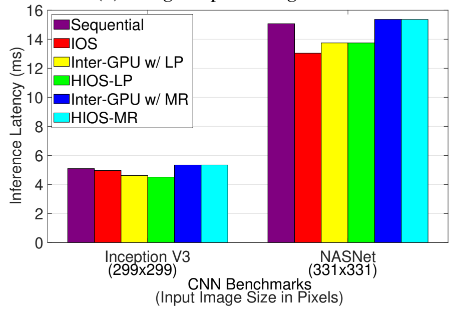


(b) *NASNet*

Fig. 12: Inference latency of different benchmarks



(a) *Large input image sizes*



(b) Default (small) input image sizes

Fig. 13: Performance gain analysis

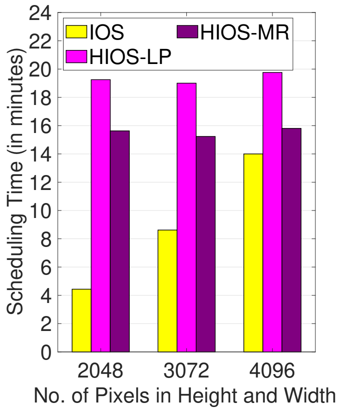
effectively validates that multiple GPUs can accelerate DL inference. Also, it can be observed that the inference latency achieved by HIOS-LP is 10.9% to 16.8% and 8.8% to 16.2% less than that achieved by HIOS-MR for Inception-v3 and NASNet, respectively. The reason for the superiority of HIOS-LP over HIOS-MR is as follows. HIOS-LP iteratively places the operators of the longest path onto the same GPU to reduce communication time for global inference latency optimization. However, HIOS-MR only places operators greedily through local optimization, involving unnecessary communication.

E. Performance Gain Analysis for the HIOS-LP Algorithm

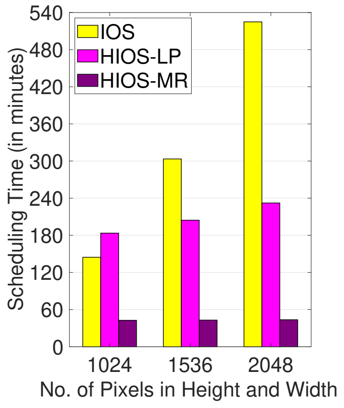
In order to further dissect the inference performance gain of hierarchical inter-operator scheduling, we display the inference latency achieved by sequential scheduling, IOS, HIOS-LP, HIOS-MR, LP-based inter-GPU operator scheduling, and MR-based inter-GPU operator scheduling for two CNN benchmarks with their default (or small) input image sizes and respective largest input image sizes in Fig. 13. This figure shows that in comparison with sequential scheduling, inference latency reduction by HIOS-LP is 9.9 and 4.5 times latency reduction by IOS for Inception v3 with large and small input image sizes, respectively, and is also 3.8 times that by IOS for NASNet with the large input image size. Because IOS is an exact algorithm for the optimal schedule in a single GPU, its inferiority means that intra-GPU operator parallelization is seriously insufficient for inference latency reduction due to limited resources in a single GPU. As shown in Fig. 13b, inference latency by HIOS-LP is 5.4% longer than that by IOS for NASNet with the small input image size. Here, performance degradation might be caused by CUDA kernel launching overheads. When directly dependent operators are in different GPUs, the succeeding CUDA kernel needs to be launched after inter-GPU data transfer completion in HIOS's current implementation version based on CUDA-aware MPI. For implementation improvement, inter-GPU data transfer based on Nvidia Collective Communication Library (NCCL) might be able to hide the launching time of succeeding CUDA kernels. In addition, we observe from Fig. 13 that latency reduction by LP-based inter-GPU operator scheduling counts towards 98.2% and 81.6% of total latency reduction by HIOS-LP for Inception v3 with large and small input image sizes, respectively. This indicates that although intra-GPU operator parallelization can group small operators for parallel execution and inference acceleration, inter-GPU operator parallelization dominates the performance gain of HIOS-LP, especially for large input image sizes. In contrast, latency reduction by LP-based inter-GPU operator scheduling counts towards almost 100% of total latency reduction by HIOS-LP for NASNet with both large and small input image sizes. This is because a limited number of independent operators in NASNet are distributed to different GPUs, which makes operators in each GPU almost non-parallelizable.

F. Time Cost of Scheduling Optimization

To compare the time cost of scheduling optimization by HIOS-LP with IOS and HIOS-MR, we measure the execution time of these three inter-operator schedulers for two CNN benchmarks with different input image sizes, shown in Fig. 14. Here, the scheduling time includes the time used to measure the execution time of each single operator and each group of parallel operators, the communication time of each possible data transfer between GPUs, the inference latency of involved sub-models, and the running time of a scheduling algorithm.



(a) *Inception V3*



(b) *NASNet*

Fig. 14: Time cost of scheduling algorithms

Fig. 14 reveals that the scheduling times of HIOS-LP and HIOS-MR increase with input image sizes much slower than that of IOS. Specifically, Fig. 14a shows that HIOS-LP's scheduling time is less than 20 minutes for Inception-v3; Fig. 14b also displays that HIOS-LP's scheduling time is up to 55.8% less than IOS's for NASNet with large input image sizes. Overall, HIOS-LP and HIOS-MR in terms of scheduling time cost are comparable with IOS and perform well for large input image sizes.

VII. Related Work

We did the survey on the related work in four aspects.

Multiple Concurrent Inferences: A few DL systems can schedule operators from different inferences for parallel execution on a single GPU. For example, Yu et al. [13] improve the runtime efficiency of a DL application composed of multiple DNN model inferences on a single GPU by wisely adjusting model concurrency and interleaving DNN model operators to maintain a continuously balanced resource utilization across the entire inference process. REEF [18] is a DNN inference serving system using preemptive scheduling for multiple workloads on a single GPU, which enables microsecond-scale kernel preemption to provide real-time guarantees and dynamically pads real-time kernels with best-effort kernels for concurrent execution to fully utilize resources. In comparison with exploiting dependence among operators from different inferences to optimize GPU utilization, it is more significant and challenging in practice that our HIOS improves the latency of a single inference by exploring dependence among its own operators.

Single Inference: Existing research work has explored inter-operator scheduling to speed up the latency of a single DL inference within a single GPU. Rammer [19] optimizes DNN inference latency on a single GPU by holistically exploiting inter- and intra-operator parallelism for a richer co-scheduling space to maximize hardware utilization. In addition, Nimble et al. [3] is a DL execution engine, which supports parallel execution of operators in a single GPU using multiple CUDA streams and thus reduces inference latency by efficiently utilizing the GPU computation power. To improve the scheduling algorithm in Nimble that does not consider the latency of each operator, Inter-Operator Scheduler (IOS) [4] is proposed as a profile-based scheduler to accelerate CNN inference by automatically scheduling multiple operators' parallel execution through a dynamic programming algorithm. However, these inter-operator scheduling algorithms do not work for the scenario of DL inference acceleration over multiple GPUs, which is targeted by our HIOS.

CPU-GPU Co-Execution: Recent efforts have coordinated heterogeneous processors to improve the latency of a single DL inference. For instance, LaLaRAND [5] is a real-time layer-level DNN scheduling framework in embedded systems, which schedules individual DNN layers onto a CPU/GPU by coupling CPU-friendly quantization with fine-grained CPU/GPU allocation schemes while mitigating accuracy loss without compromising timing guarantees. CoDL [6] is a concurrent DL inference framework to optimize the latency of DNN inference across the CPU and GPU on mobile devices, which allows heterogeneous processors to use their respective efficient data type and conducts operator partitioning by building a concurrency-aware latency predictor to extract the non-linearity of operator latency response. These frameworks make use of CPU and GPU's respective advantages to minimize DL inference latency, while our HIOS is oriented towards multiple homologous GPUs.

DAG Scheduling: Many studies [20]–[28] focus on DAG scheduling. Ueter et al. [29] study the hierarchical real-time scheduling problem of sporadic arbitrary-deadline DAG tasks. Zhao et al. [30] propose a multi-DAG scheduling approach to reduce inter-DAG interference.

VIII. Conclusion

The limited resources on a single GPU are insufficient to speed up the inference of multi-branch neural networks to meet low latency demands from real-time applications. For better inference latency acceleration, our HIOS system exploits inter-GPU operator parallelization to avoid resource contention between large operators during concurrent execution and leverages intra-GPU operator parallelization to group small operators for high resource utilization. To further solve the extremely complicated operator scheduling problem for hybrid inter-GPU and intra-GPU parallelism, we proposed a novel operator scheduling algorithm, HIOS-LP, which iteratively maps the longest path in the remaining computation graph onto one selected GPU for inter-GPU operator parallelization and then groups independent operators in a sliding window for intra-GPU operator parallelization. Experiments with two real-life CNN benchmarks and extensive simulation in various scenarios validate the superiority of our HIOS-LP over the state-of-the-art IOS and HIOS-MR methods.

Acknowledgment

This research is sponsored by National Science Foundation under Grant No. OAC-2306184 with the University of North Texas.

References

[1] G. Dong, K. G. Felker, A. Svyatkovskiy, W. Tang, and J. Kates-Harbeck, "Fully convolutional spatio-temporal models for representation learning in plasma science," Journal of Machine Learning for Modeling and Computing, vol. 2, no. 1, pp. 49–64, 2021.

[2] J. Kates-Harbeck, A. Svyatkovskiy, and W. Tang, "Predicting disruptive instabilities in controlled fusion plasmas through deep learning," Nature, vol. 568, pp. 526–531, 2019.

[3] W. Kwon, G.-I. Yu, E. Jeong, and B.-G. Chun, "Nimble: Lightweight and parallel GPU task scheduling for deep learning," in Conf. on Neural Information Processing Systems (NeurIPS), Virtual Event, Dec 2020, pp. 1–12.

[4] Y. Ding, L. Zhu, Z. Jia, G. Pekhimenko, and S. Han, "IOS: Inter-operator scheduler for CNN acceleration," in Conf. on Machine Learning and Systems (MLSys), Virtual, Apr 2021, pp. 1–14.

[5] W. Kang, K. Lee, J. Lee, I. Shin, and H. S. Chwa, "LaLaRAND: Flexible layer-by-layer CPU/GPU scheduling for real-time DNN tasks," in IEEE Real-Time Systems Symposium (RTSS), Dortmund, Germany, Dec 2021, pp. 329–341.

[6] F. Jia, D. Zhang, T. Cao, S. Jiang, Y. Liu, J. Ren, and Y. Zhang, "CoDL: Efficient CPU-GPU co-execution for deep learning inference on mobile devices," in ACM Intl. Conf. on Mobile Systems, Applications, and Services (MobiSys), Portland, Oregon, Jun 2022, pp. 209–221.

[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library," in Conf. on Neural Information Processing Systems (NeurIPS), Vancouver, Canada, Dec 2019, pp. 8026–8037.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," in USENIX Symposium on Operating Systems Design and Implementation (OSDI), Savannah, GA, USA, Nov 2016, pp. 265–283.

[9] A. Bakhtiarnia, Q. Zhang, and A. Iosifidis, "Efficient high-resolution deep learning: A survey," https://arxiv.org/abs/2207.13050, pp. 1–19, 2022.

[10] V. Thambawita, I. Strumke, S. A. Hicks, P. Halvorsen, S. Parasa, and M. A. Riegler, "Impact of image resolution on deep learning performance in endoscopy image classification: An experimental study using a large dataset of endoscopic images," Diagnostics, vol. 11, no. 12, pp. 2183:1–9, 2021.

[11] Q. Wen, Z. Luo, R. Chen, Y. Yang, and G. Li, "Deep learning approaches on defect detection in high resolution aerial images of insulators," Sensors, vol. 21, no. 4, pp. 1033:1–26, 2021.

[12] L. Liang, A. Meyarian, X. Yuanb, B. R. Runklec, G. Mihailab, Y. Qin, J. Danielse, M. L. Reba, and J. R. Rigby, "The first fine-resolution mapping of contour-levee irrigation using deep bi-stream convolutional neural networks," International Journal of Applied Earth Observation and Geoinformation, vol. 105, pp. 102 631:1–10, 2021.

[13] F. Yu, S. Bray, D. Wang, L. Shangguan, X. Tang, C. Liu, and X. Chen, "Automated runtime-aware scheduling for multi-tenant DNN inference on GPU," in IEEE/ACM Intl. Conf. on Computer Aided Design (ICCAD), Munich, Germany, Nov 2021, pp. 1–9.

[14] "NVLink and NVSwitch," 2023, https://www.nvidia.com/en-us/data-center/nvlink/.

[15] J. Ullman, "NP-complete scheduling problems," Journal of Computer and System Sciences, vol. 10, no. 3, pp. 384–393, 1975.

[16] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, Jun 2016, pp. 2818–2826.

[17] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, Jun 2018, pp. 8697–8710.

[18] M. Han, H. Zhang, R. Chen, and H. Chen, "Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences," in USENIX Symposium on Operating Systems Design and Implementation (OSDI), Carlsbad, CA, USA, Jul 2022, pp. 539–558.

[19] 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 USENIX Symposium on Operating Systems Design and Implementation (OSDI), Virtual, Nov 2020, pp. 881–897.

[20] T. Shu and C. Q. Wu, "Performance optimization of Hadoop workflows in public clouds through adaptive task partitioning," in Proc. of IEEE Intl. Conf. on Computer Communications (INFOCOM), Atlanta, GA, USA, May 2017, pp. 2349–2357.

[21] ——, "Energy-efficient mapping of large-scale workflows under deadline constraints in big data computing systems," Future Generation Computer Systems (FGCS), vol. 110, pp. 515–530, 2020, https://www.sciencedirect.com/science/article/pii/S0167739X17300468.

[22] ——, "Energy-efficient dynamic scheduling of deadline-constrained MapReduce workflows," in Proc. of IEEE eScience, Auckland, New Zealand, Oct 2017, pp. 393–402.

[23] ——, "Energy-efficient mapping of big data workflows under deadline constraints," in Proc. of Workshop on Workflows in Support of Large-Scale Science in conjunction with ACM/IEEE Supercomputing Conference, Salt Lake City, UT, USA, Nov 2016, pp. 34–43, http://ceur-ws.org/Vol-1800/paper5.pdf.

[24] T. Shu, "Performance optimization and energy efficiency of big-data computing workflows," Dissertation, New Jersey Institute of Technology, Newark, NJ, USA, Aug 2017, http://archives.njit.edu/vol01/etd/2010s/2017/njit-etd2017-096/njit-etd2017-096.pdf.

[25] T. Shu, Y. Guo, J. Wozniak, X. Ding, I. Foster, and T. Kurc, "Bootstrapping in-situ workflow auto-tuning via combining performance models of component applications," in Proc. of IEEE/ACM Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC), Virtual Event, Nov 2021, pp. 1–14.

[26] ——, "Poster: In-situ workflow auto-tuning through combining component models," in Proc. of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Virtual Event, Feb 2021, pp. 467–468.

[27] J. M. Wozniak, M. Dorier, R. Ross, T. Shu, T. Kurc, L. Tang, N. Podhorszki, and M. Wolf, "MPI jobs within MPI jobs: A practical way of enabling task-level fault-tolerance in HPC workflows," Future Generation Computer Systems, vol. 101, pp. 576–589, 2019.

[28] J. M. Wozniak, P. Davis, T. Shu, J. Ozik, N. Collier, I. Foster, T. Brettin, and R. Stevens, "Scaling deep learning for cancer with advanced workflow storage integration," in Proc. of the 4th Workshop on Machine Learning in HPC Environments in conjunction with ACM/IEEE Supercomputing Conference, Dallas, TX, USA, Nov 2018, pp. 114–123.

[29] N. Ueter, M. Günzel, G. von der Brüggen, and J.-J. Chen, "Parallel path progression DAG scheduling," IEEE Transactions on Computers, pp. 1–15, 2023.

[30] S. Zhao, X. Dai, and I. Bate, "DAG scheduling and analysis on multi-core systems by modelling parallelism and dependency," IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 12, pp. 4019–4038, 2022.