SortingHat: System Topology-aware Scheduling of Deep Neural Network Models on Multi-GPU Systems

Seok Namkoong
Yonsei University
Seoul, Republic of Korea
seoknamkoong@yonsei.ac.kr

Taehyeong Park
Yonsei University
Seoul, Republic of Korea
taehyeongpark@yonsei.ac.kr

Kiung Jung
Yonsei University
Seoul, Republic of Korea
kiung@yonsei.ac.kr

Jinyoung Kim
Yonsei University
Seoul, Republic of Korea
jinyoungkim97@yonsei.ac.kr

Yongjun Park
Yonsei University
Seoul, Republic of Korea
yongjunpark@yonsei.ac.kr

Abstract

The advent of cutting-edge AI applications has emphasized the importance of reducing inference latency. Consequently, efficient model-parallel execution on multiple GPUs represents a key challenge in achieving high performance through the partitioning of the target neural network. Nevertheless, in recent complex deep learning models, as the size of parameters continues to increase and overall inference latency is no longer solely dominated by kernel execution, performance improvements using multiple GPUs cannot be achieved by simply exploiting model parallelism without considering data transfer parallelism and the system topology. To address this challenge, this paper proposes *SortingHat*, which generates an efficient schedule of target neural network models on multi-GPU systems to minimize inference latency. Initially, *SortingHat* partitions a target model into multiple submodels based on dominator analysis to find the best solution within a reasonable time. Subsequently, *SortingHat* finds the best schedule for each submodel using Mixed Integer Linear Programming, taking system topology into account to exploit both model parallelism and data transfer parallelism. Once the schedules of all submodels are found, they are merged and executed on the ready queue-based executor. Evaluations on diverse multi-GPU environments with various large language models show that *SortingHat* achieves an average speedup of 2.28× and up to 2.96× over the single GPU on TVM baseline.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3728490

CCS Concepts

• **Computer systems organization**; • **Software and its engineering → Software system structures**;

Keywords

System topology, DNN scheduling, Multi-GPU system

ACM Reference Format:
Seok Namkoong, Taehyeong Park, Kiung Jung, Jinyoung Kim, and Yongjun Park. 2025. SortingHat: System Topology-aware Scheduling of Deep Neural Network Models on Multi-GPU Systems. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3721145.3728490

1 Introduction

Deep neural network (DNN) have become essential for cutting-edge AI services such as chatbots [17, 39] or coding assistants [2, 16]. Most of these services include interactive interfaces, which implies that latency is more important than throughput because users usually expect a fast response to their questions or requests [42]. To accelerate DNNs for low inference latency, the performance of single GPUs has continuously improved, and new hardware accelerators, such as neural processing units (NPUs) [8, 25], have been introduced. However, parallelism cannot be exploited within single-GPU systems, and the accelerators which target specific models are difficult to support newly introduced neural network architectures immediately. These limitations have rendered applying parallelism on multi-GPU systems a prevalent solution for accelerating DNNs.

In multi-GPU systems, model parallelism is typically used to minimize inference latency by maximizing the concurrent execution of multiple independent operations [19, 46]. While simple-structured models have few benefits from model parallelism, data transfer parallelism, which refers to simultaneous transmissions of parameters by fully utilizing PCIe buses

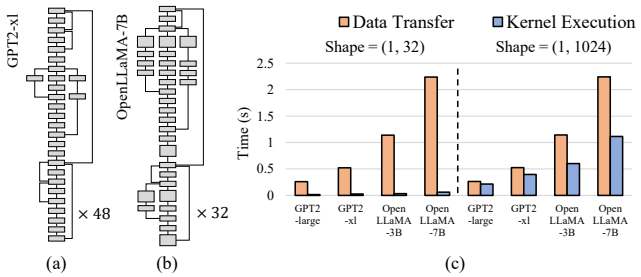

Figure 1: The structure of (a) GPT2-xl and (b) OpenLLaMA-7B models, represented by rectangles corresponding to operations such as Matmul and Transpose. (c) The elapsed time of data transfer and kernel execution on a single NVIDIA V100 GPU [30] for each model. The shape indicates (batch, sequence length).

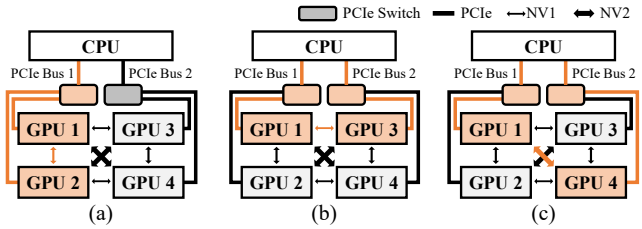

Figure 2: Different GPU selections on the V100 server of Figure 8(a): (a) One PCIe bus and (b) two PCIe buses are used, and (c) NVLink is 2× faster than (b). The orange lines represent data transfer paths for inference with selected GPUs.

and NVLinks [33], can be applied efficiently regardless of the model structure. For instance, DeepPlan [24] proposed *parallel transmission* (PT) that the half of the parameters are initially transferred to the secondary GPU and then retransferred to the first GPU through NVLink, while the other half are transferred to the first GPU at the beginning. We believe that a judicious combination of these parallelism techniques will lead to a remarkable reduction in inference latency.

While prior studies show that those parallelisms are effectively utilized, it is not guaranteed that the methodologies always reduce inference latency. In the case of model parallelism, executing partitioned subgraphs on multiple GPUs inevitably incurs additional data transfer between GPUs, and the distribution of subgraphs across different GPUs can lead to varying performance. In data transfer parallelism, PCIe contention often occurs when parallel data transfers are done over the same PCIe buses, or differing PCIe bandwidth results in different data transfer time. Taken together, the performance of individual GPUs and the configuration of data transfer paths significantly impact the overall latency. Therefore, it is crucial to find the best schedule that takes into account ***System Topology***, which encompasses both the data transfer topology and GPU performance.

In conclusion, it is essential to develop a scheduling strategy that considers model parallelism, data transfer parallelism, and system topology to minimize inference latency in diverse multi-GPU systems. Given that scheduling with a single factor approach is already complicated, attempting to address all the factors results in search space explosion. Therefore, the key challenge in minimizing latency is identifying the best schedule within a reasonable time using a sophisticated scheduling algorithm, avoiding the explosion of search space.

To tackle this challenge, this paper presents ***SortingHat***, a novel framework that finds the best schedule by maximizing both model and data transfer parallelism with consideration of system topology on multi-GPU systems. *SortingHat* first represents a target deep learning model as a computational graph and partitions it into multiple submodels based on dominator analysis, thereby preventing search space explosion. Next, *SortingHat* finds the best schedule for each submodel using Mixed Integer Linear Programming (MILP). The MILP incorporates multiple expressions for latency estimation that reflect various constraints such as data transfer parallelism, model parallelism, and system topology. Once all schedules of submodels are identified, they are merged and run on the ready queue-based executor.

We evaluated *SortingHat* on three multi-GPU environments; two homogeneous servers with four NVIDIA V100 GPUs [30] and three RTX A6000 GPUs [32], respectively, and one heterogeneous server with two high-performance GPUs and two low-performance GPUs. We analyzed its performance on representative LLMs to prove the effectiveness of *SortingHat* for different model structures. In our experiments, *SortingHat* achieved an average speedup of 2.28× and up to 2.96× over the single GPU on TVM [9] baseline, while state-of-the-art techniques such as DeepPlan [24] and EfficientMIP [46] achieved an average speedup of 1.93×, 1.16× and up to 2.14×, 1.62×, respectively.

This paper offers the following contributions:

- Avoidance of search space explosion through model partitioning based on dominator analysis.
- MILP-based scheduling with the system topology consideration: data transfer topology and GPU performances.
- Ready queue-based executor with maximized model parallelism and data transfer parallelism.
- Extensive evaluation of *SortingHat* with various neural networks in diverse multi-GPU environments.

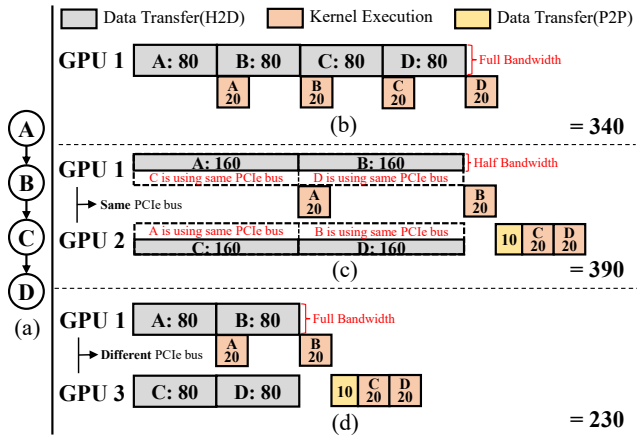

Figure 3: (a) An example neural network model. The expected most efficient execution timelines on (b) a single GPU1, (c) GPU1 and GPU2, and (d) GPU1 and GPU3, based on the topology of V100 server in Figure 2.

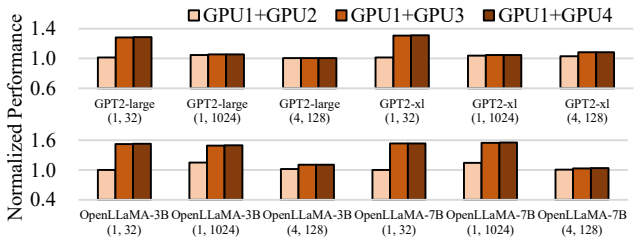

Figure 4: Performance difference due to different GPU selections on the V100 server in Figure 2, even with the same schedule. Each performance is normalized to the performance on a single GPU1.

2 Background and Motivation

2.1 Model and Data Transfer Parallelism

The effect of model parallelism is highly influenced by target model structures. Complex models, characterized by multiple branches, can obtain significant performance gains by distributing the execution of different operations across multiple GPUs. For example, Figure 1(a) and (b) depict that the OpenLLaMA-7B model with a complex structure includes many branches that can be executed concurrently, while the GPT2-xl model with a simple structure has almost no chance of applying model parallelism. This implies that more opportunities for model parallelism exist in complex models.

While models with a simple structure have few benefits from model parallelism, data transfer parallelism can be a promising alternative. Figure 1(c) shows that the data transfer overhead exceeds the kernel execution time, regardless of the input shape. This means that total latency is influenced not only by kernel execution but also by data transfer. Despite significant efforts to reduce parameter size by compression or quantization [10, 20, 23, 43, 53], the size of whole parameters in models has continued to increase steadily in recent years [15] and this trend is unlikely to change in the near future. In this context, DeepPlan [24] suggested parallel data transfer of parameters using independent PCIe buses and NVLinks to reduce inference latency. This strategy is applicable to a broad range of models, and therefore remains effective even for relatively simple model structures.

To achieve synergy by combining both types of parallelism, data transfers and computations can be overlapped [29]. However, recent studies aimed at reduction of inference latency have primarily focused on only one of these parallelism strategies. DeepPlan proposed an efficient method for data transfer parallelism, yet it did not further reduce inference latency by exploiting model parallelism. In contrast, EfficientMIP [46] introduced a method for searching the best schedule using model parallelism, but it overlooked the potential benefits of data transfer parallelism. Therefore, as focusing exclusively on either data transfer parallelism or model parallelism often misses opportunities for additional performance gains, both parallelism strategies must be considered collectively to find the most efficient schedule.

2.2 System Topology on Multi-GPU System

With the widespread adoption of multi-GPU environments, system topology can vary based on several factors such as GPU performance, the number of PCIe buses, inter-GPU data link interfaces, and support for peer-to-peer (P2P) data transfer [27]. Even worse, although advanced GPU-based servers use homogeneous all-to-all networks, real-world clusters often still consist of heterogeneous GPUs [41, 42] or rely on PCIe buses connecting the CPU or DRAM. These practical factors remain important when considering performance and scalability. As a result, the optimal configuration for maximizing both data transfer parallelism and model parallelism is not identical depending on the system environment. For instance, although the data transfers from a CPU to GPUs in Figure 2 involve the same number of GPUs, their performance differs significantly. The data transfer speed of Figure 2(a) is up to twice as slow as that of Figure 2(b), as Figure 2(b) utilizes all PCIe buses[1] at full bandwidth, while Figure 2(a) relies on a single PCIe bus operating at half bandwidth. Figure 2(c) even outperforms Figure 2(b) thanks to the doubled P2P data transfer bandwidth. If the performance of each GPU were different, determining the optimal GPU selection would be clearly more complex.

More specifically, Figure 3 demonstrates that the inference latency can vary depending on the system configurations, even with the same schedule. The only difference between Figure 3(c) and (d) is the data transfer efficiency resulting from the selection of different GPUs. As GPU1 and GPU2

[1]In this paper, PCIe bus refers to the PCIe link originating from CPU.

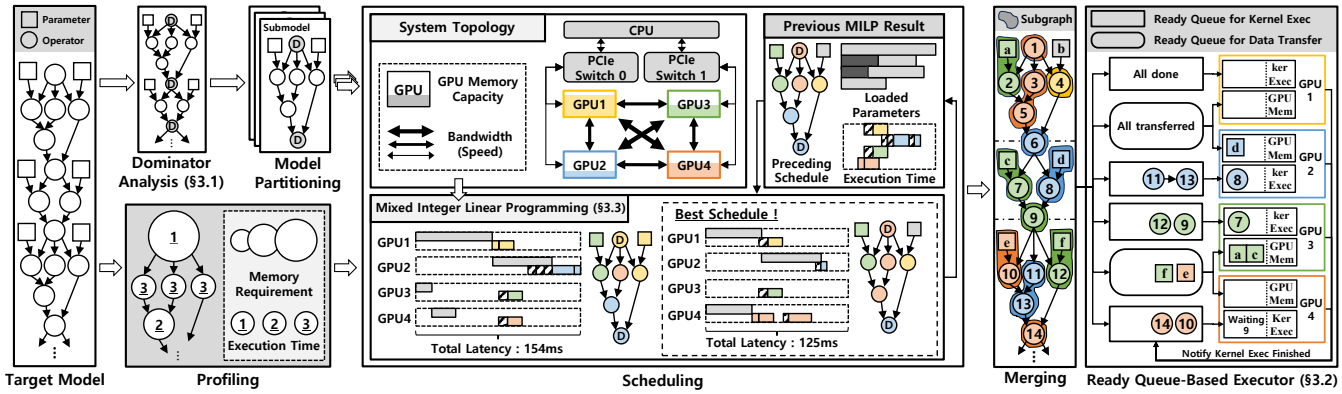

Figure 5: An overview of *SortingHat* framework.

share the same PCIe bus, data transfer speed of each GPU is half of the maximum bandwidth. In contrast, GPU1 and GPU3 can transfer data at full bandwidth, as they utilize separate PCIe buses. Due to this difference, operations C and D need to wait significantly longer in the GPU1+GPU2 configuration, resulting in even slower performance compared to a single GPU inference of Figure 3(b).

Figure 4 shows the actual performance differences when target models are executed across the configurations in Figure 2. The schedules were generated by EfficientMIP [46] assuming a two-GPU system. Experimental results show that the GPU1+GPU3 configuration is actually faster than the GPU1+GPU2 configuration, as we expected. Furthermore, GPU1+GPU4 consistently outperforms GPU1+GPU3, owing to the 2× faster NVLink connection. This analysis highlights the importance of considering system topology in multi-GPU environments to optimize end-to-end inference performance.

2.3 Search Space for Latency Estimation

As mentioned earlier, several factors including GPU performance and data transfer speed influence the inference latency of a target model. Moreover, the overlap of data transfer and kernel execution or emerging techniques such as parallel transmission further complicates latency estimation. The easiest way to simplify latency estimation is to ignore or limit certain factors, but this approach does not guarantee an optimal solution. For non-complex models, the optimal schedule cannot be found without considering data transfer parallelism, while model parallelism is crucial for complex models. Therefore, it is essential to search for the optimal schedule by comprehensively considering all factors that influence inference latency, with a realistic latency estimation.

Finding the optimal schedule using a brute-force algorithm is impractical in terms of scheduling efficiency. Heuristic algorithms may serve as potential solutions, but they do not ensure optimality. One of the most commonly used approaches in such cases is Mixed Integer Linear Programming (MILP) which involves defining an objective function and formulating constraints as inequalities consisting of integer variables to find the optimal solution [19, 46, 48, 50, 56]. Although optimality is not always guaranteed because this approach does not always explore all possible cases, it generally provides better search efficiency and the solution tends to converge closer to the optimal as the running time of MILP increases. However, for models with a large number of nodes, MILP becomes practically infeasible due to the rapid growth of the search space. Obviously, the search space would grow further when considering system topology and various latency minimization techniques. To address this challenge, *SortingHat* presents a method that provides an efficient multi-GPU schedule along with a realistic latency estimation.

3 SortingHat

The overview of *SortingHat* is illustrated in Figure 5. First, *SortingHat* profiles a target model to gather mandatory information including data transfer time, execution times of each operation on each GPU, and memory requirements. Then, based on the results of dominator analysis, *SortingHat* partitions the model into multiple submodels to reduce the search space. For each submodel, a MILP-based search process finds the best schedule, taking into account the system topology, the schedule of preceding submodels, and profiling results. The schedules for all submodels are then merged to form the entire model. Lastly, *SortingHat* enqueues the merged model into ready queues at the subgraph level and executes the subgraphs concurrently, overlapping data transfer and kernel execution. Each element is popped from the ready queue whenever the GPU is ready, until the queue is empty. Please note that the overhead and complexity of dominator analysis and MILP are effectively amortized, as they are applied only once before a series of inferences.

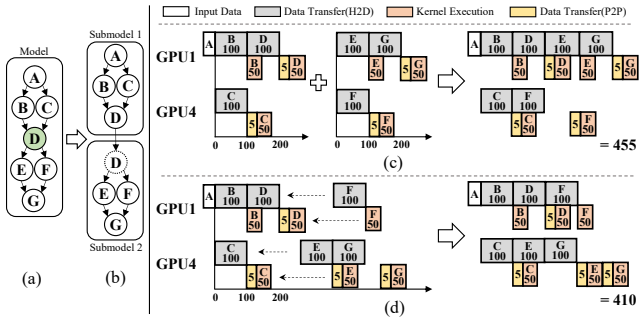

Figure 6: (a) Dominator analysis of an example model. (b) Submodels partitioned by node D. A node with dashed line is a placeholder node. (c) Submodel 1 and 2 are scheduled independently. (d) Submodel 1 is scheduled first, then Submodel 2 is scheduled considering the schedule of Submodel 1.

3.1 Model Partitioning via Dominator Analysis

While complex model structures provide opportunities for model parallelism, partitioning the entire model into submodels is necessary to reduce the search space, even though it sacrifices some degree of parallelism. Therefore, identifying appropriate partitioning points is crucial for both reducing search time and preserving the potential of parallelism.

Intuitively, branches would remain in submodels if the model is partitioned based on the nodes that must be visited to go from an input node to an output node. In Figure 6(a), the number of possible paths from input node A to output node G is 4, which are (1) A-B-D-E-G, (2) A-B-D-F-G, (3) A-C-D-E-G, and (4) A-C-D-F-G. Among these paths, node D is always traversed, except for the input and output nodes. When the model is partitioned based on node D, the branches of (A, B, C, D) and (D, E, F, G) can be maintained in two submodels.

Therefore, identifying these points, also called as *partition nodes* [50], is a critical aspect of model partitioning. These points are effectively determined through dominator analysis, a well-known control flow analysis methodology used in compiler optimization. In dominator analysis, the *dominators of node v* refer to the set of nodes that appear in all paths from the input node to node v. It implies that partition nodes can be defined as the dominators of the output node, making it straightforward to identify partitioning points by recognizing the dominators of the output node. While dominator analysis is typically associated with control flow analysis and deep learning models are inherently data flow-based as described in Figure 1(a) and (b), it is a widely used technique in deep learning compiler optimizations, such as operator fusion [9]. As a result, it can be effectively applied to *SortingHat* as well.

Once dominator analysis is completed, the target model is partitioned into several submodels based on the dominator nodes, as depicted in Figure 6(b). Please note that the size of the submodels can be adjusted by the number of dominator nodes included in each submodel. The last dominator node in each submodel becomes the output node of the submodel, while an additional placeholder node is added as the input node except for the first submodel. The placeholder node which is represented with a dashed line in Figure 6(b) is not an actual operation but a virtual node used to simply hold the data from the output node of preceding submodel. Note that the correctness of model is guaranteed because the data dependencies are not altered.

After partitioning the model into submodels, *SortingHat* finds the best schedule for each submodel, as shown in Figure 6(d). An important point here is that the schedule of preceding submodel must be considered. If each submodel is scheduled independently, the information about data transfer time and kernel execution time for each submodel cannot be shared. This makes it challenging to find the best solution from the perspective of the entire model, as demonstrated in Figure 6(c): the latency of the schedule in Figure 6(c) is higher compared to that of (d). Therefore, after reducing the search space through model partitioning based on dominator analysis, *SortingHat* should find the best schedule for each submodel using the schedule of the preceding submodel.

3.2 Ready Queue-based Executor

After model partitioning, it is important to minimize inference latency by exploiting the potential parallelism during inference. As illustrated in Figure 5, this can be accomplished by creating multiple ready queues per GPU for subgraph inference and per PCIe bus for data transfer. Also, based on the principles of DeepPlan [24], *SortingHat* implements direct host access (DHA) method and parallel transmission (PT) concepts. DHA allows direct parameter access from host memory, and PT facilitates the transfer of parameters to adjacent GPUs, which are later copied to the target GPU through the fast P2P data transfer. Algorithm 1 details how *SortingHat* maximizes parallelism through the ready queue-based executor, with the merged subgraphs as an input.

For efficient data transfer, a single GPU should exclusively utilize the PCIe bus to avoid contention and thereby the maximum bandwidth can be secured. While a simple approach might involve assigning exclusive access of a PCIe bus to a single GPU until all data transfers in the GPU are completed, this might result in long starvation for other GPUs. To address this issue, *SortingHat* proposes a round-robin preemption scheme that allows each GPU to exclusively access PCIe buses in turn. Initially, *SortingHat* creates ready queues according to the PCIe bus (line 2, 28). Then, parameters in the

Algorithm 1 Ready Queue-based Executor

```
1:  function CopyParam(pid)
2:      paramQ := Parameters that are copied via PCIe_pid,
        enqueued based on the order pre-defined by MILP
3:      while !paramQ.isEmpty() do
4:          p = paramQ.pop()
5:          Copy p to destination GPU on H2DStream_p.dstgid
6:          Wait until copying p is finished
7:          Mark p as copyFin
8:  end function
9:
10: function ExecSubgraph(gid)
11:     subgraphQ := Subgraphs that are run on GPU_gid,
        enqueued based on the order pre-defined by MILP
12:     while !subgraphQ.isEmpty() do
13:         sg = subgraphQ.pop()
14:         if sg.execType != DHA then
15:             for p in sg.InputParameters do
16:                 Wait until p is marked as copyFin
17:                 if p.dstGid != gid then
18:                     Copy p to GPU_gid on P2PStream_p.dstgid,gid
19:         for op in sg.InputOperators do
20:             Wait until op is marked as execFin
21:             if op.dstGid != gid then
22:                 Copy the result of op to GPU_gid
                    on P2PStream_op.dstgid,gid
23:         Run sg on ExecStream_gid
24:         Mark all operators in sg as execFin
25: end function
26:
27: for pid in range(numPCIe) do
28:     spawn thread CopyParam(pid)
29: for gid in range(numGPU) do
30:     spawn thread ExecSubgraph(gid)
```

ready queue are copied to the target GPU with a pre-defined order by MILP (line 3-7), as briefly outlined in Figure 7(d).

In the case of subgraph inference, ready queues are created per GPU, not PCIe bus (line 11, 30). To execute a subgraph, the input parameters are prepared in three different ways. First, if a subgraph is executed with DHA, the parameter copying process is unnecessary (line 14). Second, when the input parameters reside on adjacent GPUs by PT, additional copying is required (line 17-18). All other input parameters should be ready on the GPU by *CopyParam* function (line 16). Once all input parameters are ready, the result data from input operators is also copied by a similar process (line 19-22), and then the subgraph can be executed (line 23). This process is repeated until all ready queues are empty, and the result of final output node becomes the inference result of original model.

Table 1: Summary of constant variables used in MILP.

Constant Variables	
V	The set of all nodes (operation nodes and parameter nodes)
E	The set of edges
W	The set of parameter nodes ($W \subset V$)
k	The number of GPUs
q	The maximum number of subgraphs per GPU in a submodel (multiple of 2)
H	Large positive value that can be treated as infinity
M_i	Maximum memory size of GPU_i
m_v	Memory size of node v
$T_{0,i}$	Data transfer speed from DRAM to GPU_i (time per size)
T_{i_1,i_2}	Data transfer speed from GPU_{i_1} to GPU_{i_2} (time per size). If i_1 is equal to i_2, the value is 0.
$P^{GPU}_{v,i}$	Latency of node v on GPU_i without DHA
$P^{DHA}_{v,i}$	Latency of node v on GPU_i with DHA
$PCIeStartIdx$	The set of subgraph indices that H2D data transfer is begun at first in each PCIe bus.
$PrevM_i$	Size of memory which is already in use on the GPU_i.
$PrevDTFin_d$	Finish time of data transfer on $PCIe_d$ in the previous MILP. For the first MILP, the value is 0.
$PrevSGFin_i$	Finish time of subgraph execution on GPU_i in the previous MILP. For the first MILP, the value is 0.
$FixedNodes$	The set of (v, i) pair, representing that node v is already allocated to GPU_i in the previous MILP. For the first MILP, the set is empty.
$ParentDT_j$	Preceding subgraph index that data transfer is done right before $Subgraph_j$ ($j \notin PCIeStartIdx$)
$ParentPCIe_j$	Index of PCIe bus in which $Subgraph_j$ is included.

3.3 MILP-based Execution Plan Scheduling

SortingHat proposes a MILP model that incorporates key factors to maximize both model parallelism and data transfer parallelism, considering system topology and advanced data transfer techniques such as DHA and PT. This section explains the construction of the MILP model and demonstrates how the best schedules for submodels can be derived.

Prerequisites. The constant variables for the MILP inequalities are summarized in Table 1. Please note that these constant variables remain fixed during a single MILP run, but not across multiple MILP runs. The term "*subgraph*" in the context of the MILP refers to a subgraph in a submodel. A subgraph in MILP can hold both parameter nodes and operation nodes, and each type of node is processed by pre-defined order as described in Figure 7(c) and (d). $i(= 1, 2, ..., k)$ represents the indices of GPUs, and $j(= 0, 1, ..., kq)$ represents the indices of subgraphs. One important thing to note is that $Subgraph_0$ is a set of parameter nodes which are not copied to GPU memory but remained in DRAM for DHA. Each submodel can be split up to $(kq + 1)$ subgraphs, and GPU_i contains $Subgraph_{(i-1)q+1 \le j \le iq}$ as depicted in Figure 7(b). v

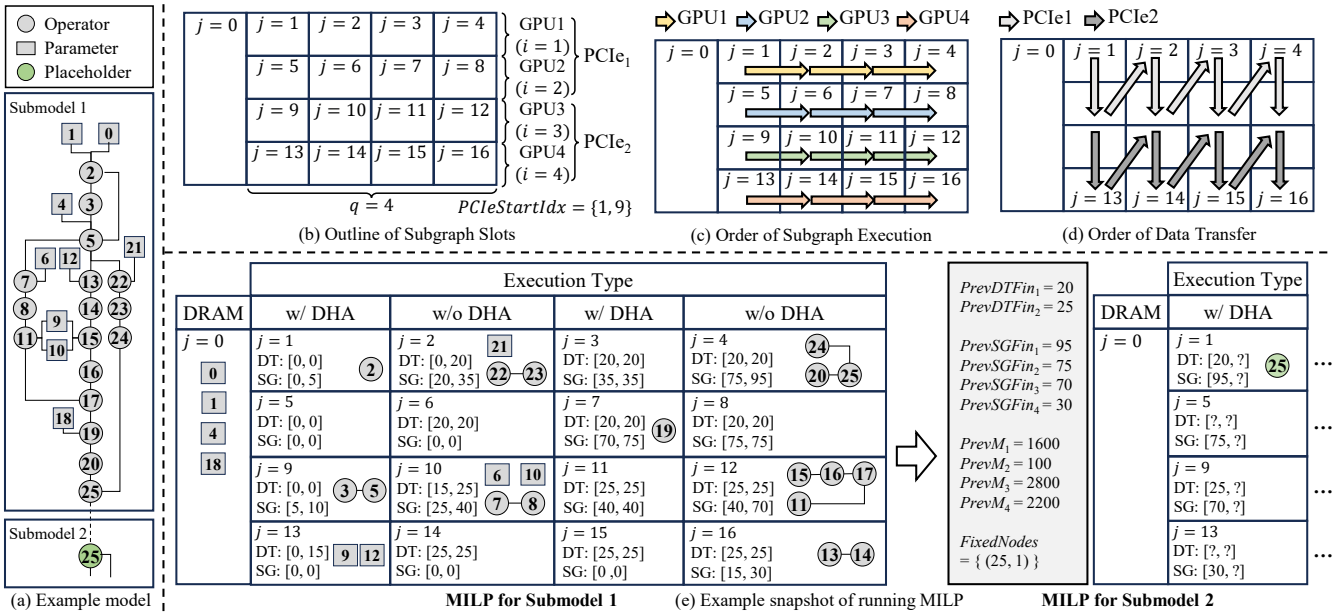

Figure 7: Details of MILP with a topology of V100 server in Figure 8(a). (a) An example model after dominator analysis. (b) Outline of subgraph slots. (c, d) Order of subgraph execution and data transfer. (e) An example intermediate result of MILP. T_{i_1,i_2} is assumed to be 0 for simplicity and square brackets [Start, Fin] of DT and SG mean $DTStart_j, DTFin_j, SGStart_j$ and $SGFin_j$, respectively.

is an index for any types of nodes and w is an index only for parameter nodes. Edges are denoted as (u, v), which represents an edge from node u to node v.

Objective. The ultimate goal of *SortingHat* is to minimize inference latency subject to the given constraints. To achieve this, *SortingHat* models the estimated latency of a submodel as $TotalLatency$ and searches for the best schedule that minimizes $TotalLatency$ while satisfying all constraints.

$$\begin{aligned} &\text{minimize} \quad TotalLatency \\ &\text{subject to} \quad \textit{Assignment Constraints, Data Movement,} \\ &\textit{Memory Limit, Parameter Copy, Subgraph Inference} \end{aligned} \tag{1}$$

Assignment Constraints. Binary variable $x_{v,j}$ indicates whether node v is a member of $Subgraph_j$, with a value of 1 assigned when node v is placed in $Subgraph_j$, otherwise 0. This variable has self-imposed constraints, regardless of data transfer and subgraph execution. For example, each node should be involved in only one subgraph (2), and already fixed nodes should be started with the first subgraph of the allocated GPU (3).

$$\sum_{j=0}^{kq} x_{v,j} = 1 \qquad (\forall v) \tag{2}$$

$$x_{v,(i-1)q+1} = 1 \qquad (\forall (v,i) \in FixedNodes) \tag{3}$$

Moreover, $Subgraph_0$ is only for parameter nodes, not for operator nodes. This implies that operator nodes must not be assigned in $Subgraph_0$ (4). On the other hand, the placement of parameter nodes is determined by the execution type of the operator nodes. If an operator node is executed with DHA, the input parameter nodes must be placed in DRAM ($Subgraph_0$). Otherwise, they should be placed in GPUs, not DRAM (5). For instance, in Figure 7(e), parameter nodes 0, 1, 4, and 18 are the input parameter nodes of $Subgraph_1$, $Subgraph_9$ and $Subgraph_7$ that are executed with DHA, respectively, and they are all involved in $Subgraph_0$. In contrast, other parameter nodes that are executed without DHA are included in $Subgraph_{j>0}$.

$$x_{v,0} = 0 \qquad (\forall v \in V \setminus W) \tag{4}$$

$$x_{w,0} = \sum_{j=1}^{kq/2} x_{v,2j-1} \qquad (\forall (w,v) \in E) \tag{5}$$

Finally, subgraphs should be *contiguous*, which executed without interrupting or interleaving. *SortingHat* borrowed the definition and inequalities of contiguity from Efficient-MIP [46], where the binary variable $z_{v,j}$ is 1 if $Subgraph_j$ is reachable from node v. A subgraph is contiguous if a set of $z_{v,j}$ values that satisfy the following inequalities exist (6-8).

$$z_{v,j} \geq x_{v,j} \qquad (\forall v)\ (\forall j > 0) \tag{6}$$

$$z_{u,j} \geq z_{v,j} \qquad (\forall (u,v) \in E)\ (\forall j > 0) \tag{7}$$

$$z_{v,j} \leq x_{v,j} - x_{u,j} + 1 \qquad (\forall (u,v) \in E)\ (\forall j > 0) \tag{8}$$

Data Movement. Incoming edges to subgraphs must be recognized when calculating data transfer time or subgraph execution time. To fulfill these requirements, two binary variables are introduced. The first variable is $EdgeIn_{u,j}$ whose value is 1 if node u has an outgoing edge to $Subgraph_j$ (9). If $x_{v,j}$ is 1 and $x_{u,j}$ is 0 for edge (u,v), $EdgeIn_{u,j}$ must be 1 due to the inequality, thus indicating that $Subgraph_j$ has an incoming edge from node u. The second variable is $CopyIn_{v,i,j}$ whose value is 1 if node v is assigned in GPU_i and has an outgoing edge to $Subgraph_j$ (10). This variable is used to determine whether the result of node v should be copied from GPU_i to $GPU_{\lfloor (j-1)/q+1 \rfloor}$. In Figure 7(e), $EdgeIn_{2,9}$ becomes 1 and thus $CopyIn_{2,1,9}$ also becomes 1, which means that node 2 with an outgoing edge to $Subgraph_9$ is allocated to GPU_1.

$$EdgeIn_{u,j} \geq x_{v,j} - x_{u,j} \ (\forall (u,v) \in E) \ (\forall j > 0) \tag{9}$$

$$CopyIn_{v,i,j} \geq \sum_{j'=(i-1)q+1}^{iq} x_{v,j'} + EdgeIn_{v,j} - 1 \quad (\forall v) \ (\forall i) \ (\forall j > 0) \tag{10}$$

Memory Limit. $PrevM_i$ is required to inherit the memory usage of each GPU from previous MILP. Along with the variable, it is assumed that the memory requirement is the sum of the nodes copied to GPU_i, the nodes allocated in GPU_i, and the memory in use on GPU_i from the previous MILP (11). However, this assumption may vary depending on the memory allocation policy of a target system and thus (11) can be modified to accurately reflect the real-world environment.

$$M_i \geq \sum_v \left(m_v \cdot \sum_{j=(i-1)q+1}^{iq} \left(\sum_{i'=1}^{i-1} CopyIn_{v,i',j} + \sum_{i'=i+1}^{k} CopyIn_{v,i',j} \right) \right) + \sum_v \left(m_v \cdot \sum_{j=(i-1)q+1}^{iq} x_{v,j} \right) + PrevM_i \quad (\forall i) \tag{11}$$

Parameter Copy. Data transfers are performed according to a round-robin preemption policy, as described in Figure 7(d). In this figure, the $ParentDT_j$ values are assigned as follows: $ParentDT_5 = 1$, $ParentDT_2 = 5$, etc. If the index of subgraph is an element of $PCIeStartIdx$, data transfer begins immediately after the last data transfer from the previous MILP on the PCIe bus is completed (12). Otherwise, the data transfer begins immediately after $Subgraph_{ParentDT_j}$ completes a transfer (13). The data transfer is finished after all parameters in the subgraph are copied (14).

$$DTStart_j = PrevDTFin_{ParentPCIe_j} \ (\forall j \in PCIeStartIdx) \tag{12}$$

$$DTStart_j = DTFin_{ParentDT_j} \ (\forall j > 0, j \notin PCIeStartIdx) \tag{13}$$

$$DTFin_j = DTStart_j + \sum_w x_{w,j} \cdot m_w \cdot T_{0,\lfloor (j-1)/q+1 \rfloor} \ (\forall j > 0) \tag{14}$$

Subgraph Inference. To determine when a subgraph is ready for beginning, the finish time of the input nodes must be known. $NodeFin_v$ means the finish time of operator node v and is set to the time at which the execution of its assigned subgraph is completed (15). Similarly, $NodeFin_w$ is the finish time of parameter node w and is set to the time at which the data transfer of its assigned subgraph is finished (16). The inequalities (15-16) are intended to be meaningless when $x_{v,j}$ or $x_{w,j}$ equals 0.

$$NodeFin_v \geq SGFin_j - (1 - x_{v,j}) \cdot H \ (\forall v \in V \setminus W)(\forall j > 0) \tag{15}$$

$$NodeFin_w \geq DTFin_j - (1 - x_{w,j}) \cdot H \ (\forall w)(\forall j > 0) \tag{16}$$

Subgraph inference is performed in parallel on each GPU, as shown in Figure 7(c). If the subgraph is the first to run on the GPU, its execution begins immediately after the last inference from the previous MILP on the GPU is completed (17). Otherwise, it starts after the inference of the preceding subgraph is finished (18). The last condition is that all input nodes must be ready for use (19). However, when an input parameter node is allocated in same subgraph, (19) could be null. For example, in the case of node 7 in Figure 7, the earliest time at which node 7 can be executed is 25, which is after the data transfer of the input parameter node 6 is completed. Since $EdgeIn_{6,10}$ is 0, (19) is negated, and thus $SGStart_{10}$ becomes 10, which is the value of $SGFin_9$, according to (18). This implies that operator node 7 can be executed even if the input parameter node 6 is not ready. (20) is added to prevent the invalidity when both an operator node and its input parameter node are placed in the same $Subgraph_j$.

$$SGStart_j \geq PrevSGFin_{\lfloor (j-1)/q+1 \rfloor} \ (\forall j = 1 \ mod \ q) \tag{17}$$

$$SGStart_j \geq SGFin_{j-1} \ (\forall j > 0, j \neq 1 \ mod \ q) \tag{18}$$

$$SGStart_j \geq NodeFin_v - (1 - EdgeIn_{v,j}) \cdot H \quad (\forall v)(\forall j > 0) \tag{19}$$

$$SGStart_j \geq NodeFin_w - (2 - x_{w,j} - x_{v,j}) \cdot H \quad (\forall j > 0)(\forall (w,v) \in E) \tag{20}$$

Once all input nodes are ready, subgraph inference is performed according to the execution type. When j is even, the subgraph is executed without DHA (21), and when j is odd, it is executed with DHA (22). Among the finish times of each subgraph inference, the largest value becomes $TotalLatency$ (23). The $TotalLatency$ of the last submodel represents the total inference latency of original model.

$$SGFin_j = SGStart_j + \sum_v \sum_i CopyIn_{v,i,j} \cdot m_v \cdot T_{i,\lfloor (j-1)/q+1 \rfloor} + \sum_v x_{v,j} \cdot P^{GPU}_{v,\lfloor (j-1)/q+1 \rfloor} \ (\forall j > 0, j = 0 \ mod \ 2) \tag{21}$$

$$SGFin_j = SGStart_j + \sum_v \sum_i CopyIn_{v,i,j} \cdot m_v \cdot T_{i,\lfloor (j-1)/q+1 \rfloor} + \sum_v x_{v,j} \cdot P^{DHA}_{v,\lfloor (j-1)/q+1 \rfloor} \ (\forall j = 1 \ mod \ 2) \tag{22}$$

$$TotalLatency \geq SGFin_j \ (\forall j > 0, j = 0 \ mod \ q) \tag{23}$$

Table 2: Details of target multi-GPU systems.

Server	V100	V100(hetero)	A6000
OS	Ubuntu 20.04 LTS		
DRAM	256GB		128GB
CPU	Intel Xeon E5-2698 v4 [22]		AMD EPYC 7262 8-Core Processor [4]
GPU	NVIDIA Tesla V100 DGXS 32GB [30] ×4	NVIDIA Tesla V100 DGXS 32GB 1) 1500MHz ×2 2) 135MHz ×2	NVIDIA RTX A6000 [32] ×3
PCIe	PCI Express gen 3		PCI Express gen 4
NVLink	Fully connected with 2-ways ×2 (GPU1-GPU4, GPU2-GPU3) 1-way ×4 (Others)		4-ways ×1 (GPU2-GPU3)
CUDA	11.4 [31]		12.4 [34]
TVM [9]	Modified based on 0.15.0		
PyTorch [5]	2.5.1		

3.4 Adjustment of Constant Variables in MILP

This section discusses how to adjust the constant variables in MILP to improve the accuracy of latency estimation. First, inevitable PCIe contention should be considered when measuring the $P_{v,i}^{DHA}$ value. Since the execution with DHA method requires direct DRAM access, data communication over PCIe buses during computation is unavoidable. This may lead to a collision with other subgraph executions with DHA or data transfers from host memory to GPUs, while latency with DHA is measured without PCIe contention in a profiling step. To resolve the problem, *SortingHat* adjusts the $P_{v,i}^{DHA}$ value by multiplying it by twice the number of GPUs sharing the same PCIe bus, under the assumption of the worst-case scenario bandwidth. As a result, *SortingHat* mitigates the overall performance degradation that would occur from selecting DHA methods more than necessary. Similarly, host-to-device (H2D) data transfer time can also be influenced by DHA. Nevertheless, *SortingHat* does not adjust for this, because H2D data transfer typically takes much longer than execution with DHA and the impact of PCIe contention is relatively negligible.

The T_{i_1,i_2} value also requires a similar process. When GPU_{i_1} and GPU_{i_2} are connected with a single direct link free from interference by other GPUs, contention in the link is only possible when both GPUs transfer data to each other simultaneously. Since the likelihood of this situation occurring is low, *SortingHat* recommends simply multiplying the T_{i_1,i_2} value by 1.5 rather than 2. On the other hand, if PCIe buses are shared by multiple GPUs for data transfer, as in the case of the PCIe buses between GPU1 and GPU3 in Figure 8(c), PCIe contention would arise often. Theoretically, the maximum number of GPUs sharing the same PCIe bus equals the total number of GPUs. If all GPUs exchange data simultaneously, the data transfer latency increases by a factor proportional to the total number of GPUs. Therefore, *SortingHat* suggests multiplying the T_{i_1,i_2} value by the total number of GPUs when

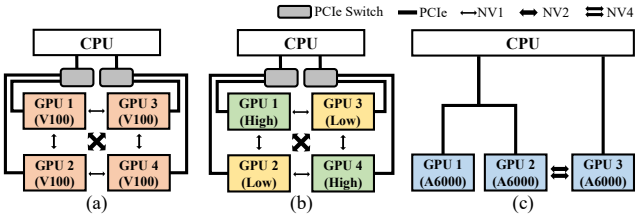

Figure 8: System topology of (a) V100, (b) V100(hetero), and (c) A6000 servers in Table 2.

GPU_{i_1} and GPU_{i_2} do not have a direct link between them. In this case, P2P data transfer may be selected only if it is still efficient even after the adjustment.

4 Evaluation

4.1 Experimental Setup

Multi-GPU System Setup. SortingHat is evaluated on three different multi-GPU servers as shown in Figure 8 to show the generality of *SortingHat*, and detailed information is provided in Table 2. One notable point is that the frequencies of GPU1 and GPU4 are set to 1500MHz, while GPU2 and GPU3 are set to 135MHz in the V100(hetero) server. To set up the heterogeneous environment, it was necessary to emulate low-performance GPUs to achieve an overall performance similar to the low performance GPUs with fewer SMs. Therefore, we set an extremely low frequency for these GPUs, but Multi-Process Service (MPS) [35] could also be a good option.

Integer Programming Solver. SortingHat used Gurobi 10.01 [18] to design and run the MILP models. The Python API of Gurobi was used to write the inequalities in MILP, and the runner to find the best solution was processed on an Intel Core i7-13700K CPU [21] with 64GB DRAM.

Implementation. We used cuBLAS [36] kernels for fast GEMM operations and modified TVM to apply the techniques proposed in Section 3.2. First, parameters are loaded into DRAM allocated by *cudaMallocHost()* and P2P data transfer is enabled by *cudaDeviceEnablePeerAccess()*. Moreover, streams created by *cudaStreamCreate()* are used for overlapping data transfer and operation execution.

Target Neural Network Models. In the evaluation, we used representative pre-trained LLMs: GPT2-large, GPT2-xl [45], OpenLLaMA-3B, and OpenLLaMA-7B [12, 14, 47] models which can be found on Hugging Face [51, 52]. As a result, we were able to prepare a variety of combinations with different parameter sizes and structures. Additionally, to verify that *SortingHat* works well for diverse model structures and ratios of data transfer to kernel execution time, various input shapes were used with different batch sizes and sequence lengths: (1, 32), (1, 1024), (4, 128), and (32, 64).

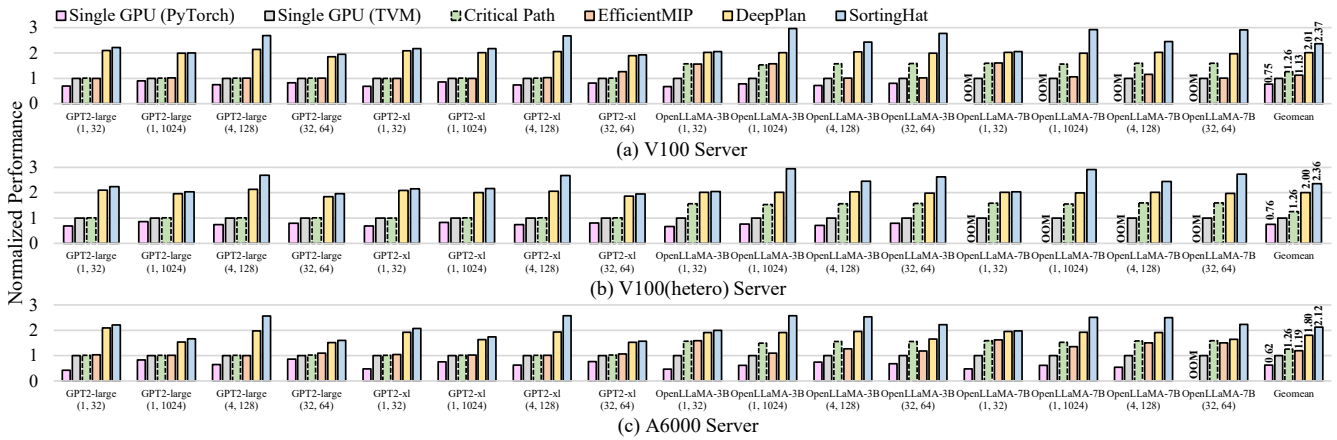

Figure 9: Normalized performance improvements of baselines and *SortingHat* over the performance of single GPU (TVM) on (a) V100, (b) V100(hetero), and (c) A6000 servers. The parentheses on the x-axis are the input shapes and *OOM* stands for *out of memory*. The boxes of Critical Path are drawn with dashed lines as these are theoretical values, not actual end-to-end execution results.

4.2 Baselines

SortingHat is evaluated with a time limit of 30 seconds per submodel for MILP running. Each submodel contains one dominator node and q in MILP was set to 6. Four scheduling methods are used to compare performance with *SortingHat*.

Single GPU: For TVM, the inference latency of serial execution on a single GPU is used as the baseline. Any additional techniques such as DHA and PT are not used, but the only changes applied are DRAM allocation via *cudaMallocHost()* and the use of cuBLAS kernels as mentioned in Section 4.1. PyTorch execution on a single GPU is also used as a baseline. Since the pure end-to-end latency of PyTorch is too slow due to memory allocation, JIT compilation, and other overheads, we only considered data transfer and kernel execution time for a fair comparison. In the V100(hetero) server, the experiments were conducted with the fastest single GPU.

Critical Path: Critical path is the longest path in the model inference process. When sufficient GPUs are available, it corresponds to the theoretical minimum inference time achievable through model parallelism alone. However, *SortingHat* outperforms the theoretical minimum by considering data transfer parallelism.

EfficientMIP: EfficientMIP only considers model parallelism and assumes that all data transfers between GPUs must pass through DRAM with the same speeds. Therefore, EfficientMIP is used to prove the effectiveness of considering the system topology and data transfer parallelism in *SortingHat*. To avoid search space explosion, we partitioned the models into submodels exactly the same as *SortingHat*. Each submodel is scheduled independently without considering the preceding schedule and the data transfer speed is set to be twice as fast to mimic a multi-GPU environment. 30 seconds of time limit per submodel is imposed, and the value of q in EfficientMIP is set to 6. Please note that EfficientMIP is not evaluated on the V100(hetero) server, as it does not support heterogeneous environments.

DeepPlan: As mentioned earlier, DeepPlan proposed DHA and PT strategies to minimize data transfer overhead by fully utilizing PCIe and NVLink bandwidth. However, it did not propose a strategy to maximize model parallelism by fully utilizing GPUs. Therefore, *SortingHat* is expected to gain additional performance improvements over DeepPlan by considering model parallelism.

4.3 Performance Evaluation

Latency Improvement. Figure 9 shows the speedup of *SortingHat* and other baselines over single GPU (TVM). For *SortingHat*, OpenLLaMA models with an input shape of (1, 1024) show remarkable performance improvements of up to 2.96×. Since these models have many branches and large parameters, opportunities for applying not only data transfer parallelism but also model parallelism exist, and thus *SortingHat* considering both parallelism is the most effective framework. Although the model structure is simple, GPT2 models with an input shape of (4, 128) also demonstrate satisfactory performance improvements, as the ratio of data transfer to kernel execution is well balanced. In contrast, the performance improvements achieved by EfficientMIP and DeepPlan are less effective, as they focus on only one of both parallelisms, disregarding system topology.

For GPT2 and OpenLLaMA models with an input shape of (1, 32), data transfer parallelism has a greater impact due

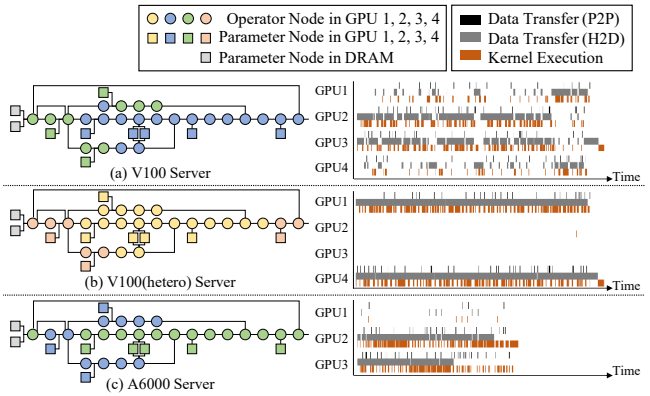

Figure 10: A partial scheduling result (left) and total execution timelines (right) by *SortingHat* for OpenLLaMA-3B (1, 1024) on (a) V100, (b) V100(hetero), and (c) A6000 servers.

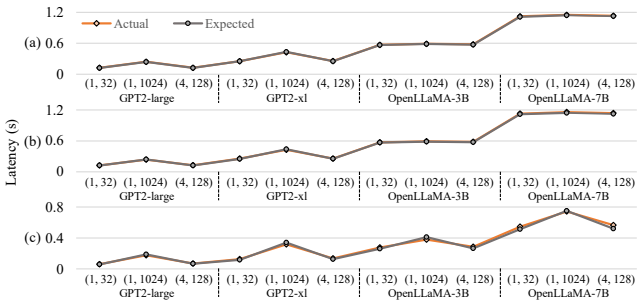

Figure 11: Comparison of expected latency and actual latency on (a) V100, (b) V100(hetero), and (c) A6000 servers.

to the small ratio of kernel execution. Because of these limitations, *SortingHat* achieves similar performance improvements compared to DeepPlan, but still finds the best schedules. EfficientMIP shows similar or even worse performance improvements than Critical Path, as each submodel is scheduled independently without considering the schedule of the preceding submodel. Unlike the input shape of (1, 32), the kernel execution time in (32, 64) has a much greater impact on performance and it means that the influence of model parallelism is significantly greater than that of data transfer parallelism. As a result, the OpenLLaMA models with a more complex structure showed a significant performance improvement, while the GPT2 models showed a smaller improvement. Nevertheless, when compared to other baselines, *SortingHat* still proved to be the most effective framework.

An interesting observation from Figure 9(b) is that *SortingHat* consistently outperforms the other baselines, even on the heterogeneous GPU servers. This is because *SortingHat* avoids low-performance GPUs for kernel execution by considering the system topology, as shown in Figure 10(b), whereas all GPUs are utilized evenly in the V100 server, as depicted in Figure 10(a). The importance of system topology-aware scheduling is also highlighted in A6000 server. Figure 8(c) illustrates that only GPU2 and GPU3 are connected via NVLink, indicating that selecting GPU2 and GPU3 for P2P data transfer is the most efficient option. Indeed, as demonstrated in Figure 10(c), *SortingHat* selected GPU2 and GPU3 for most of nodes, resulting in a notable performance improvement. These results suggest that *SortingHat* is able to search the best schedules with taking system topology into account, across various kinds of systems.

While a direct comparison between PyTorch and TVM is challenging due to the different execution mechanisms of the two frameworks, Single GPU (PyTorch) shows slightly lower performance compared to Single GPU (TVM) for all test cases. Nevertheless, *SortingHat* demonstrates consistent performance gains across various environments and models.

Accuracy of Latency Estimation. Figure 11 demonstrates that *SortingHat* provides the best performance with results nearly identical to those obtained using MILP, not by coincidence. The margin of error between actual and expected latency is 2.97% across all cases, and the overall trend of actual latency closely matches the expected latency. In conclusion, while the performance improvement may vary depending on the target model, *SortingHat* finds the best schedules with accurate latency estimation by considering data transfer parallelism, model parallelism, and system topology. Further ablation studies will prove that this is thanks to the comprehensive inclusion of all these factors.

4.4 Ablation Study

Search Space and Search Time. The number of dominator nodes in a submodel (*nDom*) indicates the size of the submodel. A large *nDom* value may lead to the discovery of schedules that are not found in the MILP with a smaller *nDom*. Similarly, the q value and the MILP time limit exhibit similar characteristics, since more efficient schedule might be found by a larger q value or a longer MILP running time.

Figure 12 proves that *SortingHat* yields enough performance improvements, even when the value of q, the submodel size (*nDom*), and the MILP running time are not large, respectively. In most cases, the performance improvement is greatest when $(q = 6, nDom = 1)$. Although the configuration of $(q = 6, nDom = 2)$ is also effective for reducing inference latency, the best schedules for OpenLLaMA models in Figure 12(b) remain far from the best. Experiments conducted with the $(q = 12, nDom = 1)$ configuration show that the scheduling results do not significantly differ from those obtained with the $(q = 6, nDom = 1)$ configuration. Therefore, q value, submodel size, and search time are not needed to be always large to identify the best solution.

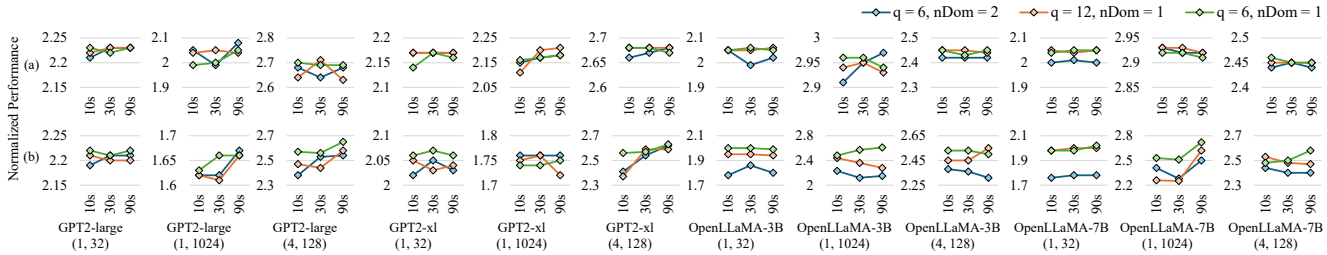

Figure 12: Performance improvements according to different q, $nDom$ values and MILP running time on (a) V100, and (b) A6000 servers. Performance is normalized to the single GPU (TVM) baseline.

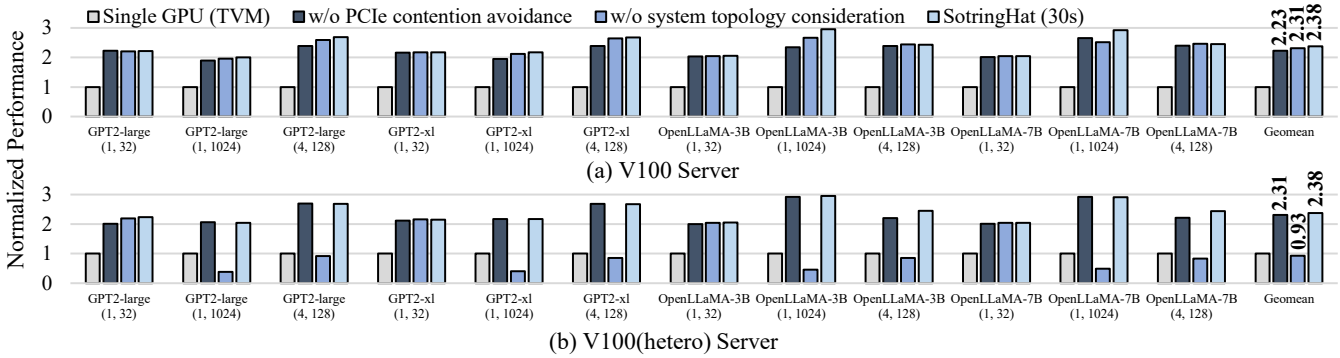

Figure 13: Performance improvements of *SortingHat* without PCIe contention avoidance, *SortingHat* without system topology consideration, and *SortingHat* on (a) V100 and (b) V100(hetero) servers. Each case has a time limit of 30 seconds for MILP running. Performance is normalized to the single GPU (TVM) baseline.

System Topology and PCIe Contention Awareness. The inequalities (12) and (13) in Section 3.3 are modified to $DTStart_j = PrevDTFin_{\lfloor (j-1)/q+1 \rfloor} (\forall j = 1 \ mod \ q)$ and $DTStart_j = DTFin_{j-1}$ $(\forall j > 0 \ \& \ j \neq 1 \ mod \ q)$ with assumption that each GPU has independent PCIe bus, for checking the effectiveness of preventing PCIe contention. The ready queue-based executor is also modified to transfer data using per-GPU ready queues instead of per-PCIe bus. Figure 13 demonstrates that it is consistently inefficient than original *SortingHat* although the performance is not significantly different. To evaluate the impact of system topology awareness, the data transfer speed and GPU performances are all set to their average values. In Figure 13(a), most of the improvements were similar to those observed with the original *SortingHat*, as all GPUs are of the same type. However, in Figure 13(b), severe performance degradation is observed, even worse than a single GPU. These results suggest that system topology and PCIe contention awareness are essential factors in identifying efficient schedules.

Remaining Parameters. Some parameters can remain in memory until they are unloaded. When the same schedule is performed again, data transfer for the remaining parameters can be skipped. Figure 14 shows that *SortingHat* is efficient even in these situations. The experiment was conducted with varying ratios of remaining parameters among previously loaded parameters, using GPT2-large and OpenLLaMA-7B models whose parameter sizes are the smallest and largest, respectively. For the GPT2-large models, re-running the same schedule is slightly slower than a single GPU when all parameters (100%) remain in memory. This is because the original schedule involves data transfers between GPUs, and the total inference latency is too small to hide the overhead of these transfers. Nevertheless, the absolute latency difference is minimal, and *SortingHat* is significantly faster than a single GPU in most other cases.

5 Related Work

Table 3 compares the key aspects of *SortingHat* with relevant works.

Topology-aware Scheduling. Several studies have explored topology-aware scheduling in multi-GPU systems. M. Amaral [1] proposes a GPU placement method designed to efficiently process each request in multi-GPU clouds. While similar to *SortingHat* in its consideration of topology, this work differs in that model parallelism is not incorporated and the scheduling policy aims to minimize the sum of the accumulated execution times of multiple requests. DAPPLE [13] introduces a topology-aware device assignment mechanism

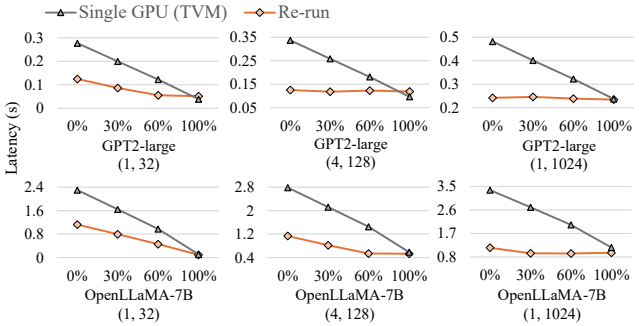

Figure 14: Latency comparison between the single GPU (TVM) baseline and re-running *SortingHat* schedule on the V100 server. The x-axis is the ratio of remaining parameters.

for efficient training. However, DAPPLE does not address PCIe and GPU interconnect structures which are critical for modern GPUs, and focuses on reducing training time.

Model Parallelism. EfficientMIP [46] proposes several algorithms for generating schedules to minimize latency for a given model. However, EfficientMIP provides only theoretical performance improvements and does not consider the underlying system topology. As a result, its applicability is limited in multi-GPU systems with varying topologies. Similarly, Pesto [19] introduces an ILP-based algorithm to find efficient schedules for training on multi-GPU systems. Since Pesto does not address key factors such as data transfer from DRAM to GPU or system topology, it is challenging to implement efficient multi-GPU inference across diverse systems.

Data Transfer Parallelism. PipeSwitch [6] accelerates deep learning applications through pipelined model transmission and unified memory management. However, PipeSwitch supports data transfer parallelism to reduce context switching costs and does not consider system topology. Additionally, PipeSwitch primarily focuses on single-GPU tasks, whereas *SortingHat* is designed for multi-GPU environments. DeepPlan [24] effectively reduces the amount of data transfers and the associated overhead by utilizing direct host access and parallel transmission. Although an additional GPU is employed for parallel transmission, it is solely dedicated to data transfer and not leveraged for model parallelism, missing the opportunity for further latency reduction. Additionally, DeepPlan relies exclusively on P2P data transfers between GPUs connected via NVLink, which requires manual identification of the system topology. This implies that parallel transmission cannot be utilized if NVLink is unavailable or if the system topology is not identified.

Table 3: A comparison table with related works.

✔ : Considered / ▲ : Partially considered / ✘ : Not considered / — : Not confirmed

Works	Topology -aware	Model Parallelism	Data Transfer Parallelism	Heterogeneity
M. Amaral[1]	✔	✘	✘	—
DAPPLE[13]	✔	✘	✘	—
EfficientMIP [46]	✘	✔	✘	✘
Pesto [19]	✘	✔	✘	▲
PipeSwitch [6]	✘	✘	✔	—
DeepPlan [24]	▲	✘	✔	—
SortingHat	✔	✔	✔	✔

6 Discussion

Extensibility to New Trends in LLM Inference. KV cache or prefix cache [40, 44] has been widely used for accelerating LLMs in recent years. PagedAttention [26] proposed partitioning the KV cache into multiple KV blocks and reusing these blocks for efficient memory management. Considering the scenario where *SortingHat* handles the remaining parameters, as depicted in Figure 14, we believe that *SortingHat* would also be efficient in an environment where KV blocks are reused. To efficiently handle multiple requests from users, batching or new serving methodologies are introduced. Orca [54] introduces selective batching, which applies batching only to a selected set of operations. If selectively batched models can be represented as graph format, we expect that *SortingHat* could be extended with Orca. Splitwise [42] suggested scheduling LLMs with a split between the prompt computation phase and the token generation phase, as each phase has different characteristics. By setting the KV cache results from the prompt computation phase as input parameter nodes for the token generation phase and scheduling each phase separately, *SortingHat* could be easily extended to such emerging inference optimizations. While inter-operator parallelism is already handled by model parallelism, we believe that *SortingHat* could be extended to support intra-operator parallelism by treating sub-operators as separate operators or by modifying model graphs. Please note that we do not consider data parallelism, as it is mainly intended to improve training efficiency, whereas *SortingHat* is focused on reducing inference latency.

Extensibility to New Types of DNN Workloads. Following the huge success of LLMs, more advanced models [7, 28, 49, 55] have emerged. These models may be further expanded or entirely new model architectures may be introduced in the near future. We believe that *SortingHat* will be able to handle the model evolution effectively as long as the models can be represented in a computational graph format. In addition to the model architecture, the methods or scenarios for serving DNNs can vary significantly. For example, heterogeneous DNN models could be served together to improve the overall utilization of server resources [11]. In such scenarios, dominator analysis can be applied to each model independently,

as these models are independent. Submodels generated by the models through dominator analysis can then be combined into an integrated submodel. The integrated submodel can be treated and processed just like any other single model, by eliminating the need for specialized handling. Therefore, *SortingHat* will still be an effective framework even in this case.

Extensibility to New Types of GPU Systems. To overcome data transfer bottlenecks, some systems have adopted new equipments such as NVSwitch [38], and we can further accelerate deep learning inference using *SortingHat* on the advanced structures. For example, the AWS p4d.24xlarge [3] instance has eight A100 GPUs fully connected by NVSiwtch, and four PCIe buses connected to DRAM. When we ran the MILP using the system topology of the p4d.24xlarge instance and four models with an input shape of (32, 64), the MILP result predicted that *SortingHat* would achieve up to a 3.7× performance improvement over the expected latency of TVM running on a single GPU in the p4d.24xlarge instance. This demonstrates that *SortingHat* could be effectively applied to emerging types of Multi-GPU systems. Furthermore, new technologies that enhance GPU resource utilization have also been developed. For example, the Multi-Instance GPU (MIG) [37] technique allows a single GPU to be partitioned into multiple virtual GPUs, and the Multi-Process Service (MPS) [35] enables multiple processes to share a single GPU. If each GPU instance is treated as an individual GPU and profiling is conducted accordingly, we believe that *SortingHat* would still be effective.

7 Conclusion

This paper presents *SortingHat* for efficient DNN scheduling on multi-GPU systems. The model is partitioned into multiple submodels based on dominator analysis, and MILP is performed on each submodel with system topology consideration. After merging the submodels based on the found MILP result, the target model is served by the ready queue-based executor with maximized model parallelism and data transfer parallelism. As a result, it achieves an average speedup of 2.28× and up to 2.96× compared to a single GPU on TVM baseline.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments, and Yoon Noh Lee who helped with this paper. This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant (No. RS-2024-00459774, No. RS-2024-00339187, No.RS-2023-00277060, No. RS-2020-II201361), and by the National Research Foundation of Korea(NRF) grant (BK21 FOUR (Department of Computer Science and Engineering, Yonsei University)) funded by the Korea government(MSIT). Yongjun Park is the corresponding author.

References

[1] Marcelo Amaral, Jordà Polo, David Carrera, Seetharami Seelam, and Malgorzata Steinder. 2017. Topology-aware GPU scheduling for learning workloads in cloud environments. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. doi:10.1145/3126908.3126933

[2] Amazon. 2023. Amazon Q Developer. https://aws.amazon.com/q/developer/

[3] Inc. Amazon Web Services. 2020. Amazon EC2 P4 Instances. https://aws.amazon.com/ec2/instance-types/p4/?nc1=h_ls

[4] AMD. 2024. AMD EPYC™ 7262. https://www.amd.com/en/support/downloads/drivers.html/processors/epyc/epyc-7002-series/amd-epyc-7262.html

[5] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*. ACM. doi:10.1145/3620665.3640366

[6] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: fast pipelined context switching for deep learning applications. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 28, 16 pages.

[7] Keqin Chen, Zhao Zhang, Weili Zeng, Richong Zhang, Feng Zhu, and Rui Zhao. 2023. Shikra: Unleashing Multimodal LLM's Referential Dialogue Magic. arXiv:2306.15195 [cs.CV] https://arxiv.org/abs/2306.15195

[8] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[9] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[10] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. PACT: Parameterized Clipping Activation for Quantized Neural Networks. arXiv:1805.06085 [cs.CV]

[11] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[12] Together Computer. 2023. *RedPajama-Data: An Open Source Recipe to Reproduce LLaMA training dataset.* https://github.com/togethercomputer/RedPajama-Data

[13] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Virtual Event, Republic of Korea) *(PPoPP '21)*. Association for Computing Machinery, New York, NY, USA, 431–445. doi:10.1145/3437801.3441593

[14] Xinyang Geng and Hao Liu. 2023. *OpenLLaMA: An Open Reproduction of LLaMA*. https://github.com/openlm-research/open_llama

[15] Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. AI and Memory Wall . *IEEE Micro* 44, 03 (May 2024), 33–39. doi:10.1109/MM.2024.3373763

[16] GitHub. 2024. GitHub Copilot · Your AI pair programmer. https://github.com/features/copilot

[17] Google. 2023. Gemini. https://gemini.google.com/app

[18] Gurobi Optimization, LLC. 2025. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[19] Ubaid Ullah Hafeez, Xiao Sun, Anshul Gandhi, and Zhenhua Liu. 2021. Towards optimal placement and scheduling of DNN operations with Pesto. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 39–51. doi:10.1145/3464298.3476132

[20] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[21] Intel. 2022. Intel® Core™ i7-13700K Processor. https://www.intel.com/content/www/us/en/products/sku/230500/intel-core-i713700k-processor-30m-cache-up-to-5-40-ghz/specifications.html

[22] Intel. 2022. Intel® Xeon® Processor E5-2698 v4. https://www.intel.com/content/www/us/en/products/sku/91753/intel-xeon-processor-e52698-v4-50m-cache-2-20-ghz/specifications.html

[23] Yongkweon Jeon, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Jeongin Yun, and Dongsoo Lee. 2020. BiQGEMM: matrix multiplication with lookup table for binary-coding-based quantized DNNs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Atlanta, Georgia) *(SC '20)*. IEEE Press, Article 95, 16 pages.

[24] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 249–265. doi:10.1145/3552326.3567508

[25] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[26] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. doi:10.1145/3600006.3613165

[27] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R. Tallent, and Kevin J. Barker. 2020. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Trans. Parallel Distrib. Syst.* 31, 1 (jan 2020), 94–110. doi:10.1109/TPDS.2019.2928289

[28] Haotian Liu, Chunyuan Li, Qingyang Wu, and Yong Jae Lee. 2023. Visual Instruction Tuning. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 34892–34916. https://proceedings.neurips.cc/paper_files/paper/2023/file/6dcf277ea32ce3288914faf369fe6de0-Paper-Conference.pdf

[29] NVIDIA. 2012. How to Overlap Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/

[30] NVIDIA. 2019. NVIDIA DGX STATION DATASHEET. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/dgx-station/nvidia-dgx-station-datasheet-uk.pdf

[31] NVIDIA. 2021. CUDA Toolkit 11.4 Downloads. https://developer.nvidia.com/cuda-11-4-0-download-archive

[32] NVIDIA. 2022. NVIDIA RTX A6000 Graphics Card. https://resources.nvidia.com/en-us-briefcase-for-datasheets/proviz-print-nvidia-1?ncid=no-ncid

[33] NVIDIA. 2023. NVIDIA NVLink: High-Speed GPU Interconnect. (2023). https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/

[34] NVIDIA. 2024. CUDA Toolkit 12.4 Downloads. https://developer.nvidia.com/cuda-12-4-0-download-archive

[35] NVIDIA. 2025. Multi-Process Service. https://docs.nvidia.com/deploy/mps/index.html

[36] NVIDIA. 2025. *NVIDIA cuBLAS*. https://docs.nvidia.com/cuda/cublas/index.html

[37] NVIDIA. 2025. NVIDIA Multi-Instance GPU. https://www.nvidia.com/en-us/technologies/multi-instance-gpu/

[38] NVIDIA. 2025. NVLink and NVLink Switch. https://www.nvidia.com/en-us/data-center/nvlink/

[39] OpenAI. 2022. ChatGPT. https://openai.com/chatgpt/

[40] Rui Pan, Zhuang Wang, Zhen Jia, Can Karakus, Luca Zancato, Tri Dao, Yida Wang, and Ravi Netravali. 2024. Marconi: Prefix Caching for the Era of Hybrid LLMs. arXiv:2411.19379 [cs.DC] https://arxiv.org/abs/2411.19379

[41] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 307–321. https://www.usenix.org/conference/atc20/presentation/park

[42] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. 2024. Splitwise: Efficient generative LLM inference using phase splitting. In *ISCA*. https://www.microsoft.com/en-us/research/publication/splitwise-efficient-generative-llm-inference-using-phase-splitting/

[43] Antonio Polino, Razvan Pascanu, and Dan Alistarh. 2018. Model compression via distillation and quantization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=S1XolQbRW

[44] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently Scaling Transformer Inference. In *Proceedings of the Sixth Conference on Machine Learning and Systems, MLSys 2023, Miami, FL, USA, June 4-8, 2023*, Dawn Song, Michael Carbin, and Tianqi Chen (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2023/hash/c4be71ab8d24cdfb45e3d06dbfca2780-Abstract-mlsys2023.html

[45] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[46] Jakub Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient algorithms for device placement of DNN graph operators. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1296, 13 pages.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[48] Christos Valouxis, Christos Gogos, Panayiotis Alefragis, George Goulas, Nikolaos Voros, and Efthymios Housos. 2013. Dag scheduling using integer programming in heterogeneous parallel execution environments. In *Proceedings of the Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2013)*. Ghent, Belgium, 392–401.

[49] Peng Wang, Shuai Bai, Sinan Tan, Shijie Wang, Zhihao Fan, Jinze Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Yang Fan, Kai Dang, Mengfei Du, Xuancheng Ren, Rui Men, Dayiheng Liu, Chang Zhou, Jingren Zhou, and Junyang Lin. 2024. Qwen2-VL: Enhancing Vision-Language Model's Perception of the World at Any Resolution. arXiv:2409.12191 [cs.CV] https://arxiv.org/abs/2409.12191

[50] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal instruction scheduling using integer programming. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (Vancouver, British Columbia, Canada) *(PLDI '00)*. Association for Computing Machinery, New York, NY, USA, 121–133. doi:10.1145/349299.349318

[51] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771 [cs.CL]

[52] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[53] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: accurate and efficient post-training quantization for large language models. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1585, 13 pages.

[54] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[55] Deyao Zhu, Jun Chen, Xiaoqian Shen, Xiang Li, and Mohamed Elhoseiny. 2023. MiniGPT-4: Enhancing Vision-Language Understanding with Advanced Large Language Models. arXiv:2304.10592 [cs.CV] https://arxiv.org/abs/2304.10592

[56] M. Yusuf Özkaya and Ümit V. Çatalyürek. 2022. A Simple and Elegant Mathematical Formulation for the Acyclic DAG Partitioning Problem. arXiv:2207.13638 [cs.DS] https://arxiv.org/abs/2207.13638