GSPO: A Graph Substitution and Parallelization Joint Optimization Framework for DNN Inference

Zheng Xu[1], Xu Dai[2], Shaojun Wei[1,3], Shouyi Yin[1,3], and Yang Hu[1*]
1. School of Integrated Circuits, 3. BNRist, Tsinghua University, Beijing, China, 100084
2. Shanghai Artificial Intelligence Laboratory, Shanghai, China, 200433

ABSTRACT

This work proposes GSPO, an automatic unified framework that jointly applies graph substitution and parallelization for DNN inference. GSPO uses a joint optimization computation graph (JOCG) to represent graph substitution and parallelization at the operator level. Then, a novel cost model customized for joint optimization is used to evaluate the computation graph execution time quickly. With the graph partition and backtracking search algorithm, GSPO can find the optimal joint optimization solution within an acceptable search time. Compared to existing frameworks applying graph substitution or parallelization, GSPO can achieve up to 27.1% end-to-end performance improvement and reduce search time by up to 94.3%.

KEYWORDS

DNN inference, Graph substitution, Parallelization, Joint optimization

1 INTRODUCTION

Deep neural networks (DNNs) have made extraordinary achievements in many fields such as computer vision[9, 13] and machine translation[2, 6, 17]. To obtain better performance, DNN models are currently becoming larger and larger, bringing enormous challenges to inference models in actual scenarios. How to improve the efficiency of DNN inference has received more and more attention.

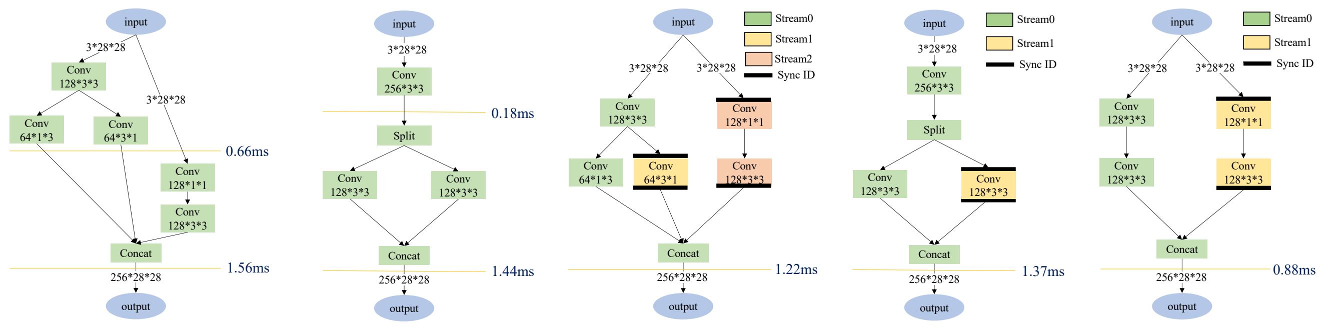
Deep learning (DL) frameworks represent the neural architecture as the computation graph, where each node represents mathematical operators (e.g., convolution, maxpool, matrix multiplication, etc.), and each edge represents the data dependency between different nodes. DNN inference can be effectively accelerated by optimizing the computational graph. To improve the runtime performance of the computation graph, the most common and effective optimization method is graph substitution.

*This work was supported in part by the National Science and Technology Major Project under Grant 2022ZD011520; the NSFC under Grant 62125403, and Grant 92164301; Beijing ST Project Z221100007722023; in part by the project funding for the 2022 Special Project on Industrial Foundation Reconstruction and High Quality Development of Manufacturing Industry CEIEC-2022-ZM02-0245; in part by the Beijing National Research Center for Information Science and Technology; and in part by the Beijing Advanced Innovation Center for Integrated Circuits. (*Corresponding author: Yang Hu, hu_yang@tsinghua.edu.cn)

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0601-1/24/06
https://doi.org/10.1145/3649329.3655683

Graph substitution will replace the original DNN computation subgraph with an equivalent subgraph with improved performance. Operator fusion is a typical example of graph substitution, which combines two or more operators into a single semantically equivalent operator, which can eliminate intermediate results to reduce system overhead, such as memory accesses and kernel launches. Existing deep learning systems such as TensorFlow[1], Pytorch[15], and TVM[4] use the operator fusion method and adopt the greedy rule-based strategy to optimize the computation graph. Fig. 1(a) shows the process of Pytorch[15] executing a particular DNN task, which uses a sequential execution strategy. However, operator fusion has strict restrictions on the operators to be fused. For example, having the same convolution kernel size is necessary for the fusion of two convolution operators. MetaFlow[12], TASO[11], etc., have proposed graph substitution based on relaxed rules to break this limitation and obtain more significant optimization space. As shown in Fig. 1(b), it implements more equivalent graph substitution operations and has less total execution time with the same DNN task. For the fusion of convolution operators with 3×3 and 1×1 kernel, the specific process is to enlarge the kernel size from 1×1 to 3×3 by padding with zeros and then perform operator fusion. Although such an operation increases the amount of computation, its final execution time will be less due to the reduction of memory accesses and kernel launches. Graph substitution can achieve a certain degree of performance improvement, but it only utilizes intra-operator parallelization and ignores inter-operator parallelization. Each kernel monopolizes all hardware resources, resulting in low hardware utilization.

Parallelization generally refers to inter-operator parallelization, which performs multiple branch operations simultaneously through GPU multi-stream or other methods to improve the utilization of hardware resources and reduce the impact of kernel launches. For the NVIDIA GPU, CUDA programming interface[5] provides Stream API, allowing users to manually allocate operators in different streams to achieve concurrent kernel execution. Recent work has focused on automatic multi-stream execution, such as Nimble[14], AutoGraph[18], etc. As shown in Fig. 1(c), it is the result of Nimble executing the same DNN task in Fig. 1(a). Although automatic parallel execution can improve performance, this optimization method is imperfect. Automatic parallel execution allocates a stream to each branch in the computation graph. If there are too many streams, the synchronization overhead between different streams will be substantial, leading to an imbalance of workloads at different periods and underutilization of hardware resources. In Fig. 1(c), Stream1 must wait to complete the execution of the first convolution operator in Stream0 before it can commence. Operators with fewer FLOPs are executed before the initiation of Stream1, leading to suboptimal utilization of inter-operator parallelization,



(a) Sequential Execution (b) Equivalent Graph Substitution (c) Parallel Execution (d) Direct Joint Execution (e) Our Work

Figure 1: Different execution methods for the same DNN task on NVIDIA Tesla V100 GPU. (a) Pytorch[15]: An example of sequential execution without any optimization. (b) TASO[11]: An example of graph substitution. (c) Nimble[14]: An example of parallel execution. (d) TASO+Nimble: An example of directly combining graph substitution and parallelization. (e) Our Work: A joint optimization framework that can fully utilize the advantages of graph substitution and parallelization.

while operators with larger FLOPs are executed after it, creating a resource contention problem that adversely impacts hardware performance.

When focusing solely on parallelization optimization, the problem arises from workload imbalance due to excessive branches in the computation graph. Graph substitution offers a solution by merging operators to reduce parallel branches. Hence, the effective integration of graph substitution and parallelization promises a more significant enhancement in DNN inference performance. As shown by our work in Fig. 1(e), the redundant Stream1 in Fig. 1(c) is eliminated by merging the convolution operators with a 1 × 3 kernel and a 3 × 1 kernel, thereby significantly reducing the inference time.

However, effectively combining graph substitution and parallelization is not an easy task. The straightforward approach involves executing both optimization methods sequentially and applying them independently. For instance, one could initially perform graph substitution and subsequently optimize the altered graph through parallel execution. The end-to-end performance may be worse than applying only one optimization method, as shown in Fig. 1(d). The fundamental reason is that the two optimization methods have different goals: graph substitution aims to merge as many operators as possible, while parallelization seeks to execute operations as concurrently as possible. The fusion of operators through graph substitution may compromise the potential for inter-operator parallelization, leading the optimization in the wrong direction.

To solve the problem of the effective combination of graph substitution and parallelization, we propose GSPO, an automatic joint optimization framework that applies graph substitution and parallelization. In summary, this work makes the following contributions:

1) **A joint optimization method of graph substitution and parallelization at the operator level.** We introduce the JOCG (Joint Optimization Computational Graph), which can perform operator fusion and add synchronization operators at the operator level. This breaks the limitations of existing graph substitution and parallelization methods, providing more optimization opportunities. At the same time, the cost of obtaining different candidate graphs is reduced since the optimization is carried out at a fine-grained level. Combined with our graph partition and backtracking search algorithm, this significantly decreases the scheduling time for complex DNN computation graphs.
2) **A novel cost model suitable for evaluating sequential and parallel execution scenarios.** By measuring execution time for representative operators and various computation subgraphs with distinct branch numbers and FLOPs, we established cost models for sequential and parallel execution scenarios. Subsequently, the total cost was derived by combining multiple metrics for each operator on the computation graph. Compared with the actual measured values on the hardware, our cost model has proven effective.
3) **An optimization framework for DNN inference further improves performance.** By applying our joint optimization mechanism and backtracking search based on our cost model, GSPO achieves lower latency and search time, yielding significant performance improvement. Compared to existing optimization frameworks, GSPO can improve end-to-end inference performance by up to 27.1% and reduce search time by up to 94.3%.

2 CHALLENGES

As the number of SM units in GPUs increases, executing a single DNN operator can no longer ensure efficient resource utilization. It is necessary to exploit the parallelization. DNN models are gradually replacing linear chains of operators with multiple branches, expanding the space for parallel scheduling. Therefore, effective methods are needed to find the optimal result. Graph substitution and parallelization joint optimization method can optimize the parallel scheduling through graph substitution and maximize performance, but it also brings significant challenges.

A joint representation of graph substitution and parallelization is necessary. Existing frameworks can only perform graph substitution or parallelization on the entire computation graph. However, graph substitution will disrupt the parallelization allocation results. To jointly apply both optimization methods during the search process, we need a representation that allows graph substitution and parallelization to coexist at a fine-grained level.

A cost model specifically for joint optimization of graph substitution and parallelization is necessary. The cost models of existing work are mainly divided into two types. One is the actual deployment time on hardware[14][18]. The other is to estimate the cost by measuring the actual deployment time of representative operators on hardware and synthesizing the essential metrics for each operator[11][12][8]. The first one can reflect the cost of the computation graph very well, but because it requires the actual deployment of each candidate graph, the actual search time will be unacceptably large. The second method greatly reduces the search time by estimating the cost, but this method ignores the parallel execution scenario. Therefore, we need a cost model that can be applied to guide joint optimization and rapidly and accurately estimate the cost of computation graphs.

A method to quickly find the optimal joint optimization results in a huge schedule space is necessary. Existing search-based optimization frameworks only focus on improving either graph substitution or parallelization. However, the complexity of their search algorithms has made optimizing complex DNNs like NasNet-A[20] very challenging. Simultaneously considering both optimization approaches only exacerbates the problem. To make the joint optimization framework practical, we must optimize the schedule space to obtain optimized results quickly.

In summary, joint optimization of graph substitution and parallelization requires a representation that can indicate two optimization methods, a cost model that can accurately estimate its execution time, and it also needs to quickly find the optimal joint optimization result in a huge schedule space. These conclusions significantly guide the design of GSPO.

3 GSPO

This work proposes GSPO, a joint graph substitution and parallelization optimization framework. Fig. 2 illustrates the overall process of GSPO. It consists of four essential optimization steps. First, GSPO will use the flow-based graph partition method to divide the initial computation graph into several disjoint subgraphs to ensure rapid optimization (Section 3.1). Second, GSPO will process subgraphs at the operator level and obtain the priority candidate queue, using JOCG to complete graph substitution and add synchronization identification operators applied to parallelization (Section 3.2). Third, GSPO will evaluate all candidate graphs using a cost model designed explicitly for joint optimization, and the candidate queue is dequeued in increasing order by their costs (Section 3.3). Fourth, GSPO will utilize a backtracking search algorithm to explore graph optimization solutions, put candidate graphs with optimized performance into a candidate list, and perform on-board measurements, using runtime results to find the optimal result (Section 3.4). Finally, all optimized subgraphs are merged to obtain the optimized computation graph.

3.1 Flow-based Computation Graph Partition

Most state-of-the-art DNN models present a challenge for direct computation graph optimization due to their enormous size. Additionally, the abundance of substitution rules, synchronization identification operators, and optimization iterations further exacerbates this situation. Therefore, an effective graph partition strategy is urgently needed to reduce the search space and maximize the opportunities for joint optimization. We use a flow-based graph

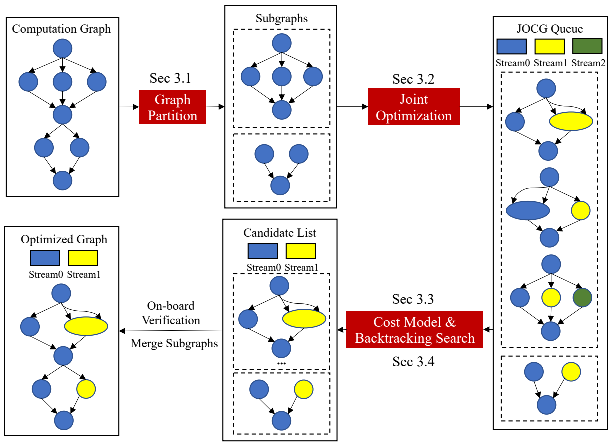


Figure 2: The overall flow of GSPO.

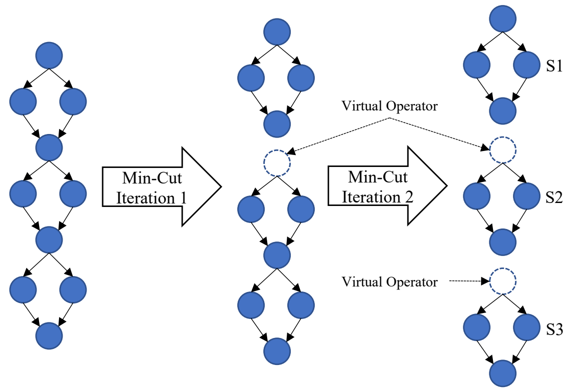


Figure 3: Flow-based algorithm for graph partition.

partition strategy to divide a computation graph into disjoint subgraphs. This is motivated by our observation that most DNN models are composed of several blocks connected sequentially, such as inception V3[16], Resnet50[9], etc. Graph substitution and parallelization can be performed generally within a block and there are almost no optimization opportunities between blocks.

To split a graph into several disjoint subgraphs, we aim to preserve most graph substitution and parallelization opportunities. For each operator $O_i \in S$, we define its weight $W(O_i)$ as the number of different mapping that maps the operator to an operator in the initial graph of the optimization rules r. To split the graph, we aim to find the set of operators O with minimum weight sum. By using the weight $W(O_i)$ for each operator, we transform the graph split problem into a minimum cut problem, and we use the Boykov algorithm[3] to find the minimum cut.

We iteratively use the minimum cut algorithm to split the computation graph into subgraphs smaller than the threshold T and obtain the subgraphs set $S_m = \{S_1, S_2, \ldots, S_n\}$, as shown in Fig. 3. It is worth noting that subgraph S_i and S_{i+1} may share the same operator O_i, and we generally choose to assign operator O_i to subgraph S_i. However, to prevent subgraph S_{i+1} from losing graph substitution and parallelization opportunities due to the lack of a head operator, we will create a virtual node O_i' for subgraph S_{i+1} to express the input of the subgraph.

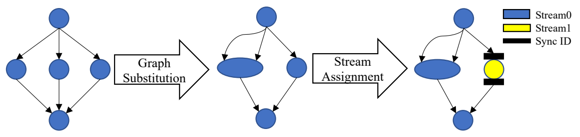


Figure 4: An example of JOCG representation.

3.2 Joint Optimization Computation Graph

To find a better-optimized computation graph that combines graph substitution and parallelization as shown in Fig. 1(e), we introduce the Joint Optimization Computation Graph (JOCG) as a unified representation of DNN inference. JOCG is a computation graph representation specially customized for joint optimization, allowing for the simultaneous expression of graph substitution and parallelization. Combined with our cost model, it can better estimate the sequential and parallel execution time.

JOCG extends traditional computation graph representations, enabling the simultaneous representation of both optimization methods at the operator level. For graph substitution, JOCG rewrites the computation graph by replacing the original subgraph with an equivalent subgraph that improves performance. In parallelization, JOCG achieves multi-stream allocation by adding parallel synchronization identifiers. Our cost model can recognize the parallel synchronization identifier and transition into a parallel execution evaluation mode. The parallel synchronization identifier can also be identified during on-board verification for rapid feedback.

The ideal multi-stream allocation strategies should maximize the logical concurrency while minimizing the number of synchronizations. We use the fast analytical algorithm in Nimble[14] to realize this process, and it can be embedded into our framework well. Fig. 4 shows an example of JOCG representing graph substitutions and parallelization at a fine-grained level. The JOCG representation will be added to the candidate queue and sorted based on the cost.

3.3 Cost Model

To solve the shortcoming that the existing cost model is unsuitable for parallel execution, we introduce a new cost model that incorporates multiple dimensions to evaluate the joint optimization results of graph substitution and parallelization. The cost model will compute various metrics of each operator in the subgraph and apply them to estimate the total cost. These metrics include FLOPs, number of kernel launches, memory usage, execution time, and hardware utilization. We compute the cost model in two scenarios: sequential execution and parallel execution:

1) Sequential execution: In this scenario, operators in the graph will be executed sequentially, each occupying all available hardware resources individually during execution. Consequently, we can consider the total cost of the computation graph as the sum of the costs of all operators. The expression for the cost model is shown in Equation (1).

$$Cost = \sum_{O_i \in V} cost(O_i) \tag{1}$$

V is the set of all operators in the computation graph. Since most DNN operators involve dense linear algebra without branches, the cost of a single operator can be predicted by measuring the execution time of representative operators. For example, for a convolution operator with a kernel size of 3*3 and a stride of 1, we can pre-measure the representative operator O_c with the same parameters and given input, and store its execution time and FLOPs. After that, we can use the metric FLOPs of the operator O_i to estimate its execution time, as shown in Equation (2).

$$cost(O_i) = \frac{FLOPs(i)}{FLOPs(c)} \times Execution\_time(c) \tag{2}$$

2) Parallel execution: In this scenario, we will execute the computation graph with multiple branches in parallel. We will measure the execution time of the representative computation graphs with multiple branches (different number of branches, FLOPs of each branch, and operator types) and store their average hardware utilization β_a. We can use the polynomial regression model to establish the relationship between various metrics and the average hardware utilization β_a in the parallel execution scenario based on the measurement results. Therefore, for a computation graph that needs to be executed in parallel, we can predict its cost by combining the average hardware utilization β_a and other metrics (FLOPs and hardware utilization executed alone of each branch, etc.), as shown in Equation (3).

$$Cost = \max_{i \in G}(\frac{\beta_i}{\beta_a}) \times cost(i) \tag{3}$$

In actual situations, we can iteratively use the cost model of sequential and parallel execution to obtain the cost of the computation graph. For example, a subgraph consists of multiple branches, and each branch has various operators. We can apply the rules for sequential execution to get the *cost(i)* for each branch and then use the rules for parallel execution to derive the final *Cost*.

3.4 Backtracking Search Algorithm

Different JOCG candidate graphs can be generated by applying graph substitution and parallelism at the operator level. Enumerating all possible candidate graphs and measuring them exhaustively is very inefficient. We adopt a cost-based backtracking search algorithm to find the optimal graph quickly. Algorithm 1 shows the pseudocode of the complete backtracking search process.

Algorithm 1 Backtracking Search Algorithm

```
Input: An initial computation subgraph $G_0$ and hyper parameter $\alpha$
Output: An optimized subgraph $G_{opt}$
 1: $Q \leftarrow \{G_0\}$; // priority queue Q
 2: $G_{best} \leftarrow G_0$;
 3: while $Q \neq \emptyset$ do
 4:     $G \leftarrow Q.pop()$;
 5:     while $JOCG_R \neq \emptyset$ do
 6:         $G' \leftarrow JOCG_R(G)$; // graph substitution
 7:         $G'' \leftarrow JOCG_P(G')$; // adding synchronization identifiers
 8:         if $Cost(G'') \leq \alpha \times Cost(G_{best})$ then
 9:             Q.push(G');
10:         end if
11:         if $Cost(G'') \leq Cost(G_{best})$ then
12:             L.insert(G”); // candidate list L for on-board verification
13:         end if
14:     end while
15: end while
16: for $all\ G_H$ in $H$ do
17:     if $Latency(G_H) \leq Latency(G_{best})$ then
18:         $G_{opt} \leftarrow G_H$;
19:     end if
20: end for
21: return $G_{opt}$;
```

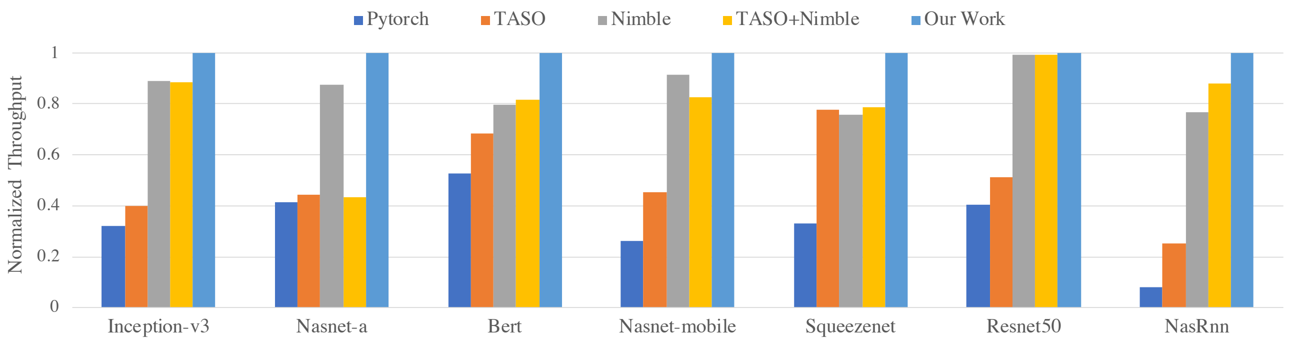


Figure 5: End-to-end performance comparison of optimization frameworks across seven real-world DNN models.

Based on our graph split method, we can perform a cost-based backtracking search on each subgraph for fast optimization. We take each subgraph as input. In each round of iterative search, we apply JOCG representation to generate candidate graphs, where $JOCG_R$(line 6) represents using rule R for graph substitution and $JOCG_P$(line 7) represents adding synchronization identifiers for parallel optimization. Candidate graphs within α times the cost of the current best value are collected and saved to the priority queue Q for the next iteration of the search(lines 8-9). Candidate graphs whose costs are strictly less than the current optimal result are saved to candidate list L(lines 11-12) for subsequent on-board verification (lines 16-20).

4 EVALUATION

4.1 Experimental Setup

All experiments were performed on the 2.70GHz Intel Xeon Platinum 8168 CPU and NVIDIA Tesla V100 GPU. We benchmark seven representative DNN models to evaluate our work, including five CNN models, one RNN model, and one transformer model. Resnet50[9], Inception-v3[16], and Squeezenet[10] are three widely used CNN architectures for image classification and object recognition. NasNet-A and NasNet-Mobile[20] are the representative CNN architectures automatically discovered by neural architecture search methods. NasRNN[19] is an RNN architecture that automates the design of efficient and effective architectures for sequential data tasks discovered through neural architecture search methods. BERT[6] is a powerful pre-trained transformer-based deep learning model widely used for natural language understanding and processing tasks.

4.2 End-to-end Performance

We compare the end-to-end inference performance of the benchmark DNN models between our work with the following 4 baselines: a framework with neither graph substitution nor parallel optimization, a framework with graph substitution, a framework with parallel optimization, and a framework with direct joint graph substitution and parallel optimization. PyTorch[15] is the most widely adopted development tool and we regard it as a baseline framework with neither graph substitution nor parallel optimization. TASO[11] is a representative framework performed graph substitution. For parallel framework, we choose Nimble[14] as a

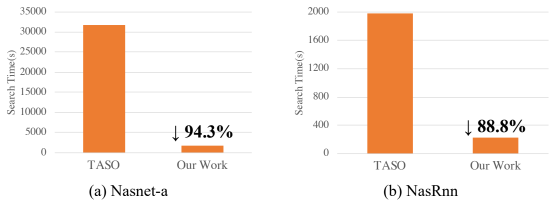


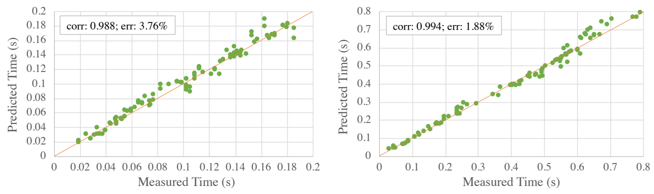
Figure 6: Search time comparison of our work and TASO without limitation across representative complex DNN models. (a) Nasnet-a. (b) NasRnn.

baseline. We regard the sequential execution of TASO and Nimble framework as a direct joint optimization baseline.

The experimental results are shown in Fig. 5. Due to the different complexity of DNN models, there will be a large gap in end-to-end performance. To express the results more intuitively, we normalize throughput to the best one for each model. We conduct each experiment 20 times and report the average performance. Our approach outperforms two state-of-the-art frameworks, TASO[11] and Nimble[14], across all benchmark models, achieving an impressive up to 27.1% improvement in end-to-end performance. This demonstrates that the joint application of graph substitution and parallelization can enhance DNN inference performance. Compared with the framework of simply combining two optimization methods, our work can achieve up to 130.3% inference acceleration. This remarkable performance is due to our JOCG representation, cost model, and search mechanism customized for joint optimization. It proves that our method is not a simple synthesis of two existing optimization methods, but an innovative joint optimization framework.

4.3 Search Time

We compare the search time of the complex DNN models between our work and TASO[11] without limiting the number of iterations. Because our method is a joint optimization framework and introduces on-board verification to obtain more accurate runtime information, the search time of our method on simple DNN models is almost not improved, or even worse. Therefore we do not consider them as our benchmark models in evaluating search time. Fig. 6 shows the search time results on two representative complex DNN models, Nasnet-a[20] and NasRNN[19]. It can be seen that compared with the existing framework, our method can reduce the



(a) Sequential Execution (b) Parallel Execution

Figure 7: Cost model accuracy test. (a) Sequential execution. (b) Parallel execution.

search time by up to 94.3%, proving that our method has improved performance for the search time of complex DNN models.

4.4 Cost Model Performance

We test the performance of the cost model under sequential execution and parallel execution. We select 100 representative computation subgraphs (different FLOPs, different numbers of subgraph branches, different numbers of operators, etc.) for each scenario and we compare them with the results measured on the GPU. As shown in Fig. 7, the predicted results of the cost model exhibit a nearly linear correlation with the measured results on the GPU. The average error rate under sequential execution is 3.76%, while the average error rate under parallel execution is 1.88%. It is proven that our cost model can be used to guide the joint optimization of graph substitution and parallelization.

5 RELATED WORK

Graph substitution. MetaFlow[12] allows graph substitutions with relaxed performance constraints to achieve better performance under a longer sequence of substitutions. TASO[11] further introduces an automated generation of substitution rules relying on its operator definitions and specifications, enabling a larger search space than manually designed ones. It also provides semi-automatic support for verifying the correctness of substitutions. OCGGS[8] also adopts the relaxed graph substitutions strategy and proposes an effective sampling algorithm to reduce the excessive search time caused by the huge search space, especially for complex models. Compared to the previous work, GSPO takes full advantage of the performance improvements brought by inter-operator parallelization instead of blindly maximizing graph substitutions.

Parallelization. Nimble[14] is an automatic deep learning execution engine supporting parallel kernel launch and proposes an ahead-of-time (AoT) scheduling technique to reduce the scheduling overhead. IOS[7] divides the computation graph into different stages, applies dynamic programming algorithms to find optimal scheduling results iteratively, and supports concurrent execution of different types of operators. Autograph[18] is an automatic parallel kernel launch optimization framework that uses a customized cost function and introduces accurate runtime information as feedback. However, existing approaches only support inter-operator parallelization and ignore the optimization opportunities brought by applying graph substitution to reduce the number of parallel execution branches. GSPO can obtain better end-to-end performance by joint graph substitution and parallelization within an acceptable search time.

6 CONCLUSION

This work proposes an automatic unified framework that jointly applies graph substitution and parallelization, GSPO, which reduces the inference time and optimization cost of DNN models. It can optimize the DNN computation graph with a JOCG representation of graph substitution and parallelization at the operator level, a customized cost model for performance estimation, and a backtracking search algorithm combined with accurate runtime feedback. Experiments show that GSPO can achieve up to 27.1% end-to-end performance improvement and reduce search time by up to 94.3% with the previous state-of-the-art framework on seven widely used DNN models. Moreover, GSPO can achieve up to 130.3% performance improvement compared to a framework that simply combines graph substitution and parallelization.

REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, and et al. Davis, A. 2016. TensorFlow: A System for Large-Scale Machine Learning. *In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2016).

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* (2014).

[3] Yuri Boykov and Vladimir Kolmogorov. 2004. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE transactions on pattern analysis and machine intelligence* (2004).

[4] T.; Jiang Z.; Zheng L. Yan E. Shen H. Cowan M. Wang L. et al. Chen, T.; Moreau. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. *13th USENIX Symposium on Operating Systems Design and Implementation* (2018).

[5] CUDA. 2022. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda.

[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR* (2018).

[7] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. Ios: Inter-operator scheduler for cnn acceleration. *Proceedings of Machine Learning and Systems* (2021).

[8] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN Computation Graph Using Graph Substitutions. *Proc. VLDB Endow.* (2020).

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016).

[10] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. *arXiv preprint arXiv:1602.07360* (2016).

[11] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*.

[12] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *Proceedings of Machine Learning and Systems*.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. *Advances in Neural Information Processing Systems* (2012).

[14] Woosuk Kwon, Gyeong-In Yu, Eunji Jeong, and Byung-Gon Chun. 2020. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. In *Advances in Neural Information Processing Systems*.

[15] PyTorch. 2022. PyTorch CUDA Semantics. https://pytorch.org.

[16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016).

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. *Advances in Neural Information Processing Systems* (2017).

[18] Yuxuan Zhao, Qi Sun, Zhuolun He, Yang Bai, and Bei Yu. 2023. AutoGraph: Optimizing DNN Computation Graph for Parallel GPU Kernel Execution. *Association for the Advancement of Artificial Intelligence* (2023).

[19] Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578* (2016).

[20] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018).