AutoGraph: Optimizing DNN Computation Graph for Parallel GPU Kernel Execution

Yuxuan Zhao, Qi Sun, Zhuolun He, Yang Bai, Bei Yu
The Chinese University of Hong Kong
{yxzhao21,qsun,zlhe,ybai,byu}@cse.cuhk.edu.hk

Abstract

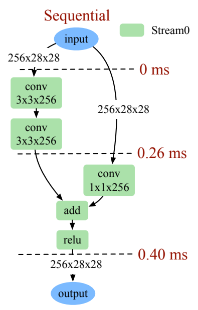
Deep learning frameworks optimize the computation graphs and intra-operator computations to boost the inference performance on GPUs, while inter-operator parallelism is usually ignored. In this paper, a unified framework, AutoGraph, is proposed to obtain highly optimized computation graphs in favor of parallel executions of GPU kernels. A novel dynamic programming algorithm, combined with backtracking search, is adopted to explore the optimal graph optimization solution, with the fast performance estimation from the mixed critical path cost. Accurate runtime information based on GPU Multi-Stream launched with CUDA Graph is utilized to determine the convergence of the optimization. Experimental results demonstrate that our method achieves up to 3.47$\times$ speedup over existing graph optimization methods. Moreover, AutoGraph outperforms state-of-the-art parallel kernel launch frameworks by up to 1.26$\times$.

Introduction

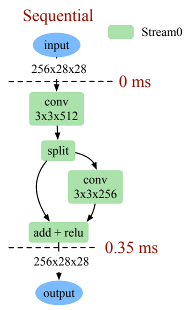
Deep neural networks (DNNs) have shown remarkable power on many tasks such as computer vision (He et al. 2016; Redmon et al. 2016) and natural language processing (Vaswani et al. 2017; Devlin et al. 2018). The state-of-the-art models get deeper and larger to achieve better performance, which causes a great challenge to deployment in actual scenarios. As a result, the acceleration of modern DNNs is in great demand.

Deep learning (DL) frameworks (Chen et al. 2015; Abadi et al. 2016; Paszke et al. 2019) represent the neural networks as graphs and optimize the graphs to accelerate inference. In the graph, a node represents the atomic DL operator (*e.g.*, convolution, pooling), and an edge represents the data dependency between two nodes. A widely used graph optimization approach is *equivalent graph substitutions* (Jia et al. 2019a; Zhao et al. 2022; Xing et al. 2022). TensorFlow, PyTorch and TVM (Chen et al. 2018) perform greedy rule-based substitutions in a predefined order. For example, one rule used by them is to replace convolution, batch normalization, and activation with an equivalent fused operator, which can eliminate intermediate results and reduce memory access. MetaFlow (Jia et al. 2019b) allows graph substitutions with relaxed performance constraints to enlarge the search space. It adopts a flow-based method to split the graph and uses a cost-based backtracking algorithm to find the optimized graphs. To save the engineering efforts for emerging new operators, TASO (Jia et al. 2019a) takes operator definitions and specifications, then automatically generates and verifies graph substitutions, enabling a much larger search space than manually designed ones. It is non-trivial to find the graphs with optimal execution time on hardware in this exponential search space. OCGGS (Fang et al. 2020) introduces partial orders to substitution sequence to prune the redundant sequences and reuses the valid subgraph matching from the previous step. TENSAT (Yang et al. 2021) relies on the equality saturation technique to divide the graph optimization into two phases, exploration and extraction, to reduce the complexity. Despite these advancements, these existing graph optimization methods (Jia et al. 2019a,b; Fang et al. 2020; Yang et al. 2021; Bai et al. 2021) focus on sequential kernel launch frameworks, which is of low efficiency in the inference stage, given a static neural network with fixed input shape. In other words, only *intra-operator parallelism* is considered while no *inter-operator parallelism*. Each kernel occupies all the on-chip resources exclusively during execution though most resources are idle.

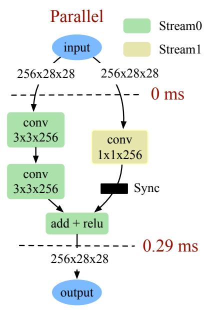
Modern GPUs possess many computation and memory resources that provide great potentials for high-performance computations and are ideal for intensive tensor computations in DNN models. With the rapid advances in GPUs (NVIDIA A100 reaches 19.5 TFLOPS of FP32 performance), the exclusive execution of a single DNN operator in the previous arts can no longer achieve sufficient resource utilization. In addition, there is a trend in the DL community to replace the linear chains of operators with multiple branches (Szegedy et al. 2015; Zhang et al. 2022) to capture diverse and deep information. The neural networks obtained by neural architecture search (NAS) (Zoph et al. 2018) show similar structures. Such networks can perform better on GPUs by leveraging inter-operator parallelism. IOS (Ding et al. 2021) divides the computation into different stages and uses a dynamic programming technique to find the optimized launch schedule for CNN models. Nimble (Kwon et al. 2020) is a framework supporting parallel kernel launch for the whole DNN model and leverages the ahead-of-time (AOT) scheduler to minimize hardware scheduling overhead. Parallel execution for multi-tenant DNN inference (Yu et al. 2021) is explored by



(a) PyTorch



(b) TASO



(c) Ours

Figure 1: Performance of different methods of the same DNN task on NVIDIA GeForce RTX 2080Ti. PyTorch runs the computation graph sequentially and gets the worst latency 0.40 ms. TASO (Jia et al. 2019a) optimizes the computation graph aggressively while still launching the computation graph sequentially. Our method fuses the element-wise operators and keeps the parallel branches while using multiple streams, achieving the best latency of 0.29 ms.

combining multi-stream and fine-grained scheduling in the runtime stage. These advancements demonstrate the superiority of the parallel paradigm in many scenarios.

Despite the successes of the inter-operator parallelism on GPUs, existing graph optimization methods are limited by their unawareness of the runtime system. The optimization is only guided by a naive performance estimation for graphs without runtime feedback. The cost of a node in the graph is the execution time of the corresponding operator on GPUs, and the cost of the whole graph is the total costs of all the nodes in it. Such metric ignores parallel kernel launch scenario and leads the optimization in the wrong direction. For example, given the DNN task in Figure 1a, PyTorch emits each operator to a single stream (PyTorch 2022) and executes them sequentially. TASO (Jia et al. 2019a) enlarges the kernel size from 1×1 to 3×3 by padding with zeros, then fuses the two convolutions to generate the functionally equivalent graph in Figure 1b. It has a better cost in terms of the sum of the execution time of operators, but it breaks the inter-operator parallelism. Therefore, the actual deployment performance is unsatisfying. Our method fuses the element-wise operators but keeps the parallel branches. Each branch is assigned to an independent GPU stream (CUDA 2022), with proper synchronization inserted to satisfy data dependency. It achieves the best performance, as shown in Figure 1c.

To mitigate the above issues, we present AutoGraph to optimize computation graphs for parallel kernel launch frameworks by a unified approach that uses both customized cost function and accurate runtime information. The major contributions of this paper are: (**i**) we propose a mixed critical path cost for fast estimation, which reflects the computation graph performance on the parallel kernel launch framework. (**ii**) A novel dynamic programming algorithm, combined with backtracking search, is proposed to find promising candidates for on-board measurements. (**iii**) We introduce accurate runtime information in the optimization flow. The latencies of selected candidates based on GPU Multi-Stream launched with CUDA graph determine the convergence of the optimization. (**iv**) We evaluate AutoGraph on representative DNN models. The results demonstrate that AutoGraph outperforms existing graph optimization methods with speedup ranging from 1.04× to 3.47×. Moreover, AutoGraph achieves 1.06× to 1.26× speedup over state-of-the-art parallel kernel launch frameworks.

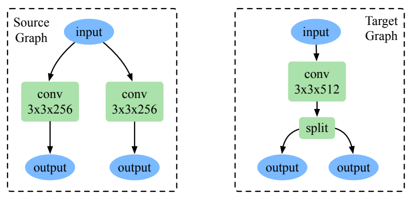


Figure 2: A typical substitution rule r.

Preliminaries

Computation Graph Optimization. A DNN model is defined by a directed acyclic computation graph $G = (V, E)$, where V represents the set of nodes, E is the set of edges. An edge $(u, v) \in E$ represents a tensor that is the output of node u and the input of node v. Following prior arts (Jia et al. 2019b; Fang et al. 2020), we introduce the concept of the graph substitution rule which defines how to replace a subgraph with a functionally equivalent one. A substitution rule contains two templates, the source graph and the target graph. For any valid inputs, they output the same results. Following the one-to-one matching between the input and output nodes, the source graph can be replaced by the target graph without any influence on the computation results. We denote a graph substitution rule as r. A typical substitution rule is shown in Figure 2.

To apply the rule r to a computation graph G, we have to find a subgraph of G that can match up with the source graph in r. To guarantee such substitution is valid, each node in the source graph can only be mapped to a node with the same type. Additional constraints on operators can be included to confine the mapping. For example, the nodes of the source graph in Figure 2 can only be mapped to convolution operators with same padding, stride and kernel size. After finding such mapping, we can replace the subgraph of G with the target graph in r, and the newly generated computation graph will compute the same results as G. Different substitution rules can be performed repeatedly to form complex graph optimization, as shown in Figure 1.

Parallel Kernel Launch. GPUs support parallel kernel execution by using multiple streams (CUDA 2022). A single stream is a sequence of kernels that execute in order, while the multiple streams can execute their kernels in parallel with higher utilization of memory and computation resources. When launching DNN models in multiple streams, explicit synchronizations across streams are needed to satisfy the data dependencies of some operators since operations from different streams can execute in any relative order. CUDA Graph (CUDA 2022) is the ideal multi-stream

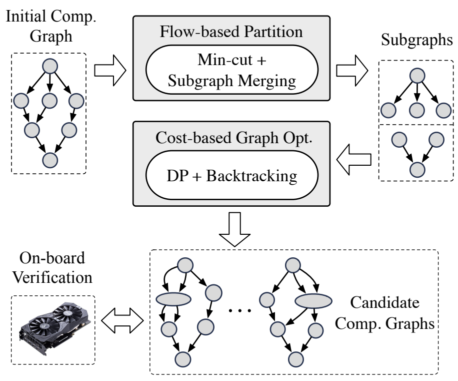


Figure 3: The brief flow of AutoGraph framework.

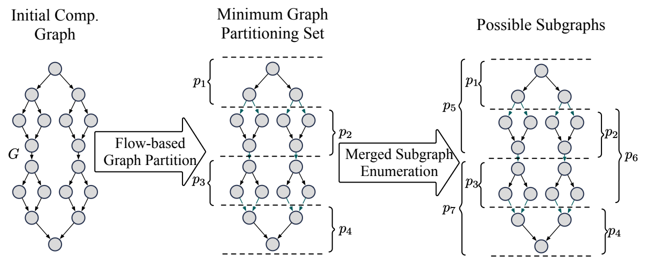

Initial Comp. Graph
Minimum Graph Partitioning Set
Possible Subgraphs
G
Flow-based Graph Partition
Merged Subgraph Enumeration
p1
p2
p3
p4
p5
p6
p7

Figure 4: A flow-based graph partition method is recursively applied to graph G until the size of each subgraph is less than the lower threshold $Size_L$. Then we scan the minimum partitioning set P_m and merge the adjacent subgraphs to form new subgraphs with a size less than the upper threshold $Size_U$, which is also performed recursively.

scheduling method on GPU to minimize the scheduling overhead (*e.g.*, kernel initialization and context switching). It is a record of the tasks that a stream and its dependent streams perform. CUDA Graph only needs to be defined ahead of time and then can be launched any number of times fast and efficiently, which meets the requirements of DNN model deployment seamlessly.

Cost Measurement. There are two types of costs in our context, one is estimated by inference frameworks in the unit of an operator, and the other one is the end-to-end on-board performance. For a computation graph, the actual execution latency on GPU is denoted by C_L, and the estimated performance is denoted by C_E. Ideally, C_E should align with C_L. Existing techniques (Jia et al. 2019a,b; Fang et al. 2020; Yang et al. 2021) compute C_E by summing up the costs of nodes in the graph. The cost of a node is the execution time of the operator (with specific input shape and parameters) on the hardware. This naive cost estimation method degrades the performance of graph optimization algorithms. A more accurate cost function is needed for the parallel kernel launch scenario.

Problem Formulation. Given an initial computation graph G, a set of substitution rules $R = \{r_1, r_2, \dots, r_m\}$, and the latency cost measurement $\texttt{Measure}(\cdot)$, the problem is to find an optimized graph G_{opt} with minimal latency cost $C_{L_{opt}}$ by applying a sequence of substitution rule r in R.

AutoGraph

The brief flow of our framework is shown in Figure 3. First, the initial computation graph is divided into multiple subgraphs to enable fast optimization. Then, a dynamic programming algorithm, combined with backtracking search, is adopted to explore the graph optimization solutions. Candidate graphs with low customized cost are selected for on-board measurement, and the runtime feedback is utilized to guide the optimization.

Flow-based Graph Partition

Most modern DNN models are too large for direct computation graph optimization, and numerous substitution rules and optimization iterations worsen the situation. Therefore, an effective graph partitioning strategy, which can reduce the search space while maximizing optimization opportunities, is highly desired. We first use a flow-based graph partition method to divide a computation graph into independent subgraphs. Then, possible merged subgraphs are enumerated to increase the optimization opportunities. Figure 4 illustrates the process.

Splitting a graph into two disjoint parts disables the graph substitutions spanning them because the substitution rules can not be applied to either side. Therefore, we aim to preserve most graph substitutions when partitioning the initial graph. For each node $v \in V$, we define its capacity $cap(v)$ as the number of different mappings that map the node to a node in the source graph of a substitution rule r. Our objective is to find the node set V_{cut} with minimal capacity sum to split the graph G. To find such node set, we construct a flow network G_n from the graph G to enable the network flow algorithm. The idea is to use two separate nodes and an edge with weight $cap(v)$ to denote each single node from the original graph. The details of the network construction can be found in the appendix. Finding the node set V_{cut} in G is equivalent to finding the minimum cut in G_n. The Boykov-Kolmogorov algorithm (Boykov and Kolmogorov 2004) is used to find the minimum cut. The entire computation graph G is recursively divided into independent subgraphs smaller than a threshold $Size_L$, which form the minimum partitioning set $P_m = \{p_1, p_2, \dots, p_n\}$, as shown in Figure 4.

We notice that the node capacity keeps changing during the graph optimization process. However, it is unrealistic to frequently update the partitioning result for the entire graph during the search. Regarding this issue, possible merged subgraphs are enumerated to increase optimization opportunities. After the flow-based partition, the minimum partitioning set P_m is scanned. If the size of two adjacent subgraphs (p_i, p_{i+1}) do not exceed an upper threshold $Size_U$, we merge them to form a new subgraph and repeat this process in a recursive manner as shown in Figure 4. Each subgraph now is suitable for subsequent optimization. We denote the set of the beginning subgraphs containing input nodes as P_b and collect all the subgraphs' adjacency information for further reference.

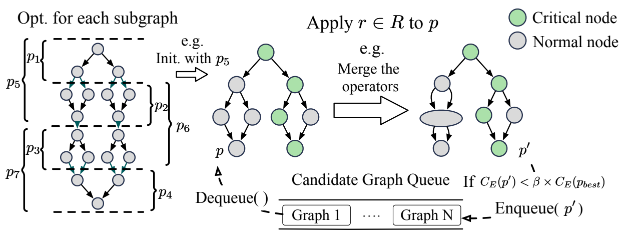


Figure 5: An illustration of our backtracking search. Our method takes each subgraph as the input graph for fast optimization. Available substitution rules are applied to generate new graphs. Nodes in the critical path are assigned with higher weights when calculating costs. Graphs whose mixed critical path cost is β times worse than the best current value are stopped from further exploration.

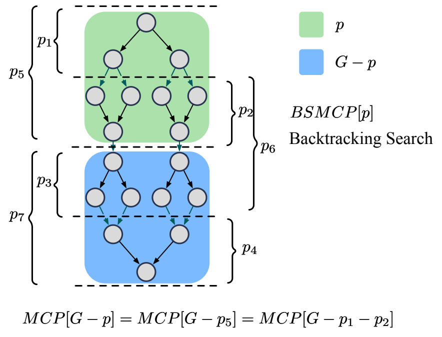


Figure 6: An illustration of a transition state in our dynamic programming + backtracking search algorithm. At current state $p = p_5$, we directly retrieve the solution for the subproblem $MCP[G - p_5]$ cached in previous steps by $MCP[G - p_1 - p_2]$. Optimization for current subgraph p is performed via our backtracking search.

Cost-based Graph Optimization

We first introduce the backtracking search method, which is utilized to optimize each subgraph. The optimization solutions for the entire graph are explored by the proposed dynamic programming algorithm. Our customized cost, considering inter-operator parallelism, serves as the selection criterion.

Backtracking Search via Mixed Critical Path Cost. Equivalent computation graphs with different runtime performances can be generated by applying suitable substitution rules to a computation graph. Exhaustively enumerating all the possible results is of low efficiency, and we use a backtracking search method to find optimized graphs instead. It relies on the cost of each graph to prune the search space. Fast and accurate estimation is needed to perform the optimization efficiently.

As shown in Figure 1, the naive cost estimation in existing works ignores the inter-operator parallelism and misguides the optimization for the parallel kernel launch framework. We propose the mixed critical path cost in Equation (1) as the selection criterion. We define V_c as the set of nodes on the critical path. And the critical path is the longest path in terms of cost sum. Because the computation graph is a DAG, the critical path can be extracted efficiently via topological sort. Since the operators on the critical path must be executed in order, the critical path cost is the lower bound for the computation graph execution. But we notice that there are numerous graphs with the same critical path cost, while the costs of the rest routes still need to be considered to make a further selection. Thus, we consider both the critical path cost and the total cost. And we use a hyperparameter α to control the critical path cost weight. As is shown, when calculating the total cost, the critical path cost is already collected once.

$$\begin{aligned} C_E &= \alpha \sum_{v \in V_c} cost(v) + \sum_{v \in V} cost(v) \\ &= (1+\alpha) \sum_{v \in V_c} cost(v) + \sum_{v \in V - V_c} cost(v). \end{aligned} \tag{1}$$

Figure 5 illustrates our backtracking search. Thanks to our graph partition procedure, we can perform the backtracking search on each subgraph to enable fast graph optimization. We take each subgraph as the input graph. In each search round, we find all the possible mappings and generate corresponding new candidate graphs via the rules. Candidate graphs of mixed critical path cost within β times of the current best value are collected for the next search round. Complex graph optimizations composed of multiple substitutions can be found automatically. We set $\beta = 1.1$ to allow temporarily increasing the cost in intermediate steps, encouraging more exploration. The extra critical path cost prevents the optimization from forming large kernels in the critical path, preserving the inter-operator parallelism. The experimental results show its effectiveness. Our backtracking search is flexible. It can return the best graph in terms of the mixed critical path cost or a batch of promising candidates for further verification.

DP-based Optimization Solution Search. Among all the possible subgraphs, we need to perform graph substitutions on an optimal graph partitioning sequence that leads to the best performance for the entire graph. A valid graph partitioning sequence spans all parts of the graph G. After optimizing each subgraph in the sequence, we can stitch them to construct the entire optimized computation graph. The minimum partitioning set P_m is one of them, but it is not the optimal choice in most cases. We observe that different graph partitioning sequences share the same sub-sequence, and thus we design a dynamic programming algorithm to find the optimization solution efficiently.

To find the optimal solution for a computation graph G, we first divide it into $G - p$ and p. p is the subgraph generated in the graph partition process. And p is always chosen from the beginning part of current G, which can have multiple choices. Specifically, if G is the initial graph, p is chosen from the beginning subgraph set P_b. Otherwise, p is chosen from the subgraphs adjacent to the previous choice that induces the current divided graph. For simplicity, we denote current available subgraph set as P_c. We enumerate all the $p \in P_c$, optimize p with the backtracking search method and reduce the original problem to a sub-problem, finding

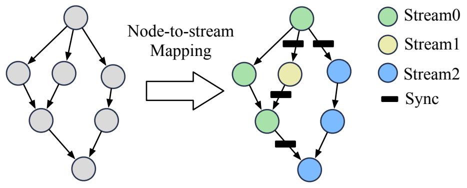


Figure 7: An example of stream assignment.

the optimal solution for $G - p$. We denote $MCP[G]$ as the mixed critical path cost of the optimal solution for G, and denote $BSMCP[p]$ as the mixed critical path cost returned by applying backtracking search on p. The above idea can be described by the recursive formulation in Equation (2).

$$MCP[G] = \min_{p \in P_c} (MCP[G-p] + BSMCP[p]). \tag{2}$$

Figure 6 shows a transition state in our dynamic programming + backtracking search algorithm. There are seven different subgraphs from Figure 4. At start, the subgraphs in the beginning subgraph set $P_b = \{p_1, p_5\}$ are enumerated, and p_1 is explored by the backtracking search to perform graph substitutions first. p_2 and p_6 are the subgraphs adjacent to p_1, and they will be explored next. When exploring p_2, the original problem $MCP[G]$ is reduced to $MCP[G-p_1-p_2]$, which is solved in the same manner. And the mixed critical path cost of the optimal solution for $G-p_1-p_2$ is stored in a hash table. When we reach the current state with $p = p_5$, the backtracking search is applied to optimize the current subgraph, and the problem is reduced to $MCP[G-p_5]$ that is the same as $MCP[G-p_1-p_2]$. Thus, we can directly obtain the result from the hash table to avoid redundant computations. After running the dynamic programming algorithm, we can construct the complete optimization solution based on the hash table, which achieves the best mixed critical path cost on graph G. The pseudocode of our DP-based method is provided in the appendix.

On-board Verification

Previous computation graph optimization methods terminate before on-board measurement, lacking actual runtime information in the optimization process. This downgrades the performance of the final optimization result, and meanwhile the complex parallel kernel launch scenario aggravates such a problem. Therefore, we propose to introduce runtime information to guide the optimization. To acquire such information, we implement the parallel kernel launch runtime system in our framework.

Specifically, we leverage GPU multi-stream to exploit the inter-operator parallelism of the computation graph. The operator nodes on different branches are assigned to different streams, and proper synchronization events are inserted into the streams to guarantee the data dependencies. Figure 7 shows an example. An ideal stream assignment should maximize the operator concurrency while minimizing the number of synchronizations. We adopt the maximum matching algorithm in Nimble (Kwon et al. 2020) to find such assignment. It is a fast analytical approach, which can be embedded into our framework seamlessly. Based on the node-to-stream mapping, CUDA Graph is used to pre-run the computation graph once and trace invocations of GPU kernels and allocations of GPU memory. In the inference stage, CUDA Graph replays the recorded parallel kernel launch schedule without the extra overhead. The computation graph is launched in this way to achieve its best deployment performance, and the end-to-end latency is treated as the golden metric for guiding the optimization flow.

Since it is infeasible to measure every new graph on the runtime system, we only sample the top-k candidates for on-board verification each time. Candidates with improved performance are selected for next optimization iteration.

Overall Optimization Flow

Combining all the aforementioned methodologies our optimization flow are shown in Algorithm 1. All the selected graphs are maintained in a priority queue Q and popped in increasing order of end-to-end latency. For each popped graph, independent subgraphs are found via our flow-based graph partition method and stored in a set P. Then the optimal optimization solution is found by our dynamic programming algorithm to generate new candidate graphs, and candidates with top-k mixed critical path cost are collected in a heap H. These candidates are measured on the parallel kernel launch runtime system, and candidates with improved performance are inserted to Q for further exploration. Finally, the optimized computation graph G_{opt} is returned.

Algorithm 1: Optimization Flow

```
Input: Initial computation graph $G$, a set of substitution rules $R$,
    latency cost measurement Measure$(\cdot)$.
Output: An optimized graph $G_{opt}$.
 1: $G_{opt} \leftarrow G, C_{L_{opt}} \leftarrow$ Measure$(G)$;
 2: $Q \leftarrow \{G_{opt}, C_{L_{opt}}\}$;            ▷ graph priority queue Q
 3: while $Q \neq \varnothing$ do
 4:     $G_{cand}, C_{L_{cand}} = Q$.pop();
 5:     if $C_{L_{cand}} \leq C_{L_{opt}}$ then
 6:         $G_{opt} \leftarrow G_{cand}, C_{L_{opt}} \leftarrow C_{L_{cand}}$;
 7:     GraphPartition$(G_{cand}, R, P)$;        ▷ subgraph set $P$
 8:     GraphOptimizatioin$(G_{cand}, R, P, H)$; ▷ candidate
    heap $H$
 9:     for all $G_{new} \in H$ do
10:         $C_{L_{new}} \leftarrow$ Measure$(G_{new})$;
11:         if $C_{L_{new}} \leq C_{L_{opt}}$ then
12:             $Q$.insert$(G_{new}, C_{L_{new}})$;
13: return $G_{opt}$;
```

Experiments

Experimental Settings

We conduct all the experiments on an Intel(R) Xeon(R) Silver 4114 CPU@ 2.20GHz. The hardware platform is an NVIDIA GeForce RTX 2080Ti GPU with CUDA 11.0, cuDNN 8.0.5, and PyTorch 1.7. Seven modern DNNs are benchmarked in the experiments, and the details of the models are shown in Table 1. Inception-v3 (Szegedy et al. 2016)

Table 1: DNN Models Used in Our Experiments.

Type	Name	block#	input shape
CNN	Inception-v3	11	[1, 3, 299, 299]
	ResNet-50	16	[1, 3, 224, 224]
	ResNeXt-50	16	[1, 3, 224, 224]
	NasNet-A	18	[1, 3, 224, 224]
	NasNet-Mobile	12	[1, 3, 224, 224]
RNN	RNNTC-SRU	10	[1 × 10, 1024]
Transformer	BERT	8	[16 × 64, 1024]

Table 2: Model inference latency results (ms).

Model	JIT	TASO+JIT	IOS	Nimble	TASO+Nimble	Ours
Inception-v3	8.839	7.819	3.788	3.174	2.928	**2.799**
ResNet-50	4.566	4.554	3.284	2.144	1.988	**1.905**
ResNeXt-50	7.540	7.369	3.056	7.708	5.933	**2.892**
NasNet-A	13.891	10.843	9.583	6.483	13.086	**5.850**
NasNet-Mobile	10.155	8.085	3.821	2.320	6.540	**1.883**
RNNTC-SRU	1.496	1.307	-	0.486	**0.387**	**0.387**
BERT	11.011	9.026	-	6.923	6.473	**6.240**

and ResNet-50 (He et al. 2016) are widely used networks for image classification. ResNeXt-50 (Xie et al. 2017) introduces a new grouped convolution operator to replace the residual block and improves the model accuracy. NasNet-A (Zoph et al. 2018) and NasNet-Mobile (Zoph et al. 2018) are the representative CNN models with complicated model structures discovered by neural architecture search. RNNTC (Lei et al. 2017), a model for natural language processing tasks, is also tested. It is a sequence-to-sequence RNN model built on the simple recurrent unit (SRU) (Lei et al. 2017). BERT (Devlin et al. 2018), *i.e.*, Bidirectional Encoder Representation from Transformers, is a powerful model which stacks the complicated transformers and has obtained state-of-the-art results on many tasks.

End-to-end model inference latency is adopted as the performance metric. We compare our method with both sequential and parallel kernel launch frameworks. All the frameworks are based on cuDNN libraries for fair comparisons. PyTorch (Paszke et al. 2019) is the most widely adopted development tool and the representative sequential kernel launch framework. We use PyTorch with Just-In-Time compilation (PyTorch-JIT 2022) to enhance the inference performance. Nimble (Kwon et al. 2020) and IOS (Ding et al. 2021) are the state-of-the-art parallel kernel launch frameworks, which exploit inter-operator parallelism extensively to accelerate model inference. To compare with existing graph optimization methods, we transform the optimized computation graphs found by TASO (Jia et al. 2019a) to formats accepted by PyTorch-JIT and Nimble to obtain the end-to-end latencies. They are also included as baselines.

In our method, the 157 substitution rules from TASO (Jia et al. 2019a) are used as the substitution rule set R. The correctness of these rules has been verified. We set $\alpha = 0.25$ as the weight of critical path cost and set $\beta = 1.1$ for the backtracking search. The lower threshold $Size_L$ and the upper threshold $Size_U$ are set as 40 and 120, respectively. In each iteration, the top-20 candidate graphs are collected for on-board verification. The optimization results are not sensitive to most of the hyperparameters. Detailed ablation studies on the hyperparameter settings are provided in the appendix.

End-to-end Performance

We compare the end-to-end inference performance of the benchmark models between our method and the baselines. We adopt the same inference setting with (Jia et al. 2019a; Kwon et al. 2020; Ding et al. 2021) for a fair comparison.

The experimental results are shown in Table 2. JIT in Table 2 represents PyTorch with JIT optimization. We conduct each model inference 50 times and report the average performance. Our method consistently obtains the best results on all the benchmark models, which demonstrates the significant advantages of our framework over the baselines. Compared with PyTorch-JIT (Paszke et al. 2019) and PyTorch-JIT with computation graphs optimized by TASO (Jia et al. 2019a), our method achieves huge speedup, demonstrating the necessity to make use of inter-operator parallelism. In the parallel kernel launch scenario, our method outperforms the state-of-the-art frameworks IOS (Ding et al. 2021) and Nimble (Kwon et al. 2020) on all benchmark models with speedup ranging from 1.06$\times$ to 1.26$\times$, which proves that exploiting graph optimization and inter-operator parallelism simultaneously can further improve the inference performance. IOS (Ding et al. 2021) only supports CNN model execution, therefore the latency results of IOS for RNNTC-SRU and BERT are unavailable. Compared with applying TASO (Jia et al. 2019a) to Nimble (Kwon et al. 2020), our method achieves speedup ranging from 1.04$\times$ to 3.47$\times$ on the benchmark models. This significant performance gain comes from our customized cost and accurate runtime information, which helps our framework find the most suitable graph optimizations for the parallel kernel launch runtime system.

Case Studies for Graph Optimizations

To understand how our method finds better optimized computation graphs compared with baselines, we exemplify the NasNet and ResNeXt in detail.

Figure 8 illustrates the different optimization choices between our method and TASO on NasNet cell. Due to the naive cost function, TASO accepts any graph substitution rule which reduces the total cost of all the operators. As a result, TASO applies all the three substitutions for each cell in the network and transforms the multi-branch structures into a single link with large kernels. The reduction of kernel launch overhead fails to offset the extra computation workload induced by these fused operators. These large kernels accumulate in the critical path and cause significant performance degradation on parallel kernel launch runtime system. Therefore, TASO fails to find computation graphs with improved performance for NasNet-A and NasNet-Mobile on Nimble, only achieving poor performance of 13.086 ms and 6.540 ms, respectively. With the help of the mixed critical path cost and the advanced DP-based optimization method, our method rejects the two substitutions which break the

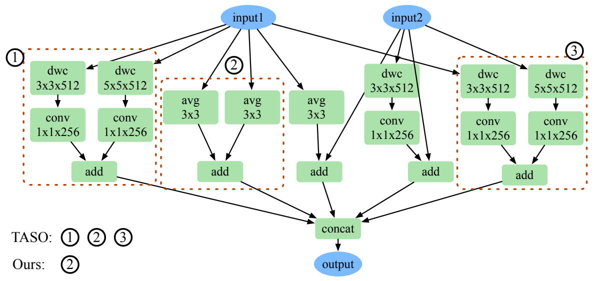


Figure 8: Different graph optimization choices on NasNet cell. dwc and avg refer to depth-wise convolution and average pooling. There are three valid graph substitutions. In ① and ③, the two dwc and two conv can be fused. In ②, the two avg followed by add can be replaced by a single operation. TASO applies all the three substitutions and forms large kernels in the critical path. Our method only chooses ② and keeps the parallel branches, which reduces the computation and preserves the inter-operator parallelism.

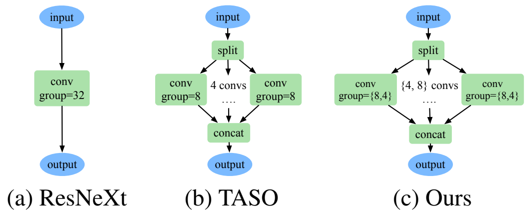


Figure 9: Different optimizations on ResNeXt block. TASO changes the group number from 32 to 8 for all the convolutions. Our method sets some convolutions with group number 8 and some with 4, guided by the runtime information.

inter-operator parallelism and only merges the two average pooling operators to reduce computation. Therefore, our method keeps most parallel structures in NasNet-A and NasNet-Mobile, which leads to the best performance of 5.850 ms and 1.883 ms.

Figure 9 shows the different computation graphs found by TASO and our method for ResNeXt block. ResNeXt replaces the residual block in ResNet with 32 branches of small convolutions, and the implementation is equivalent to a group convolution with group number 32. TASO changes the group number from 32 to 8 for all the blocks in ResNeXt based on the estimated cost and achieves improved inference performance of 5.933 ms on Nimble. Our mixed critical path cost tends to set the group number as 4 for all the blocks to increase inter-operator parallelism. However, with the DP-based optimization and the on-board verification, our method finds a mixture of group number 8 and 4 can maintain a balanced resource usage and improve the overall efficiency. Our method sets the blocks in ResNeXt with group number 8 and 4 alternately, which achieves the best inference latency of 2.892 ms.

Ablation Studies

To validate the graph optimization and the dynamic programming method, we compare the results of different settings where graph optimization or dynamic programming is disabled, as shown in Figure 10. It demonstrates that we can improve the inference latency by performing suitable graph substitutions. The optimal optimization solution found by our DP-based method leads to optimized graphs with better performance.

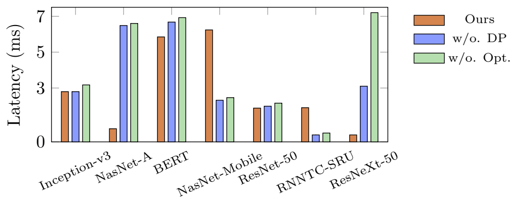


Figure 10: Performance comparisons between different settings. "w/o. Opt." means directly measuring the initial computation graph. "w/o. DP" means directly using the minimum partitioning set without our DP-based method.

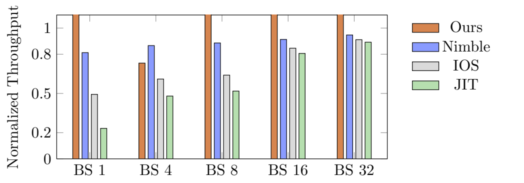


Figure 11: The normalized throughput comparisons of different frameworks on various batch sizes for NasNet-Mobile.

We optimize NasNet-Mobile on our framework with different batch sizes. Figure 11 shows our method consistently outperforms all the baselines, which demonstrates the generality of our method. Although a larger batch size provides more intra-operator parallelism, we can still exploit inter-operator parallelism and graph optimization to further improve the inference performance.

Conclusion

Existing computation graph optimization methods lean on sequential GPU kernel execution, which fails to utilize inter-operator parallelism and thus impairs system capability within a parallel kernel launch framework. We identify the potential of combining graph optimization and inter-operator parallelism to boost the inference performance and propose a unified approach. The proposed framework, AutoGraph, can optimize DNN computation graphs with a novel dynamic programming + backtracking search algorithm using customized performance estimation and accurate runtime information. We show that AutoGraph can achieve up to $3.47\times$ performance improvement on various widely used DNNs compared with previous arts. Moreover, AutoGraph outperforms state-of-the-art parallel kernel launch frameworks by up to $1.26\times$.

Acknowledgment

This work is partially supported by SmartMore and ITF Partnership Research Programme (No. PRP/65/20FX).

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 265–283.

Bai, Y.; Yao, X.; Sun, Q.; and Yu, B. 2021. AutoGTCO: Graph and Tensor Co-Optimize for Image Recognition with Transformers on GPU. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

Boykov, Y.; and Kolmogorov, V. 2004. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 26(9): 1124–1137.

Chen, T.; Li, M.; Li, Y.; Lin, M.; Wang, N.; Wang, M.; Xiao, T.; Xu, B.; Zhang, C.; and Zhang, Z. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274.

Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 578–594.

CUDA. 2022. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Devlin, J.; Chang, M.; Lee, K.; and Toutanova, K. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805.

Ding, Y.; Zhu, L.; Jia, Z.; Pekhimenko, G.; and Han, S. 2021. IOS: Inter-Operator Scheduler for CNN Acceleration. *Proceedings of Machine Learning and Systems (MLSys)*, 3.

Fang, J.; Shen, Y.; Wang, Y.; and Chen, L. 2020. Optimizing DNN computation graph using graph substitutions. *International Conference on Very Large Databases (VLDB)*, 13(12): 2734–2746.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 770–778.

Jia, Z.; Padon, O.; Thomas, J.; Warszawski, T.; Zaharia, M.; and Aiken, A. 2019a. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, 47–62.

Jia, Z.; Thomas, J.; Warszawski, T.; Gao, M.; Zaharia, M.; and Aiken, A. 2019b. Optimizing DNN computation with relaxed graph substitutions. *Proceedings of Machine Learning and Systems (MLSys)*, 1: 27–39.

Kwon, W.; Yu, G.-I.; Jeong, E.; and Chun, B.-G. 2020. Nimble: Lightweight and Parallel GPU Task Scheduling for Deep Learning. *Conference on Neural Information Processing Systems (NeurIPS)*, 33.

Lei, T.; Zhang, Y.; Wang, S. I.; Dai, H.; and Artzi, Y. 2017. Simple Recurrent Units for Highly Parallelizable Recurrence. arXiv:1709.02755.

Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Conference on Neural Information Processing Systems (NeurIPS)*, volume 32, 8026–8037.

PyTorch. 2022. PyTorch CUDA Semantics. https://pytorch.org/docs/stable/notes/cuda.html.

PyTorch-JIT. 2022. TorchScript. https://pytorch.org/docs/stable/jit.html.

Redmon, J.; Divvala, S.; Girshick, R.; and Farhadi, A. 2016. You only look once: Unified, real-time object detection. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 779–788.

Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; and Rabinovich, A. 2015. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1–9.

Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; and Wojna, Z. 2016. Rethinking the inception architecture for computer vision. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2818–2826.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Conference on Neural Information Processing Systems (NeurIPS)*, 5998–6008.

Xie, S.; Girshick, R.; Dollár, P.; Tu, Z.; and He, K. 2017. Aggregated residual transformations for deep neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1492–1500.

Xing, J.; Wang, L.; Zhang, S.; Chen, J.; Chen, A.; and Zhu, Y. 2022. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. *Proceedings of Machine Learning and Systems (MLSys)*, 4: 204–216.

Yang, Y.; Phothilimthana, P.; Wang, Y.; Willsey, M.; Roy, S.; and Pienaar, J. 2021. Equality saturation for tensor graph superoptimization. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 3.

Yu, F.; Bray, S.; Wang, D.; Shangguan, L.; Tang, X.; Liu, C.; and Chen, X. 2021. Automated Runtime-Aware Scheduling for Multi-Tenant DNN Inference on GPU. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 1–9.

Zhang, H.; Wu, C.; Zhang, Z.; Zhu, Y.; Lin, H.; Zhang, Z.; Sun, Y.; He, T.; Mueller, J.; Manmatha, R.; Li, M.; and Smola, A. 2022. ResNeSt: Split-Attention Networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2736–2746.

Zhao, J.; Gao, X.; Xia, R.; Zhang, Z.; Chen, D.; Chen, L.; Zhang, R.; Geng, Z.; Cheng, B.; and Jin, X. 2022. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization. *Proceedings of Machine Learning and Systems (MLSys)*, 4: 1–19.

Zoph, B.; Vasudevan, V.; Shlens, J.; and Le, Q. V. 2018. Learning transferable architectures for scalable image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 8697–8710.