Collage: Seamless Integration of Deep Learning Backends with Automatic Placement

Byungsoo Jeon*
byungsoj@cs.cmu.edu
Carnegie Mellon University

Sunghyun Park*†
spark@octoml.ai
OctoML

Peiyuan Liao
peiyuanl@andrew.cmu.edu
Carnegie Mellon University
Praxis Pioneering

Sheng Xu†
jackyxu1997@gmail.com
Amazon Web Services

Tianqi Chen
tqchen@cmu.edu
Carnegie Mellon University
OctoML

Zhihao Jia
zhihao@cmu.edu
Carnegie Mellon University

ABSTRACT

The strong demand for efficient and performant deployment of Deep Learning (DL) applications prompts the rapid development of a rich DL ecosystem. To keep up with this fast advancement, it is crucial for modern DL frameworks to efficiently integrate a variety of optimized tensor algebra libraries and runtimes as their **backends** and generate the fastest possible executable using these backends. However, current DL frameworks require significant manual effort and expertise to integrate every new backend while failing to unleash its full potential. Given the fast-evolving nature of the DL ecosystem, this manual approach often slows down continuous innovations across different layers; it prevents hardware vendors from the fast deployment of their cutting-edge libraries, DL framework developers must repeatedly adjust their hand-coded rules to accommodate new versions of libraries, and machine learning practitioners need to wait for the integration of new technologies and often encounter unsatisfactory performance.

In this paper, we propose *Collage*, a DL framework that offers seamless integration of DL backends. *Collage* provides an expressive backend registration interface that allows users to precisely specify the capability of various backends. By leveraging the specifications of available backends, *Collage* automatically searches for an optimized backend placement strategy for a given workload and execution environment. Our evaluation shows that *Collage* outperforms the best existing framework for each hardware by 1.26×, 1.43×, 1.40× on average on NVIDIA's RTX 2070 GPU, V100 GPU, and Intel's Xeon 8259CL CPU, respectively. Collage has been open-sourced [1] and deployed in Apache TVM.

*Both authors contributed equally to this research.
†The work was done during their degree programs.
[1]https://github.com/cmu-catalyst/collage

PACT '22, October 10–12, 2022, Chicago, IL, USA

ACM ISBN 978-1-4503-9868-8/22/10.
https://doi.org/10.1145/3559009.3569651

CCS CONCEPTS

• **Software and its engineering → Compilers**.

KEYWORDS

Machine Learning System, Compiler, Software Library

ACM Reference Format:
Byungsoo Jeon, Sunghyun Park, Peiyuan Liao, Sheng Xu, Tianqi Chen, and Zhihao Jia. 2022. Collage: Seamless Integration of Deep Learning Backends with Automatic Placement. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '22), October 10–12, 2022, Chicago, IL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3559009.3569651

1 INTRODUCTION

Due to the explosive popularity of Deep Learning (DL) applications, there are tremendous demands for performant and efficient software/hardware stacks for DL computations. These strong demands have driven both industry and academia to invest a significant amount of effort in developing various hardware devices [1, 6, 40], software libraries [3, 4, 7, 23, 43], compilers [12, 16, 17, 19, 20, 44, 47, 49, 56, 59, 70], and DL frameworks [9, 10, 24, 55, 58, 61]. Both the hardware and software stacks for DL have been diversified, resulting in a rich and fast-evolving ecosystem.

Within this ecosystem, today's DL frameworks can leverage a variety of optimized software libraries [23, 67] and runtimes [7, 43] as their **backends** [2] to deliver fast execution. Existing backends can be grouped into two categories based on their capabilities. First, *operator kernel libraries* [23, 43, 67] provide efficient low-level kernel API for individual DL operators (e.g., convolution). These libraries often support *operator fusion*, which combines multiple operators into a single kernel based on certain fusion rules (e.g., cuDNN fusion engine) [21, 23, 27, 48, 53, 73]. Second, *graph inference libraries* [3, 7] take an entire DL model as input and produce efficient run-time code. In addition to the optimizations that operator kernel libraries provide, the graph inference libraries also consider graph-level cross-kernel optimizations, such as memory optimizations [62].

There are strong demands for high-performance DL backends in both industry and academia. However, seamless integration of diverse and rapidly advancing DL backends requires addressing

[2]We define a *backend* as a kernel library or a runtime framework that takes DL workloads as inputs and provides an optimized low-level target code.

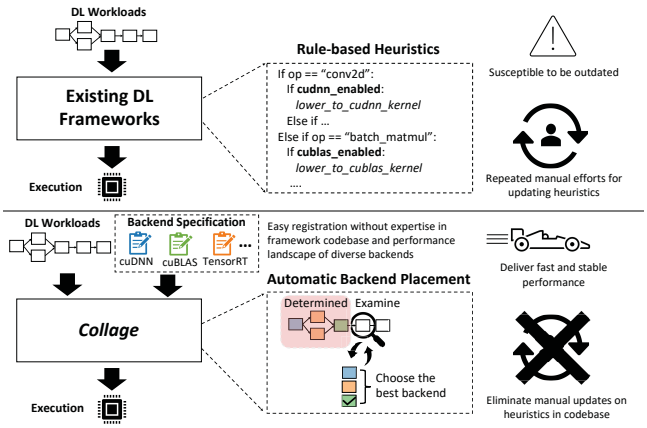


Figure 1: A comparison between existing DL frameworks and *Collage*. Existing frameworks (top) use rule-based heuristics to integrate different backends. In contrast, *Collage* provides an automatic search algorithm to find optimized placement of backends for a given hardware platform. New backends can be easily integrated into *Collage* through the backend registration interface.

two key challenges: (1) incorporating a wide variety of available backends with different programming models and performance characteristics, and (2) optimizing placement of backends to effectively assign DL computations to various backends by leveraging the performance advantages of each backend. We refer to this overall problem as **backend integration problem**.

To deal with the backend integration problem, existing DL frameworks [10, 55] rely on rule-based heuristics manually designed by experts (Figure 1). These heuristics often directly offload the entire workload to a single backend (e.g., TensorRT) whenever applicable. Otherwise, DL frameworks lower individual operators to different backends based on a fixed priority-based strategy; for example, in PyTorch, cuDNN has the highest priority for convolution, while cuBLAS is the first choice for matrix multiplication.

However, even for the same type of operators, the optimal backend varies depending on the hardware (e.g., different types of GPUs) and operator configuration (e.g., tensor shape, padding) as depicted in Figure 2. As a result, the hand-coded heuristics in current DL frameworks may leave substantial performance on the table. Besides, existing frameworks require significant expertise in both framework and performance landscape of diverse backends as developers need to directly modify the complex lowering heuristics (e.g., more than ten thousand lines of code in PyTorch) in a framework to introduce a new backend or reflect any backend updates. These handcrafted heuristics are hard to maintain and keep up with the rapid developments in backends. This is a major bottleneck for various machine learning personas, since the integration workflow requires repetitive manual efforts to accommodate new backends. This integration overhead hinders hardware vendors from deploying their cutting-edge libraries and delays machine learning practitioners from employing newest system-level supports.

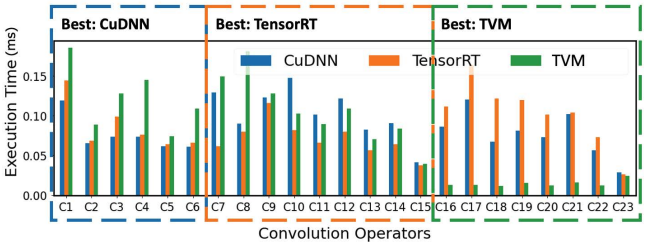


Figure 2: Performance of various convolutions (C#) with different configurations (e.g., input tensor shape, kernel size) in ResNext-50 on NVIDIA RTX 2070; Note that there is no single *backend* that is the best for all convolutions.

In this paper, we aim to design a system that can provide seamless backend integration workflow with high performance. Building such a solution requires addressing two key challenges. First, it is non-trivial to integrate diverse backends with different characteristics into a system while maintaining their full capabilities. Often times, backend capability is intricate to capture accurately since today's DL backends generally support sophisticated operator fusion with various constraints (e.g., fusing convolution ops with 3x3 kernel). Second, the search space of backend placement is extremely large, whose size grows exponentially in the number of operators in a DNN and the number of available backends. The search space is also highly irregular due to diverse backend capabilities and operator fusion patterns.

In *Collage*, we advocate for a new approach to tackling these challenges, as shown in the bottom of Figure 1. *Collage* contains two key components. First, to integrate diversified backends, *Collage* provides a descriptive *backend registration interface* to specify a backend's capability based on its supported operator type (e.g., conv), configurations (e.g., kernel size), and its fusion rule. This interface only requires basic understanding of our pattern language and backend capability in contrast to existing frameworks that require considerable expertise in both the performance landscape of varied backends and the coding skills for backend placement rules in existing frameworks. *Collage* allows easy backend registration for a new backend (e.g., 100 LoC for all possible operators) or a new operator pattern support (e.g., 1 LoC in most cases). Second, to efficiently optimize backend placement, *Collage* employs a *two-level optimization* to deal with unique chacteristics of two backend categories (i.e., operator kernel library and graph inference library). Our system automatically explores possible matches between an input computation graph and backend operator patterns to find optimized placements by taking available backends and an underlying hardware into consideration.

To sum up, *Collage* significantly lowers the bar in the current backend integration workflow by eliminating the need to modify the placement heuristic. With simple registration from users, *Collage* can immediately launch the automatic placement optimizer without any intricate manual consideration for the capability of new backend and its performance relation with other backends across different workloads and hardware architectures.

This paper makes the following contributions:

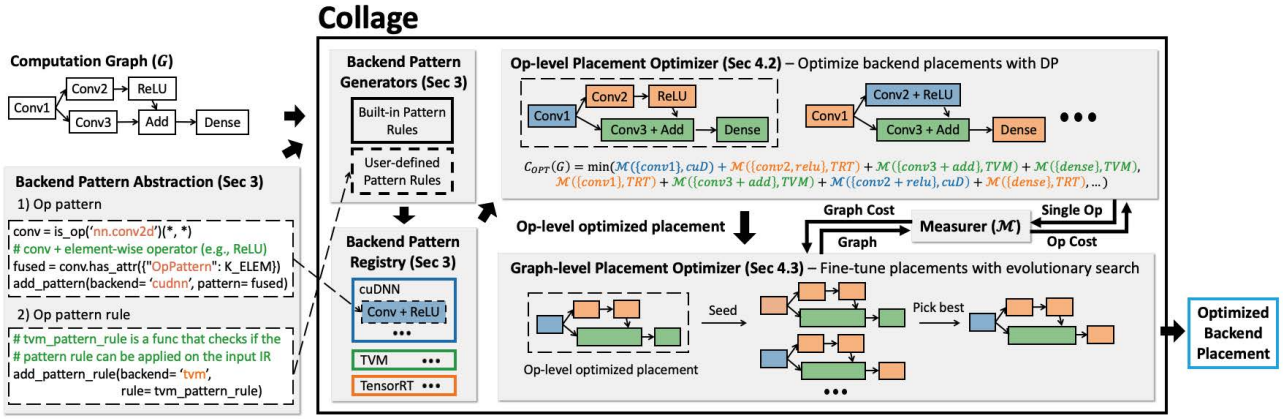


Figure 3: System overview of *Collage*. By using our backend specification interface, users can efficiently register diverse backend patterns supported by diverse backends. Then, with its two-level optimization process, *Collage* automatically optimizes backend placement for an underlying execution environment.

- We identify system and optimization challenges in integration of diversified DL backends and build *Collage* to tackle these challenges.
- We provide a pattern-based interface for quick registration of various backends and their updates with significantly less user efforts and expertise in performance landscape of varied backends and the placement heuristic in the framework codebase.
- We develop a two-level search method to automatically optimize placement of diverse backends for a given hardware.

Our evaluation shows that *Collage* stably outperforms existing DL frameworks across a variety of models and hardware architectures by effectively mix-using multiple backends with their own unique strengths. On average, *Collage* brings 1.26×, 1.43×, and 1.40× speedup on two different NVIDIA GPUs and an Intel CPU respectively, compared to the best framework for each hardware.

2 OVERVIEW

Figure 3 illustrates the overarching design of *Collage*, which takes a DNN model and the specifications of available backends as inputs, and optimizes backend placement for the underlying hardware. Note that *Collage* considers different sets of backends based on a given target environment (e.g., Intel CPU, NVIDIA GPU) and reflects performance characteristics of backends via the measurer component ($\mathcal{M}$). *Collage* consists of two key components.

Backend pattern abstraction. Existing backends provide a variety of programming models for performing DL computations. To decouple backend capability from the placement algorithm and eliminate the manual effort for backend integration, we introduce *backend pattern*, a new abstraction for capturing the capability of varied backends. Specifically, a backend pattern defines a set of operators and their possible fusion combinations (e.g., Conv+ReLU) that can be deployed on each backend. Based on this pattern abstraction, *Collage* provides a straightforward interface to register a backend and specify supported operator patterns.

Accurate specification is crucial to leverage the full capability of diverse backends. To achieve this goal, *Collage* offers two levels of abstraction. For simple patterns, *Collage* allows users to enumerate the supported operator patterns. However, this approach may not cover the full capability of backends with advanced operator fusion engines [7, 21, 23, 53]. To enable more flexible specification, *Collage* also allows users to bring their pattern rules that specify supported operator kinds and complex operator fusion rules. When those rules are provided, the *pattern generator* automatically identifies all legitimate operator fusion patterns on a given computation graph and adds them into the backend pattern registry. §3 provides details.

Backend placement optimizer. Once all available patterns are registered in the pattern registry, *Collage* uses a *two-level optimization* approach to discovering an optimized backend placement strategy for a given execution environment. As existing operator libraries offer operator-level point of view while graph inference libraries additionally apply cross-kernel optimizations, *Collage* takes two different optimization strategies to exploit their differences. First, the *op-level placement optimizer* explores promising candidates for individual operators, without considering cross-kernel optimizations. By adopting a Dynamic Programming (DP) algorithm, the op-level placement optimizer can efficiently find an optimized backend placement strategy within a minute. Second, the *graph-level placement optimizer* fine-tunes the optimized backend placement using evolutionary search [30]. This approach compensates for the missing opportunities from the op-level placement optimizer by examining the impact of cross-kernel optimizations. §4 discusses the two optimizers in detail.

3 BACKEND PATTERN ABSTRACTION

As an important component of DL ecosystem, there are diverse fast-evolving DL backends with different programming models and performance characteristics. Depending on their target hardware and design principles, each backend has its own unique strength and coverage. In addition, many backends support various complex operator fusion rules [7, 21, 21, 23, 53], which add significant complexity in their integration with the full capability. Under the

```
1  import collage
2
3  # [Method 1] Explicit pattern specification
4  # Pattern language to describe conv2d + add + relu.
5  conv = is_op('conv2d')(wildcard(), wildcard())
6  conv_constr = conv.has_attr({"data_layout": "NCHW"})
7  conv_add = is_op('add')(conv_constr, wildcard())
8  conv_add_relu = is_op('relu')(conv_add)
9
10 # Introduce new backend pattern to Collage.
11 collage.add_backend_pattern(backend='cuDNN',
12                             pattern=conv_add_relu)
13
14 # [Method 2] Pattern rule specification
15 class MyPatternRule(collage.BasePatternRule):
16    # Define variables
17    kFusable = 0
18    kElemwise = 1
19    # ...
20    # Checker for the supported operators.
21    @staticmethod
22    def op_rule(op):
23       if op.name == "dense":
24          # Dense operator is always supported.
25          return True
26       elif op.name == "conv2d":
27         # constraints can be verified as well.
28         return op.attr["data_layout"] == "NCHW"
29       # ... rest of the op rule ...
30       return False
31
32    # Checker for fusion patterns.
33    #   -- cur_type: type of current fusion group
34    #   -- src: seed operator node
35    #   -- sink: post-dominator of src
36    @staticmethod
37    def fusion_rule(cur_type, src, sink):
38       # If current fusion group contains
39       #   at least one conv/matmul (kFusable)
40       if cur_type == MyPatternRule.kFusable:
41         # Helper functions can be defined.
42         def fchecker(node_pattern):
43            return (node_pattern == MyPatternRule.kElemwise)
44         # Check if every operator between src and sink.
45         # Helper function can be passed as a checker.
46         if collage.check_path(src, sink, fchecker)):
47            return True
48       # ... rest of the fusion rule ...
49       return False
50
51 # Introduce new pattern generation rule to Collage.
52 collage.add_backend_pattern_rule(backend='TVM',
53                                  pattern_rule=MyPatternRule())
```

Listing 1: Example of the backend registration interface. To register a new backend, users can directly enumerate patterns or write a pattern rule that consists of valid operator checker and fusion rule in Python classes.

hood, existing operator fusion engines often fuse operators based on heuristic fusion rules that examine the type of each operator and the relationship between different types. For instance, a fusion engine may combine multiple operators across different branches into a single kernel as long as they satisfy its fusion rule.

For an adoption of various backends, our system provides two levels of abstraction: *pattern* and *pattern rule*. Pattern is a direct way to specify all supported operator patterns in *Collage*'s pattern language, which extends the Relay pattern language [60]. However, supported patterns can be too complicated to explicitly specify. To incorporate sophisticated patterns, pattern rules offer an expressive way to specify a valid set of operator fusion rules in the form of Python; users can use any Python features to describe complex fusion algorithms. Each pattern rule is used to generate valid patterns for the input workload with our automatic pattern generator. With

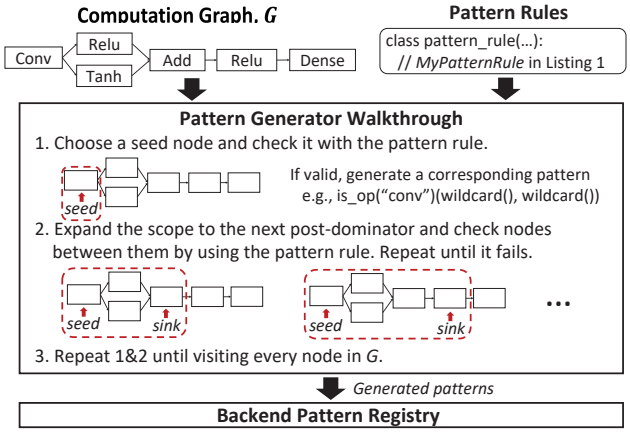


Figure 4: Example illustrating how the backend pattern generator would automatically generate valid patterns with the pattern rule presented in Listing 1.

two levels of abstraction, users can easily incorporate an additional backend by specifying its patterns and pattern rules with an intuitive programming interface. By default, *Collage* provides built-in patterns and pattern rules for popular backends [5, 7, 21, 23, 67].

Listing 1 presents an example of use-case scenarios. If a backend only supports a few simple patterns, users may enumerate those patterns and add them directly to the backend pattern registry (line 3-12). Users can easily check the operators (line 5), their configurations such as data layouts and kernel sizes (line 6), and the the relationship between operators (line 7-8). A wildcard operator is a special placeholder that matches any operator.

To fully support advanced backends [7, 21, 23, 53], users can bring their pattern rules to incorporate more complicated patterns with *Collage*'s pattern generator (line 14-53). To use this feature, users need to provide operator checkers with their potential constraints (line 20-30) and a fusion rule (line 32-49) in the form of Python methods. Then, the automatic pattern generator in *Collage* will search for valid operator patterns satisfying these rules and add them to the backend pattern registry before optimizing backend placement.

Figure 4 exhibits how our pattern generator searches for legitimate patterns using given pattern rules on an input computation graph. By visiting every operator in an input computation graph, the pattern generator investigates how far a pattern can grow without breaking the pattern rule. For each operator, the pattern generator first validates whether the operator can be executed on a backend (line 20-30). If valid, it enlarges the scope one step further and validates whether a set of operators satisfies the fusion rule (line 32-49). For instance, line 40-47 specify that the assumed backend can fuse element-wise operators following an operator of type `kFusable`, which includes convolution and matrix multiplication. Whenever a group of operators satisfying the rule is found, the pattern generator produces a corresponding pattern and adds it to the backend pattern registry. Then, it enlarges the scope of interests one step further again to see if a bigger pattern can be found. This

approach allows *Collage* to incorporate advanced backends, such as TVM, cuDNN, DNNL and TensorRT, without missing any pattern.

4 BACKEND PLACEMENT OPTIMIZATION

4.1 Problem Definition

Collage attacks the backend placement problem to find the best use of available backends and maximize performance. Consider a computation graph $\mathcal{G}$ and a set of backend patterns $\mathcal{B}$ in *Collage*'s backend pattern registry. $\mathcal{G}$ is a Directed Acyclic Graph (DAG) where each node represents a tensor operator (e.g., convolution, matrix multiplication). $b = (p, d) \in \mathcal{B}$ is a pair of an operator pattern p and a backend identifier d, such as cuDNN, cuBLAS, etc.

With M matched subgraphs g_i and backend patterns b_i for $i \in \{1, 2, \cdots M\}$, let $\mathcal{P}(\mathcal{G}) = \{(g_i, b_i) | b_i \in \mathcal{B}, \bigcup_{i=1}^{M} g_i = \mathcal{G}, g_i \cap g_j = \emptyset$ for all $i, j \in \{1, 2, \cdots, M\}$ where $i \neq j\}$ be a backend placement strategy on a computation graph $\mathcal{G}$ and $Cost(\mathcal{P}(\mathcal{G}))$ be the execution time of a placement $\mathcal{P}(\mathcal{G})$. In this work, we aim to find a backend placement strategy $\mathcal{P}_{opt}$ that minimizes $Cost(\mathcal{P}(\mathcal{G}))$. This problem can be formalized as follows:

$$\mathcal{P}_{opt}(\mathcal{G}) = \underset{\mathcal{P}(\mathcal{G})}{\arg\min}\, Cost(\mathcal{P}(\mathcal{G})) \tag{1}$$

4.2 Op-level Placement Optimizer

To efficiently evaluate numerous candidates with different placement and prune the search space, *Collage* conducts an op-level placement optimization as the first step. Its goal is to map all operators on the computation graph to the most efficient set of low-level kernel implementations from available backends fast without considering cross-kernel optimizations in graph inference libraries. As discussed earlier, the graph-level placement optimizer (§4.3) would make up for the possible performance loss from this simplification.

With this simplification, low-level kernel executions become independent to each other in a single device execution. Let s_1 and s_2 be subgraphs of $\mathcal{G}$ where $s_1 \cup s_2 = \mathcal{G}$, $s_1 \cap s_2 = \emptyset$. Then, the following additive relationship [37] between the run-time cost of $\mathcal{P}(s_1)$ and $\mathcal{P}(s_2)$ can be used to determine $Cost(\mathcal{P}(\mathcal{G}))$:

$$Cost(\mathcal{P}(\mathcal{G})) = Cost(\mathcal{P}(s_1)) + Cost(\mathcal{P}(s_2)) + \epsilon \tag{2}$$

where ϵ is a context switching cost (e.g, driver overhead), which is nearly constant empirically. Note that *Collage* avoids data transfers between different backends on the same device by only exchanging data pointers to the tensors (e.g., s1 and s2) using the zero-copy mechanism. With this cost model, it is possible to cheaply approximate the cost of a graph by partitioning a graph into smaller subgraphs and summing up their cost. Despite the efficient cost model, excessively large number of possible placement strategies and a variety of fusion patterns make search non-trivial.

To address this challenge, we propose a Dynamic Programming (DP) method for optimizing backend placement at the operator level. By using the additive relation (Equation 2), we deduce the following recurrence relation of optimized backend placement $\mathcal{P}_{opt}(s)$ and its cost $C_{opt}(s)$ for any subgraph $s \subset \mathcal{G}$. This breaks down a problem

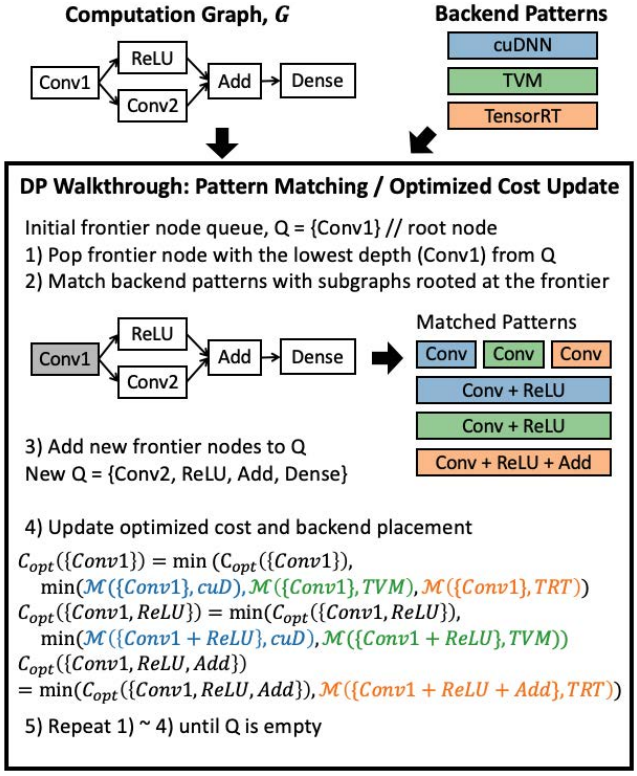


Figure 5: Example of Dynamic Programming (DP) procedures. By visiting over each frontier node, DP algorithm matches backend patterns and update the optimized placement and its cost. For simplicity, optimized placement update is omitted.

of finding $\mathcal{P}_{opt}(\mathcal{G})$ into smaller problems of finding $\mathcal{P}_{opt}(s)$.

$$\mathcal{P}_{opt}(s) = \mathcal{P}_{opt}(s_{min}) \cup \mathcal{P}(g_{min})$$
$$C_{opt}(s) = \begin{cases} 0 & \text{if } s = \emptyset \\ C_{opt}(s_{min}) + \mathcal{M}(\mathcal{P}(g_{min})) + \epsilon & \text{otherwise} \end{cases} \tag{3}$$

where s_{min} and g_{min} are

$$\underset{s' \cup g' = s, s' \cap g' = \emptyset}{\arg\min} \{C_{opt}(s') + \mathcal{M}(\mathcal{P}(g')) + \epsilon\} \tag{4}$$

s' represents a subgraph that is already examined while g' is a subgraph that is going to be evaluated with a measurer $\mathcal{M}(\cdot)$, which takes a backend placement strategy and returns its actual run-time cost on the execution environment. We query the measurer at the granularity of a backend pattern that matches with g', which is either single or multiple operators (operator fusion) that will be lowered to a single low-level kernel. This approach ensures that we always measure a single kernel and add it up to compute the cost of larger subgraphs. To avoid the repetitive and expensive measurement overhead (i.e., compilation + multiple runs on the actual hardware), we cache the result to the log for the future usage. With this approach, we can efficiently explore possible backend placements and evaluate them.

Algorithm 1 Op-level Placement Optimization: DP

Input: Computation graph $\mathcal{G}$ and set of backend patterns $\mathcal{B}$
Output: Optimized placement $\mathcal{P}_{opt}(\mathcal{G})$

1: // v_0: a root of $\mathcal{G}$, $\mathcal{Q}$: a priority queue sorted by node depth
2: $\mathcal{Q} = \{v_0\}$
3: **repeat**
4: // v_s is a frontier node
5: $v_s = \mathcal{Q}$.dequeue()
6: **for** $b_i \in \mathcal{B}$ **do**
7: // Find a subgraph g rooted at v_s that matches b_i
8: **if** g = get_match(v_s, b_i) **then**
9: // $\mathcal{F}$ is a set of new frontier nodes after matching
10: **for** $v_j \in \mathcal{F}$ **do**
11: **if** v_j has never been added to $\mathcal{Q}$ **then**
12: $\mathcal{Q}$.enqueue(v_j)
13: **end if**
14: **end for**
15:
16: // $\mathcal{P}(g) = \{(g, b_i)\}$
17: // $\mathcal{M}$ is a measurer
18: // $\mathcal{S}$ is a set of subgraphs, each of which includes all nodes before v_s in post-order and does not include g
19: // ϵ is a constant for context switching cost
20: **for** $s_j \in \mathcal{S}$ **do**
21: **if** $C_{opt}(s_j \cup g) > C_{opt}(s_j) + \mathcal{M}(\mathcal{P}(g)) + \epsilon$ **then**
22: $C_{opt}(s_j \cup g) = C_{opt}(s_j) + \mathcal{M}(\mathcal{P}(g)) + \epsilon$
23: $\mathcal{P}_{opt}(s_j \cup g) = \mathcal{P}_{opt}(s_j) \cup \mathcal{P}(g)$
24: **end if**
25: **end for**
26: **end if**
27: **end for**
28: **until** $\mathcal{Q} = \emptyset$
29:
30: **return** $\mathcal{P}_{opt}(\mathcal{G})$

Figure 5 illustrates an simplified walkthrough example of our DP method. By traversing a computation graph $\mathcal{G}$, it solves smaller problems of finding $\mathcal{P}_{opt}(s)$ for a subgraph $s \subset \mathcal{G}$ and eventually discovers $\mathcal{P}_{opt}(\mathcal{G})$. First, it puts a root node in the priority queue as an initial frontier node; we define a *frontier node* as a node that has the lowest depth among unvisited nodes on a path from the root. Then it pops a frontier node with the lowest depth from the queue and examines if any subgraph rooted at the current frontier node can match any valid backend pattern. Once a matching is found, we add new frontier nodes to the priority queue and measure the cost of the subgraph matched with the backend pattern. If a better placement strategy is found, we update the optimized cost and backend placement strategy based on Equation (3). We repeat these steps until the priority queue is empty. Given that graph inference libraries, such as TensorRT, can also provide competitive operator-level implementations (Figure 2), we also include them in the op-level optimization. Algorithm 1 formalizes our DP method.

Time complexity. We derive the time complexity of Algorithm 1. Let N be the number of nodes (operators) in computation graph

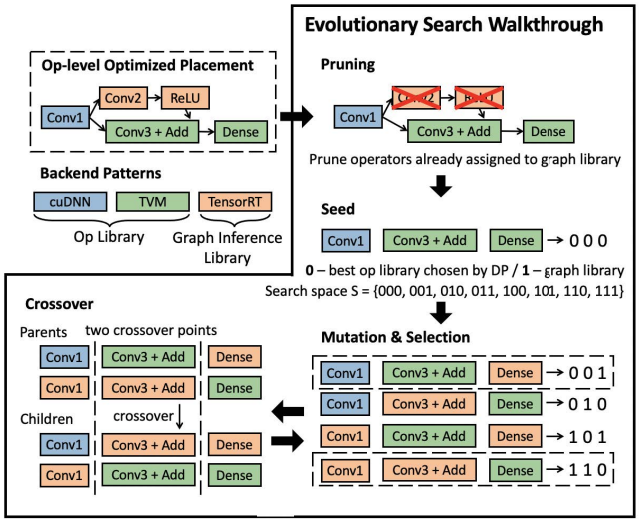


Figure 6: Example of Evolutionary Search (ES) procedure. After pruning search space, it iterates over mutation, selection, and crossover until it reaches saturation or time limit.

$\mathcal{G}$, P be the average number of backend pattern matches per frontier, F be the maximum possible number of frontiers for a single match, and S be the maximum number of subgraphs in $\mathcal{S}$ (line 20). In Algorithm 1, the outermost while loop (line 3) takes $O(N)$ times to traverse each frontier node in $\mathcal{G}$. For each frontier, there can be $O(P)$ matches (line 6-8). For each match, the algorithm iterates over its $\mathcal{F}$ (line 10) and $\mathcal{S}$ (line 20) and takes $O(F + S)$. Therefore, the overall time complexity of our op-level placement optimizer is $O(NP(F + S))$. In all workloads that we have investigated, $N < 1000, P < 20, F < 10, S < 200$. As a result, our DP method optimizes placement within a minute by effectively pruning candidates.

4.3 Graph-level Placement Optimizer

As the op-level placement optimization ignores the effect of cross-kernel optimizations (e.g., scheduling and memory optimizations) in graph inference libraries, *Collage* introduces the graph-level placement optimizer to fine-tune the potentially sub-optimal backend placement strategies from the op-level. To do so, we need to identify additional operators that are not assigned to graph inference libraries but can benefit from cross-kernel optimizations. Once identified, we offload them to graph inference libraries to extract further improvement. However, a key challenge we must address in this approach is deciding which operators to offload to graph inference libraries among a myriad of candidates..

To address this challenge, we represent each backend placement strategy by using a sequence of digits. Each digit implies whether to offload to graph inference libraries. Since our goal is to offload more operators that can benefit from the cross-kernel optimization, we exclude operators already mapped with a graph inference library from this encoding. This straightforward state representation eliminates the complexity from various graph partitions and their topology.

We adopt an evolutionary search algorithm [30] for graph-level placement optimization. Figure 6 describes the procedure of our evolutionary search method. For state representation, 0 indicates keeping the decision of the op-level optimizer and 1 means overriding the decision and offloading it to a graph inference library (e.g., TensorRT). To facilitate the search process, we include the op-level optimized placement strategy as one of the seeds to provide a good starting point. The evolutionary algorithm iterates over rounds of mutation, selection, and two-point crossover to fine-tune the backend placement.

5 EVALUATION

This section aims to answer the following questions:

- Can *Collage* effectively optimize real-world DL model execution over diverse backends and target devices compared to the existing DL frameworks? (§5.2)
- Is optimization time affordable? How much time does each optimization take? (§5.3)
- Does adding more backends improve the performance of *Collage*? (§5.4)
- How does backend placement optimized by *Collage* look like? (§5.5)

5.1 Experimental Setup

Implementation. We built the core of *Collage* in the form of a portable Python library and leveraged diverse backends in different hardware architectures: cuDNN [23], cuBLAS [5], TVM [21], TensorRT [7], MKL [67] and DNNL [3]. To orchestrate a runtime execution with multiple backends, Collage uses DLPack to minimize data movement (e.g., tensor) across different backend runtimes by efficiently exchanging pointers of data with zero-copy approach [2]. Still, even such optimized communications incur certain run-time overhead (e.g., deserialization overhead of the engine in graph inference libraries [8]). Thus, Collage takes this run-time overhead into account when measuring execution time of various placement candidates. If such run-time overhead is too excessive, Collage will choose another candidate with better performance. To leverage full capabilities of backends, their supported patterns and pattern rules are provided based on their official documentation and codebases. Each backend specification with full operator supports only takes about 100 LoC with Collage API.

Baselines. We examine TensorFlow (TF) [10], TF-XLA [9], PyTorch [55], TVM [21], and TensorRT [7] as DL framework baselines. For TVM, we use AutoTVM to automatically generate the optimized operator schedules for each target. Note that we also integrate TensorRT and TVM as high-performance graph inference libraries in this experiment.

Workload. We evaluate five popular real-world DL inference workloads that cover a wide range of application. BERT [25] is a transformer-based language model that achieved the state-of-the-art performance on a spectrum of natural language processing tasks. DCGAN [57] is an extension of the GAN [32] with an unsupervised representation learning mainly for image generation. NasNet-A [75] is one of the most popular machine-generated DL workloads that show strong performance on popular image recognition tasks. 3D-ResNet50 [33] is an extension of widely adopted ResNet50 [34] for 3D image tasks such as action recognition. ResNeXt50 [68] introduces a grouped convolution to ResNet50 architecture and improves its model accuracy and computational complexity for image recognition.

Each workload has its own characteristics in terms of its operators and structure. Most of recent models for language application such as BERT are basically a series of the Transformer layers that consist of batch matrix multiplication, layer normalization, softmax, etc. On the other hand, models for vision application such as ResNeXt50 and NasNet-A has a series of layers that has operators including convolutions and non-linear activation functions (e.g., ReLU). In these models, operator configuration (e.g., number of channels and hidden nodes) varies across different layers as you see in Figure 2, which leads to performance diversity of DL backends.

5.2 End-to-end Evaluation

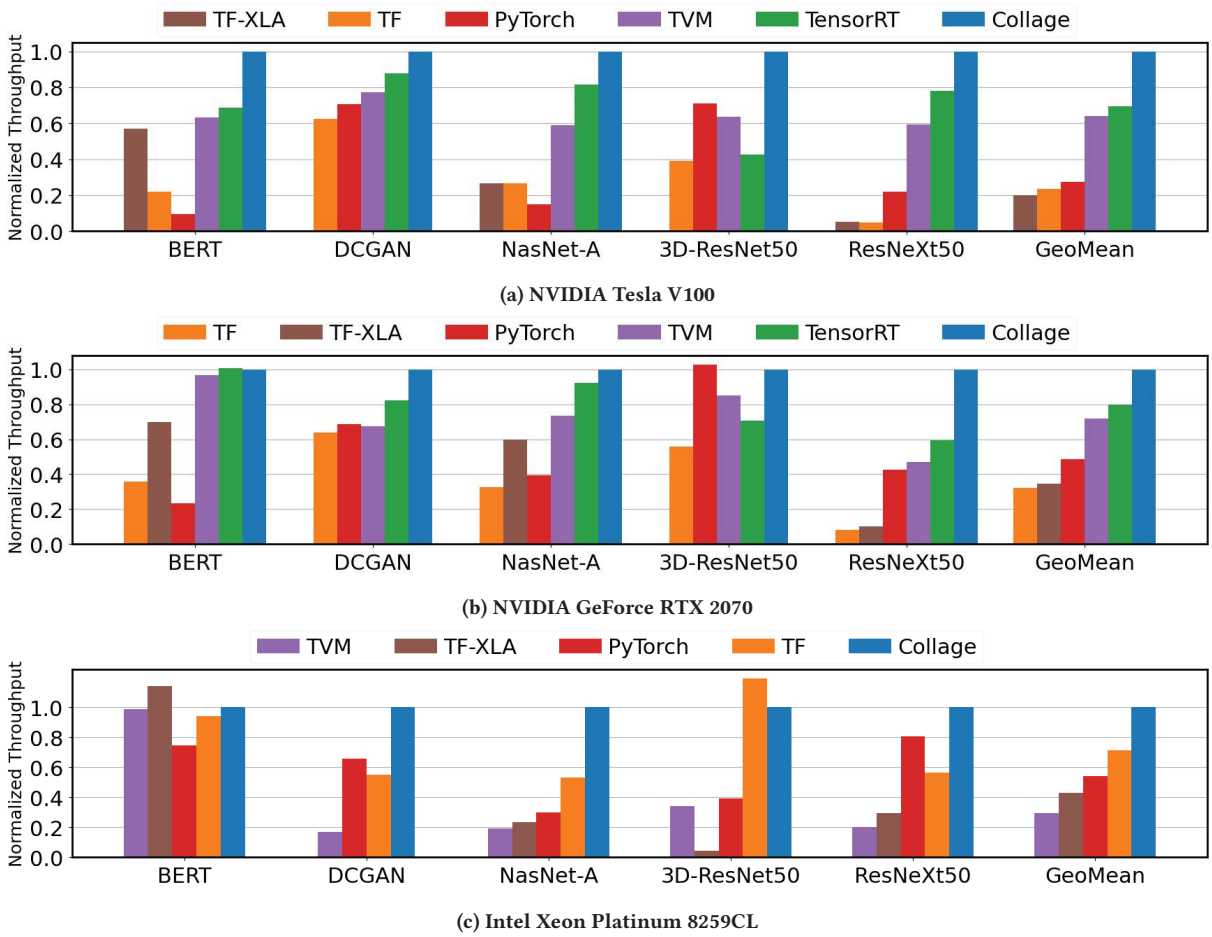
To discuss the effectiveness of our approach, we evaluate the end-to-end performance of *Collage* against the baseline frameworks; note that we omit error bars from our figures because we observe marginal standard deviation (less than 3%) for all results. Note that the performance of TF-XLA is missing for some pairs of workload and targets (e.g., 3D-ResNet50 and NVIDIA GPU) because it has issues with some 3D convolutions for GPU targets and certain image resizing operators.

Figure 7a and Figure 7b presents the end-to-end normalized throughput of *Collage* and existing DL frameworks on two different NVIDIA GPU architectures, Tesla V100 and GeForce RTX2070. Normalized throughput is the throughput of each framework normalized by the throughput of *Collage*. Overall, *Collage* consistently produces the most efficient executable across different workloads and hardware architectures: In terms of geometric mean, *Collage* outperforms the state-of-the-arts by 1.43× on V100 and 1.26× on RTX 2070, respectively. This improvement comes from *Collage*'s backend placement optimization that effectively leverages the unique strength of various backends.

Figure 7c exhibits the experimental results on the Intel CPU. Likewise, *Collage* showcases the most stable performance across different workloads on this Xeon architecture while beating the state-of-the-arts by 1.40× in the geometric mean. However, on BERT and 3D-ResNet50, TF-XLA and TF are faster possibly due to their optimizations customized for Intel CPU such as data layout optimization with non-uniform memory access, which is orthogonal to backend placement.

As the representative case, different batch sizes are also examined with ResNeXt50 on V100. Figure 8 indicates that *Collage* consistently outperforms the state-of-the-art frameworks across different batch sizes as well.

Since backends and their performance vary depending on the underlying execution environment, backend placement should be carefully customized by considering their performance landscape. Our experimental results indicate that *Collage* can stably offer a faster DL execution than existing frameworks with the rigid hand-written heuristics across different hardware architectures.



(a) NVIDIA Tesla V100

(b) NVIDIA GeForce RTX 2070

(c) Intel Xeon Platinum 8259CL

Figure 7: End-to-end performance of state-of-the-arts DL frameworks and *Collage* in five real-life workloads on NVIDIA GPUs and Intel CPU. Throughput of each framework is normalized by the throughput of *Collage*. Following backends are employed for each framework according to target hardware and its capabilities: NVIDIA GPU (cuDNN, cuBLAS, TVM, TensorRT), Intel CPU (MKL, DNNL, TVM).

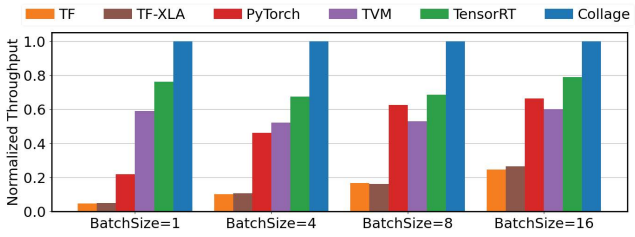


Figure 8: End-to-end performance with different batch sizes in ResNeXt50 on NVIDIA V100. Normalized throughput is the throughput normalized by the throughput of *Collage*.

5.3 Optimization Time

To evaluate the overhead from our automated optimizer, this subsection studies the overall optimization cost of the two-level approach. For this section, we use NVIDIA V100 as our target.

Figure 9 shows the breakdown of our operator-level optimization time. If the optimization is launched from scratch, the entire optimization process takes up to two minutes. This optimization time consists of two parts: measurement of the operator cost and overhead from the DP algorithm. Due to the high evaluation cost, the optimization time is dominated by the profiling overhead. However, as discussed in §4.2, the repetitive profiling for operator cost can be avoided by saving the cost of each operator. When the cost of every operator is profiled in advance, our op-level placement optimization takes less than a minute on all of the five networks.

Figure 10 exhibits how our graph-level placement optimization gradually improves from the op-level placement optimization over time. The evolutionary searcher could boost the performance by leveraging more cross-kernel optimizations as it goes through several generations of mutations and crossovers. In BERT and DCGAN, the effect of cross-kernel optimization is quite notable and thus, our graph-level placement optimizer accelerate its execution by $1.09 - 1.20\times$ from the op-level optimization. For the rest of the

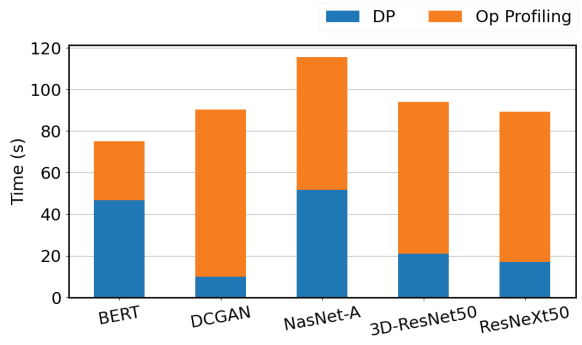


Figure 9: The breakdown of op-level placement optimization time. On average, profiling overhead for operator cost measurements takes up 68% of the entire optimization time. Note that profiling is only necessary for unseen operators. Once the cost of a new operator is measured, its information will be saved in the logging database in *Collage* to avoid the repetitive profiling. If profiling log is available, op-level optimizer only takes less than a minute.

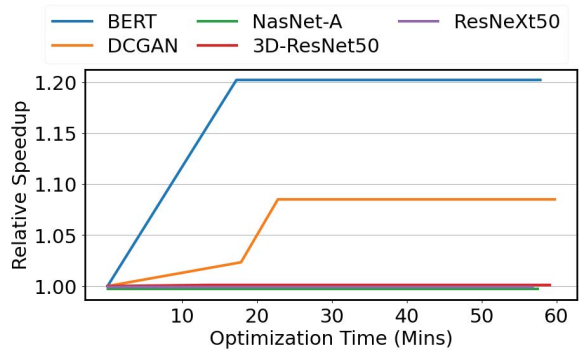


Figure 10: Performance improvement of graph-level placement optimization over time. The y-axis presents the speedup relative to the op-level placement optimization.

workloads, graph-level placement optimization cannot improve any further since the placement from the op-level optimization is already hard to beat. Overall, most of workloads are observed to reach the saturation within thirty minutes.

Due to the lack of the efficient cost model that can factor in the cross-kernel optimization effect, graph-level placement optimization has expensive evaluation overhead that leads to the longer optimization time compared to the op-level. Given that our op-level placement optimizer can identify high-performance backend placement for the most workloads within just a minute, we recommend the graph-level placement optimization as the optional tool for the users interested in squeezing the last drop of performance.

5.4 Backend Ablation Study

To assess the impact of integrating backends, we conduct an ablation study by adding backends one-by-one to *Collage*.

Figure 11 shows the experimental result on V100. Overall, *Collage* monotonically improves performance as we integrate more backends. This reinforces the importance of smart mixed-use of multiple backends and also corroborates the robustness of our backend placement optimization. It is worth noting that the performance improvement from a new backend varies depending on a network. In the case of BERT and DCGAN, we see relatively consistent enhancement from each backend. This is because *Collage* identifies a way to utilize every backend for the different part of the workload depending on its own unique strength. In case of NasNet-A and ResNeXt50, TVM offers the majority of the performance improvement while cuDNN significantly benefits *Collage* for the 3D-ResNet50.

These observations show that *Collage* can stably improve performance by having more backends. By leveraging the unique strength of available backends, our automated optimizer delivers the performance with a set of backends that surpasses or guarantees the performance with its subset.

5.5 Case Study of Backend Operator Placement

To understand the source of performance improvement from *Collage*, we examine two representative workloads in detail. Figure 12 illustrates *Collage*'s final backend placement for ResNeXt50 and BERT on V100.

Even within a single network, we observe that the same type of operator is mapped to different backends due to the performance diversity depending on its configuration, such as data shape and kernel size, and the operator fusion with its neighbor nodes. For example, batch matrix multiplication operators in BERT are assigned to two different backends (cuBLAS and TVM) while convolution operators in ResNeXt50 are assigned to three different backends (cuDNN, TVM, and TensorRT). Interestingly, the graph inference library (e.g., TensorRT) can be a competitive choice even for a single operator as observed with some convolution operators in ResNeXt50.

This figure also demonstrates that *Collage* is capable of leveraging various fusion patterns from each backend. For instance, we discover a variety of operator fusion patterns selected by *Collage* such as Conv+ReLU, Conv+Add+ReLU, and Add+ReLU. Although it is omitted from this figure for simplicity, we observe the fusion pattern involved with more than ten operators. Again, as in a single operator, *Collage* chooses the different backends for the identical fusion pattern of Conv+Relu in ResNeXt50 because the best backend choice varies depending on specific operator configurations.

This study confirms that *Collage* can accelerate DL workload execution by leveraging diverse operator patterns from multiple backends given their performance characteristics.

6 RELATED WORK

Diversified Backend Ecosystem. To extract the best performance from the underlying hardware, there have been substantial efforts to design high-performance DL backends. Hardware vendors have released various specialized optimized libraries and inference engines. NVIDIA has actively developed cuDNN [23] to deliver optimized implementations of DL operators, cuBLAS [5] to offer efficient BLAS kernels, and TensorRT [7] to create fast execution plans for DL workloads. Particularly, TensorRT considers various graph-wide

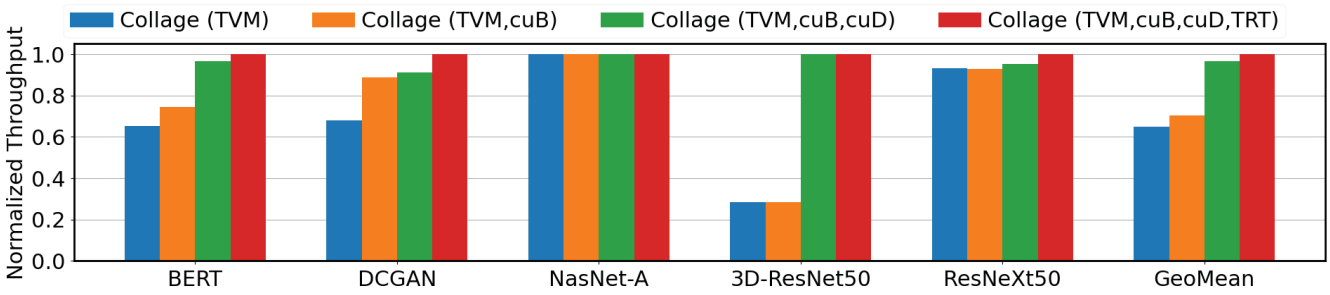


Figure 11: End-to-end performance of *Collage* with different number of backends on NVIDIA Tesla V100. Each throughput is normalized by the throughput of *Collage* (TVM,cuB,cuD,TRT). TVM, cuB, cuD, and TRT represents TVM, cuBLAS, cuDNN, and TensorRT.

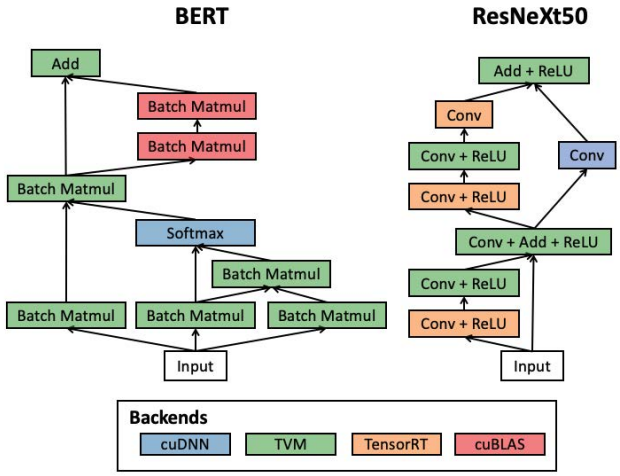


Figure 12: Representative backend placements discovered by *Collage* on V100 (Figure 7a). Note that *Collage* leverages various backends given their unique strength to enhance performance.

cross-kernel optimizations for scheduling, memory footprint and etc. Meanwhile, Intel has released oneDNN [3] for optimized DL operator kernels and OpenVINO [4] as an inference engine for Intel CPUs. AMD also has driven MIOpen [43], an open source GPU library for DL primitives.

Today's DL frameworks exploit tensor compilers [12, 14, 16, 17, 19–21, 29, 35, 41, 44, 45, 47, 49, 56, 58, 59, 65, 66, 70, 72] as their backends to generate operator kernels for various target devices. While some tensor compilers rely on manual scheduling [16, 58, 72], automatic approaches [12, 14, 17, 20–22, 41, 41, 44, 49, 56, 58, 59, 66, 70] has been actively studied to optimize tensor operator kernels for a given DL workload and device. For instance, Tensor Comprehension [66] uses black-box auto-tuning to optimize CUDA kernels along with polyhedral optimizations. To speed up the optimization time, cost model has been also widely examined together with automated approaches [22, 42, 70, 72].

By providing an expressive registration interface and automatic placement optimizer, *Collage* enables seamless integration of a wide variety of DL backends without any expertise in complex performance dynamics of varied backends.

DL Frameworks. To provide easy and powerful platform of running a variety of DL workloads, different frameworks have been continuously released and improved. Google maintains TensorFlow [10] and XLA [9] to optimize the execution on various hardware devices including TPUs [40]. Facebook develops Pytorch [55] that supports dynamic eager execution for usability while preserving compelling DL execution performance. For NVIDIA GPUs, TensorRT [7] is developed as a runtime framework that optimizes DL model execution. As an open-source C++ library and compiler suite for CPUs, Intel has launched nGraph [24]. Also, TVM [21] offers the efficient compilation pipeline that is designed to support diverse hardware devices and DL workloads. On the other hand, Glow [61] is proposed to efficiently generate the optimized code for multiple targets of heterogeneous hardware. While such existing DL frameworks employ handwritten rules to integrate new backend, *Collage* reduces the manual effort with the backend pattern abstraction and extracts further performance gain with the automated backend placement.

Operator Fusion. Fusion is one of the most efficient techniques to optimize DL workloads by combining multiple high-level operators on the computation graph into a single kernel. To maximize the benefit, advanced fusion techniques [9, 11, 15, 18, 21, 26, 27, 29, 41, 46, 48, 53, 63, 73] introduce their own unique fusion rules to apply this optimization beyond a few special cases. For instance, by iterating over every operator, TVM seeks for an opportunity to merge each operator with its neighbors by using the union-find algorithm [21]. To efficiently explore the fusion opportunities, DNNFusion [53] employs a detailed classification of operation type and makes the fusion decisions. To identify the best fusion plan, FusionStitching [73] conducts Just-In-Time tuning. NVIDIA has actively improved the fusion engine in cuDNN to merge certain patterns of operators at runtime [23]. Internally, TensorRT [7] also actively apply the fusion to optimize the memory access and scheduling overhead. By offering the highly flexible user interface for the pattern rules, *Collage* can support such complicated fusion patterns from a variety of such backends. With fusion patterns and their rules, *Collage* naturally considers diverse fusion possibilities in multiple backends.

Graph Rewriting. To accelerate a DL execution, DL frameworks can rewrite an input computation graph by considering a number of graph substitution rules. Most DL frameworks such as TensorFlow [10], TensorRT [7], and TVM [21] rely on the greedy approach by opportunistically applying a few important hand-coded rules. In contrast, MetaFlow [38] suggests an automated graph rewriting approach that optimizes an input graph using backtracking search. TASO [37] extends MetaFlow's backtracking search and further automates graph substitution generation for every new input graph. To further improve graph substitution search efficiency, sampling-based approach [28] has also been explored. To overcome the inefficiency in making sequential rewriting decisions, [69] proposes e-graph and equality saturation method. As these graph rewriting techniques are orthogonal to *Collage*, *Collage* can improve the performance of a rewritten computation graph by optimizing the backend placement.

Device Placement. There are two major categories of work that investigates how to place DL operators across devices. One category is to learn a placement policy [31, 50, 51] that places each operator onto one of given set of devices and generalize it to new workloads via transfer learning [13, 54, 74]. Another category is to algorithmically find good graph partitions of DL workloads and their schedules [36, 39, 52, 64, 71]; for example, FlexFlow [39] uses stochastic search method with delta simulation to partition a single operator into multiple computation and place them on devices. Compared to device placement, backend placement itself has its unique challenges of modeling complicated and fast-evolving operator fusion patterns and constraints from diverse backends in addition to different backend characteristics (e.g., cross-kernel optimization of graph inference library). To tackle this challenge, *Collage* provides an expressive backend pattern abstraction and a two-level optimizer, each level of which considers different characteristics of backends. Our work is complementary to existing device placement works.

7 CONCLUSION

This work investigates an efficient DL backend integration system, called *Collage*. For the seamless integration of various backends, Collage offers an user interface that allows the flexible specification of diverse backend capabilities. To find the best uses of available backends, Collage introduces a two-level optimization method and automatically customizes the best possible backend placement for the underlying execution environment. The experimental results demonstrate that *Collage* outperforms the best manual approach in the state-of-the-arts DL framework by up to 1.43× on average over real-life DL models and various hardware architectures. More importantly, unlike existing approaches, it offers stable performance across diverse hardware architectures and models by selecting the most beneficial backends for each part of workload.

ACKNOWLEDGEMENT

We would like to thank members of Catalyst group at CMU for their helpful comments on our work and manuscript. We would also like to thank the anonymous PACT reviewers for constructive feedbacks. This work was partially supported by the National Science Foundation under grant number CNS-2147909 and the Real Time Machine Learning (RTML) DARPA project.

REFERENCES

[1] [n.d.]. Apple Neural Engine (ANE). https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/. Accessed: 2021-08-25.
[2] [n.d.]. DLPack: Open In Memory Tensor Structure. https://github.com/dmlc/dlpack. Accessed: 2022-04-05.
[3] [n.d.]. Intel OneDNN. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onednn.html. Accessed: 2021-09-27.
[4] [n.d.]. Intel OpenVINO. https://software.intel.com/content/www/us/en/develop/tools/openvino-toolkit.html. Accessed: 2021-09-27.
[5] [n.d.]. NVIDIA cuBLAS. https://developer.nvidia.com/cublas. Accessed: 2021-08-05.
[6] [n.d.]. NVIDIA Deep Learning Accelerator (NVDLA). http://nvdla.org/. Accessed: 2021-08-25.
[7] [n.d.]. NVIDIA TensorRT. https://developer.nvidia.com/tensorrt. Accessed: 2021-08-05.
[8] [n.d.]. NVIDIA TensorRT Deserialization. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html. Accessed: 2022-06-09.
[9] [n.d.]. Tensorflow XLA. https://www.tensorflow.org/xla. Accessed: 2021-09-27.
[10] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.
[11] Amirali Abdolrashidi, Qiumin Xu, Shibo Wang, Sudip Roy, and Yanqi Zhou. 2019. Learning to fuse. In *NeurIPS ML for Systems Workshop*.
[12] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize Halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (2019), 121:1–121:12.
[13] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2018. Placeto: Efficient progressive device placement optimization. In *NIPS Machine Learning for Systems Workshop*.
[14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. 2014. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 303–316.
[15] Arash Ashari, Shirish Tatikonda, Matthias Boehm, Berthold Reinwald, Keith Campbell, John Keenleyside, and P Sadayappan. 2015. On optimizing machine learning workloads via kernel fusion. *ACM SIGPLAN Notices* 50, 8 (2015), 173–182.
[16] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 193–205.
[17] Cedric Bastoul, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Adilla Susungi, Javier de Juan, Etienne Filhol, Baptiste Jarry, Gianpietro Consolaro, and Renwei Zhang. 2022. Optimizing GPU Deep Learning Operators with Polyhedral Scheduling Constraint Injection. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 313–324.
[18] Matthias Boehm, Berthold Reinwald, Dylan Hutchison, Alexandre V Evfimievski, and Prithviraj Sen. 2018. On optimizing operator fusion plans for large-scale machine learning in systemml. *arXiv preprint arXiv:1801.00829* (2018).
[19] Anupama Chandrasekhar, Gang Chen, Po-Yu Chen, Wei-Yu Chen, Junjie Gu, Peng Guo, Shruthi Hebbur Prasanna Kumar, Guei-Yuan Lueh, Pankaj Mistry, Wei Pan, et al. 2019. Igc: The open source intel graphics compiler. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 254–265.
[20] Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. 2020. Automatic generation of multi-objective polyhedral compiler transformations. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 83–96.
[21] 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.
[22] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. *arXiv preprint arXiv:1805.08166* (2018).
[23] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[24] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. 2018. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058* (2018).

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[26] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent rnns: Stashing recurrent weights on-chip. In *International Conference on Machine Learning*. PMLR, 2024–2033.

[27] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning.. In *CIDR*.

[28] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN computation graph using graph substitutions. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2734–2746.

[29] Pratik Fegade, Tianqi Chen, Phillip Gibbons, and Todd Mowry. 2021. Cortex: A Compiler for Recursive Deep Learning Models. *Proceedings of Machine Learning and Systems* 3 (2021).

[30] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *The Journal of Machine Learning Research* 13, 1 (2012), 2171–2175.

[31] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*. PMLR, 1676–1684.

[32] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[33] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. 2018. Can Spatiotemporal 3D CNNs Retrace the History of 2D CNNs and ImageNet?. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6546–6555.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[35] Geonhwa Jeong, Gokcen Kestor, Prasanth Chatarasi, Angshuman Parashar, Po-An Tsai, Sivasankaran Rajamanickam, Roberto Gioiosa, and Tushar Krishna. 2021. Union: A unified HW-SW Co-Design ecosystem in MLIR for evaluating tensor operations on spatial accelerators. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 30–44.

[36] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks.. In *ICML*. 2279–2288.

[37] 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*. 47–62.

[38] Zhihao Jia, James Thomas, Tod Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing dnn computation with relaxed graph substitutions. *SysML 2019* (2019).

[39] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks.. In *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia (Eds.), Vol. 1. 1–13. https://proceedings.mlsys.org/paper/2019/file/c74d97b01eae257e44aa9d5bade97baf-Paper.pdf

[40] 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.

[41] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. 2021. DeepCuts: a deep learning optimization framework for versatile GPU workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 190–205.

[42] Samuel Kaufman, Phitchaya Mangpo Phothilimthana, and Mike Burrows. 2019. Learned TPU cost model for XLA tensor programs. In *Proc. Workshop ML Syst. NeurIPS*. 1–6.

[43] Jehandad Khan, Paul Fultz, Artem Tamazov, Daniel Lowell, Chao Liu, Michael Melesse, Murali Nandhimandalam, Kamil Nasyrov, Ilya Perminov, Tejash Shah, et al. 2019. MIOpen: An open source library for deep learning primitives. *arXiv preprint arXiv:1910.00078* (2019).

[44] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29.

[45] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.

[46] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic horizontal fusion for GPU kernels. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 14–27.

[47] Guei-Yuan Lueh, Kaiyu Chen, Gang Chen, Joel Fuentes, Wei-Yu Chen, Fangwen Fu, Hong Jiang, Hongzheng Li, and Daniel Rhee. 2021. C-for-metal: high performance SIMD programming on intel GPUs. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 289–300.

[48] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[49] Linjian Ma, Jiayu Ye, and Edgar Solomonik. 2020. AutoHOOT: Automatic high-order optimization for tensors. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 125–137.

[50] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *International Conference on Learning Representations*.

[51] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*. PMLR, 2430–2439.

[52] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 1–15.

[53] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

[54] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2020. Reinforced Genetic Algorithm Learning for Optimizing Computation Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rkxDoJBYPB

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[56] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, et al. 2021. A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 1–16.

[57] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[58] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[59] Ari Rasch, Richard Schulze, and Sergei Gorlatch. 2019. Generating portable high-performance code via multi-dimensional homomorphisms. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 354–369.

[60] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. 2018. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 58–68.

[61] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

[62] Omais Shafi, Chinmay Rai, Rijurekha Sen, and Gayathri Ananthanarayanan. 2021. Demystifying TensorRT: Characterizing Neural Network Inference Engine on Nvidia Edge Devices. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 226–237. https://doi.org/10.1109/IISWC53511.2021.00030

[63] Muthian Sivathanu, Tapan Chugh, Sanjay S Singapuram, and Lidong Zhou. 2019. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 909–923.

[64] Jakub M Tarnawski, Amar Phanishayee, Nikhil Devanur, Divya Mahajan, and Fanny Nina Paravecino. 2020. Efficient algorithms for device placement of dnn graph operators. *Advances in Neural Information Processing Systems* 33 (2020), 15451–15463.

[65] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A language, compiler, and runtime for elegant and efficient deep neural networks. In *Proceedings of the 37th*

ACM SIGPLAN Conference on Programming Language Design and Implementation. 209–223.

[66] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[67] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[68] Saining Xie, Ross Girshick, Piotr Dollar, Zhuowen Tu, and Kaiming He. 2017. Aggregated Residual Transformations for Deep Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[69] Yichen Yang, Phitchaya Mangpo Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. arXiv:cs.AI/2101.01332

[70] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. 2020. Ansor: Generating High-Performance Tensor Programs for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[71] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Joseph E Gonzalez, et al. 2022. Alpa: Automating Inter-and Intra-Operator Parallelism for Distributed Deep Learning. *arXiv preprint arXiv:2201.12023* (2022).

[72] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.

[73] Zhen Zheng, Pengzhan Zhao, Guoping Long, Feiwen Zhu, Kai Zhu, Wenyi Zhao, Lansong Diao, Jun Yang, and Wei Lin. 2020. Fusionstitching: boosting memory intensive computations for deep learning workloads. *arXiv preprint arXiv:2009.10924* (2020).

[74] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. 2019. Gdp: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578* (2019).

[75] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. https://arxiv.org/pdf/1707.07012.pdf