DietCode: High-Performance Code Generation for Dynamic Tensor Programs

Abstract

Achieving high performance for compute intensive operators in machine learning (ML) workloads is a crucial but challenging task. This is why many ML and systems practitioners rely on vendor libraries or auto-schedulers to do the job. While the former requires significant engineering efforts (especially for new ML chips), the latter has a limited application domain (e.g., only static-shape workloads are supported). It is impractical to apply existing auto-schedulers directly to dynamic-shape workloads (which are essential in the sequence learning domain such as language modelling, machine translation, and speech recognition), as this leads to both (1) extremely long auto-scheduling/compilation time and (2) a huge binary size.

To address those problems, we propose DietCode, an automated tensor program generation framework for dynamic-shape workloads. The design of DietCode is based on two key ideas: (1) sample programs from a shape-generic search space, and (2) group workload instances into categories. It has three major components: (i) search space, (ii) cost model, and (iii) dispatcher. When auto-scheduling, DietCode learns those three submodules *jointly* to balance the tradeoff between the runtime performance and the binary size.

We evaluate DietCode using various dynamic-shape workloads that are extracted from state-of-the-art machine learning workloads on both CPUs and GPUs. Our evaluation shows that DietCode has the following strengths compared with the state-of-the-art auto-schedulers: (1) reduction in auto-scheduling time by as much as $128\times$, (2) smaller binary size by as much as $6.4\times$, and (3) performance improvement by up to 18% and on par with vendor libraries. All these improvements make DietCode practical to be applied to dynamic-shape workloads.

1 Introduction

Deep neural networks are an important class of machine learning algorithms, having applications in domains such as image classification [15], object detection [14], machine translation [32, 33], speech recognition [5], and language modelling [10]. In those applications, the networks are executed repeatedly for a large number of iterations to capture the internal pattern of the input data samples. Because of this, it is crucial to guarantee the efficiency of every operator in those networks so as to have high performance during the execution on real hardware (e.g., CPUs and GPUs).

However, it is a notoriously challenging task to implement all operators efficiently, especially those that are compute-intensive (e.g., convolution and matrix multiply), because nontrivial expertise is needed across the software and hardware stack in order for those operators to be programmed efficiently. Therefore, most state-of-the-art machine learning frameworks rely on heavily optimized, hand crafted vendor libraries (e.g., oneDNN [26] on Intel CPUs; cuDNN [9] and cuBLAS [24] on NVIDIA GPUs). Despite being able to deliver high performance, the development time and release cycle for vendor libraries to support new workloads and operators on new hardware platforms could be extremely long.

To deliver portable and high-performance tensor programs across various hardware platforms in a relatively short time, auto-scheduler frameworks (e.g., Ansor [36], AutoTVM [7], Halide auto-scheduler [3], and Tensor Comprehensions [31]) have been proposed to bridge the gap between high-level compute definition and low-level schedule primitives. Although having different implementations, to our best knowledge, all existing auto-schedulers have a common limitation: they only support static-shape workloads where all shapes have to be known at compile time.

Unfortunately, real world workloads are not always static due to the following reasons: (1) varying hyperparameters that machine learning practitioners would like to experiment with, (2) dynamic by design (e.g., many models in the sequence learning domain [5, 10, 32, 33] have to dynamically adopt to the input sequence length at runtime), and (3) varying shapes depending on the position of the layer in the model [10, 15]. All those situations push for the need to support *dynamic-shape* workloads in the auto-scheduler, where certain shapes are only known at runtime.

A straightforward way for existing auto-schedulers [3, 7, 31, 36] to support dynamic-shape workloads is by having them operate on one instance of the dynamic-shape workloads (defined as an instantiation with a specific static shape of the dynamic-shape workload) at a time. However, such an auto-scheduling workflow is impractical for two reasons: (1) It takes many hours to complete (can be more than 100 hours on a modern machine equipped with 8 Intel® Xeon® Platinum 8259CL CPUs [27] and 1 NVIDIA Tesla T4 GPU [22]), and (2) it generates a huge binary that includes many possible shape variations (can be more than 100 in some cases). All of these numbers are for a single dynamic-shape workload. These two issues exist because existing auto-schedulers face the following key challenges on dynamic-shape workloads:

(1) *Shape-Dependent Search Space*: The search space construction in existing auto-schedulers is shape-dependent [3, 7, 31, 36], making it hard for one workload instance to share its own search space with others. This results in an overall search space that is too large to be explored efficiently.

(2) *One Schedule for One Instance*: Existing auto-schedulers can only generate one schedule for one workload instance at a time [3, 7, 31, 36]. Therefore, to effectively cover all the possible workload instances, they need to incorporate all the generated programs to a single binary. Each program can only be applied to one static workload instance.

Based on those two key challenges, we propose DietCode, an automated tensor program generation framework for dynamic-shape workloads. To efficiently address those challenges, DietCode adopts two key ideas:

(1) *Sample programs from a shape-generic search space* that can be applied not only to a single but to a collection of workload instances. This allows for simultaneous search of schedules for multiple workload instances – a significantly more efficient approach than the one used by existing auto-schedulers [3, 7, 31, 36].

(2) *Group workload instances into categories* and have all the instances within each category share the same shape-generic program. The sharing strategy reduces the number of programs that are needed to cover all the workload instances.

Based on these key ideas, we design the DietCode framework that has three major components: (i) a shape-generic search space, (ii) a cost model, and (iii) a dispatcher. The search space is composed of *micro-kernels*, small programs that compute a piece of the complete programs and are executed multiple times to carry out an instance of a dynamic-shape workload. Each micro-kernel is shape-generic and form a *category* that can be applied to multiple workload instances of different shapes. In the case when a workload instance cannot fit perfectly into a micro-kernel, automatic padding is applied to pad the instance by the spatial dimension size of the micro-kernel. The cost model accurately and efficiently evaluates how performant all the workload instances are under each category, and the dispatcher sends each workload instance to its preferred category.

We put all three submodules under the same framework, DietCode, and adopt a *joint learning* approach to optimize those three components. We then make the key observation that all three submodules are closely coupled and therefore should be optimized together. Specifically, the cost model and the dispatcher work cooperatively to categorize all the workload instances. One major challenge that we have to address is the tradeoff between the performance on all the workload instances (predicted by the cost model) and the number of categories (i.e., binary size, given by the dispatcher). These two objectives are competing, and we design a new unified cost function to address this challenge. The two submodules (cost model and dispatcher) can also dynamically expand/prune the search space to search for more efficient micro-kernels or speedup the auto-scheduling process.

Our major contributions can be summarized as follows:

(1) We find and address the key challenges in making auto-scheduling practical for dynamic-shape workloads by sampling programs from a *shape-generic* search space and grouping workload instances into *categories*.

(2) We formulate the auto-scheduling problem for dynamic-shape workloads as a *joint learning* problem where we collectively optimize three major components: (i) a shape-generic search space, (ii) a cost model, and (iii) a dispatcher.

(3) We build DietCode, an automated tensor program generation framework for dynamic-shape workloads that adopts the above joint learning approach.

(4) We evaluate DietCode on a dense layer with dynamic sequence lengths from BERT [10], a state-of-the-art language modelling application, and show that it can effectively reduce the auto-scheduling time and the binary size by $128\times$ and $6.4\times$ correspondingly, while improving the runtime performance by 18% compared with a state-of-the-art auto-scheduler, TVM [7, 36], on the GPUs. To demonstrate the generality of our approach, we also evaluate on a different dynamic axis (hidden dimension), a different layer type ($\mathrm{conv2d}$) and achieve $1.09\times$ and $1.05\times$ of the TVM performance. We further show results on the CPUs with the previous two layer types and achieve $0.97\times$ and $0.88\times$ of the TVM performance. In all cases, DietCode only performs the auto-scheduling process once and hence significantly reduces the auto-scheduling time.

2 Background and Motivation

In this section, we present an overview of the key characteristics and shortcomings of both vendor libraries [9, 24, 26] and state-of-the-art auto-schedulers [3, 7, 31, 36] on dynamic-shape workloads.

Achieving high performance for compute-intensive operators (e.g., convolution and matrix multiplication) has always been a challenging task. Therefore, most state-of-the-art machine learning frameworks (e.g., TensorFlow [2], PyTorch [28], and MXNet [6]) rely on heavily optimized, hand crafted vendor libraries (e.g., oneDNN [26] on Intel CPUs; cuDNN [9] and cuBLAS [24] on NVIDIA GPUs). Despite already providing high performance for many operators, those libraries come with a non-trivial price: they require significant expertise across the software and hardware stacks to be efficient for every important operator. As a result, the development time and release cycle for vendor libraries to support new workloads and operators on new hardware platforms could be extremely long and labor intensive.

To deliver portable and high-performance tensor programs across various hardware platforms in a relatively short time, auto-scheduling frameworks (e.g., Ansor [36], AutoTVM [7], Halide auto-scheduler [3], and Tensor Comprehensions [31])

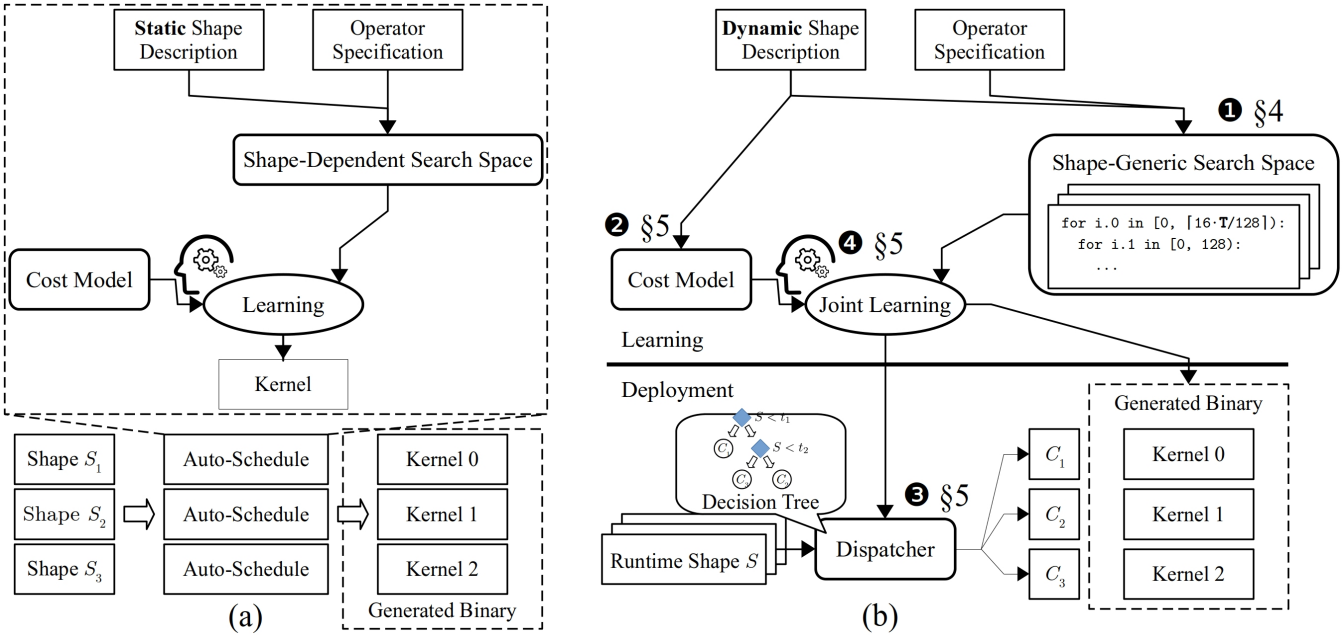

Figure 1. Code-generation comparison between (a) existing auto-schedulers and (b) DietCode. Existing approaches apply learning for each runtime shape and are not applicable for shape dependent settings. DietCode solves the problem by learning a dispatcher that categorizes the shapes into groups, and optimizes the kernel for each group by searching a shape generic search space.

$Y = XW^T, X : [1024, 768], W : [2304, 768]$

```
X = placeholder('float32', (1024, 768))
W = placeholder('float32', (2304, 768))
Y = compute((B,H), lambda b,h: sum(X[b,i]*W[h,i], [i])
```

Figure 2. A static-shape workload $Y = XW^T$ and its corresponding tensor program.

have been proposed. The input to the auto-scheduler is a compute description, which is usually in the form of a mathematical expression (see Figure 2). An auto-scheduler automatically constructs the search space by analyzing the expression and searching for the optimal schedule. Despite different in the implementation, to the best of our knowledge, all existing auto-schedulers have a common limitation: they require workloads to be *static* (i.e., all shapes known at compile-time) to perform analysis and schedule evaluation.

Unfortunately, real world workloads are not always static due to the following major reasons:

(1) *Varying Hyperparameters*: When developing machine learning models, machine learning practitioners might yearn for experimenting with different hyperparameter settings such as batch sizes and/or hidden dimensions by doing neural architecture search [38].

(2) *Dynamic by Design*: Even if machine learning practitioners choose to freeze the models for deployment, certain models, especially those in the sequence learning domain, are dynamic by their nature. For instance, machine translation [32, 33], speech recognition [5], and language modeling [10] all involve dynamic sequence lengths that are the property of the training data samples.

(3) *Varying Shapes at Different Stages*: Even within a static model, one operator class might exhibit distinct shapes at different stages of the model. For example, in the case of BERT [10], a state-of-the-art language modeling application, the hidden dimensions of a dense layer can take on the value of $\{768, 2304, 3072\}$ depending on the position of the layer in the model. Although this might not be a big issue because the number of stages tends to be small, it still opens up potential optimization opportunities.

Consequently, tensor programs in these aforementioned models have dynamic shapes that are only known at runtime. We refer to those programs as *dynamic-shape workloads* in later text (and *static-shape workloads* are used to denote tensor programs whose shapes are known at compile-time, as in Figure 2). The restriction that only static-shape workloads are accepted in the existing auto-schedulers [3, 7, 31, 36] poses challenges in applying these auto-schedulers to dynamic-shape workloads, as their compilation time can become extremely long (shown in following paragraphs). In contrast, one of the key features of DietCode is the efficient support for the dynamic-shape workloads.

We use a dynamic-shape workload example to demonstrate the key differences between vendor libraries [9, 24, 26], existing auto-schedulers [3, 7, 31, 36], and DietCode. Figure 1 illustrates the comparison on the code-generation procedure

$Y = XW^T, X : [16 \times \mathbf{T}, 768], W : [2304, 768], T \in [1, 128]$

```
X = placeholder('float32', (16*Var('T', [1, 128]), 768))
W = placeholder('float32', (2304, 768))
Y = compute((B,H), lambda b,h: sum(X[b,i]*W[h,i], [i])
```

Figure 3. A dynamic-shape workload $Y = XW^T$ (with **T** being dynamic) and its corresponding tensor program.

```
for i.0 in range(0, I0):                    ∏_i I_i = 16 · T
  for j.0 in range(0, J0):                  ∏_i J_i = 2304
    for k.0 in range(0, K0):                ∏_i K_i = 768
      sync_threads();
      X_local = X[...];  # fetch
      W_local = W[...];
      sync_threads();
      for i.1 in range(0, I1):
        for j.1 in range(0, J1):
          for k.1 in range(0, K1):
            Y_local += X_local * W_local  # compute
    for i.1 in range(0, I1):
      for j.1 in range(0, J1):
        Y[...] = Y_local;  # writeback
```

Listing 1. Sample schedule for $Y = XW^T$ in Figure 3. Note that the schedule has been simplified to help understanding.

between the two learning-based approaches. The example that we use is the dense layer $Y = XW^T$ from the BERT [10] model (see Figure 3). The dynamism of the tensor program lies in the sequence length T, which represents the length of a sentence in a corpus. Without loss of generality, we pick its range to be $[1, 128]$ for illustrative purpose. The generated schedule for the compute in Figure 3 will take on a form similar to the one shown in Listing 1. For simplicity, we only show the skeleton of the three main stages: fetch, compute, and writeback. In Figure 1, we abbreviate the schedule using the inner and outer loop tiles on the dynamic dimension $16 \cdot T$ respectively as $i.1$ and $i.0$ in the code snippets). The sample schedule will also be used in Section 4.2 to illustrate the details of the shape-generic search space design.

Vendor Libraries. Vendor libraries [9, 24, 26] are usually hand-crafted to include several optimized kernels to cover all the possible shapes, and workloads will be dispatched on the fly to the most suitable schedule (determined by hard-coded heuristics) for execution based on their shapes. When the workload shape does not entirely fit into the schedule, the runtime performance will be sub-optimal on specific hardware or workload (as much as 13× performance degradation in some extreme cases [4]). This sub-optimality might be because the shape is not a multiple of the hard-coded tile sizes, and/or because the kernel launch configuration corresponding to that shape is not ideal. Both cases are observed by prior works in the form of redundant computations [4] and latency stair-casing [35], respectively.

Existing Auto-Schedulers. In contrast to the vendor libraries, existing auto-schedulers [3, 7, 31, 36] can overcome the redundant computation and latency stair-casing problems by optimizing the schedule for each possible shape. These approaches can also be deployed to various hardware (e.g., CPUs and GPUs).

Despite these advantages, directly using existing auto-schedulers to support dynamic-shape workloads is still challenging. Figure 1.(a) provides a straightforward workflow if one would like to adopt existing auto-schedulers to deal with dynamic-shape workloads. Due to the fact that existing auto-schedulers can only operate on one static instance of the dynamic-shape workloads at a time, they have to be instantiated on every possible workload. This is depicted in Figure 1.(a) as multiple instances of the auto-schedulers. Since, by our evaluation, each instance takes roughly 1 CPU hour to optimize the schedule for a single shape on a modern machine equipped with 8 Intel® Xeon® Platinum 8259CL CPUs [27], we need more than 100 CPU hours to optimize a single dynamic-shape dense layer workload specified in Figure 3. Even worse, as each instance delivers one schedule for one shape, 128 kernels have to be compiled and reside in the generated binary. Such a huge binary size is an obstacle for deploying models on edge devices with limited resources.

The reason why the existing auto-schedulers [3, 7, 31, 36] are not practical in the case of dynamic-shape workloads is because they face the following key challenges:

(1) *Shape-Dependent Search Space*: They have search spaces that are *shape-dependent*, which makes it difficult for one workload instance (defined as an instantiation with a specific static shape of the dynamic-shape workload) to share its own search space with others. As a result, to sample a valid program for a collection of workload instances, one either has to look into the intersection of all their search spaces or sample within each of their search space individually. While the former has a very limited search space due to the intersection behavior (and can even give an empty search space in practice), the latter has an unaffordable runtime complexity that is equal to the number of workload instances.

(2) *One Schedule for One Instance*: They can only generate one schedule for one workload instance at a time, but not for multiple ones. As a result, the size of the generated binary explodes in the case when there is a large number of workload instances.

Those challenges limit the practical value of existing auto-schedulers for dynamic-shape workloads.

DietCode. To adequately address the challenges of existing auto-schedulers [3, 7, 31, 36] while preserving the ability to do auto-tuning, we propose DietCode, an automated tensor program generation framework for dynamic-shape workloads based on two key ideas:

(1) *Shape-Generic Search Space*: Sample programs from a *shape-generic* search space that can be applied not only to a single but to a collection of workload instances.

	Tuning?	Complexity	Binary Size
Vendor Libraries	✘	-	$\|C\|$
Existing AutoSched	✓	$O(\|S\|)$	$\|S\|$
DietCode	✓	$O(\|C\|)$	$\|C\|$

Table 1. Comparison between all three options in terms of the ability to do auto-tuning, the runtime complexity of tuning, and the binary size.

(2) *One Category for Multiple Instances*: Group workload instances into *categories*, and workload instances that belong to the same category share the same shape-generic program.

With those key ideas, we present the workflow of DietCode as in Figure 1.(b), where submodule ❶-❹ will be further elaborated in Section 3-5. The workflow gives DietCode the ability to operate on a per-category basis rather than a per-workload-instance basis, which is significantly more efficient. Table 1 summarizes the comparison between the three major approaches we described so far (we use $|S|$ to represent the size of workload instances and $|C|$ the number of shape-generic categories). Since $|C|$ is guaranteed to be $\leq |S|$ and in most cases $\ll |S|$, DietCode can significantly reduce the auto-scheduling time and binary size when compared with existing auto-schedulers [3, 7, 31, 36].

3 Framework Overview

DietCode is an automated tensor program generation framework for dynamic-shape workloads. Figure 1.(c) shows the workflow of DietCode framework at the search and deploy stages. The input of DietCode is the to-be-optimized compute definition of a dynamic-shape tensor program and its shape description (e.g., $Y = XW^T$ with $T \in [1, 128]$ for the example that we present in Figure (3)). DietCode addresses the challenges faced by existing auto-schedulers [3, 7, 31, 36] using the following key ideas:

(1) Sample programs from a *shape-generic* search space.

To address the challenge of *shape-dependent search space* that would result in an overall search space that is too constrained or extremely large for all the workload instances, we construct a search space that is *shape-generic* (❶ in Figure 1.(c)). In contrast to a shape-dependent search space, all programs in a shape-generic search space can be applied not only to a single but to a collection of workload instances. This therefore allows us to search for multiple instances simultaneously, which is more efficient compared with existing auto-schedulers [3, 7, 31, 36]. More details are in Section 4.

(2) Group workload instances into categories.

To address the challenge of *one schedule for one instance* that would result in a huge binary size, we group workload instances into categories and have all the instances within each category share the same shape-generic program. The sharing strategy reduces the number of programs that are needed to cover all the workload instances.

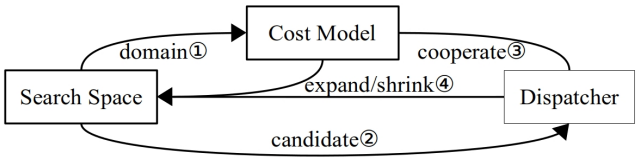

Figure 4. Relationship between the three DietCode submodules: search space, cost model, and dispatcher.

The categorization is achieved by a workload instance categorizer, which can be decomposed into two sub-components: a cost model and a dispatcher (❷ and ❸ in Figure 1.(c), respectively). While the job of the former is to accurately and efficiently evaluate how performant all the workload instances are under each category, that of the latter is to dispatch each workload instance to its preferred category. One key challenge that we address is to balance between the performance on all the workload instances and the number of categories (i.e., binary size), which are two contradictory objectives, by designing a unified cost function for the two submodules and optimizing *jointly* (see Section 5 for more details).

Joint Learning: Putting It All Together. Now that we have proposed three submodules in our design of the new auto-scheduler: (i) a shape-generic search space, (ii) a cost model, and (iii) a dispatcher, we place them together under the same framework, DietCode, and learn them *jointly* (❹ in Figure 1.(c)). We then make the key observation that all three submodules are closely coupled and therefore should be optimized together. Figure 4 summarizes the relationship between the three submodules. Note that the shape-generic programs in the search space serve as the analysis domain of the cost model and a set of potential dispatch candidates (① and ② in Figure 4); the cost model and the dispatcher work in tandem to cluster workload instances into shape-generic categories (③); they can also dynamically expand and/or shrink the search space upon seeing a workload instance that is not covered well by the current search space or a shape-generic program that is not mapped by any of the workload instances (④) (see more details in Section 5).

4 Shape-Generic Search Space

In this section, we show how to construct a shape-generic search space that is composed of micro-kernels. We then show how to sample shape-generic programs from the search space that can be efficiently applied to all the workload instances using automatic padding.

4.1 Micro-Kernels

The search space that an auto-scheduler explores determines the optimal program it can find. Existing auto-schedulers [3, 7, 31, 36] construct search spaces that are shape-dependent, which either leads to an overall search space that is too constrained or extremely large to explore. To address this

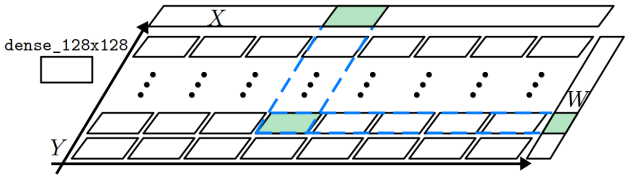

Figure 5. Micro-kernel dense_128x128 used to realize $Y = XW^T$ as in Figure 3 with $T = 64$. The horizontal and vertical axis represent the output dimensions of Y.

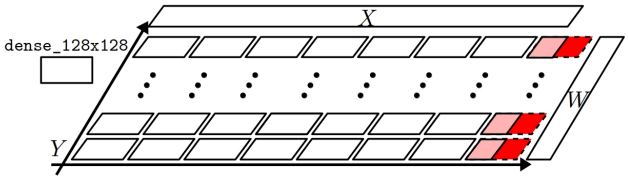

Figure 6. Micro-kernel dense_128x128 used to realize $Y = XW^T$ as in Figure 3 with $T = 60$. The tiles at the last column are padded (shown in red) to fit the micro-kernel.

challenge, DietCode constructs a search space that is shape-generic and is made up of micro-kernels, a program that computes a piece (i.e., tile) of the complete program. Those micro-kernels serve as the building blocks and are executed multiple times to carry out an instance of the dynamic-shape workload. For example, Figure 5 shows how the micro-kernel dense_128x128, whose functionality is to compute

$$Y = XW^T, X : [128, 768], W : [128, 768]$$

can be leveraged to perform an instance of $Y = XW^T$ as in Figure 3 with $T = 64$ by dissecting the complete program into 8×18 pieces along the spatial dimensions and carry out each piece (of size 128×128) individually.

Because micro-kernels only compute a piece of the complete programs, each one can be easily ported to multiple workload instances of different shapes and hence it is shape-generic. If we look at the example described above, the micro-kernel dense_128x128 can be leveraged to realize not only $T = 64$, but also $T = 1, 2, \ldots, 127, 128$. In the case when the workload cannot perfectly fit into the micro-kernel (e.g., $T = 60\%(^{128}/_{16}) \neq 0$ in the previous example, where 16 is the constant that is multiplied to T in the compute definition), *automatic padding* is applied to pad the program by the size of the micro-kernel, as we show in Figure 6.

4.2 Automatic Padding

It is common for workload instances to not fit perfectly into the micro-kernels, as is illustrated in Figure 6 where the micro-kernels at the last column are not fully materialized. In these cases, out-of-boundary `if`-checks need to be injected inside the micro-kernel body at the fetch, compute, and writeback stages (i.e., line 5, 11, 14 of Listing 1, respectively) to make sure that the program does not operate on invalid data values (see Figure 7 that is related to the example from Figure 6). However, those `if`-checks bring large performance degradation to the program (as much as 17×

```
5:  if i.0*N+tIdx < 16*60:
      X_local = X[...];
11: if i.0*I1+i.1 < 16*60:
      Y_local += ...;
14: if i.0*I1+i.1 < 16*60:
      Y[...] = Y_local;
```

(a)

```
pad X to [16x64, 768];
5:  X_local = X[...];
11: Y_local += ...;
14: Y[...] = Y_local;
unpad Y to [16x60, 2304];
```

(b)

```
    if i.0 < I0 - 1:
5:    X_local = X[...];
11:   Y_local += ...;
14:   Y[...] = Y_local;
    else:
5:    if i.0*N+tIdx < 16*60:
        X_local = X[...];
11:   if i.0*I1+i.1 < 16*60:
        Y_local += ...;
14:   if i.0*I1+i.1 < 16*60:
        Y[...] = Y_local;
```

(c)

```
5:  if i.0*N+tIdx < 16*60:
      X_local = X[...];
11: Y_local += ...;
14: if i.0*I1+i.1 < 16*60:
      Y[...] = Y_local;

# N - Number of Threads
# tIdx - Thread Index
```

(d)

Figure 7. (a) Out-of-boundary checks that correlate to Figure 6. For simplicity we only show changes made to line 5, 11, 14 in Listing 1. (b-d) Three optimization strategies: (b) global padding, (c) loop partitioning, and (d) local padding.

in some cases shown in Section 4.3) [30]. There are three possible solutions to mitigate this problem, namely:

(1) *Global Padding*: Pad the input tensors by the size of the micro-kernel and unpad the output tensors after the compute stage. This can effectively remove all the out-of-boundary checks in the compute kernel, but would require extra storage and injections of pad/unpad operators (see Figure 7.(b)).

(2) *Loop Partitioning* [30]: Partition the complete programs into regions where out-of-boundary checks are needed and regions where they are not (see Figure 7.(c)).

(3) *Local Padding*: Pad the local workspace when fetching values from the input tensors, and unpad the local workspace when writing back. This is equivalent to preserving the out-of-boundary checks at the fetch and writeback stage while removing those at the compute stage (see Figure 7.(d)).

The idea of local padding is based on the two key observations: (i) Only the out-of-boundary checks at the compute stage have the dominant impact on the runtime performance (as we will show in Section 4.3), and (ii) Invalid data values that are computed in the compute stage will be filtered out by the out-of-boundary checks in the writeback stage, hence they do not affect the program correctness.

In this work, we pick the third option because it has the merits of both being transparent (incurring no storage overhead and extra operators) and relatively good performance (as we will show in our evaluation in Section 4.3).

Now that we have presented our generic techniques to sample programs from the shape-generic search spaces, we provide the implementation details that are specific to the

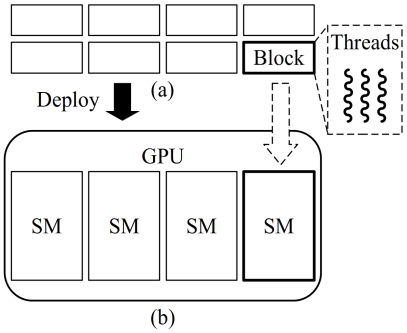

Figure 8. A simplified view of the GPU programming model and hardware architecture: (a) A GPU program is a grid of blocks of threads. (b) SM is the basic execution unit for one or multiple GPU thread blocks.

underline hardware (e.g., GPU), as the low-level architectural details affect how the initial search space is formed. For the CPUs, we use the generic techniques as are described above.

4.3 GPU-Specific Techniques

Before diving deep into how the shape-generic search space is formed on the GPUs, we present an basic introduction on the GPU architecture and programming model. We use the CUDA programming language [25] and NVIDIA GPU architectures [21] as an example, but our ideas similarly applicable to other GPUs (e.g., AMD GPUs [1]). The background information will also be used in Section 5.3 when we introduce the GPU cost model design.

The GPU programming model can be simplified as a grid of blocks of threads (Figure 8.(a)). All thread blocks are programmed to share the same CUDA kernel body, despite they might exhibit distinct behavior at runtime. During execution, thread blocks are dispatched to the execution units of the GPU, denoted as the streaming multiprocessors (SMs), in a round-robin fashion (Figure 8.(b)). It takes the GPU multiple waves (i.e. full rounds) to finish all the thread blocks. Within each wave, one or more thread blocks are dispatched to one SM. In the case when the number of thread blocks cannot fully saturate all the SMs, the GPU hardware is underutilized. Although a high SM occupancy does not necessarily lead to good performance, a low occupancy always means low performance [23]. Low occupancy can be caused by either insufficient number of thread blocks launched (Figure 9.(a)) or partial last wave (Figure 9.(b), where the final wave is not fully occupied) [20], and can be resolved by changing the inner loop tile size of the thread blocks to be smaller/larger (Figure 9.(c-d)).

Based on the programming paradigm above, we therefore observe that there are two major factors that affect the runtime performance of a GPU program: a *local* factor that corresponds to how one thread block behaves on one SM, and a *global* factor that corresponds to how the thread blocks occupy the SMs.

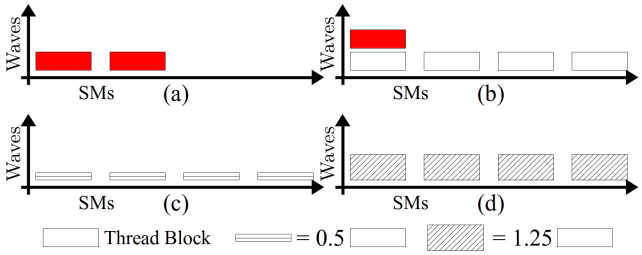

Figure 9. Underutilization of SMs caused by (a) insufficient number of thread blocks launched and (b) partial last wave. (c) Stretch the thread blocks to all the SMs by making each $0.5\times$ smaller. (d) Eliminate the partial last wave by making all thread blocks $1.25\times$ larger.

From this observation, we further notice that in the case when there is a sufficient number of thread blocks to saturate all the SMs for enough waves, the global factor will be less important or even negligible [18]. Therefore, we construct the initial search space by having an existing auto-scheduler [3, 7, 31, 36] optimize for the *largest* instance of the dynamic workload and extract the generated CUDA kernel body as the initial micro-kernel. The motivation for choosing the largest workload instance is based on the goal of minimizing the effect of SM occupancy during the optimization process (which includes underutilization and partial last wave, as in Figure 9.(a-b)), so we can decouple the global factor from the local factor.

After obtaining the initial micro-kernel M_0, we mutate its inner loop tile size T_0 and consider all the viable tile sizes within $[\frac{T_0}{2}, 2T_0]$ to cover the cases when the SM occupancy is low (Figure 9.(a-b)). The reason why we pick this heuristic is because: (i) Micro-kernels that have inner loop tile sizes within the neighbor of T_0 have similar local workspace sizes, and hence might also have decent runtime performance that is close to M_0, and (ii) $2\times$ is the theoretical maximum factor the micro-kernel can grow to resolve the partial last wave by squeezing it in its predecessor (see Figure 9.(d)).

The above mutation ($[\frac{T_0}{2}, 2T_0]$) builds up the micro-kernel search space, from which we are able to sample shape-generic programs. To adapt the sampled programs well to all the instances of the dynamic-shape workload, and also to backup our previous claim that local padding is indeed an efficient approach to optimize for out-of-boundary checks, we compare three optimization techniques (Figure 7.(b)-(d)) using the example in Figure 6 on an NVIDIA Tesla T4 GPU [22]. The micro-kernel in the figure (i.e., `dense_128x128`) is obtained by having Ansor [36], a state-of-the-art auto-scheduler, schedule for the largest instance (i.e., $T = 128$) of the dynamic-shape workload in Figure 3.

Figure 10 shows the throughput comparison (measured as TFLOPs/sec) between the three techniques (detailed methodology in Section 7), where higher throughput means better performance. We observe from the figure that local padding delivers the best performance among the three options, which

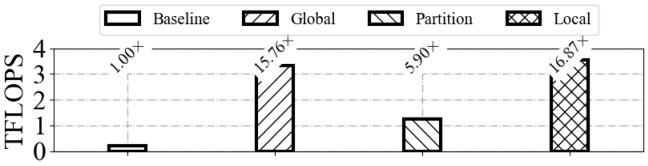

Figure 10. Comparison between optimization techniques of out-of-boundary checks for the example in Figure 6 on a Tesla T4 GPU, where *Baseline*, *Global*, *Partition*, and *Local* correspond respectively to Figure 7.(a)-(d).

is why we adopt it in DietCode to sample shape-generic programs from the micro-kernel search space.

5 Joint Learning

In this section, we first show how to build the cost model for shape-generic programs that are sampled from the search space. We then show how the cost model can be jointly optimized with the dispatcher on the workload categorization problem to balance between the runtime performance and the binary size.

5.1 Cost Model

The cost model function is to accurately and efficiently predict the performance of a shape-generic program on all the instances of a dynamic-shape workload. This is similar to the notion of cost models in prior works for static-shape workloads (e.g., Ansor [36], AutoTVM [8], see Figure 11.(a)), where the performance of the program is predicated either by extracting low-level features (e.g., how many times a certain buffer is accessed) or through direct measurements on real hardware. However, because of the major change in programs from being shape-dependent to shape-generic, the cost model has to be redesigned accordingly. Figure 11.(b) shows our cost model design for dynamic-shape workloads. Specifically, because the complete programs are represented in a shape-generic form, they have to be materialized first using the shape descriptions that users input in Figure 1.(b) (❶ in Figure 11.(b)). The performance of the materialized programs can then be predicted using either feature extraction or direct measurements, similarly to static-shape workloads.

The above procedure gives us the the cost of all workload instances W's under each shape-generic program M, which is denoted as $\mathrm{Cost}_M(W)$. We then select the shape-generic program that has the maximum

$$\sum_{i \in \mathrm{VoteFor}(M)} w_i p_i \mathrm{Cost}_M(i) \tag{1}$$

value and put it in the dispatcher (❷ in Figure 11.(b)), where w_i is the weight of the workload instance i (a reasonable choice can be the number of compute operations performed in TFLOPs), p_i is the probability for i to be executed (either specified by the shape description or assumed to be uniform),

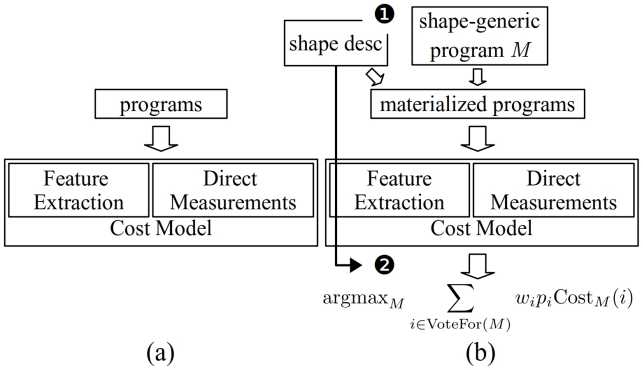

Figure 11. Cost model design for (a) static workloads and (b) dynamic workloads, where w_i, p_i, and

and $\mathrm{VoteFor}(M)$ is all workload instances that vote to be dispatched to M by the value of $\mathrm{Cost}_M(W)$ (for example, a simple voting mechanism is to have each workload instance W pick its most-preferred program, i.e., $\mathrm{argmax}_M\ \mathrm{Cost}_M(W)$). The shape-generic program selection is an iterative process: At each iteration, the shape-generic program M that has the optimal cost as in Equation 1 and the workload instances that vote for M are moved to the dispatcher as one category. This process continues until all the workload instances have been placed into certain category.

5.2 Workload Instance Categorization

We connect the cost model with the dispatcher, whose functionality is to dispatch workload instances to categories that share the same shape-generic programs, and optimize both submodules *jointly* to categorize workload instances. Before diving deep into the details of categorization, we first show why such a joint learning approach is needed.

There are two main objectives for the workload instance categorization: (i) achieve the optimal runtime performance across all the workload instances in the cost model and (ii) have the minimum number of categories in the dispatcher to maintain a small binary size. It is, however, challenging to accomplish those two objectives at the same time. The reason is because: If we only consider from the cost model standpoint and have each workload instance pick its most preferred category, we can have a large number of shape-generic programs included in the generated binary. On the other hand, if we only consider from the dispatcher standpoint and have a small binary size, certain workloads have to make compromise by not selecting the categories that work best for them. This hence poses a trade-off between the performance and the binary size, which requires *joint* optimization of the cost model (performance) and the dispatcher (binary size) submodules.

As an example, we look at the auto-scheduling the dynamic-shape workload $Y = XW^T$ in Figure 3 on a Tesla T4 GPU [22]. We observe that there is a total number of 34 shape-generic programs selected if we have all the workload instances pick

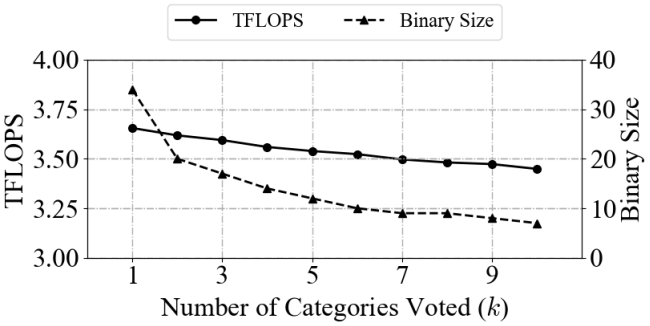

Figure 12. The performance and the binary size on the dynamic-shape workload $Y = XW^T$ in Figure 3 with various values of k on a Tesla T4 GPU [22], the number of categories voted by every workload instance.

the categories freely. On the other hand, if we push to the other extreme and restrict the number of categories to be 1, then there would be an average performance degradation of $1.11\times$ across all the workload instances compared with the previous approach.

To further study the trade-off between the performance and the binary size, we continue on the previous example and let each workload instance select k candidate categories, where k is a learnable parameter. The size of the generated binary is determined by the minimal set that covers at least one candidate for each instance. Figure 12 shows the relationship between the compute throughput (measured as $^{\text{TFLOPs}}/_{\text{sec}}$) averaged across all the workload instances and the binary size with respect to k. We observe from the figure that both the compute throughput and the binary size decrease with the increase of k, indicating that we are trading performance for smaller binary size.

Based on the above observation, we hence propose a unified objective function for workload instance categorization:

$$G = g(\text{Perf}(k), \text{BinarySize}(k)) \tag{2}$$

where Perf is the runtime performance predicted by the cost model and BinarySize is the binary size given by the dispatcher. The choice of g depends on the hardware platform and can theoretically be any function that strictly increases with Perf but decreases BinarySize (since we are optimizing for higher performance and smaller binary size). For example, in hardware platforms where binary sizes are not a concern, one might simply pick $G = \text{Perf}(k)$ so as to maximize the performance. The value k is optimized during the auto-scheduling process to maximize G.

5.3 GPU-Specific Techniques

Instead of using the generic cost models, hardware-specific properties can allow us to design specialized cost models that are more suitable for dynamic-shape workloads. Specifically, since the micro-kernel in the case of GPUs is the CUDA kernel body, we can have three separate terms in the cost model that respectively account for the following: (i) the performance of the micro-kernel, (ii) the number of times the micro-kernels are executed to form the complete program, and (iii) automatic padding. Based on this formulation, the mathematical expression of the cost model is as follows:

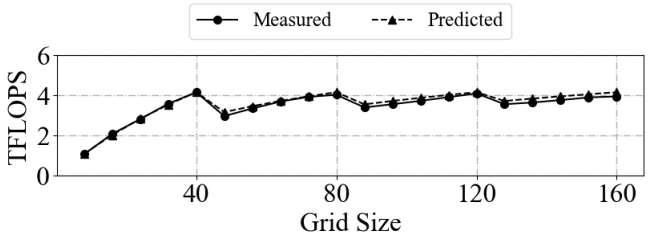

Figure 13. The compute throughput of the micro-kernel `dense_128x128` with various grid sizes on a Tesla T4 GPU. The dashed line is the predicted throughput.

$$\text{Cost}_M(W) = f_{\text{local}}(M) \cdot f_{\text{spatial}}(^W/_M) \cdot f_{\text{pad}}(W, M) \tag{3}$$

where M denotes the micro-kernel and W the workload instance. The output value $\text{Cost}_M(W)$ is the predicted performance of W using M. The three functions f_{local}, f_{spatial}, and f_{pad} correspond respectively to:

f_{local}: The program behavior local to M, which is modelled from program measurements on real hardware [8, 36]. This is achieved by executing a full wave of the micro-kernel M and measuring its compute throughput. Such a strategy eliminates the impact of SM occupancy (i.e., the global factor in Section 4.3) and allows us to see the performance of the micro-kernel (i.e., the CUDA kernel body) itself.

f_{spatial}: The behavior of executing M multiple times along the spatial dimensions to form the complete program of W. To study the effect of the term $^W/_M$ (i.e., the grid size in CUDA term because it denotes the times the CUDA kernel body is executed) on the performance. We once again take the micro-kernel `dense_128x128` evaluate it on a Tesla T4 GPU [22] with various grid sizes.

Figure 13 shows the relationship between the compute throughput (measured as $^{\text{TFLOPs}}/_{\text{sec}}$) and the grid size. From the figure, we propose to model the behavior of $f_{\text{spatial}}(^W/_M)$ using a linear regression model:

$$f_{\text{spatial}}(^W/_M) = k\frac{\text{pad}(^W/_M, \text{SMs})}{^W/_M} + b \tag{4}$$

where the pad primitive pads the grid size by the number of SMs on the GPUs (40 on a Tesla T4 GPU [22]). Both k, b are learnable parameters. In fact, because the compute throughput peaks every time an exact number of full waves are executed, we can further derive:

$$\text{Cost}_M(W) = f_{\text{local}}(M), \text{ when } \frac{\text{pad}(^W/_M, \text{SMs})}{^W/_M} = 1 \tag{5}$$
$$\Rightarrow k + b = 1 \Rightarrow b = 1 - k$$

Applying Equation 4-5 to the measurements in the Figure, we obtain $k \approx 0.5$ on a Tesla T4 GPU [22].

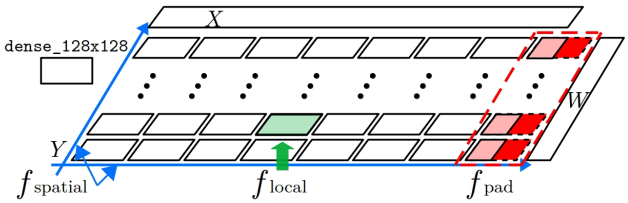

Figure 14. f_{local}, f_{spatial}, and f_{pad} account for different program behaviors (example taken from Figure 6).

f_{pad}: The amount of automatic padding made, which can be modelled as $^{\text{pad}(W,M)}/_{W}$ and is hardware-agnostic (the pad primitive pads W by the size of M).

Figure 14 illustrates the correlation between those three terms and how the micro-kernels form the complete program, where we can see the one-to-one correspondence between the two. The specialized cost model has the strength of being accurate. This can be seen from the predicted curve in Figure 13 that almost overlaps with the measured curve. Furthermore, since the cost model has a separate term for the micro-kernel itself (i.e., f_{local}), we can early prune micro-kernels that do not perform well from the search space. For example, in the case of the dynamic-shape workload $Y = XW^T$ in Figure 3, early pruning removes 28% of all the micro-kernels in the search space, making the overall auto-scheduling process 1.39× more efficient.

Although we have been using the dynamic-shape workload $Y = XW^T$ and the micro-kernel `dense_128x128` so far to demonstrate our ideas, our evaluations in Section 7 will show that the same cost model applies to other micro-kernels and dynamic-shape workloads as well.

6 Methodology

Infrastructure. Our main compute platform is a single Amazon G4 instance, which is equipped with 8 Intel® Xeon® Platinum 8259CL CPUs [27] and 1 NVIDIA Tesla T4 GPU [22], with CUDA 11.0 [25], cuDNN 8.0.5 [9], and TVM v0.8.dev0 [7].

Applications. We evaluate DietCode by auto-scheduling on the dense layers from BERT-base [10], a state-of-the-art language modelling application, with dynamic sequence lengths and dynamic hidden dimensions. We also show results on the conv2d layer from ResNet-50 [15], a state-of-the-art image classification model, with dynamic batch sizes.

Baselines. We compare our approach, DietCode, with two baselines: the vendor libraries (cuBLAS [24] and cuDNN [9] on the GPUs), which we refer to as *Vendor*, and the state-of-the-art auto-scheduler implementation in TVM [7, 36] that targets static-shape workloads, which we refer to as *TVM*.

Metrics. We show the results on the compute throughput, which is measured as $^{\text{TFLOPS}}/_{\text{sec}}$ and are all normalized with respect to *TVM* under each shape configuration. As it is impractical to complete the entire auto-scheduling procedure on *TVM* (it can take more than 100 CPU hours to complete a single dynamic-shape workload), we sample shape configurations uniformly within the dynamic range to do the comparison on the performance, and estimate that on the auto-scheduling time.

7 Evaluation

Dense layer with dynamic sequence lengths. We compare the performance between *Vendor*, *TVM*, and DietCode on the dense layer with dynamic sequence lengths (denoted as T) as in Figure 3. Figure 15 shows the comparison of the compute throughput under different sequence lengths. We observe from the figure that the performance of DietCode is on average 18% better than that of *TVM* and is on par with that of *Vendor*. The reason why DietCode has better performance than *TVM* is because (i) DietCode has different formulations of the search space (see Section 4.3) and (ii) the search space that DietCode defined contains schedules that have higher performance and was overlooked by *TVM*. Furthermore, we also notice that there is only a 1.4% gap in between the empirical performance achieved by DietCode compared and the performance predicted by the specialized cost model, indicating that our design of the specialized cost model in Section 5.3 is accurate.

Moreover, DietCode is more efficient compared with *TVM* on the auto-scheduling procedure, as it takes about 1 hour to complete the entire process, which is 128× faster than *TVM*. In terms of the binary size, the generated binary of DietCode incorporates 20 shape-generic programs, which is 6.4× smaller than that of *TVM* (128 programs in total).

Dense layer with dynamic hidden dimensions. To evaluate the effectiveness of DietCode on other dynamic axes, we extract all the hidden dimensions from BERT-base [10] (denoted as (I, H)) and compare the performance between *Vendor*, *TVM*, and DietCode on the dense layer

$$Y = XW^T, X : [16 \times 64, \mathbf{I}], W : [\mathbf{H}, \mathbf{I}] \tag{6}$$

Figure 16 illustrates the comparison of the compute throughput under different hidden dimensions. We observe from the figure that the performance of DietCode is on average 8.9% better than that of *TVM* but is slightly behind *Vendor* by 2.8%. We also observe that there is only a 2.3% gap between the achieved and the predicted performance. In terms of the auto-scheduling time, DietCode is about 4× more efficient than *TVM*.

Convolution layer with dynamic batch sizes. To evaluate the effectiveness of DietCode on other dynamic-shape workloads, we select the conv2d layer from ResNet-50 [15] and compare the performance between *Vendor*, *TVM*, and DietCode on the layer with dynamic batch sizes within $[16, 32]$ range (denoted as B). Figure 17 illustrates the comparison of the compute throughput under different batch sizes. We observe from the figure that the performance of DietCode is on average 5% better than *TVM* and *40%* better than *Vendor*.

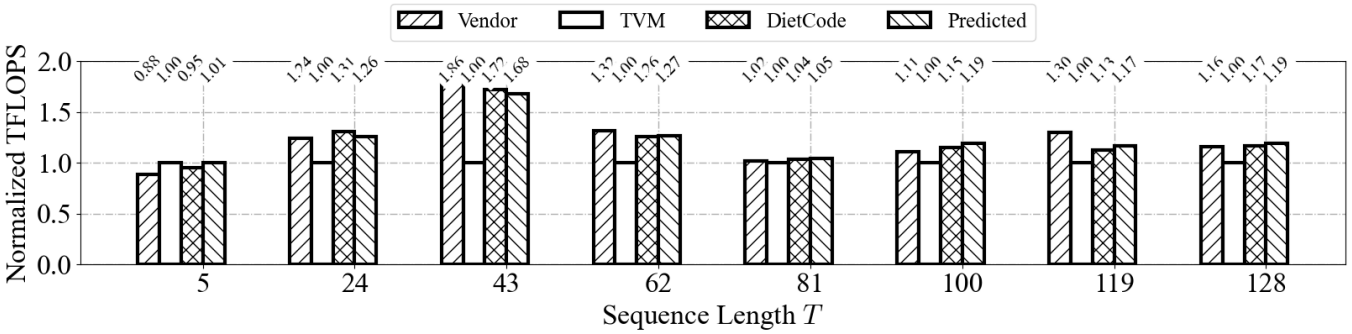

Figure 15. Performance comparison between *Vendor*, *TVM*, and DietCode on the dense layer with dynamic sequence lengths.

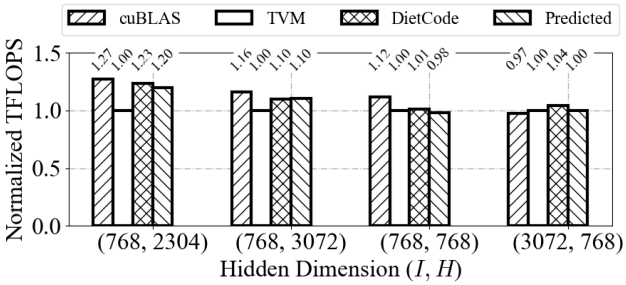

Figure 16. Performance comparison between *Vendor*, *TVM*, and DietCode on the dense layer with dynamic hidden dimensions extracted from BERT.

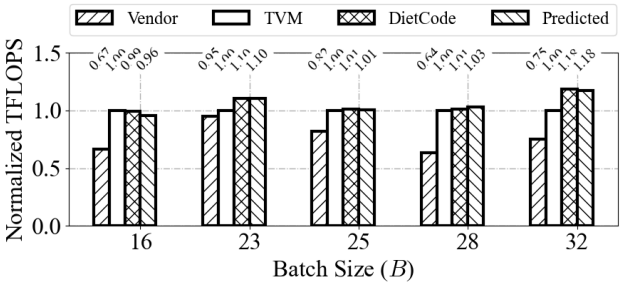

Figure 17. Performance comparison between *Vendor*, *TVM*, and DietCode on the conv2d layer with dynamic batch sizes.

In terms of the auto-scheduling time, DietCode is about 17× more efficient than *TVM*.

CPU Evaluations. To evaluate the effectiveness of DietCode on other hardware platforms other than the GPUs, we repeat the experiments of the dense layer with dynamic sequence lengths (Figure 15) and the conv2d layer with dynamic batch sizes (Figure 17) and compare between *TVM* and DietCode on the CPU. Figure 18 and 19 show the results of the two experiments respectively. We observe from the two figures that DietCode is able to achieve 97% and 88% of the performance of *TVM*, and is still able to reduce the auto-scheduling time by about 128× and 17×, respectively.

Summary. From the evaluations that we have presented, we conclude that DietCode achieves comparable or better performance with the state-of-the-art auto-schedulers [7, 36] on various dynamic axes, dynamic-shape workloads, and hardware platforms and is even on par with vendor libraries

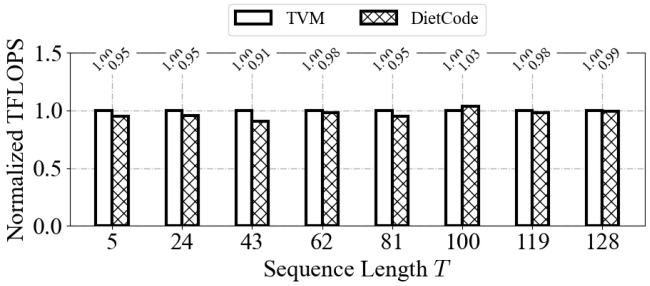

Figure 18. Performance comparison between *Vendor*, *TVM*, and DietCode on the dense layer with dynamic sequence lengths on the CPU.

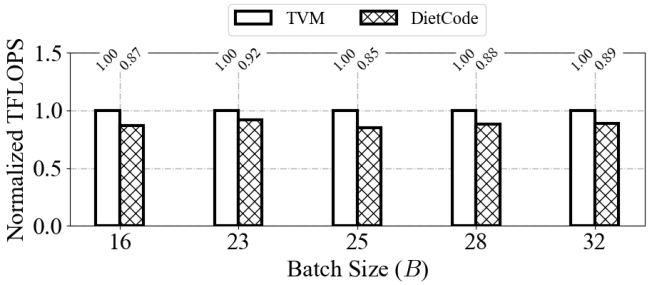

Figure 19. Performance comparison between *Vendor*, *TVM*, and DietCode on the conv2d layer with dynamic batch sizes on the CPU.

on some workloads. It can reduce the auto-scheduling time and the binary size by an order of magnitude, and therefore is more practical compared with existing auto-schedulers [3, 7, 31, 36] on dynamic-shape workloads.

8 Related Works

In this work, we present DietCode, an automated tensor program generation framework for dynamic-shape workloads. DietCode address the challenges of auto-scheduling dynamic-shape workloads by constructing a shape-generic search space and grouping workload instances into categories, which is unseen in exist auto-schedulers that mostly target static-shape workloads (Ansor [36], AutoTVM [7], Halide auto-scheduler [3], TensorComprehensions [31]).

Compilers for Tensor Programs. Halide [29] designs a scheduling language for image processing pipeline, which

can represent tensor programs and execute loop optimizations with built-in primitives. MLIR [16] designs a compiler infrastructure to support defining dialect and sharing optimizations across dialects. Although DietCode is implemented in TVM [7], it is possible for its idea to be applied to these two frameworks as well.

Search-based Auto-tuning. ProTuner [13] uses Monte Carlo tree search to tune the Halide programs. AutoTVM [8] and FlexTensor [37] leverage program templates to define the search space and guide the search procedure. Those auto-tuners face the same challenges with existing auto-schedulers and hence are not practical for dynamic-shape workloads.

Dynamic Neural Networks. Dynamic batching is a common graph level optimization adopted by frameworks such as DyNet [19], Cavs [34], BatchMaker [12], and TensorFlow Fold [17], for addressing cases when the batch axis is dynamic. Nimble [30] designs a compiler with features like dynamic type system and virtual machine to represent and execute dynamic neural networks. Cortex [11] is a compiler-based framework focus on recursive neural networks. The optimizations proposed in those works are mostly orthogonal with DietCode, which focuses on the auto-scheduling the code-generation process, and can leverage DietCode for effective operator code generation.

9 Conclusions

In this work, we propose DietCode, an automated tensor program generation framework for dynamic-shape workloads. Our evaluation on DietCode shows that it can significantly reduce the auto-scheduling time and generated binary size by $128\times$ and $6.4\times$ maximum, respectively. It is able to achieve matched performance and even outperform the state-of-the-art auto-schedulers by 18% maximum in terms of runtime performance. We hope that DietCode would become an efficient platform for further research on efficient system design for key machine learning applications.

References

[1] Tor M. Aamodt, Wilson Wai Lun Fung, and Timothy G. Rogers. 2018. *General-Purpose Graphics Processor Architectures.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00848ED1V01Y201804CAC044

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] 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. https://doi.org/10.1145/3306346.3322967

[4] Alibaba Group's Machine Translation Platform team and PAI-Blade team. 2018. *Bringing TVM into TensorFlow for Optimizing Neural Machine Translation on GPU.* https://tvm.apache.org/2018/03/23/nmt-transformer-optimize

[5] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings, Vol. 48)*, Maria-Florina Balcan and Kilian Q. Weinberger (Eds.). JMLR.org, 173–182. http://proceedings.mlr.press/v48/amodei16.html

[6] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[8] Tianqi Chen, Lianmin Zheng, Eddie Q. Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to Optimize Tensor Programs. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 3393–3404. https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html

[9] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186. https://doi.org/10.18653/v1/n19-1423

[11] Pratik Fegade, Tianqi Chen, Phil Gibbons, and Todd Mowry. 2020. Cortex: A Compiler for Recursive Deep Learning Models. *CoRR* abs/2011.01383 (2020). arXiv:2011.01383 https://arxiv.org/abs/2011.01383

[12] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency RNN inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 31:1–31:15. https://doi.org/10.1145/3190508.3190541

[13] Ameer Haj-Ali, Hasan Genc, Qijing Huang, William Moses, John Wawrzynek, Krste Asanovic, and Ion Stoica. 2020. ProTuner: Tuning Programs with Monte Carlo Tree Search. arXiv:2005.13685 https://arxiv.org/abs/2005.13685

[14] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. 2017. Mask R-CNN. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*. IEEE Computer Society, 2980–2988. https://doi.org/10.1109/ICCV.2017.322

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778. https://doi.org/10.1109/CVPR.2016.90

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. , 2–14 pages. https://doi.org/10.1109/CGO51591.2021.9370308

[17] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=ryrGawqex

[18] Paulius Micikevicius. 2012. *GPU Performance Analysis and Optimization*. https://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf

[19] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. arXiv:1701.03980 http://arxiv.org/abs/1701.03980

[20] NVIDIA. 2015. *Achieved Occupancy*. https://docs.nvidia.com/gameworks/index.html#developertools/desktop/analysis/report/cudaexperiments/kernellevel/achievedoccupancy.htm

[21] NVIDIA. 2018. *NVIDIA Turing GPU Architecture*. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[22] NVIDIA. 2020. *NVIDIA T4 70W Low Profile PCIe GPU Accelerator*. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-product-brief.pdf

[23] NVIDIA. 2021. *Best Practices Guide :: CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/

[24] NVIDIA. 2021. *cuBLAS :: CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/cublas

[25] NVIDIA. 2021. *Programming Guide :: CUDA Toolkit Documentation*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

[26] oneAPI open source projects. 2021. *oneAPI Deep Neural Network Library (oneDNN)*. https://github.com/oneapi-src/oneDNN

[27] PassMark Software. 2020. *Intel Xeon Platinum 8259CL @ 2.50GHz*. https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Platinum+8259CL+%40+2.50GHz&id=3671

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[29] 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. *SIGPLAN Not.* 48, 6 (June 2013), 519–530. https://doi.org/10.1145/2499370.2462176

[30] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. 2021. Nimble: Efficiently Compiling Dynamic Neural Networks for Model Inference. In *Proceedings of Machine Learning and Systems*, Vol. 3. https://proceedings.mlsys.org/paper/2021/hash/4e732ced3463d06de0ca9a15b6153677-Abstract.html

[31] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2020. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4 (2020), 38:1–38:26. https://doi.org/10.1145/3355606

[32] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008. https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[33] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR* abs/1609.08144 (2016). arXiv:1609.08144 http://arxiv.org/abs/1609.08144

[34] Shizhen Xu, Hao Zhang, Graham Neubig, Wei Dai, Jin Kyu Kim, Zhijie Deng, Qirong Ho, Guangwen Yang, and Eric P. Xing. 2018. Cavs: An Efficient Runtime System for Dynamic Neural Networks. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 937–950. https://www.usenix.org/conference/atc18/presentation/xu-shizen

[35] Fuxun Yu, Zirui Xu, Tong Shen, Dimitrios Stamoulis, Longfei Shangguan, Di Wang, Rishi Madhok, Chunshui Zhao, Xin Li, Nikolaos Karianakis, Dimitrios Lymberopoulos, Ang Li, Chenchen Liu, Yiran Chen, and Xiang Chen. 2020. Towards Latency-aware DNN Optimization with GPU Runtime Analysis and Tail Effect Elimination. *CoRR* abs/2011.03897 (2020). arXiv:2011.03897 https://arxiv.org/abs/2011.03897

[36] 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 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 863–879. https://www.usenix.org/conference/osdi20/presentation/zheng

[37] 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 859–873. https://doi.org/10.1145/3373376.3378508

[38] Barret Zoph and Quoc V. Le. 2017. Neural Architecture Search with Reinforcement Learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r1Ue8Hcxg