DIETCODE: AUTOMATIC OPTIMIZATION FOR DYNAMIC TENSOR PROGRAMS

Bojian Zheng [*1 2 3] **Ziheng Jiang** [*4] **Cody Hao Yu** [2] **Haichen Shen** [5] **Josh Fromm** [6]
Yizhi Liu [2] **Yida Wang** [2] **Luis Ceze** [7 6] **Tianqi Chen** [8 6] **Gennady Pekhimenko** [1 2 3]

ABSTRACT

Achieving high performance for compute-intensive operators in machine learning (ML) workloads is a crucial but challenging task. Many ML and system practitioners rely on vendor libraries or auto-schedulers to do the job. While the former requires large engineering efforts, the latter only supports static-shape workloads in existing works. It is difficult, if not impractical, to apply existing auto-schedulers directly to dynamic-shape workloads, as this leads to extremely long auto-scheduling time.

We observe that the key challenge faced by existing auto-schedulers when handling a dynamic-shape workload is that they cannot construct a unified search space for all the possible shapes of the workload, because their search space is shape-dependent. To address this, we propose *DietCode*, a new auto-scheduler framework that efficiently supports dynamic-shape workloads by constructing a *shape-generic* search space and cost model. Under this construction, all shapes *jointly* search within the same space and update the same cost model when auto-scheduling, which is therefore more efficient compared with existing auto-schedulers.

We evaluate *DietCode* using state-of-the-art machine learning workloads on a modern GPU. Our evaluation shows that *DietCode* has the following key strengths when auto-scheduling an entire model end-to-end: (1) reduces the auto-scheduling time by up to $5.88\times$ less than the state-of-the-art auto-scheduler on the uniformly sampled dynamic shapes ($94.1\times$ estimated if all the possible shapes are included), (2) improves performance by up to 69.5% better than the auto-scheduler and 18.6% better than the vendor library. All these advantages make *DietCode* an efficient and practical solution for dynamic-shape workloads.

1 INTRODUCTION

Deep neural networks (DNNs) form an important class of ML algorithms (He et al., 2016; Vaswani et al., 2017; Amodei et al., 2016; Devlin et al., 2019). They are made of tensor operators which are often executed for tens of thousands of iterations to capture the pattern of the training data samples. Because of this, it is crucial to guarantee the efficiency of every operator during the execution on real hardware, as even an improvement of 5% in performance could mean a cost reduction of up to $4000 in training a single DNN model end-to-end (Sharir et al., 2020).

However, it is a challenging task to implement all operators efficiently, because extensive expertise is needed across the software and hardware stack in order for those operators to be programmed efficiently. Therefore, most state-of-the-art ML frameworks rely on heavily optimized, hand-crafted vendor libraries (e.g., oneDNN (oneAPI, 2021) on Intel CPUs; cuDNN (Chetlur et al., 2014) and cuBLAS (NVIDIA, 2021) on NVIDIA GPUs) to provide highly optimized operators (Abadi et al., 2016; Paszke et al., 2019; Chen et al., 2015). Despite delivering high performance, the development time and release cycle for vendor libraries to support new operators on new hardware platforms could be extremely long (e.g., there is roughly a one year gap between each cuBLAS major release (NVIDIA, 2021b)).

To address these challenges, auto-scheduler frameworks have been proposed to bridge the gap between high-level compute definition and low-level implementation details (e.g., Ansor (Zheng et al., 2020a), TVM (Chen et al., 2018a), Halide auto-scheduler (Adams et al., 2019), and Tensor Comprehensions (Vasilache et al., 2020)). Although being powerful, the existing auto-schedulers can only support static-shape workloads where all shapes have to be known at compile-time because they need this information to construct the search space that describes all possible operator implementations (i.e., schedules) under consideration.

[*]Equal contribution [1]University of Toronto [2]Amazon Web Services [3]Vector Institute [4]ByteDance [5]Scroll Tech [6]OctoML [7]University of Washington [8]Carnegie Mellon University. Correspondence to: Bojian Zheng <bojian@cs.toronto.edu>.

Proceedings of the 5th MLSys Conference, Santa Clara, CA, USA, 2022. Copyright 2022 by the author(s).

However, real world workloads can take on different shapes from time to time, such as in the following scenarios: (1) neural architecture search (Zoph & Le, 2017), (2) dynamic by design (e.g., models in the sequence learning domain have to dynamically fit to the input sequence length at runtime (Wu et al., 2016; Vaswani et al., 2017; Amodei et al., 2016; Devlin et al., 2019)), and (3) varying shapes depending on the layer position in the model (He et al., 2016; Devlin et al., 2019). A straightforward way for the existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) to support those cases is by tuning for all the possible shapes, but this can take days to complete (i.e., up to 42 hours for a single BERT (Devlin et al., 2019) layer on a modern CPU), which is therefore not a practical solution.

In this work, we observe that if we group the same type of operators with different shapes together as *a single dynamic-shape workload* and only auto-schedule the workload *once*, we can significantly reduce the overall auto-scheduling time. However, existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) cannot collectively schedule for different shapes of the same operator directly, because those auto-schedulers construct different search spaces for different shapes. To address this shape-dependent search space construction, we propose *DietCode*, a new auto-scheduler framework that efficiently supports dynamic-shape workloads by constructing a *shape-generic* search space that is made up of *micro-kernels*, incomplete programs that carries out a tile of the complete computation. Because every micro-kernel can be ported to every shape of the workload, this gives *DietCode* a unified space to search for efficient schedules for all the shapes.

With this change in the search space foundation, *DietCode* designs a cost model that is made up of a convoluted shape-generic component and a simple shape-dependent component to guide its exploration in the search space. While the former requires extracting program features from the micro-kernels (e.g., loop structures, memory access patterns) and constantly learning on real hardware measurements to be accurate, the latter does not. This design allows for efficient cost model predictions, as to adapt a micro-kernel to all the possible shapes of the workload, only the shape-dependent component needs to be updated for each shape.

We highlight that *DietCode* has all the shapes of a workload explore the same search space and learn the same cost model *jointly*, therefore making the auto-scheduling process have a constant-time runtime complexity with respect to the number of shapes, which is much more efficient compared with the existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020).

Our contributions can be summarized as follows:

(1) We find and address the key challenges in making auto-scheduling practical for the important class of dynamic-shape workloads by using a *shape-generic* search space.

(2) We build *DietCode*, a new auto-scheduler framework for dynamic-shape workloads that adopts the *joint learning* approach that optimizes all the possible shapes of the workload collectively within the same shape-generic search space, and learns the same shape-generic cost model.

(3) We evaluate *DietCode* on BERT (Devlin et al., 2019), a state-of-the-art language modelling application, and show that *DietCode* can greatly reduce the auto-scheduling time by up to $5.88\times$ compared with Ansor (Zheng et al., 2020a), a state-of-the-art auto-scheduler, on the uniformly sampled dynamic shapes ($94.1\times$ estimated if all the possible shapes are included). At the same time, *DietCode* improves the runtime performance by up to 69.5% better than Ansor and 18.6% better than the vendor library (cuBLAS and cuDNN (NVIDIA, 2021; Chetlur et al., 2014)) on a modern GPU.

2 Background and Motivation

In this section, we present an overview of the key characteristics and shortcomings of both vendor libraries (oneAPI, 2021; Chetlur et al., 2014; NVIDIA, 2021) and state-of-the-art auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) on dynamic-shape workloads to motivate *DietCode*.

2.1 Vendor Libraries and Existing Auto-Schedulers

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 (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and MXNet (Chen et al., 2015)) rely on heavily optimized, hand-crafted vendor libraries (e.g., oneDNN (oneAPI, 2021) on Intel CPUs; cuDNN (Chetlur et al., 2014) and cuBLAS (NVIDIA, 2021) on NVIDIA GPUs). Despite providing high performance for a pool of operators, those libraries come with a non-trivial price: developing the efficient implementations often requires (1) in-depth expertise across the software and hardware stacks; and (2) significant amount of engineering effort which lead to prolonged release cycle for vendor libraries to support new operators on new hardware.

To deliver high-performance tensor programs on diverse hardware in a relatively short time, auto-scheduler frameworks (e.g., Ansor (Zheng et al., 2020a), TVM (Chen et al., 2018a), Halide auto-scheduler (Adams et al., 2019), and Tensor Comprehensions (Vasilache et al., 2020)) have been proposed. The input to the auto-scheduler is a tensor expression, which includes an operator specification and input shape descriptions (see Figure 2(a)). The auto-scheduler

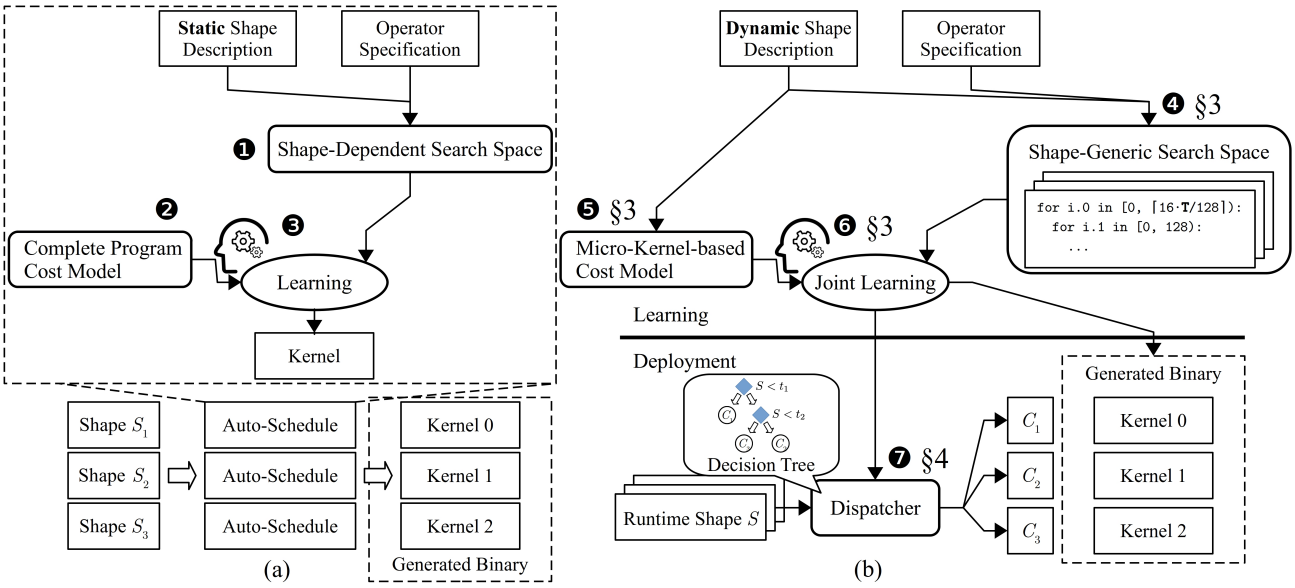

Figure 1. Code-generation comparison between (a) existing auto-schedulers and (b) *DietCode*. Existing approaches auto-schedule each shape individually. *DietCode* solves the problem by having all shapes jointly search within the same shape-generic search space and update the same micro-kernel cost model.

$$Y = XW^{\top} \quad \begin{array}{l} \text{(a)} \quad X : [1024, 768], W : [2304, 768] \\ \text{(b)} \quad X : [16 \times \mathbf{T}, 768], W : [2304, 768], \\ \qquad \mathbf{T} \in [1, 128] \end{array}$$

Figure 2. (a) A static-shape workload where all shapes are compile-time constants. (b) A dynamic-shape workload ($\mathbf{T}$ is dynamic).

automatically constructs high-performance programs by analyzing the expression. This is done by first formulating a search space that consists of numerous possible implementations (i.e., *schedules*, ❶ in Figure 1(a)) derived from the tensor expression, and then building a cost model that can predict the runtime performance of each schedule (❷). Throughout the auto-scheduling process, the cost model is constantly updated by real hardware measurements and used to guide the exploration within the search space (❸). The schedule delivered at the end of the process can lead to operator implementations that are up to $3.8\times$ faster than their counterparts in vendor libraries.

2.2 Dynamic Tensor Programs

Despite having different implementations, to the best of our knowledge, those existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) have a common limitation: they require workloads to be *static* (i.e., all shapes must be known at compile time) to perform analysis. Such a restriction makes it hard and even impractical to use them for many real world scenarios, as large amount of time has to be spent on auto-scheduling (can be around 42 hours on a single operator as we will demonstrate later). Those scenarios include:

(1) *Neural architecture search (Zoph & Le, 2017)*: NAS aims to deliver a DNN that fits the given training dataset in a predefined search space of hyperparameters (e.g., batch sizes, hidden dimensions). Each hyperparameter can lead to a network of layers with distinct shapes. To achieve the best search results, a large number of network architectures have to be explored (e.g., $12,800$ architectures are examined in Zoph & Le (2017)).

(2) *Dynamic by design*: Although machine learning practitioners usually choose to freeze the models for deployment, certain models, especially those in the sequence learning domain, are dynamic by nature. For instance, machine translation (Wu et al., 2016; Vaswani et al., 2017), speech recognition (Amodei et al., 2016), and language modeling (Devlin et al., 2019) all involve dynamic sequence lengths that vary depending on the input data samples. Each sequence length again corresponds to a network of layers with distinct shapes. For example, in the case of BERT (Devlin et al., 2019), a state-of-the-art language modelling application, the sequence length can take on any value from 1 to 128.[1]

(3) *Varying shapes with different layer positions*: Even if the model is static, one operator class can exhibit distinct shapes at different positions of the model. Take the BERT model (Devlin et al., 2019) again as an example: the hidden dimension of a dense layer can take on the value of $\{768, 2304, 3072\}$ depending on where it is in the model.

Consequently, tensor programs of the same operator type in these scenarios can take on various shapes. The restriction that only static-shape workloads are accepted by the existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a;

[1] Although padding can be used to pad the dynamic sequence lengths to static values, this nevertheless leads to performance degradation (can be as much as $2\times$ (Kosec et al., 2021)).

Adams et al., 2019; Vasilache et al., 2020) poses challenges in using them in these scenarios, as their compilation time can be extremely long (roughly estimated to be 4200, 42, and 1 hour(s) respectively on a modern CPU for a single layer type, 20× more for an entire network (Zheng et al., 2020a)). We observe, however, that if those programs can be grouped together as a single *dynamic-shape workload* and auto-scheduled once, the compilation time can be significantly shortened by up to 5.88× (94.1× projected if all the possible shapes are included).

2.3 Why A New Auto-Scheduler Framework?

This work proposes a new auto-scheduler framework that efficiently supports dynamic-shape workloads. However, a reasonable question to ask is why such a new framework is needed in the first place, or in other words, why existing solutions are not adequate. More precisely, could vendor libraries (oneAPI, 2021; Chetlur et al., 2014; NVIDIA, 2021) or existing auto schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) plus their extensions be used instead? Below, we outline some of the key challenges with these alternative approaches to highlight why existing solutions are not satisfactory in practice. The example that we use is the dense layer $Y = XW^\top$ from the BERT-base (Devlin et al., 2019) model under a commonly used batch size of 16 (see Figure 2(b)) (Zheng et al., 2020a). The dynamism of the tensor program lies in the sequence length $\mathbf{T}$, 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.

Vendor libraries (oneAPI, 2021; NVIDIA, 2021; Chetlur et al., 2014) are 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 pathological cases (Alibaba, 2018)). The reason for this sub-optimality can be padding the shape to the multiple of the hard-coded kernel tile sizes, which introduces padding overhead and decreases the computation efficiency. Previous works observe such situations in the form of redundant compute operations (Alibaba, 2018) and latency stair-casing (Yu et al., 2020). Those inefficiencies hence push for the need to extend the hand-crafted kernels in the vendor libraries, which is nevertheless a nontrivial task due to the complexity of those kernels.

Existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) can overcome the above shortcomings by optimizing the schedule for each possible shape, but directly applying them to dynamic-shape workloads can be challenging. Figure 1(a) provides a straightforward workflow if one would like to adopt existing auto-schedulers to handle dynamic-shape workloads. As existing auto-schedulers can only operate on one static shape at a time, every possible static shape needs to be auto-scheduled individually. This is illustrated in Figure 1(a) as multiple instances of the auto-schedulers (next to S_i's). Since each instance takes roughly 0.33 CPU hour to optimize the schedule for a single shape on a modern machine equipped with 16 Intel® Xeon® Platinum 8259CL CPUs (PassMark Software, 2020), we empirically estimate 42 CPU hours to optimize a single dynamic-shape dense layer workload specified in Figure 2(b).

Although extensions based on the existing auto-schedulers have been proposed to support dynamic-shape workloads (Yu, 2019; Shen et al., 2021; Wang, 2019), to our best knowledge, those prior works all face challenges of being not fully automatic and/or producing low-performance tensor programs: (1) Selective tuning (Yu, 2019) heuristically groups static-shape workloads into clusters and optimizes for each cluster separately, which involves human expertise and therefore is not fully automatic. (2) Nimble (Shen et al., 2021) auto-schedules a dynamic-shape workload by letting it take on a large shape and apply its schedule generically to all shapes. However, as our evaluation in Section 5 will show, a schedule that is optimal on the large shape might be sub-optimal on the others. (3) Bucketing (Wang, 2019) splits the dynamic range into small sub-ranges (e.g., $T \in [1, 128]$ in Figure 2(b) into $T \in [1, 8], [9, 16], \ldots$) and tunes only for the maximum value within each sub-range while padding the rest to the maximum. There are, however, two sources of inefficiency with this approach: (i) It requires extra padding and slicing operations before and after the main computation respectively that bring performance and storage penalty. (ii) Performing computation on the padded tensors results in low efficiency, as many computations are useless (e.g., in the previous bucketing example, $T = 8$ requires 7× more computations than $T = 1$). These weaknesses hinder the use of existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) to support dynamic-shape workloads in a practical and efficient way, which therefore motivates for a new auto-scheduler design.

3 DIETCODE: KEY IDEAS

We further dive into the challenges of the existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) to understand why they cannot be easily improved to be efficient on dynamic-shape workloads. We then present the key ideas behind the proposals of our design.

3.1 Shape-Generic Search Space

```
for (i = 0; i < T; ++i)
  A[i] = ...
```
(a)

```
for (io = 0; io < ⌈T/t⌉; ++io)
  for (ii = 0; ii < t; ++ii)
    A[io*t+ii] = ...
```
(b)

Figure 3. A loop tile schedule (b) transformed from the input tensor expression (a), where T is the user-provided loop extent and t is a tunable tile size parameter by the auto-scheduler.

Instead of tuning every possible shape one by one using existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020), we observe that the search spaces of different shapes can in fact overlap and hence can potentially form a **shape-generic search space**. Take the loop tile schedule in Figure 3(b) that is transformed from the tensor expression in (a) as an example. When tuning for the tile size t, existing auto-schedulers only consider all factors of T as candidates, meaning that $t \in \{1, 7, 49\}$ when $T = 49$ and $t \in \{1, 2, 5, \ldots, 50\}$ when $T = 50$. However, any integer smaller than T are in fact all valid candidates, and they form a shape-generic search space that can be used by all possible T's. We observe from this example that the search space construction is essentially to make a trade-off between a large, unified search space for all the possible shapes and the challenge of addressing extra boundary checks (which are needed in the case of non-perfect tiling, e.g., $t = 10$ when $T = 49$). As we will show in Section 4.1, we could avoid the boundary checking overhead by carefully handling the non-perfect tiling in the code generation. Accordingly, non-factor tile candidates can also lead to the same or even better performance than the factor candidates. We show how non-factor candidates can positively affect the performance of the generated tensor programs in Section 5.

From these merits, we construct **a shape-generic search space that consists of micro-kernels** (❹ in Figure 1(b)), an incomplete program that carries out a tile of the complete computation, to efficiently support dynamic-shape workloads. We use the hardware constraints (e.g., the maximum number of threads, the amount of shared and local memory) rather than the shape information to determine the micro-kernel candidates. Those candidates serve as the building blocks and are executed repeatedly to carry out a workload instance (defined as an static-shape instance of the dynamic-shape workload). For example, Figure 4 shows how the micro-kernel `dense_128x128`,[2] which evaluates

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

can be leveraged to perform an instance of $Y = XW^\top$ as in Figure 2(b) with $T = 64$. This is achieved by dissecting the complete program into 8×18 pieces along the spatial dimensions (since $16 \cdot T = \underline{8} \times 128$, $2304 = \underline{18} \times 128$) and carry out each piece individually. Because micro-kernels only realize a piece of the complete computation, each one

[2]Although we use the dense layer and the micro-kernel size 128×128 here to demonstrate how micro-kernels work, the exact same idea applies to other workloads and/or micro-kernel sizes.

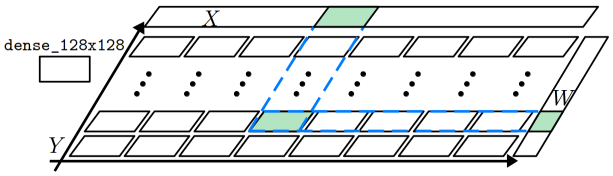

Figure 4. Micro-kernel `dense_128x128` used to realize $Y = XW^\top$ as in Figure 2(b) with $T = 64$. The horizontal and vertical axis represent the output dimensions of Y.

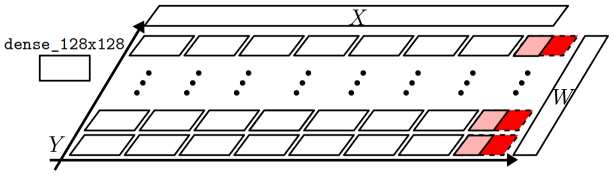

Figure 5. Micro-kernel `dense_128x128` used to realize $Y = XW^\top$ as in Figure 2(b) with $T = 60$. The tiles at the last column are padded (shown in red) to fit the micro-kernel.

can be easily ported to multiple workload instances of different shapes and hence it is shape-generic. If we revisit the previous example, the micro-kernel `dense_128x128` can be leveraged to realize not only $T = 64$, but $T = 1, 2, \ldots, 127, 128$ as well.

Despite being a generic solution, the use of micro-kernels poses two new and important challenges: (1) How to compose high-performance complete programs using micro-kernels, especially in the case when the workload cannot perfectly fit into the micro-kernel (see Figure 5 as an example), and (2) How to accurately and efficiently predict the performance of micro-kernel-based complete programs. In this work, we address the first challenge using *local padding* that automatically pads the local workspace when fetching the input tensors from global memory (see Section 4.1 for more details) and the second challenge by building a *micro-kernel-based cost model*.

3.2 Micro-Kernel-based Cost Model

To search for high-performance micro-kernels efficiently, we use cost models to guide the search process. Existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) also have cost models that predict the compute throughput of a program by extracting its features (e.g., loop structures, memory access patterns), but those cost models can only accept complete programs as inputs:

$$\text{Cost}(P) = f(\text{FeatureExtractor}(P)) \quad (1)$$

f : cost function (e.g., XGBoost (Chen & Guestrin, 2016))

P : **complete** program Cost : compute throughput

However, as micro-kernels are incomplete (they are a tile of the complete programs), they cannot be applied to the existing cost models. To address this challenge, we build a **micro-kernel-based cost model**. The key insight is that the cost of a complete program P that is made up of a micro-kernel M can be decomposed into two parts: (1) a

	Tuning?	Complexity
Vendor Libraries	✘	-
Existing Auto-Schedulers	✓	$O(\|S\|)$
DietCode	✓	$O(1)$

Table 1. Comparison between the three options in terms of the ability to do auto-tuning and the runtime complexity of tuning. $|S|$ represents the number of possible shapes.

shape-generic cost function f_{MK} that predicts the cost of M, and (2) a shape-dependent adaption cost function f_{adapt} that defines the penalty of porting M to P. While f_{MK} is a function that has to be learned and updated by real hardware measurements during the auto-scheduling process, f_{adapt} is a simple term that can be evaluated using the core occupancy and the padding ratio (in other words, it does not require feature extraction, see Section 4.2 for more details). Given below is the mathematical form of our cost model:

$$\text{Cost}_M(P) = f_{\text{MK}}(\text{FeatureExtractor}(M)) \cdot f_{\text{adapt}}(P, M) \tag{2}$$

Compared with Equation 1, Equation 2 has the key advantage of being efficient: To predict the performance of many complete programs that share the same micro-kernel M, the program feature extraction only needs to be done once on the micro-kernel M, and each program only needs to update the adaption cost individually. This significantly removes the redundancies in the feature extraction, leading to a more efficient auto-scheduling pipeline.

3.3 Joint Learning with DietCode

From the above search space and cost model formulations, we propose *DietCode*, a new auto-scheduler framework that has three key components: (1) a shape-generic search space, (2) a micro-kernel-based cost model, and (3) a dispatcher (❼ in Figure 1(b), see Section 4.3 for details) that automatically dispatches static shapes to micro-kernels based on Equation 2 at runtime.

The optimization workflow of *DietCode* is a **joint learning process** (❻ in Figure 1(b)), where all workload instances of a dynamic-shape workload share the same search space and collectively learn the same cost model. This workflow gives *DietCode* the ability to operate on a per-*category* basis, where each category shares the same shape-generic program (i.e., micro-kernel). Table 1 shows the comparison between the three major approaches we described so far: vendor libraries (oneAPI, 2021; NVIDIA, 2021; Chetlur et al., 2014), existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020), and *DietCode*, where we use $|S|$ to denote the number of possible shapes of a dynamic-shape workload. Because *DietCode* can produce all the shape-generic categories in one run, it can significantly reduce the auto-scheduling time compared with the existing auto-schedulers on dynamic-shape workloads, as our evaluation will show in Section 5.

```
for i.0 in [0, T):
  for i.1 in [0, t):
    if i.0*t+i.1 < T:
      X_local = X[...]
    if i.0*t+i.1 < T:
      Y_local = ...
    if i.0*t+i.1 < T:
      Y[...] = Y_local
```

(a)

```
X_pad = pad X to [⌈T/t⌉ · t]
for i.0 in [0, T):
  for i.1 in [0, t):
    X_local = X_pad[...]
    Y_local = ...
    Y_pad[...] = Y_local
slice Y_pad to [T]
```

(b)

```
for i.0 in [0, T):
  for i.1 in [0, t):
    if i.0 < ⌊T/t⌋:
      X_local = X[...]
      Y_local = ...
      Y[...] = Y_local
    else:
      if i.0*t+i.1 < T:
        X_local = X[...]
      if i.0*t+i.1 < T:
        Y_local = ...
      if i.0*t+i.1 < T:
        Y[...] = Y_local
```

(c)

```
for i.0 in [0, T):
  for i.1 in [0, t):
    if i.0*t+i.1 < T:
      X_local = X[...]
    Y_local = ...
    if i.0*t+i.1 < T:
      Y[...] = Y_local
```

(d)

Figure 6. (a) A tiled loop with boundary checks. (b-d) Three optimization strategies: (b) global padding, (c) loop partitioning, and (d) local padding.

4 Implementation Details

We integrate *DietCode* as a part of TVM (Chen et al., 2018a) in the form of an auto-scheduler submodule. In this section, we highlight some of the details of the *DietCode*'s design.

4.1 Local Padding

One of the key ideas of *DietCode* is to apply micro-kernels generically to all workload instances, each corresponding to a static-shape instance of the dynamic-shape workload. However, it is common for workload instances to not fit the micro-kernels perfectly, as is illustrated in Figure 5 where the micro-kernels at the last column are not fully materialized. In these cases, boundary checks need to be injected inside the micro-kernel to make sure that the program does not operate on invalid data values, but they also bring large performance degradation to the program (as much as 17× in our evaluation with the example illustrated in Figure 5 on a modern Tesla T4 GPU (NVIDIA, 2020)). This is because those checks bring in extra branching and compute instructions in the generated program.

To our best knowledge, there are three solutions to mitigate this problem. To demonstrate them, we use the code snippet with boundary checks in Figure 6(a) as an example, where we greatly simplify the schedule by keeping its skeleton only to help understanding (*fetch* the input tensor from the global memory to the local workspace, *compute*, and *writeback* the output tensor to the global memory).

(1) *Global padding* pads the input tensor before the main computation and slice the output tensor afterwards. This can remove all the boundary checks in the compute kernel, but would require extra storage and injections of pad/slice

operations (see Figure 6(b)).

(2) *Loop partitioning* (Shen et al., 2021) partitions the complete programs into regions where boundary checks can be eliminated and regions where they cannot (see Figure 6(c)). The performance benefits of loop partitioning depend on the relative ratio between the inner and the outer loop: If $t \ll T$, there will be more iterations of `i.0` without boundary conditions and hence more benefits. However, if $t \ll T$ does not hold, the benefits can be limited.

(3) *Local padding* pads the local workspace when fetching the input tensor values from the global memory, and slices the local workspace when writing back. This is equivalent to preserving the boundary checks at the fetch and writeback stages while removing those at the compute stage (see Figure 6(d)). The idea of local padding is based on the two key observations: (i) Only the boundary checks at the compute stage have the dominant impact on the runtime performance, because those at the fetch and writeback stages can be hidden by the latency of the global memory transfers (NVIDIA, 2019), and (ii) Padded data values that are computed in the compute stage will be filtered out by the boundary checks in the writeback stage, hence they do not affect the output tensor values (i.e., 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 can be generically applied to both large and small shapes (whereas loop partitioning only works best on large shapes). We prove that local padding is an efficient solution to address the boundary checks challenge with our evaluation in Section 5.

4.2 Micro-Kernel-based Cost Model

To accurately predict the performance of micro-kernel-based complete programs, we devise three terms in the cost model that respectively account for: ① the performance of the micro-kernel, ② the hardware core occupancy penalty, which correlates to the number of times this micro-kernel is executed to compose the complete program, and ③ the padding penalty. The mathematical expression of the cost model is as follows:

$$\text{Cost}_M(P) = f_{\text{MK}}(\text{FeatureExtractor}(M))① \cdot \underbrace{②f_{\text{OCC}}(P/M) \cdot f_{\text{pad}}(P, M)③}_{f_{\text{adapt}}(P,M)} \quad (3)$$

where M denotes the micro-kernel and P the complete program. The two functions f_{OCC}, f_{padding} of the adaption cost correspond respectively to:

f_{OCC}: The hardware core occupancy penalty, which we model using a linear regression model:

$$f_{\text{OCC}}(P/M) = k \underbrace{\frac{P/M}{\text{ceil_by}(P/M, \text{NumCores})}}_{④} + b$$

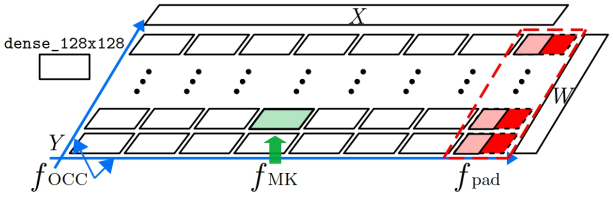

Figure 7. f_{MK}, f_{OCC}, and f_{pad} account for different program behaviors (example taken from Figure 5).

where the term P/M corresponds to the number of times M is executed to form P and the coefficients k, b are learnable parameters. Since each micro-kernel is dispatched to one hardware core, the term P/M also correlates to the number of occupied hardware cores. Therefore, the term ④ in the above equation denotes the hardware occupancy ratio of a complete program P that is composed using M. This ratio is further weighted using the two coefficients k, b, whose values are dynamically adapted to the hardware platforms when auto-scheduling, to estimate the hardware occupancy cost. In fact, since we want the occupancy cost to be 1 when all the cores are fully occupied, we can further derive:

$$f_{\text{OCC}}(P, M) = 1, \text{ when } \frac{P/M}{\text{ceil_by}(P/M, \text{NumCores})} = 1$$

$$\Rightarrow k + b = 1 \Rightarrow b = 1 - k$$

to reduce one of the dynamic parameters b.

f_{pad}: The padding penalty, which can be modelled in a hardware-agnostic way as $\text{pad_by}(P,M)/P$ (the pad_by primitive pads the complete program P by the size of M).

Figure 7 illustrates the correlation between the three terms in Equation 3 and how the micro-kernels form the complete program, where we can see the one-to-one correspondence between the equation and the complete program composition. Such correspondence allows us to accurately predict the performance of micro-kernel-based complete programs, and hence search for high-performance micro-kernels efficiently using evolutionary search (Vikhar, 2016) in the joint learning process (❻ in Figure 1).

4.3 Automatic Dispatching

After the joint learning process has been completed, a set of micro-kernels are generated by *DietCode* as the auto-scheduling outcomes. To dispatch all the possible shapes to those micro-kernels, we have each shape S vote for its favorite micro-kernel based on the cost formula in Equation 3:

$$\text{vote}(S) = \text{argmax}_M(\text{Cost}_M(P(S, M)))$$

where the LHS denotes the voted micro-kernel, and the term $P(S, M)$ refers to the complete program of shape S composed using the micro-kernel M.[3] After all S's have voted, we train a decision tree using the scikit-learn frame-

[3]The reason why it is argmax in the formula is because the term *cost* usually refers to the compute throughput (Chen et al., 2018b; Zheng et al., 2020a), hence the higher the better.

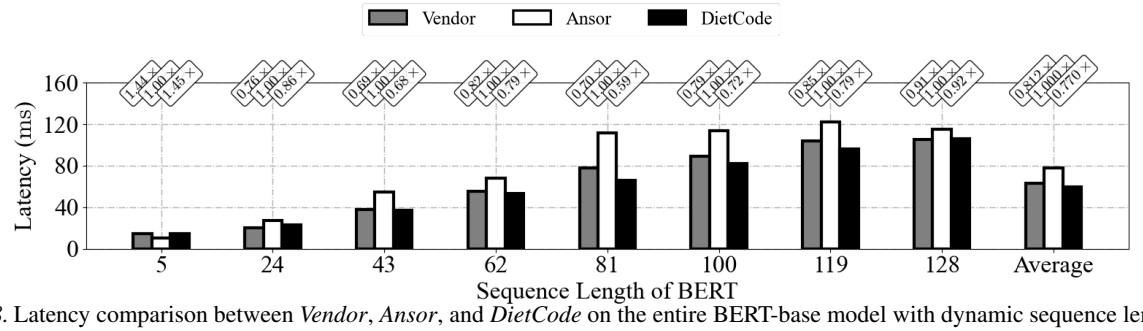

Figure 8. Latency comparison between *Vendor*, *Ansor*, and *DietCode* on the entire BERT-base model with dynamic sequence lengths.

work (Pedregosa et al., 2011) (❼ in Figure 1). The input to the decision tree is all the possible shapes and the output labels are their selected micro-kernels. The generated decision tree automatically categorizes shapes that select the same micro-kernel together, and is exported in the C-style source code format for efficient dispatching from the input shape to its corresponding micro-kernel at runtime.

Now that we have presented the details of the *DietCode*'s design, we evaluate its efficiency on state-of-the-art machine learning workloads.

5 Evaluation

5.1 Methodology

Infrastructure. Our major compute platform is an Amazon EC2 G4dn instance (Amazon, 2021), which is equipped with 16 Intel® Xeon® Platinum 8259CL CPUs (PassMark Software, 2020) and 1 NVIDIA Tesla T4 GPU (NVIDIA, 2020), with CUDA 11.3 (NVIDIA, 2021a), cuDNN 8.2 (NVIDIA, 2021), and TVM v0.8.dev0 (Chen et al., 2018a).

Applications. We evaluate our new auto-scheduler framework, *DietCode*, first on the BERT-base model (Devlin et al., 2019) end-to-end with dynamic sequence lengths. After that, we show how the performance is achieved by evaluating on the dense and batched matrix multiplication layers extracted from BERT-base, also with dynamic sequence lengths.

As it is impractical to complete the entire auto-scheduling procedure using the existing auto-schedulers (Zheng et al., 2020a; Chen et al., 2018a; Adams et al., 2019; Vasilache et al., 2020) (it can take about 42 CPU hours to complete a single dynamic-shape workload for the sequence length within the range of $[1, 128]$), we sample 8 shape configurations uniformly within the range of $[1, 128]$ to compare the performance and the auto-scheduling time, and use the auto-scheduling time on those configurations to project the total auto-scheduling time across the entire range.

Baselines. We compare *DietCode* with three state-of-the-art baselines: (1) the vendor library (cuBLAS (NVIDIA, 2021) and cuDNN (Chetlur et al., 2014) on the GPUs), which we refer to as *Vendor*, (2) the state-of-the-art auto-scheduler implementation, Ansor (Zheng et al., 2020a; Chen et al., 2018a), that targets static-shape workloads, which we refer to as *Ansor*, (3) the state-of-the-art prior work on dynamic code generation, Nimble (Shen et al., 2021), that extends *Ansor* to support dynamic-shape workloads by tuning the largest shape and applying its schedule generically to all shapes using loop partitioning, which we refer to as *Nimble*. On each static-shape workload, *Ansor* is sufficiently tuned for at least 1000 trials.

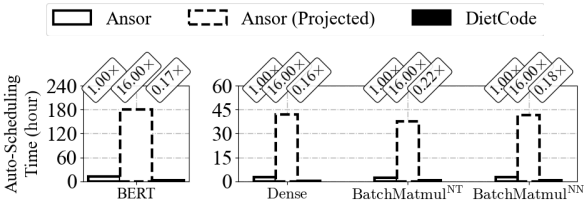

Figure 9. Auto-scheduling time comparison between *Ansor* and *DietCode* on various dynamic workloads. The lower the better.

Metrics. We show the results on (1) the end-to-end latency on the entire model, measured as msec and averaged over 5 runs (2) the runtime cost of the generated tensor programs on a single operator, measured as μsec and averaged over 100 runs, and (3) the total auto-scheduling time used, measured as hours. For all metrics, lower is better.

5.2 End-to-End Model Evaluation

Figure 8 shows the end-to-end latency comparison between *Vendor*, *Ansor*, and *DietCode* on the BERT-base model (Devlin et al., 2019) with dynamic sequence lengths, where all numbers are normalized to *Ansor* under each sequence length. We observe from the figure that the latency of *DietCode* is up to 69.5% better than that of *Ansor* and 18.6% better than that of *Vendor* (29.9% and 5.4% better on average, respectively). The reason of the speedup will be evident in the following sections as we show the performance comparison on each individual layer of the model.

Figure 9 shows the amount of auto-scheduling time taken by *Ansor* and *DietCode* on the sampled sequence lengths. Since those 8 sequence lengths are uniformly chosen within the range of $[1, 128]$, we project the auto-scheduling time of *Ansor* on the entire range to be $16\times$ of that on the sampled ones. We observe from the figure that *DietCode* takes $5.88\times$ less time to auto-schedule for the entire model than *Ansor* on the sampled sequence lengths. This speedup is estimated to

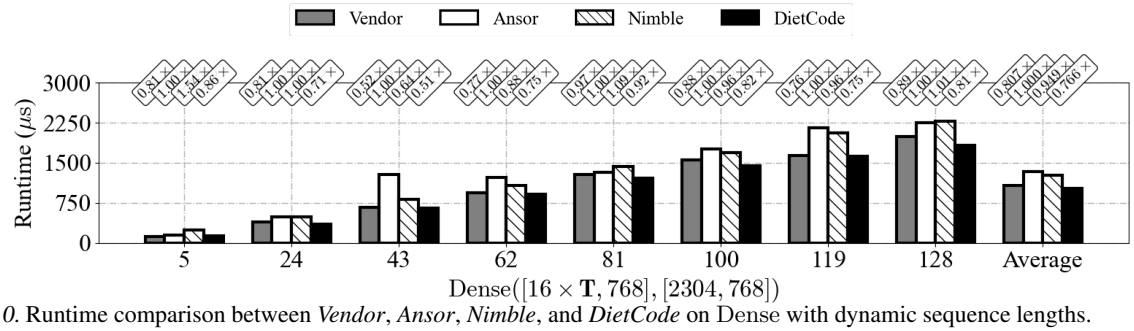

Figure 10. Runtime comparison between *Vendor*, *Ansor*, *Nimble*, and *DietCode* on Dense with dynamic sequence lengths.

increase to $94.1\times$ on the entire range, since *Ansor* needs to auto-schedule for each individual sequence length, whereas *DietCode* only needs to do the auto-scheduling once.

We now connect these improvements in the latency and the auto-scheduling time with results on individual layers as they are the building blocks for BERT (Devlin et al., 2019).

5.3 Dynamic Dense Layer

We compare the performance between *Vendor*, *Ansor*, *Nimble* and *DietCode* on the dense layer with dynamic sequence lengths (denoted as T) as in Figure 2(b). Figure 10 shows the comparison of the runtime cost under different sequence lengths. We observe from the figure that the runtime of *DietCode* across all the sequence lengths is on average 30.5% less than that of *Ansor*, 23.9% than that of *Nimble*, and 5.3% than that of *Vendor*. The reason why *DietCode* has better performance than *Ansor* is because, as is explained in Section 3.1, the shape-generic search space construction of *DietCode* allows it to explore tensor programs that are overlooked by the shape-dependent construction of *Ansor* (same reason for *Nimble* since it uses the schedule generated by *Ansor*). We cannot analyze the exact reason why *DietCode* can have better performance than *Vendor* since cuBLAS (NVIDIA, 2021) is a proprietary library.

Since *DietCode* only needs to run the auto-scheduling procedure once, it is much more efficient than *Ansor* in terms of the total auto-scheduling time. Figure 9 shows the auto-scheduling time taken by *Ansor* and *DietCode* on various types of dynamic workloads, where we observe *DietCode* takes $6.3\times$ less time than *Ansor* for auto-scheduling the dynamic dense layer on the sampled sequence lengths (and estimated to be $100.8\times$ less on the entire range). The reason why it is below the theoretical $8\times$ limit can be because the shape-generic search space construction and exploration is more complicated than that of shape-dependent search space, since hardware constraints need to be examined (as is explained in Section 3.1). However, the improvement in the auto-scheduling time is still noticeable and it will only be greater if we allow the dynamic sequence length T to take on more diverse values.

$$Y = \text{BatchMatmul}^{\text{NT}}(X, W)$$
$$X : [192, \mathbf{T}, 64], W : [192, \mathbf{T}, 64], \mathbf{T} \in [1, 128]$$

$$Y = \text{BatchMatmul}^{\text{NN}}(X, W)$$
$$X : [192, \mathbf{T}, \mathbf{T}], W : [192, \mathbf{T}, 64], \mathbf{T} \in [1, 128]$$

Figure 11. Dynamic-shape workload $Y = \text{BatchMatmul}(X, W)$ (with $\mathbf{T}$ being dynamic) and its corresponding tensor program.

5.4 Dynamic BatchMatmul Layer

We adopt the same methodology on the dynamic batched matrix multiplication workload as is defined in Figure 11, which is potentially more challenging as there are more than one axis being dynamic. We evaluate on both NT and NN data layout (where the NT layout transposes the second operand) to see whether *DietCode* can be generically applied to cases when there are dynamic spatial and reduction axes at the same time.

Figure 12 shows the comparison of the runtime cost under dynamic sequence lengths, from which we observe that in terms of the runtime across all the sequence lengths *DietCode* is on average 21.5% better than *Ansor*, 351% better than *Nimble*, and 12.3% better than *Vendor* on the NT layout. Moreover, it is 24.2% better than *Ansor*, 40.4% better than *Nimble*, and 15.4% better than *Vendor* on the NN layout.

The reason why *Nimble* does not deliver good performance (in the NT layout case particularly) is two-fold: (1) It directly applies the schedule generated on the largest shape to other shapes, which can lead to sub-optimal performance due to issues such as padding (see Figure 5). (2) It does not efficiently handle the boundary checks in the tensor programs, which are more critical in this case since there are two axes being dynamic.[4] We observe that since the shapes are relatively small compared with the previous benchmark (e.g., $192, 64$ versus $768, 2304$ in the case of the dynamic dense layer, as in Figure 10), the loop partitioning technique employed by *Nimble* is unable to show any benefits (since there is not enough loops to be partitioned without the boundary checks, as is explained in Section 4.1).

[4] Although the NN layout case also has two dynamic axes, one of them is the reduction axis. Because *Ansor* usually unrolls the reduction axes, the boundary checks on the those axes are automatically optimized after the loops are unrolled.

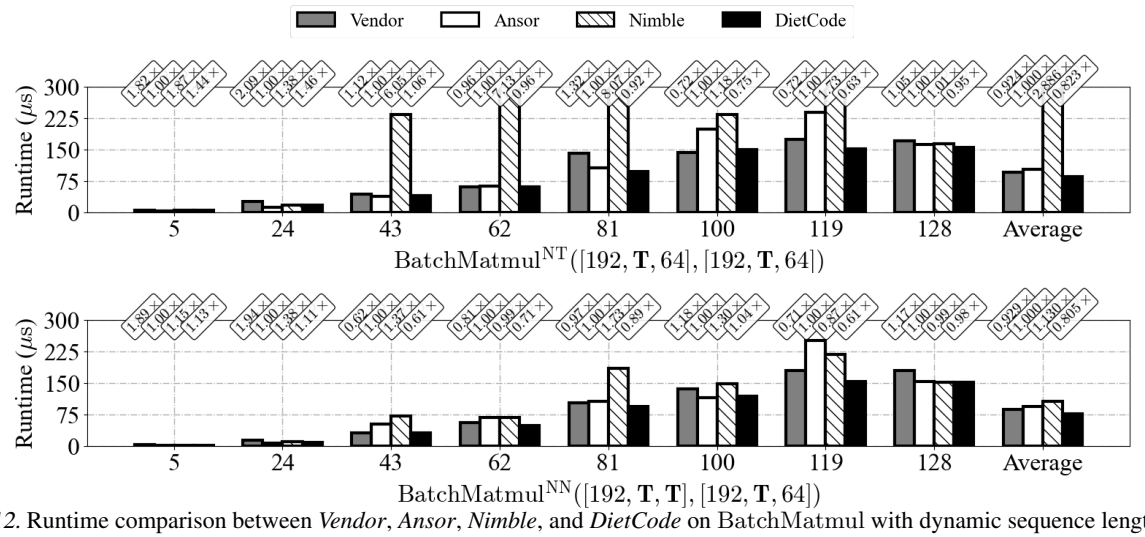

Figure 12. Runtime comparison between *Vendor*, *Ansor*, *Nimble*, and *DietCode* on BatchMatmul with dynamic sequence lengths.

Similar to the dynamic dense layer benchmark, the auto-scheduling time of *DietCode* is $4.55\times$ and $5.56\times$ less than that of *Ansor* on the NT and NN layout respectively (see Figure 9). Such benefits allow us to tune on the entire model end-to-end within a reasonable amount of time, as we have demonstrated in Section 5.2.

We conclude with the above experiments that *DietCode* is efficient in terms of the auto-scheduling time while preserving the ability to deliver high-performance tensor programs. It can be generically applied to cases when there are multiple dynamic axes, or when there is a hybrid of dynamic spatial and reduction axes, and even to an entire model. Such efficiency and generality in auto-scheduling make *DietCode* a practical solution for dynamic-shape workloads.

6 Related Works

DietCode addresses the key challenges of auto-scheduling dynamic-shape workloads by constructing a shape-generic search space and have all the possible shapes jointly search within the same space and update the same cost model, which is unseen in existing auto-schedulers that target static-shape workloads (Ansor (Zheng et al., 2020a), TVM (Chen et al., 2018a), Halide auto-scheduler (Adams et al., 2019), TensorComprehensions (Vasilache et al., 2020)). Although *DietCode* is currently implemented on top of TVM (Chen et al., 2018a), its ideas can be generically applied to other compiler frameworks as well, such as Halide (Ragan-Kelley et al., 2018), TensorComprehensions (Vasilache et al., 2020), Tiramisu (Baghdadi et al., 2019), XLA (Sabne, 2020), MLIR (Lattner et al., 2021), Glow (Rotem et al., 2018), taco (Kjolstad et al., 2017), and TASO (Jia et al., 2019).

Reuse-based Tuner. Selective Tuning (Yu, 2019) and ETO (Fang et al., 2021) group workloads into clusters based on a set of pre-defined rules (e.g., similarity ratio in Selective Tuning (Yu, 2019)) and reuse the same schedule in a single cluster. Their approaches are parallel with the joint learning adopted by *DietCode*.

Auto-Tuners. AutoTVM (Chen et al., 2018b), ProTuner (Haj-Ali et al., 2020), and FlexTensor (Zheng et al., 2020b) leverage program templates to define the search space and guide the search procedure. They are different from auto-schedulers in that the search space has to be manually defined. However, if there exists a pre-defined micro-kernel search space, then the key ideas *DietCode* can be applied to those auto-tuners as well.

Dynamic Neural Networks. Dynamic batching is a common graph-level optimization adopted by frameworks such as DyNet (Neubig et al., 2017), Cavs (Xu et al., 2018), BatchMaker (Gao et al., 2018), and TensorFlow Fold (Looks et al., 2017) for cases when the batch size is dynamic. Nimble (Shen et al., 2021) and DISC (Zhu et al., 2021) both design a compiler to represent and execute dynamic neural networks. Cortex (Fegade et al., 2020) is a compiler-based framework on recursive neural networks. Those works focus on the graph-level optimizations and therefore are orthogonal to *DietCode*, which operates on each individual layer. In fact, those graph-level solutions can also leverage *DietCode* for efficient operator code generation.

7 Conclusion

In this work, we propose *DietCode*, a new auto-scheduler framework for dynamic-shape workloads. Our evaluation shows that on *DietCode* can significantly reduce the auto-scheduling time by $5.88\times$ ($94.1\times$ projected if all the possible shapes are included), while achieving up to 69.5% better performance than the state-of-the-art auto-schedulers and 18.6% than the vendor library on a full state-of-the-art DNN model end-to-end. We hope that *DietCode* would become an efficient platform for further research on efficient system design for key machine learning applications.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P. A., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A system for large-scale machine learning. In Keeton, K. and Roscoe, T. (eds.), *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 265–283. USENIX Association, 2016. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. Learning to optimize Halide with tree search and random programs. *ACM Transactions on Graphics*, 38(4):121:1–121:12, 2019. doi: 10.1145/3306346.3322967. URL https://doi.org/10.1145/3306346.3322967.

Alibaba. Bringing TVM into TensorFlow for optimizing neural machine translation on GPU, 2018. URL https://tvm.apache.org/2018/03/23/nmt-transformer-optimize.

Amazon. Amazon EC2 G4 instances, 2021. URL https://aws.amazon.com/ec2/instance-types/g4/.

Amodei, D., Ananthanarayanan, S., Anubhai, R., Bai, J., Battenberg, E., Case, C., Casper, J., Catanzaro, B., Chen, J., Chrzanowski, M., Coates, A., Diamos, G., Elsen, E., Engel, J. H., Fan, L., Fougner, C., Hannun, A. Y., Jun, B., Han, T., LeGresley, P., Li, X., Lin, L., Narang, S., Ng, A. Y., Ozair, S., Prenger, R., Qian, S., Raiman, J., Satheesh, S., Seetapun, D., Sengupta, S., Wang, C., Wang, Y., Wang, Z., Xiao, B., Xie, Y., Yogatama, D., Zhan, J., and Zhu, Z. Deep Speech 2 : End-to-end speech recognition in English and Mandarin. In Balcan, M. and Weinberger, K. Q. (eds.), *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, volume 48 of *JMLR Workshop and Conference Proceedings*, pp. 173–182. JMLR.org, 2016. URL http://proceedings.mlr.press/v48/amodei16.html.

Baghdadi, R., Ray, J., Romdhane, M. B., Sozzo, E. D., Akkas, A., Zhang, Y., Suriana, P., Kamil, S., and Amarasinghe, S. P. Tiramisu: A polyhedral compiler for expressing fast and portable code. In Kandemir, M. T., Jimborean, A., and Moseley, T. (eds.), *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, pp. 193–205. IEEE, 2019. doi: 10.1109/CGO.2019.8661197. URL https://doi.org/10.1109/CGO.2019.8661197.

Chen, T. and Guestrin, C. XGBoost: A scalable tree boosting system. In Krishnapuram, B., Shah, M., Smola, A. J., Aggarwal, C. C., Shen, D., and Rastogi, R. (eds.), *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, pp. 785–794. ACM, 2016. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL http://arxiv.org/abs/1512.01274.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In Arpaci-Dusseau, A. C. and Voelker, G. (eds.), *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pp. 578–594. USENIX Association, 2018a. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Chen, T., Zheng, L., Yan, E. Q., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In Bengio, S., Wallach, H. M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 3393–3404, 2018b. URL https://proceedings.neurips.cc/paper/2018/hash/8b5700012be65c9da25f49408d959ca0-Abstract.html.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cuDNN: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T. (eds.), *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)*, pp. 4171–4186. Association for Computational Linguistics, 2019.

doi: 10.18653/v1/n19-1423. URL https://doi.org/10.18653/v1/n19-1423.

Fang, J., Shen, Y., Wang, Y., and Chen, L. ETO: Accelerating optimization of DNN operators by high-performance tensor program reuse. *Proceedings of the VLDB Endowment*, 15(2):183–195, 2021. URL http://www.vldb.org/pvldb/vol15/p183-chen.pdf.

Fegade, P., Chen, T., Gibbons, P., and Mowry, T. Cortex: A compiler for recursive deep learning models. *CoRR*, abs/2011.01383, 2020. URL https://arxiv.org/abs/2011.01383.

Gao, P., Yu, L., Wu, Y., and Li, J. Low latency RNN inference with cellular batching. In Oliveira, R., Felber, P., and Hu, Y. C. (eds.), *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, pp. 31:1–31:15. ACM, 2018. doi: 10.1145/3190508.3190541. URL https://doi.org/10.1145/3190508.3190541.

Haj-Ali, A., Genc, H., Huang, Q., Moses, W., Wawrzynek, J., Asanovic, K., and Stoica, I. ProTuner: Tuning programs with monte carlo tree search, 2020. URL https://arxiv.org/abs/2005.13685.

He, K., Zhang, X., Ren, S., and Sun, J. 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*, pp. 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90. URL https://doi.org/10.1109/CVPR.2016.90.

Jia, Z., Padon, O., Thomas, J. J., Warszawski, T., Zaharia, M., and Aiken, A. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In Brecht, T. and Williamson, C. (eds.), *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, pp. 47–62. ACM, 2019. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630.

Kjolstad, F., Chou, S., Lugato, D., Kamil, S., and Amarasinghe, S. P. taco: A tool to generate tensor algebra kernels. In Rosu, G., Penta, M. D., and Nguyen, T. N. (eds.), *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pp. 943–948. IEEE Computer Society, 2017. doi: 10.1109/ASE.2017.8115709. URL https://doi.org/10.1109/ASE.2017.8115709.

Kosec, M., Fu, S., and Krell, M. M. Packing: Towards 2x NLP BERT acceleration. *CoRR*, abs/2107.02027, 2021. URL https://arxiv.org/abs/2107.02027.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J. A., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: Scaling compiler infrastructure for domain specific computation, 2021. URL https://doi.org/10.1109/CGO51591.2021.9370308.

Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. 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, 2017. URL https://openreview.net/forum?id=ryrGawqex.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. DyNet: The dynamic neural network toolkit, 2017. URL http://arxiv.org/abs/1701.03980.

NVIDIA. CUDA fundamental optimization, part 1, 2019. URL https://www.olcf.ornl.gov/wp-content/uploads/2019/12/03-CUDA-Fundamental-Optimization-Part-1.pdf.

NVIDIA. NVIDIA T4 70W low profile PCIe GPU accelerator, 2020. URL https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-t4/t4-tensor-core-product-brief.pdf.

NVIDIA. Programming guide :: CUDA toolkit documentation, 2021a. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

NVIDIA. CUDA toolkit archive, 2021b. URL https://developer.nvidia.com/cuda-toolkit-archive.

NVIDIA. cuBLAS :: CUDA toolkit documentation, 2021. URL https://docs.nvidia.com/cuda/cublas.

NVIDIA. Developer guide :: NVIDIA deep learning cuDNN documentation, 2021. URL https://docs.nvidia.com/deeplearning/cudnn/developer-guide/index.html.

oneAPI. oneAPI deep neural network library (oneDNN), 2021. URL https://github.com/oneapi-src/oneDNN.

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

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. PyTorch: An imperative style, high-performance deep learning library. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., VanderPlas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL http://dl.acm.org/citation.cfm?id=2078195.

Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S. P., and Durand, F. Halide: Decoupling algorithms from schedules for high-performance image processing. *Communications of the ACM*, 61(1):106–115, 2018. doi: 10.1145/3150211. URL https://doi.org/10.1145/3150211.

Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL http://arxiv.org/abs/1805.00907.

Sabne, A. XLA: Compiling machine learning for peak performance, 2020.

Sharir, O., Peleg, B., and Shoham, Y. The cost of training NLP models: A concise overview. *CoRR*, abs/2004.08900, 2020. URL https://arxiv.org/abs/2004.08900.

Shen, H., Roesch, J., Chen, Z., Chen, W., Wu, Y., Li, M., Sharma, V., Tatlock, Z., and Wang, Y. Nimble: Efficiently compiling dynamic neural networks for model inference. In *Proceedings of Machine Learning and Systems*, volume 3, 2021. URL https://proceedings.mlsys.org/paper/2021/hash/4e732ced3463d06de0ca9a15b6153677-Abstract.html.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically. *ACM Transactions on Architecture and Code Optimization*, 16(4): 38:1–38:26, 2020. doi: 10.1145/3355606. URL https://doi.org/10.1145/3355606.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In Guyon, I., von Luxburg, U., Bengio, S., Wallach, H. M., Fergus, R., Vishwanathan, S. V. N., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pp. 5998–6008, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Vikhar, P. A. Evolutionary algorithms: A critical review and its future prospects. In *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, pp. 261–265, 2016. doi: 10.1109/ICGTSPICC.2016.7955308.

Wang, Y. [RFC] dynamic shape support - graph dispatching, 2019. URL https://github.com/apache/incubator-tvm/issues/4118.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016. URL http://arxiv.org/abs/1609.08144.

Xu, S., Zhang, H., Neubig, G., Dai, W., Kim, J. K., Deng, Z., Ho, Q., Yang, G., and Xing, E. P. Cavs: An efficient runtime system for dynamic neural networks. In Gunawi, H. S. and Reed, B. (eds.), *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pp. 937–950. USENIX Association, 2018. URL https://www.usenix.org/conference/atc18/presentation/xu-shizen.

Yu, C. [RFC][AutoTVM] selective tuning, 2019. URL https://github.com/apache/incubator-

tvm/issues/4188. [code] https://github.com/apache/incubator-tvm/pull/4187.

Yu, F., Xu, Z., Shen, T., Stamoulis, D., Shangguan, L., Wang, D., Madhok, R., Zhao, C., Li, X., Karianakis, N., Lymberopoulos, D., Li, A., Liu, C., Chen, Y., and Chen, X. Towards latency-aware DNN optimization with GPU runtime analysis and tail effect elimination. *CoRR*, abs/2011.03897, 2020. URL https://arxiv.org/abs/2011.03897.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. 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*, pp. 863–879. USENIX Association, 2020a. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

Zheng, S., Liang, Y., Wang, S., Chen, R., and Sheng, K. FlexTensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In Larus, J. R., Ceze, L., and Strauss, K. (eds.), *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pp. 859–873. ACM, 2020b. doi: 10.1145/3373376.3378508. URL https://doi.org/10.1145/3373376.3378508.

Zhu, K., Zhao, W., Zheng, Z., Guo, T., Zhao, P., Bai, J., Yang, J., Liu, X., Diao, L., and Lin, W. DISC: A dynamic shape compiler for machine learning workloads. *CoRR*, abs/2103.05288, 2021. URL https://arxiv.org/abs/2103.05288.

Zoph, B. and Le, Q. V. 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, 2017. URL https://openreview.net/forum?id=r1Ue8Hcxg.

A Artifact Appendix

A.1 Abstract

We provide the source code and scripts that correspond to Section 3 and 4 as our artifact. We use the AWS G4 instance as the hardware platform, and NVIDIA driver and Docker to build up the software stack. After the installation, each of the experiment is automated by a single script file.

A.2 Artifact check-list (meta-information)

- **Algorithm:** *DietCode* Auto-Scheduling Workflow
- **Program:** TVM (Chen et al., 2018a) submodules (one integrated with *DietCode* and the other with minor changes) + Benchmarking Test Cases
- **Compilation:** Please use the provided script file `scripts/1-compile.sh` in the repository for the compilation.
- **Transformations:** N/A
- **Binary:** N/A
- **Data set:** N/A
- **Run-time environment:** Please use the provided Dockerfile to build a containerized environment under AWS Deep Learning AMI (Ubuntu 18.04) Version 50.0
- **Hardware:** Amazon EC2 G4 Instance (g4dn.4xlarge)
- **Run-time state:** No contentions on hardware resources (CPU, GPU, RAM, PCIe) with other processes.
- **Execution:** Please use the provided script file `scripts/2-experiment_*.sh` in the repository for each of the experiment.
- **Metrics:** Compute Throughput (in TFLOPs/s) and Wall-Clock Time
- **Output:** CSV Files
- **Experiments:** Auto-scheduling Dense and BatchMatmul operators with dynamic sequence lengths. Auto-scheduling key operators of the BERT (Devlin et al., 2019) model (also with dynamic sequence lengths).
- **How much disk space required (approximately)?:** At least 20 GB (15 GB for the Docker image and 1.5 GB for the compilation artifacts)
- **How much time is needed to prepare workflow (approximately)?:** 1 hr
- **How much time is needed to complete experiments (approximately)?:** 10 hrs for the full experiments
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** N/A
- **Data licenses (if publicly available)?:** N/A
- **Workflow framework used?:** N/A
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.6326726

A.3 Description

A.3.1 How delivered

The source code is publicly available on GitHub

https://github.com/UofT-EcoSystem/DietCode
Branch MLSys2022_AE

A.3.2 Hardware dependencies

We use the AWS G4 instance (of type g4dn.4xlarge) to evaluate our results.

A.3.3 Software dependencies

The major software dependency is the NVIDIA GPU driver and the NVIDIA Docker container toolkit (nvidia-docker2 and nvidia-container-runtime). These two are automatically included in the AWS Deep Learning AMI (ubuntu 18.04) Version 50.0. The rest of the dependencies can be obtained by building the Docker image. To facilitate the Docker commands, we use docker-compose, which is a wrapper on top of Docker.

A.3.4 Data sets

N/A

A.4 Installation

Please follow the steps below:

- Clone the project by

```
git clone https://github.com/UofT-EcoSystem/DietCode -b MLSys2022_AE
```

- Install docker-compose, which is a wrapper on top of Docker.

```
sudo -H pip3 install docker-compose
```

- Build the Docker image that includes all the software dependencies required to run the experiments:

```
DietCode$ docker-compose build tvm-dev
```

- Create a running container out of the image:

```
DietCode$ docker-compose run --rm tvm-dev
```

- Build the DietCode and the TVM baseline.

```
/mnt$ ./scripts/1-compile.sh tvm
/mnt$ ./scripts/1-compile.sh tvm_base
```

A.5 Experiment workflow

- Dense Layer with Dynamic Sequence Length (Section 5.3 of the main text)

```
/mnt$ ./scripts/2_1-experiment_dynamic_dense.sh
```

- BatchMatmul Layer with Dynamic Sequence Length (Section 5.4 of the main text)

```
/mnt$ ./scripts/2_2-
    experiment_dynamic_batch_matmul_nt.
    sh
/mnt$ ./scripts/2_3-
    experiment_dynamic_batch_matmul_nn.
    sh
```

- BERT (Devlin et al., 2019) with Various Sequence Lengths (Section 5.2)

```
/mnt$ ./scripts/2_4-experiment_bert.sh
```

A.6 Evaluation and expected result

After each experiment has been run, a CSV file named

`temp_workspace.csv`

will be generated in each folder

`ops/dense`, `ops/batch_matmul`, and `networks/bert`

respectively that reports the latency numbers (in seconds, the lower the better). At the same time,

`ansor/dietcode_autosched_timer.csv`

will be generated in the same folder that reports the time to complete the auto-scheduling process (also in seconds, the lower the better).

A.7 Experiment customization

N/A

A.8 Notes

Note that the entire auto-scheduling workflow takes time to complete. Therefore, we one can use the

```
AUTO_SCHED_NTRIALS=200 ./scripts/...
```

prefix that uses fewer number auto-scheduling trials. The resulting tensor programs will still be functionally correct but the performance can be sub-optimal.

A.9 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging