# Grape: Practical and Efficient Graph-based Executions for Dynamic Deep Neural Networks on GPUs


Bojian Zheng*†
bojian@cs.toronto.edu
University of Toronto, CentML
Toronto, ON, Canada

Cody Hao Yu
comaniac0422@gmail.com
Amazon
Santa Clara, CA, USA

Jie Wang
jwangma@amazon.com
Amazon
Santa Clara, CA, USA

Yaoyao Ding†
yaoyao.ding@mail.utoronto.ca
University of Toronto, CentML
Toronto, ON, Canada

Yizhi Liu
eazhi.liu@gmail.com
Amazon
Santa Clara, CA, USA

Yida Wang
wangyida@amazon.com
Amazon
Santa Clara, CA, USA

Gennady Pekhimenko†
pekhimenko@cs.toronto.edu
University of Toronto, CentML
Toronto, ON, Canada



## ABSTRACT

Achieving high performance in machine learning workloads is a crucial yet difficult task. To achieve high runtime performance on hardware platforms such as GPUs, graph-based executions such as CUDA graphs are often used to eliminate CPU runtime overheads by submitting jobs in the granularity of multiple kernels. However, many machine learning workloads, especially *dynamic* deep neural networks (DNNs) with varying-sized inputs or data-dependent control flows, face challenges when directly using CUDA graphs to achieve optimal performance. We observe that the use of graph-based executions poses three key challenges in terms of efficiency and even practicability: (1) Extra data movements when copying input values to graphs' placeholders. (2) High GPU memory consumption due to the numerous CUDA graphs created to efficiently support dynamic-shape workloads. (3) Inability to handle data-dependent control flows.

To address those challenges, we propose *Grape*, a new graph compiler that enables practical and efficient graph-based executions for dynamic DNNs on GPUs. *Grape* comprises three key components: (1) an alias predictor that automatically removes extra data movements by leveraging code positions at the Python frontend, (2) a metadata compressor that efficiently utilizes the data redundancy in CUDA graphs' memory regions by compressing them, and (3) a predication rewriter that safely replaces control flows with predication contexts while preserving programs' semantics. The three components improve the efficiency and broaden the optimization scope of graph-based executions while allowing machine learning practitioners to program dynamic DNNs at the Python level with minimal source code changes.

We evaluate *Grape* on state-of-the-art text generation (GPT-2, GPT-J) and speech recognition (Wav2Vec2) workloads, which include both training and inference, using real systems with modern GPUs. Our evaluation shows that *Grape* achieves up to 36.43× less GPU memory consumption and up to 1.26× better performance than prior works on graph-based executions that directly use CUDA graphs. Furthermore, *Grape* can optimize workloads that are impractical for prior works due to the three key challenges, achieving 1.78× and 1.82× better performance on GPT-J and Wav2Vec2 respectively than the original implementations that do not use graph-based executions.



## CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; *Machine learning*; *Artificial intelligence.*

## KEYWORDS

machine learning compilers, CUDA graphs, dynamic neural networks

**ACM Reference Format:**
Bojian Zheng, Cody Hao Yu, Jie Wang, Yaoyao Ding, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Grape: Practical and Efficient Graph-based Executions for Dynamic Deep Neural Networks on GPUs. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3613424.3614248


## 1 INTRODUCTION

DNNs form an important class of machine learning algorithms and have made significant advancements in numerous domains such as image classification [41, 43, 107], text generation [11, 78, 87, 106, 114], and speech recognition [5, 8, 36]. In those applications, the DNNs are first trained and then deployed for inference, both for

*Part of the work done while interning at Amazon.
†Also with Vector Institute.




*MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada*



ACM ISBN 979-8-4007-0329-4/23/10.
https://doi.org/10.1145/3613424.3614248

many iterations. Due to the high cost of the two operations in terms of time and hardware resources [81, 100], it is crucial to achieve efficient execution of DNNs on specialized hardware platforms such as GPUs.

Graph-based executions such as CUDA graphs [35] on NVIDIA GPUs are an effective way of improving the performance of DNNs by submitting jobs to GPUs in the granularity of multiple kernels instead of a single kernel. Specifically, given a DNN model, CUDA graphs *capture* the model's computations on GPUs at the first time they are launched and *replay* those computations in subsequent iterations with a single graph launch call. To keep the correctness and the consistency between the captured operations and the replayed ones, CUDA graphs request that all workloads be *deterministic* (i.e., no control flows) and have their parameters fixed when capturing, which include CUDA kernel launch configurations and function arguments. In order for CUDA graphs to operate on different input values, synthetic inputs are used at capture time as *placeholders*, which are populated with real input values (i.e., input data from machine learning workloads) at replay time.

Although CUDA graphs are powerful in eliminating CPU overheads for general-purpose GPU applications, it is difficult to directly apply them to many DNNs (especially dynamic DNNs that have varying-sized inputs or data-dependent control flows, which are ubiquitous in modern machine learning applications [5, 8, 11, 36, 78, 87, 106, 114]). This is because such dynamic DNNs pose three key challenges in terms of efficiency and even practicability:

(1) *Extra data movements from real inputs to the graphs' placeholders*: The design of CUDA graphs introduces extra copies from real input values to the graphs' placeholders, attributing to up to 34% of the models' execution time.

(2) *Prohibitively large GPU memory consumption to efficiently support dynamic-shape workloads*: To efficiently support workloads with dynamic input shapes, we have to create a CUDA graph for each shape, and every created graph consumes a certain amount of GPU memory (20–100 `MB`). Although the amount is small by itself and hence not a challenge in static-shape DNNs, they sum up to be a huge value (20–100 `GB`) that exceeds the GPU memory capacity (e.g., 24 `GB` on an NVIDIA RTX 3090 GPU [71], 40/80 `GB` on an A100 GPU[70], or 80 `GB` on an H100 GPU[74]) when the number of possible shapes is 1024 (used in state-of-the-art language modeling applications [87, 114]).

(3) *Unable to handle programs with data-dependent control flows*: CUDA graphs can only capture deterministic computation and thus fail to capture modules with data-dependent control flows. This restriction prevents us from using them to optimize important modules that take a large portion of the total runtime but nonetheless have control flows within them. For example, our runtime measurements show that the beam search module [112] of state-of-the-art text generation applications [87] constitutes 30% of the total execution time, but it is out of the optimization scope of CUDA graphs for having data-dependent control flows.

These three challenges hurt the efficiency and the practical use of CUDA graphs in many real-world scenarios. To adequately address those challenges, we propose *Grape*, a graph compiler that makes graph-based executions practical and efficient for dynamic DNNs on GPUs. *Grape* is made up of three key components:

(1) An **alias predictor** that automatically and accurately predicts if a tensor will be copied to a graph's placeholder. This enables direct forwarding of the placeholder's memory region to the tensor (which we denote as that placeholder's *alias*), eliminating the need for future data movements. We observe the main reason why the data movements are needed in the existing graph-based executor [80] is that the memory allocators of machine learning frameworks [2, 80] are unable to tell whether an arbitrary tensor created in Python's dynamic programming environment will be moved to a placeholder in the future. Therefore, they can only offer a general memory region to the tensor first and then move it later to the regions that are reserved for placeholders. To our best knowledge, this is a common problem that exists for all frameworks adopting a Python frontend and a C++ runtime (e.g., PyTorch [80] and TensorFlow [2]).

We, however, notice that such data movements can be avoided if the Python frontend is considered when allocating memory. Specifically, we observe that DNN executions are highly regular and a Python code position that yields a placeholder's alias in one iteration is also likely to yield another alias for the same placeholder in the next iteration. Based on this insight, we design a frontend-aware alias predictor that dynamically records the connection between the Python frontend and the placeholders. The predictor requires zero changes to frontend applications' source code and can accurately forward the reserved memory regions for placeholders to their respective aliases.

(2) A **metadata compressor** that significantly reduces the GPU memory consumption of CUDA graphs by compressing their memory regions at capture time and efficiently decompresses them at replay time. Although the precise technical details regarding the CUDA graphs' memory regions are proprietary, we speculate that they are for caching the graphs' metadata on GPUs after carefully studying their dumped values. The metadata describes what is captured in the graph and contains information such as the CUDA kernels that reside in the graph along with their function arguments.

We further observe from the dumped values that the CUDA graphs' memory regions for many important models are very sparse and repetitive. This is because most of the CUDA kernels in state-of-the-art DNNs [8, 87, 114] do not fully leverage the function argument space provided to them, and they are usually invoked using similar pointer values. Therefore, those CUDA kernels leave abundant sparsity and value redundancy in their memory regions, allowing us to compress them using simple compression algorithms such as run-length encodings [31]. With efficient compression, we are able to fit the extremely large GPU memory consumption when using CUDA graphs to execute dynamic-shape DNNs (up to 100 `GB`) into the limited memory of modern GPUs.

(3) A **predication rewriter** that replaces data-dependent control flows with predication contexts, a new Python context that controls whether GPU operations within it can go through or not using a predicate. Although the idea of predication has been proposed in prior works [39, 42, 45, 63] at the architecture level to handle short `if-else` statements, we significantly broaden its scope of applicability by allowing machine learning practitioners to implement common control logic (e.g., `if`, `break`, and `continue`) at the Python level. Predication contexts stitch fragmented basic blocks together into a monolithic block that does not have control flows

while preserving the programs' semantics, which hence enables conversion to efficient graph-based executions.

Our contributions can be summarized as follows:

• We identify the key challenges in making graph-based executions practical and efficient for dynamic DNNs, an important class of DNNs that involve varying-sized inputs and/or data-dependent control flows.

• We build *Grape*, a new graph compiler that is integrated in PyTorch [80], a state-of-the-art machine learning framework. *Grape* adequately addresses those challenges and allows machine learning practitioners to program their models at the Python level with minimal source code changes. Its design is based on three key insights: (1) regularity in mappings from the Python frontend to graphs' placeholders, (2) high data redundancy in graphs' memory regions, and (3) predication contexts in place of data-dependent control flows.

• We evaluate *Grape* using state-of-the-art DNN training and inference workloads in text generation (GPT-2 [87], GPT-J [114]) and speech recognition (Wav2Vec2 [8]) on NVIDIA RTX 3090 [71] and A100 [70] GPUs. Our evaluation shows that *Grape* achieves up to 1.26× better performance on GPT-2 [87] than prior works on graph-based executions [80]. Furthermore, as *Grape* reduces the GPU memory consumption of CUDA graphs by up to 36.43×, it is able to practically optimize dynamic-shape workloads that are challenging for prior works [80], achieving up to 1.78× and 1.82× better performance than the original implementations that do not use graph-based executions on GPT-J [114] and Wav2Vec2 [8], respectively.

## 2 BACKGROUND AND MOTIVATION

In this section, we present an overview of (1) what CUDA graphs are and why they are important (Section 2.1), and (2) the challenges of using CUDA graphs to efficiently execute machine learning workloads (Section 2.2), especially those that involve dynamic shapes and/or dynamic control flows. After we present those challenges, we will also demonstrate why prior works [69, 86, 105, 118] are inadequate in addressing them (Section 2.3). This motivates us to propose a new graph compiler that allows for practical and efficient graph-based executions on GPUs.

### 2.1 CUDA Graph

Modern machine learning frameworks such as PyTorch [80] and TensorFlow [2] commonly follow a hierarchical design: To achieve high efficiency and programmability, they have a Python programming frontend that communicates with hardware backends such as NVIDIA GPUs via a C++ runtime. CUDA graphs are a GPU programming model that can significantly boost the performance of machine learning workloads on NVIDIA GPUs by eliminating the overheads on the CPU side, which exist across the frontend (e.g., Python invokes C APIs), the runtime (e.g., operator implementations verify input data values and shapes), and the backend (e.g., CUDA [72] launches GPU kernels using the `cudaLaunch` API). In addition to being easily programmable by machine learning practitioners at the Python level, CUDA graphs are orthogonal to other optimizations such as reduced precision [66, 67, 116] and quantization [38, 44, 52, 57, 103, 108, 121], and their ideas can be generically applied to other hardware platforms (e.g., HIP graphs [4] on AMD GPUs are similar in spirit to CUDA graphs).

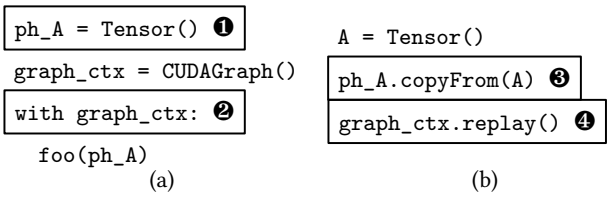



**Figure 1: CUDA graph (a) Capture and (b) Replay**

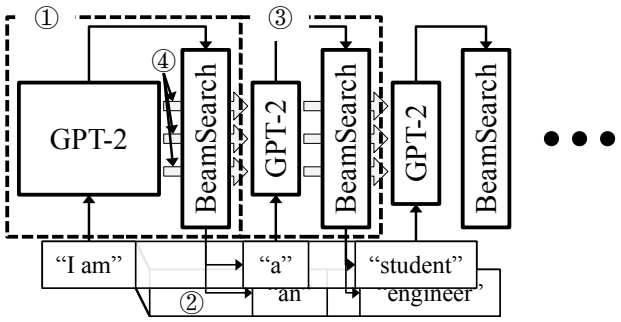



**Figure 2: Text Generation Pipeline of GPT-2**

CUDA graphs aim at eliminating CPU overheads by *capturing* and *replaying* only the effective computations on the GPU side (Figure 1): To construct a CUDA graph, the workload is first captured (❷ in Figure 1) to record all the GPU operations that happen within a given capture context. In subsequent runs, the exact same operations in the context can be replayed (❹) by launching the graph object instead. Due to the consistency between the operations that are captured and the ones that are replayed, CUDA graphs request that all GPU kernels be deterministic (i.e., no control flow when capturing) and have their parameters fixed. These parameters include launch configurations (e.g., block and grid dimensions) and function arguments (e.g., pointer values that refer to the input and output data). To have CUDA graphs operate on different data values, synthetic inputs are used at capture time as *placeholders* (❶), and their contents are populated at replay time with real input values from machine learning workloads (❸).

### 2.2 Challenges of Using CUDA Graphs

Although CUDA graphs are effective in removing CPU overheads, it is challenging to apply them out-of-the-box to many machine learning workloads for runtime efficiency. We use the GPT-2 [87] model as an example[1]. Note that as we introduce the model in the next paragraph, we highlight key points that hinder the practicability or the efficiency of CUDA graphs in **bold** texts.

GPT-2 [87] is an important language model in the text generation domain. It has structural similarity with numerous large-scale language models [11, 78, 106, 114] and many state-of-the-art DNNs

[1]The GPT-2 [87] example is for *illustrative* purposes only. In other words, the same challenges depicted in the case of GPT-2 [87] apply generically to other models as well (see the later text in this section and our evaluations in Section 5).

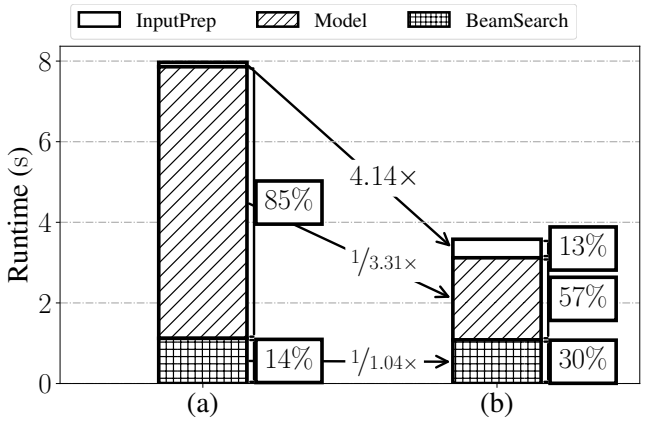



**Figure 3: Runtime breakdown of the GPT-2's text generation pipeline on an RTX 3090 GPU: (a) PyTorch (b) PyTorch + CUDA graphs**

in other domains such as speech recognition [8, 36]. Figure 2 illustrates the text generation pipeline of the GPT-2 [87] model. In each decoding iteration (e.g., ① in Figure 2), the GPT-2 model [87] first decodes the current input tokens for the probabilities of the next token on all the possible vocabularies. The input tokens have **dynamic sequence lengths** whose value can only be known at runtime. The model then applies a searching algorithm (e.g., the **beam search algorithm** [112]) that cherry-picks tokens to compose sentences that have the highest overall probabilities (②). This process continues until the stopping criteria are met (e.g., when the maximum generation length is reached or when the end-of-sequence token <EOS> is generated). To avoid re-evaluating the intermediate values of past input tokens, those **values are usually cached and forwarded to the next iteration** as inputs [84] (e.g., ① passes its intermediate values evaluated for "*I am*" to ③ via ④). Although the input tokens that need to be decoded in iteration ③ have a sequence length of 1, they need the intermediate values from the previous iteration, whose **sequence length is again dynamic**, to generate the next token.

Figure 3 illustrates the runtime breakdown of the GPT-2's [87] text generation pipeline on a modern NVIDIA RTX 3090 GPU [71], using the PyTorch [80] framework without and with the CUDA graphs applied (see Section 5.1 for the detailed methodology). We categorize the runtime into three portions: (1) *InputPrep*: the time spent on preparing input values to the model (e.g., filling non-provided arguments with their default values), (2) *Model*: the time spent on the model's forward pass (i.e., "GPT-2" in Figure 2), and (3) *BeamSearch*: the time spent on the beam search (i.e., "BeamSearch" in Figure 2). We make the following key observations from Figure 3:

• The *InputPrep* portion becomes 4.14× longer with the CUDA graphs applied (23% of the time on *Model*). This is because the data copying from runtime input values into the graphs' placeholders (❹ in Figure 1) incurs runtime overheads (**Challenge #1**), but such copying is an essential step for CUDA graphs to operate on distinct input values (see Section 2.1). More fundamentally, this is because by the time of tensor A's creation in Figure 1(b), the memory allocator of the machine learning framework is unable to foresee the future

| **Model** | GPT-2 [87] | GPT-J [114] | Wav2Vec2 [8] |
|---|---|---|---|
| **Memory** (MB) | 20 | 102 | 98 |
| **Operators** | 446 | 1666 | 1476 |

**Table 1: GPU memory consumption increase per CUDA graph creation and number of executed operators of the three models on an RTX 3090 GPU**

and tell that A will be copied to the placeholder. Therefore, it can only allocate a general memory region to A and copy its contents to the placeholder later, resulting in extra data movements. Despite the fact that one could manually force A to directly take the memory space of ph_A when A is being created by modifying the machine learning applications' source code, it is a tedious backtracking process that has to be done for all the placeholders. For example, there are 27 placeholders in the GPT-2 [87] model. Most of them are the intermediate values coming from the previous decoding iterations [84] (④ in Figure 2). Those values are created at different source code locations that span multiple files, making them hard to trace.

• Although CUDA graphs can speedup the *Model* portion by 3.31×, they nevertheless come with a significant increase in the GPU memory consumption due to the extra metadata: Each time a CUDA graph is created, a certain amount of GPU memory is allocated. The precise technical details regarding how the memory is used have not been revealed by NVIDIA, but we speculate that it is used for the CUDA graph's metadata by looking at the actual GPU memory content. This can be empirically observed in the results we show in Table 1, where we present the GPU memory consumption increase per CUDA graph creation on three state-of-the-art machine learning models: (1) GPT-2 [87] (introduced in the previous text), (2) GPT-J [114], whose architecture is similar to GPT-2 [87] but possesses 48.3× more parameters, and (3) Wav2Vec2 [8], which is the state-of-the-art speech recognition model. We observe from the table that the GPU memory consumption differs in each model and it scales proportionally with the number of operators executed in the model. We present more detailed evidence on the memory being the CUDA graph's metadata in Section 3.2.

Each memory allocation made by CUDA graphs is not particularly large in isolation, however, their cumulative impact can become significant when attempting to optimize dynamic-shape workloads whose input shapes vary upon each model invocation (**Challenge #2**). For example, to support the maximum sequence length of 1024, which is the context size when training the GPT-2 model [87], 20 GB of the GPU memory has to be allocated just for the CUDA graphs, which is in the same order as the GPU memory capacity of a modern RTX 3090 GPU [71] (24 GB). This does not even consider several complications. For example, when serving models like GPT-2 [87] and GPT-J [114], their inputs could be batched dynamically at runtime to have variable batch size per iteration [125]. Yet another example, when using CUDA graphs to train a model, the backward pass [97] of the model also requires distinct graph construction from its forward pass. All these complications further increase the number of CUDA graphs required and exacerbate the

GPU memory consumption problem, causing CUDA graphs to be less practical for dynamic-shape workloads.

• CUDA graphs are unable to optimize the *BeamSearch* portion because it involves complex data-dependent control flows (**Challenge #3**). Specifically, it needs to check whether the next token generated is the <EOS> token and decide whether the current sentence should be inserted into its internal scoreboard that maintains top-rank sentences based on the sentence's length and overall probability. These program flows cannot be handled by CUDA graphs due to the requirement that they have to be deterministic (as in Section 2.1). However, as Figure 3 shows, *BeamSearch* grows to 30% of the total execution time after the CUDA graphs are applied, and we are not able to squeeze the last drop of performance from GPUs with it being unoptimized.

## 2.3 Why A New Compiler?

Now that we have presented the key challenges on the practical and the efficient aspects of CUDA graphs, a natural question to ask is why a new compiler is needed to address them. Below, we answer this question by showing why prior works [69, 86, 105, 118] and some obvious solutions fall short in resolving these challenges.

**Eagerly release CUDA graphs**. One possible solution to the GPU memory challenge posed by the CUDA graphs is to construct them just-in-time before invoking the model and destruct them upon the model's completion. This solution is not feasible in practice, as the CUDA API that constructs the CUDA graph (namely `cudaGraphInstantiate` [72]) is prohibitively expensive runtime-wise (3–5 ms for one sequence length of the GPT-2 model [87] on an RTX 3090 GPU [71], which is more than 2× longer than running the model itself). Despite the high cost, the API call is necessary for the CUDA graphs to function correctly, as our experiments find that different CUDA graphs populate their respective memory regions with distinct values, even if they come from the same DNN but with different shapes. This is expected because machine learning frameworks such as PyTorch [80] are likely to launch the same CUDA kernel with distinct grid dimensions or even invoke distinct kernels when the input shapes vary [76, 130].

**Bucketing**. Another possible solution to the GPU memory challenge is to reduce the number of CUDA graphs needed using bucketing [69, 118]. Bucketing [69, 118] implements dynamic-shape workloads by dividing the dynamic ranges into buckets, each of which denotes a static shape. At runtime, the workload is dispatched to one of the buckets that fits its input shapes best by padding its inputs to the bucket's shape. Despite being straightforward, bucketing [69, 118] suffers from two key weaknesses: (1) It causes performance degradation due to padding [40, 130], and (2) it requires careful insertion of many additional masking operations into the models to prevent the padded values from affecting the output results [69], which leads to nontrivial engineering efforts. These weaknesses limit the practical use of bucketing [69, 118] in dynamic-shape workloads.

**One CUDA graph for every basic block/possible execution path**. We now study possible solutions to the data-dependent control flow challenge. S. Stevenson et al. [105] propose to convert every basic block (defined as contiguous statements without control flows) into its corresponding CUDA graph. This solution, however, is not practical for fragmented program flows such as the beam search algorithm [112] (where the average length of the basic blocks is only 2 statements approximately by our calculation), as launching multiple small CUDA graphs requires frequent CPU-GPU communications just like normal CUDA kernels, ruining the purpose of using them. TorchDynamo [86] uses graph breaks at control flows and instantiates one graph for every possible execution path. Hence, the number of paths it needs to handle grows exponentially with the number of control flows (i.e., $2^n$ where $n$ is the number of `if-else` statements in the program), resulting in huge GPU memory consumption (as every CUDA graph consumes GPU memory). This does not even consider the complications brought by `break` and `continue` statements in loops, hindering its practical use in graph-based executions.

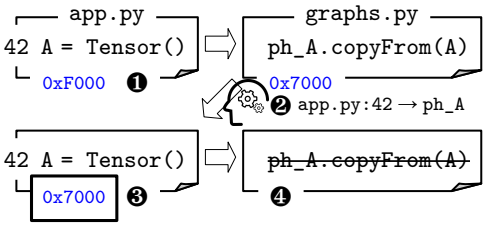



**Figure 4: An example that illustrates how the alias predictor eliminates data movements by predicting the placeholders' alias using the information from the Python frontend. The pointer values (`0x`'s) are used to demonstrate the aliasing.**

We now conclude that those prior works [69, 86, 105, 118] and obvious solutions are insufficient to resolve the challenges faced in graph-based executions for dynamic DNNs, and hence a solid new solution is required.

# 3 GRAPE: KEY IDEAS

To adequately address the challenges described in Section 2.2, we propose *Grape*, a new graph compiler that enables practical and efficient graph-based executions for dynamic DNNs on GPUs. *Grape* resolves the aforementioned Challenge #1-3 using three key components: (1) an **alias predictor** that accurately foretells if a tensor will be forwarded to the placeholders, eliminating extra data movements, (2) a **metadata compressor** that efficiently compresses CUDA graphs' memory regions, making them practical for dynamic-shape workloads, and (3) a **predication rewriter** that safely replaces data-dependent control flows with Python contexts, stitching the fragmented basic blocks together into a monolithic block that enables conversion to efficient graph-based executions. We now elaborate on those components.

## 3.1 Frontend-Aware Alias Prediction

Our first key idea, *frontend-aware alias prediction*, is based on the observation that smarter memory allocations can be made if the frontend Python code positions are considered. Current state-of-the-art machine learning frameworks [2, 80] allocate memory independently of the Python frontend. This works in general use cases, but is inefficient when graph-based executions are applied, as data movements are always required from general-purpose memory

(a)
```
struct Argument { int arr[ ArraySize ]; };
__global__ void Sample(Argument arg) {}
```

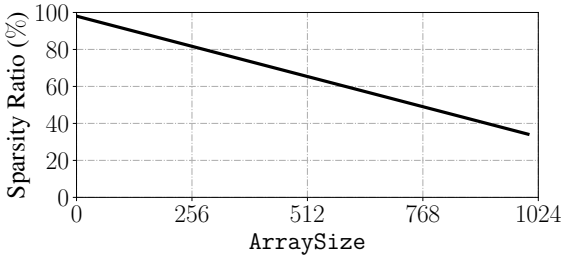
(b)



**Figure 5: (a) The microbenchmark that is used to study the CUDA graphs' memory regions. It captures a CUDA kernel `Sample` that takes `ArraySize` integers as an argument. (b) Sparsity of the captured CUDA graph's memory region versus the value of `ArraySize` (with every byte of `arr` being `0xFF`).**

regions to specially reserved placeholders' regions. We, however, notice that such data movements can be avoided by leveraging the regular nature of DNN executions and the rich lexical information from the Python frontend. Specifically, if a code position in Python yields a tensor that is copied into a placeholder in one iteration, it is also likely to yield another tensor that will be copied into the same placeholder in the subsequent iterations. The tensors are what we refer to as that placeholder's *aliases*.

Based on this insight, we devise an alias predictor that accurately predicts the placeholders' alias directly from the Python frontend. Figure 4 shows an example that illustrates its workflow: In the first iteration when a tensor is created, the alias predictor fetches a memory region from the memory allocator to it and records its code position in the Python script (file `app.py` line 42, ❶). By the time the tensor is copied into a placeholder, the predictor links its code position with the placeholder (❷), so that in the next iteration when the same position is encountered, it will proactively forward the placeholder's memory region to the tensor (❸). This eliminates the data movements (❹) and improves the efficiency of the graph-based executions, as our evaluation in Section 5.3 will show. Furthermore, the alias predictor is a transparent optimizer that requires zero changes to machine learning applications' source code.

## 3.2 Metadata Compression

Our second key idea, *metadata compression*, is based on the observation that the memory regions created by the CUDA graphs exhibit a huge level of data redundancy (e.g., usually more than 90% of all values are zeros), and therefore can be easily compressed with simple techniques such as run-length encodings [31]. Although the exact details regarding the CUDA graphs' memory regions are proprietary, by carefully studying their dumped values, we infer that they are for caching the CUDA graphs' metadata on the GPUs. The metadata of a CUDA graph describes what the graph captures and contains information such as the CUDA kernels that reside in the graph.

To prove our hypothesis that the memory regions are indeed used to store the metadata, and also to locate the source of the high sparsity of the CUDA graphs' memory regions, we consider the microbenchmark as in Figure 5(a), where we use the CUDA graphs to capture a single CUDA kernel `Sample`. The CUDA kernel accepts an object of type `Argument` as its function argument, whose member is an integer array of size `ArraySize`. In Figure 5(b), we examine the sparsity of the memory region of the CUDA graph that captures `Sample` while increasing the value of `ArraySize` and filling every byte with the value `0xFF`. We observe from Figure 5(b) that the sparsity of the memory regions decreases as we populate the data structure `Argument` with more non-zero values.This verifies our claim that the CUDA graphs' memory regions are used to cache the metadata that describes the CUDA kernels the graphs capture on GPUs, and a piece that forms the metadata is the function arguments of the CUDA kernels. Furthermore, since most of the CUDA kernels in the state-of-the-art machine learning workloads (e.g., GPT-2 [87], GPT-J [114], and Wav2Vec2 [8] in Table 1) underutilize the argument spaces that are provided to them and use pointer values as function arguments, they leave abundant sparsity and value redundancy (usually more than 90%) in the CUDA graphs' memory regions, making the latter highly compressible. Our evaluation in Section 5.4 shows that, by leveraging efficient data compression, we are able to use CUDA graphs to execute state-of-the-art dynamic DNNs [8, 114] that would normally consume up to 100 GB GPU memory on modern GPUs whose memory capacity is only 24 GB [71].

## 3.3 Predication Contexts

Our third key idea, *predication contexts*, is based on the observation that many data-dependent control flows can in fact be converted into equivalent forms that do not have control flows using the *predication contexts*, a new Python context that is illustrated in Figure 6(a). We define a predication context as follows: If the input $x$ in Figure 6(a) is evaluated to true, then all GPU operations within the context can go through and execute normally. Otherwise, all GPU operations within the context are *nullified*. The predication contexts realize the nullification by adding an extra argument `predicate` to every CUDA kernel of the machine learning framework's operator pool (see Figure 6(b)). At runtime, the input to a predication context is passed to all the CUDA kernels within it as a predicate, and the predicate's value controls whether the kernels' body will be executed or not.

Although the idea of predication has been proposed before in prior works [39, 42, 45, 63] at the architecture-level to handle short `if-else` statements, we significantly broaden its scope of applicability by enabling it to work on many common control flow patterns at the Python level. We show this in Figure 6(c)) where we demonstrate how common control flows such as `if`, `break`, and `continue` can be replaced by predication contexts, yielding equivalent programs with those control flows removed. Despite having the limitation of not being able to handle `break` statements in loops whose upper bound is unknown (e.g., `while True` statements), we notice that predication contexts are generic enough to cover important modules like the beam search module of GPT-like models [11, 78, 87, 106, 114] (shown in Figure 2).

```
with Predicate(x):
  # body statement
```

(a)

```
__gloabl__ void cudaKernel(
    …, bool* predicate) {
  if (*predicate) {
    // original kernel body
  }
}
```

(b)

| | if | break | continue |
|---|---|---|---|
| Original | `if x:`<br>`  # do A`<br>`else:`<br>`  # do B` | `for …:`<br>`  if x:`<br>`  break`<br>`  # do A` | `for …:`<br>`  if x:`<br>`  continue`<br>`  # do A` |
| Transformed | `with Predicate(x):`<br>`  # do A`<br>`with Predicate(!x):`<br>`  # do B` | `break_flag = CUDATensor(False)`<br>`for …:`<br>`  with Predicate(!break_flag):`<br>`  break_flag = x`<br>`  with Predicate(!break_flag):`<br>`  # do A` | `for …:`<br>`  with Predicate(!x):`<br>`  # do A` |

(c)

**Figure 6: (a) An example predication context. (b) The predication contexts control whether the GPU operations within them are nullified or not by using a predicate. (c) A showcase of how common control flows such as `if`, `break`[2], and `continue` (from left to right) can be replaced by the predication contexts.**

We further observe from the design of predication contexts that they do not change the launch configurations (e.g., block and grid dimensions) of CUDA kernels within them. This complies with the *deterministic* constraint imposed by graph-based executions (see Section 2.1). Therefore, even if a program module has control flows and fragmented basic blocks, we can rewrite it into a form that does not have control flows using the predication contexts, which can be captured as a monolithic CUDA graph. The graph behaves as if we stitch the fragmented basic blocks together and is more efficient than the original implementation with control flows as it no longer needs to frequently synchronize with the CPUs for control flow operations (see our evaluation in Section 5.3). In contrast to TorchDynamo [86], which needs to instantiate a number of graphs that grows exponentially with the number of `if-else` statements and cannot practically handle `break` and `continue` statements in loops, predication contexts create a single monolithic graph that can incorporate any number of common control flows (e.g., `if`, `break`, and `continue`) and therefore do not lead to huge GPU memory consumption (24 MB for the beam search module in Section 2.2).

## 4 IMPLEMENTATION DETAILS

We implement *Grape* in PyTorch [80], a state-of-the-art machine learning framework. To enable straightforward conversions from PyTorch [80] neural network modules into their equivalent compiled forms, we develop a top-level interface (shown in Figure 8(a)) that is easy for machine learning practitioners to use. The interface accepts two key arguments: (1) `module`: the module that is to be compiled and optimized, (2) `module_args_generator`: a tensor generator that yields a list of tensor arguments to the module (to be used as placeholders). Figure 8(b) demonstrates an example of using the interface, where we compile the GPT-2 model [87] from the HuggingFace Transformers repository [120] with dynamic sequence lengths ($\in [1, 1024)$). The compiled `GrapeModule` can work as a drop-in replacement of the original `module` and supports both training and inference.

[2]The statement `break_flag = CUDATensor(False)` means the `break_flag` is a boolean scalar on the GPU and it is set to false.

| Component | Timing | Code Changes |
|---|---|---|
| Alias Predictor | Runtime | None |
| Metadata Compressor | Compile time* and Runtime† | None |
| Predication Rewriter | Compile time | Rewrite as per Table 6(c). |

* Compression † Decompression

**Table 2: Overview of *Grape*'s Key Components**

Table 2 provides an overview of the three key components of *Grape*, including whether they are done at compile time or runtime and the changes that they require to machine learning applications' source code. In the following subsections, we highlight some of the important details of the design of those key components.

### 4.1 Alias Predictor

We implement the alias predictor by monitoring the traffic between PyTorch's [80] Python frontend and its GPU memory allocator (i.e., `CUDACachingAllocator`). Every time a GPU memory allocation is requested by the frontend, we attach the current Python thread state to the data pointer object that is returned by the caching allocator. The thread state is queried from the Python C APIs [85] and contains information such as the current code object and the last instruction executed. When a copy is made from a tensor to a CUDA graph's placeholder (marked by specially reserved memory regions in PyTorch [80]), the alias predictor will be invoked to record the connection between the tensor's thread state when it was created and the corresponding placeholder by marking the thread state as generating the placeholder's alias. The next time another thread state that has the same code object and last instruction executed

```
from torch.nn import Module

class GrapeModule(Module):
  def __init__(
    module,
    module_args_generator,
  ):
    …
```

(a)

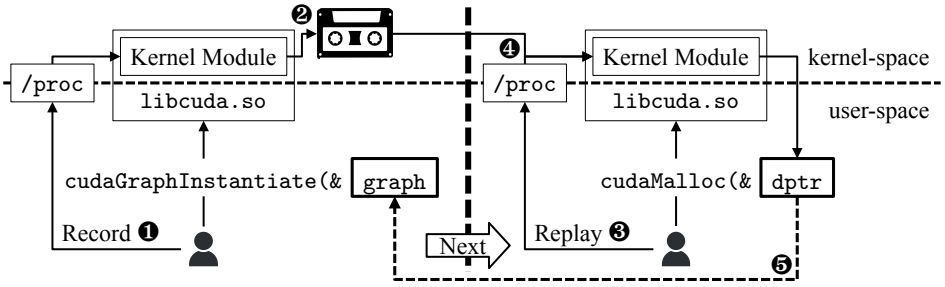



**Figure 7: The workflow of white-boxing CUDA graphs' memory regions**

```
from transformers import gpt2

def _args_generator():
  for seq_len in range(1, 1024):
    yield RandomCUDATensor(shape=(1, seq_len))

compiled_gpt2 = GrapeModule(gpt2, _args_generator)
```

(b)

**Figure 8: (a) *Grape*'s interface and (b) An example use case of *Grape* on compiling GPT-2**

as the recorded thread state is encountered, the alias predictor will predict it to be an alias of the recorded placeholder and proactively forward the placeholder's memory region to it, eliminating extra data movements in the future.

A misprediction happens when a tensor is predicted to be a placeholder's alias but is later discovered not to be so. We detect misprediction by checking whether the real inputs have their pointer addresses aligned with the placeholders at runtime (e.g., we check if `A`'s pointer address aligns with `ph_A` at ❹ in Figure 4, and mark ❷ as mispredicted if `A` is reassigned and no longer takes on the address `0x7000`). In the case of a misprediction, the mispredicted tensor will have to move its memory contents out from the placeholder's memory region. This incurs runtime overheads but still preserves program correctness, as it is possible to use a placeholder's memory region for general purposes but not vice versa. Furthermore, we notice from our evaluation in Section 5 that the misprediction rarely happens thanks to the regular nature of DNN executions.

The alias predictor is enabled by default during graph-based executions and it allows for automatic and transparent removals of data movements into placeholders (i.e., no changes to machine learning applications' source code). Our evaluation in Section 5.3 shows that the alias predictor is able to speed up the input preparation time of graph-based executions by up to 2.07×.

## 4.2 Metadata Compressor

As a prerequisite to implementing the metadata compressor, we first need to find a way to access (i.e., white-box) the CUDA graphs' memory regions as CUDA [72] is NVIDIA's proprietary software and does not give access to the memory that it allocates for CUDA graphs. We accomplish this by using a customized GPU kernel module that is modified from the open-sourced release [75]. The kernel module is able to *record* and *replay* GPU memory allocations depending on the command that we send to it via the `/proc` filesystem. It grants us access to private GPU memory allocations that are made by CUDA [72] but not exposed to its users.

Figure 7 illustrates the white-boxing workflow: Before a CUDA graph is created using the `cudaGraphInstantiate` function call, we issue a *record* command to the kernel module (❶ in Figure 7) to have it record the memory resource parameter assigned to that CUDA graph (including information such as its size, address, and unique handle, ❷). Immediately after the instantiation call resolves, we issue a *replay* command to the kernel module (❸) while invoking a `cudaMalloc` function call from the user space that requests a memory region of the same size as the CUDA graph. In the replay mode, the kernel module bypasses physical memory allocations and directly tapes out the previously recorded memory resource parameter to serve the request (❹). This hence has the data pointer returned by `cudaMalloc` point to the memory region that was previously allocated to the CUDA graph (❺), allowing us to inspect and manipulate it (i.e., compress and decompress) in the user space.

As we examine the dumped values of CUDA graphs' memory regions, we notice that they are full of zeros and repeated values (see Section 3.2 for the explanation). Based on this observation, we use the run-length encoding algorithm [31] to compress them. To make the algorithm more friendly to GPUs, we develop a page-based variant of it that allows each GPU thread to decompress each page in parallel. Specifically, at compile time, we fetch CUDA graphs' memory regions from the GPU to the CPU and have the CPU split the regions into pages[3], compress those pages sequentially, and send the compressed pages to the GPU upon completion. Each time a page is compressed, we record the current size of the compressed data in a separate array, so that when decompression happens, each thread knows exactly which part of the compressed data corresponds to its own page by reading the array. Our evaluation in Section 5.4 shows that this page-based run-length encoding is both effective and efficient, as it achieves up to 36.43× compression ratio with negligible performance overhead (less than 1% of the time spent on the graph-based executions of DNNs).

[3]We use a page size of 2 KB to have good compression ratios and enough parallelism in decompression

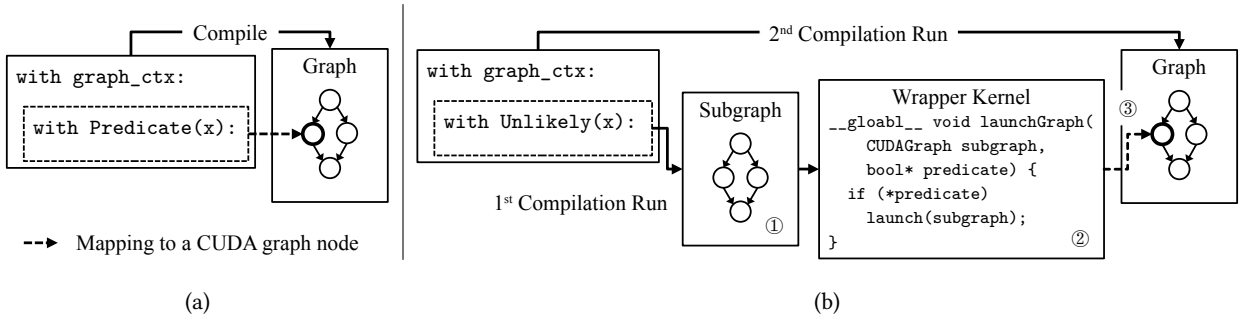



**Figure 9: Comparison between compilation workflow of (a) `Predicate` and (b) `UnlikelyPredicate`**

## 4.3 Predication Rewriter

We present two important implementation details on the predication contexts, including (1) the source code transformations that we do to make them functional in PyTorch [80], and (2) the `UnlikelyPredicate` context variant that we support, which provides machine learning practitioners with the flexibility of marking contexts as *unlikely* for further performance benefits.

*4.3.1 Predication-Enabled Framework.* To enable the predication contexts, we need to enhance the PyTorch's [80] C++ runtime. We first do a lookup in the PyTorch's [80] operator pool for all the CUDA `__global__` functions. For each `__global__` function we find, we make a copy of its implementation and append an extra argument `predicate` to the copy. The predicate governs the copy's entire body (as in Figure 6(b)) and controls whether the kernel is nullified or not. In addition to transforming GPU kernels, we also look for CUDA APIs [72] that implicitly invoke GPU operations under the hood (e.g., `cudaMemcpyAsync`) and develop their equivalent CUDA kernel implementations (also with a `predicate` boolean variable appended at the end of their arguments and governing their kernel bodies). Furthermore, we maintain a global variable named `gCurrentPredicate` in the C++ runtime. The variable points to the current predicate value in effect (evaluated from the inputs to predication contexts) and can be accessed by all GPU kernels. Whenever a kernel in a predication context is invoked, we pass this predicate variable as the last argument. If this predicate variable is evaluated to true, the kernel will behave as the original. Otherwise, the kernel will be nullified.

When executing a DNN, we switch between the CUDA kernel implementations that have a predicate and those that do not by checking whether there is a predication context in effect. This ensures that the enhancements that we implement do not affect the performance and the correctness in the case of non-graph-based executions.

*4.3.2 Unlikely Predicates.* Although the `predicate` in Figure 6(b) is able to shorten most of the execution time of the CUDA kernel when it is evaluated to false, executing an empty CUDA kernel still takes time (in the order of several microseconds on a modern RTX 3090 GPU [71]). This small amount of time could nevertheless sum up to be a noticeable runtime overhead when there are many GPU operations within a predication context. To derive further performance benefits, we allow machine learning practitioners to specify a predication context that is unlikely to be executed using an `UnlikelyPredicate` context, similar to the `unlikely` attribute in the C++ programming language [19].

Figure 9 compares the compilation workflow of a `Predicate` context with that of an `UnlikelyPredicate` context. Unlike a `Predicate` context that can be compiled in a single run (Figure 9(a)), an `UnlikelyPredicate` context requires two compilation runs (Figure 9(b)). In the first run, the capture for the parent graph context `graph_ctx` is turned off and the GPU operations within the child `Unlikely` context are first captured as a standalone CUDA subgraph (① in Figure 9(b)). The subgraph is configured to be launchable from the GPU side and wrapped in a CUDA kernel (②). In the second run, the capture for the parent graph context is turned on and the wrapper kernel is used in replacement of the body statements of `Unlikely` (③).

Compared with `Predicate` contexts, `UnlikelyPredicate` contexts possess the strength of having constant-time latencies when they are not taken (regardless of how many statements they have). However, they have the weakness of not being nestable (i.e., we cannot nest one `UnlikelyPredicate` inside another one due to the lack of programming language support in CUDA [72]). They also require domain-specific knowledge inputs from frontend machine learning practitioners to indicate which predicates are unlikely. These shortcomings make them less generic when compared to `Predicate` contexts.

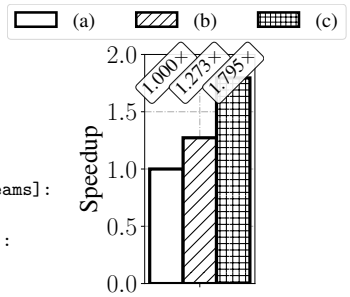
As an example application of the `UnlikelyPredicate` contexts, we consider the beam search module [112] of GPT-2 [87]. In the current state-of-the-art implementation of beam search [120], the decoding output from the GPT-2 model [87] (which represents the probabilities of the next token on all the possible vocabularies) is first cherry-picked to filter out the top $2\times$*nbeams* tokens (where *nbeams* is the beam width of the beam search algorithm [112]) before given to the beam search module. The reason for this $2\times$ over-provisioning is because it is possible for a filtered token to be the end-of-sequence token `<EOS>`, which is not selected by the beam search module for generating future tokens. The beam search module iterates through those filtered tokens until it has *nbeams* tokens that are not `<EOS>` before giving them to the GPT-2 model [87] for the next decoding iteration. The program that shows the high-level control flow and its semantically equivalent form using `Predicate` contexts are illustrated respectively in Figure 10(a) and (b)[4].

[4]For conciseness, the loop body has been simplified.

```
idx = 0
for t in tokens[:2*nbeams]:
  if t != EOS:
    idx += 1
  if idx == nbeams:
    break
```
(a)

```
idx = CUDATensor(0)
flag = CUDATensor(True)
for t in tokens[:2*nbeams]:
  with Predicate(flag):
    with Predicate(t != EOS):
      idx += 1
    flag = idx != nbeams
```
(b)

```
idx = CUDATensor(0)
for t in tokens[:nbeams]:
  with Predicate(t != EOS):
    idx += 1
flag = idx != nbeams
for t in tokens[nbeams:2*nbeams]:
  with Unlikely(flag):
    with Predicate(t != EOS):
      idx += 1
    flag = idx != nbeams
```
(c)



(d)

**Figure 10: (a) The original Python program and its semantically equivalent forms that use (b) `Predicate` contexts and (c) `UnlikelyPredicate` contexts. (d) Performance comparison between (a-c) on an RTX 3090 GPU.**

We observe, however, that since the probability of generating an <EOS> token is low (once per roughly one hundred decoding iterations in our evaluation), the first *nbeams* tokens are usually enough to fulfill the *nbeams* requirement of the beam search module. Based on this observation, we can split the loop [7, 79] into two (before and after *nbeams*) and mark the second loop as *unlikely*. This is illustrated in Figure 10(c).

We measure the runtime of the three implementations (the original program, the one using `Predicate` contexts, and the one using `UnlikelyPredicate`) on an RTX 3090 GPU [71] (with the latter two compiled using *Grape*). Figure 10(d) shows the relative speedup over the original program (higher is better). We observe from the figure that while `Predicate` contexts achieve a speedup of 1.27× over the original program, `UnlikelyPredicate` contexts further increase this speedup to 1.79× (hence is 1.41× better than pure `Predicate` contexts), demonstrating their value if domain-specific knowledge can be provided.

# 5 EVALUATION

## 5.1 Methodology

**Infrastructure**. Our main compute platform is a single machine that is equipped with a 12-core AMD Ryzen Threadripper 2920X CPU [3] (with 64 GB DDR4 [48] memory) and an NVIDIA RTX 3090 GPU [71] (Ampere architecture [71] with 24 GB GDDR6 memory [49]), connected via the PCIe v3 interconnect [82]. The platform is installed with the NVIDIA GPU driver version 525.89.02 [73] (using the open-sourced GPU kernel module [75] of the same version), CUDA 11.6 [72], cuDNN 8.4.0.27 [16], TensorRT 8.2.5 [111], and PyTorch 1.12 [80].

To prove that *Grape*'s key ideas apply generically to other hardware platforms, we also evaluate on a GCP `a2-highgpu-1g` [34] instance that is equipped with 12 Intel Xeon 8273CL virtual CPUs [20] (with 85 GB RAM) and an NVIDIA A100 GPU [70] (Ampere architecture [70] with 40 GB HBM2 [50]), using the same software stack as the main compute platform.

**Applications**. We evaluate our new compiler, *Grape*, on three state-of-the-art DNN models from the HuggingFace Transformers repository ver. 4.18 [120]: GPT-2 [87], GPT-J [114] (with 6 billion parameters), and Wav2Vec2 [8]. We use a maximum sequence length of 1024, which corresponds to the context size that the GPT-2 model [87] is trained on, a batch size of 1, and a beam width of 5 [112] when generating texts with both the GPT-2 [87] and the GPT-J [114] model. Additionally, we use a training batch size of 8 to fine-tune the Wav2Vec2 model [8] on the TIMIT dataset [30] for one epoch [113] (with 391 different input shapes).

**Baselines**. We compare *Grape* with two main baselines: (1) the original implementation from PyTorch [80] without CUDA graphs, denoted as *Baseline*, and (2) the graph-based executions from PyTorch [80] using CUDA graphs that handle dynamic-shape workloads by capturing one graph for one input shape, denoted as *PtGraph*. All three systems use cuDNN [16] and invoke the exact same set of CUDA kernels under the hood.

**Metrics**. We show results on (1) the end-to-end latency on the entire model, measured in seconds (lower is better), (2) the runtime of each submodule (*InputPrep*, *Model*, and *BeamSearch*, as in Figure 3) in one decoding iteration, measured in seconds (lower is better), and (3) the GPU memory consumption per CUDA graph created, measured in MB (lower is better).

## 5.2 End-to-End Results

Figure 11 shows the end-to-end latency comparison between *Baseline*, *PtGraph*, and *Grape* on the three machine learning workloads and the two hardware platforms. We are unable to get *PtGraph* functional on the GPT-J [114] and the Wav2Vec2 [8] model due to the GPU memory consumption challenge (Section 2.2 Challenge #2). We make the following key observations from Figure 11(a):

(1) *Grape* achieves noticeable speedups (up to 2.97×) over *Baseline* on all the three models evaluated [8, 87, 114], showing the importance of using graph-based executions to boost the performance of machine learning workloads.

(2) On the GPT-2 [87] model where both *PtGraph* and *Grape* can be applied, *Grape* outperforms *PtGraph* by 1.26×. This is because *Grape*'s alias predictor saves the extra data movements of copying runtime input values to the graph's placeholders and its predication rewriter allows it to optimize the beam search module, which is not optimizable by *PtGraph* due to its data-dependent control flows (see Section 5.3 for the breakdown between the two components).

(3) *Grape* can be applied to workloads such as the GPT-J [114] and Wav2Vec2 [8] model where *PtGraph* is not practical due to the extremely large GPU memory consumption, achieving a speedup of 1.41× and 1.38× respectively. This is due to its capability of reducing the GPU memory consumption of CUDA graphs by leveraging the high sparsity and value redundancy of their memory regions (see Section 5.4).

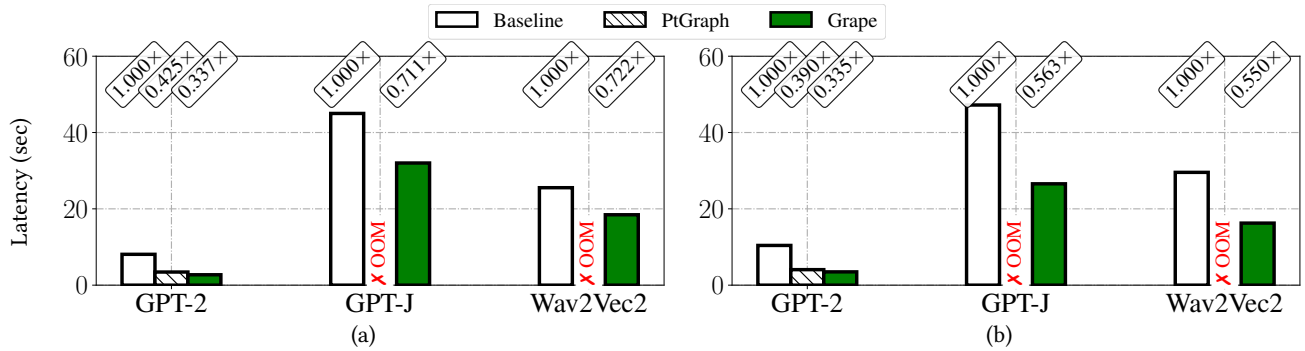



**Figure 11: End-to-end performance of GPT-2, GPT-J (both are inference), and Wav2Vec2 (training) compared between *Baseline*, *PtGraph*, and *Grape* on (a) RTX 3090 and (b) A100**

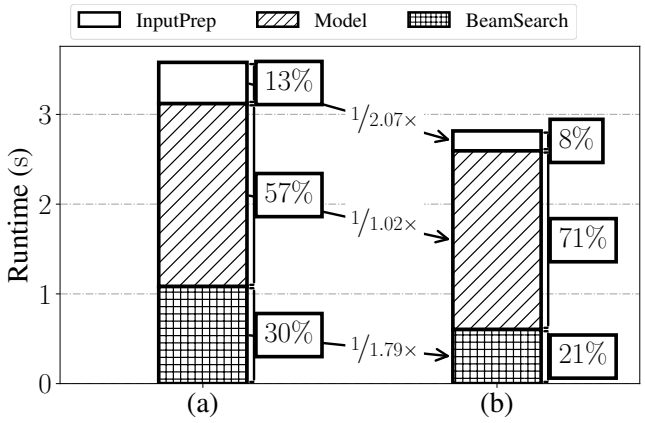



**Figure 12: Runtime breakdown of the GPT-2's text generation pipeline: (a) *PtGraph* (b) *Grape***

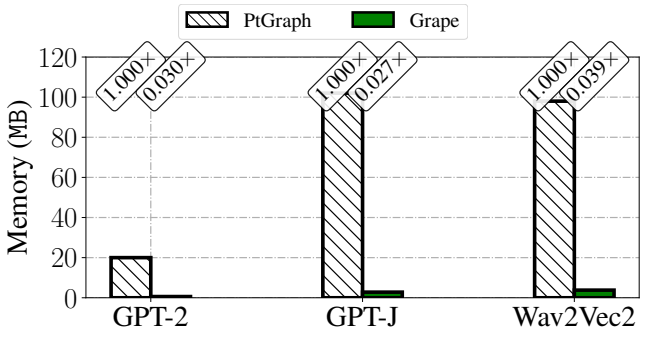



**Figure 13: Comparison between *PtGraph* and *Grape* on the GPU memory consumption per CUDA graph creation**

We further observe from Figure 11(b) that *Grape* works equally well on the A100 [70] hardware platform, achieving even greater speedup (up to 2.99×) compared with *Baseline*. This not only proves *Grape*'s generality across different hardware platforms, but stresses its importance further as GPUs with stronger compute power are used [10].

## 5.3 Breakdown of Performance Speedup

To better understand the performance speedup of *Grape* over *PtGraph*, we analyze again the runtime breakdown of the GPT-2's [87] text generation pipeline on the RTX 3090 GPU [71], which is similar to Figure 3 but compares between *PtGraph* and *Grape*. Figure 12 shows the runtime breakdown comparison. We observe from the figure that *Grape* is able to speed up the *InputPrep* portion by 2.07× and the *BeamSearch* portion by 1.79×. While the former is contributed by the alias predictor (34% of the 1.26× speedup of *Grape* over *PtGraph* in Figure 11(a), the latter is by the predication rewriter (66% of the speedup). This indicates that both components are necessary to maximize the performance of graph-based executions.

## 5.4 Metadata Compression

To better understand how the GPU memory consumption challenge is resolved by *Grape*, we show the GPU memory consumption per CUDA graph creation in Figure 13 on the RTX 3090 GPU [71]. We observe from the figure that *Grape* is able to achieve significant compression ratios over *PtGraph* across all the three models evaluated [8, 87, 114] (up to 36.43× GPU memory reduction).

We also study the compilation and the runtime overheads that are caused respectively by compressing and decompressing the CUDA graphs' memory regions. We notice that while the compression overheads are large (constituting 19.8%, 58.5%, and 30.3% of the total compilation time for GPT-2 [87], GPT-J [114], and Wav2Vec2 [8] respectively). They are done ahead-of-time and hence do not affect the model's runtime performance (which is why we implement the compression on the CPU and do not parallelize it for simplicity). We also observe that the decompression runtime overheads are negligible when compared with the models' execution time (i.e., less than 1%), this is because the decompression of the page-based run-length encoding that we adopt is highly parallelizable and friendly to GPUs.

We further repeat the experiments on A100 [70] and observe similar numbers. This is expected since the GPU memory allocations are made by the CUDA software stack [72] and hence are largely independent of the underlying GPUs, indicating *Grape* brings in

practical and efficient graph-based executions for dynamic-shape workloads on different GPUs.

## 6 RELATED WORKS

*Grape* addresses key challenges in making graph-based executions both practical and efficient for dynamic DNNs. As *Grape* aims at eliminating CPU overheads, it is orthogonal to many system optimization techniques such as reduced precision [66, 67, 116], quantization [38, 44, 52, 57, 103, 108, 121], and operator fusions [14, 21, 59, 62, 98, 99, 117, 128, 136]. In fact, those optimizations can be used jointly with *Grape* to derive even greater performance benefits.

**DNN Benchmarking and Profiling Analyses**. *Grape* focuses on the efficiency of state-of-the-art dynamic DNNs and uses models that are prevalent in the area of text generation and speech recognition for its evaluation. TBD [137], MLPerf Training [64], MLPerf Inference [91], and DawnBench [18] are benchmark suites that bring together state-of-the-art DNNs from various domains (such as image classification [41], object detection [61], and recommendation [68]). To help machine learning practitioners understand the runtime and GPU memory allocation behaviors of those important DNNs, there are profiling tools that accurately diagnose DNN executions with visualizations, such as DeepView [12], Skyline [126], RL-Scope [32], and Hotline [104]. Those useful tools greatly inspire part of the ideas in *Grape*.

**Machine Learning Compilers**. *Grape* is a compiler for machine learning workloads, hence it is related to numerous research works in the area of machine learning compilers, including those that rewrite DNNs at the graph level [23, 51, 115, 123, 133], those that allow easy development of low-level tensor programs [9, 14, 21, 26, 37, 60, 101, 110, 119, 122, 129, 132, 134, 135, 138], those that target efficient code generation for irregular [25, 109], sparse [124], dynamic-shape [102, 130, 139], or recursive [24] workloads, those that propose expressive graph-level representations [77, 86, 92], and those that cover multi-level optimizations [58, 98]. These machine learning compilers are orthogonal to *Grape*. In fact, we notice that some compiler frameworks such as Hidet [21] and TVM [14] already adopt graph-based executions in their codebase. It is hence possible for them to benefit from *Grape*.

**GPU memory optimizations**. As DNNs become large, GPU memory optimizations are usually applied to avoid hitting the GPU memory capacity wall. Common optimization strategies include: virtualization [89, 93, 95], data compression [17, 46, 96, 127], rematerialization [15, 47, 54, 83, 88, 90, 131], and reverse computation [6, 13, 33]. Those techniques are related to the metadata compressor of *Grape*, but *Grape* focuses on the memory regions for CUDA graphs rather than those for DNNs' data values.

**Data-dependent control flows**. It is challenging to handle data-dependent control flows on a SIMT hardware platform like GPUs [1]. To address this challenge, prior works propose solutions such as warp compaction [27–29], multi-path execution [22, 65, 94], MIMD [53, 55, 56], and predication [39, 42, 45, 63]. Although the idea of predication [39, 42, 45, 63] is similar in spirit to that of *Grape*'s predication contexts, the former is proposed at the architecture level to handle short `if-else` statements, whereas the latter works at the Python level and can handle common control flow patterns (e.g., `if`, `break`, `continue`) in state-of-the-art DNN workloads.

## 7 CONCLUSION

We build *Grape*, a new graph compiler that enables practical and efficient graph-based executions for dynamic DNNs. On the three state-of-the-art DNNs evaluated, *Grape* improves performance by up to 1.26× better than prior works on graph-based executions. Moreover, *Grape* significantly broadens the optimization scope of graph-based executions by compressing their GPU memory consumption by up to 36.43×, enabling it to optimize workloads that are impractical for prior methods and resulting in up to 1.82× better performance. System researchers and machine learning practitioners can all benefit from *Grape*, as it speeds up training and inference workloads using graph-based executions while allowing easy development at the Python level. We hope that *Grape* would become a platform for further research on efficient system design for key machine learning applications.

## ACKNOWLEDGMENTS

We want to express our sincere gratitude to the anonymous MICRO reviewers for their valuable and constructive feedback and suggestions, and the artifact evaluation reviewers for their precious time in reproducing our experiments. The authors with the University of Toronto are supported by the Canada Foundation for Innovation JELF grant, NSERC Discovery grant, AWS Machine Learning Research Award (MLRA), Facebook Faculty Research Award, Google Scholar Research Award, and VMware Early Career Faculty Grant.

## A ARTIFACT APPENDIX

### A.1 Abstract

This appendix works through the key experiments in Section 5. Specifically, we compare the runtime performance and the GPU memory consumption against the original implementation from PyTorch [80] without CUDA graphs and the graph-based executions from PyTorch [80] using CUDA graphs. In the public GitHub repository, we provide scripts that automatically set up the software environment and run the experiments that correspond to Figure 11 to 13.

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

- **Compilation:** Open GPU kernel module (ver. 525.89.02) [75] and PyTorch (ver. 1.12.0) [80].
- **Transformations:** Convert DNNs into their corresponding graphed forms.
- **Model:** GPT-2 [87], GPT-J [114], and Wav2Vec2 [8] (all from the HuggingFace Transformers repository [120]).
- **Data set:** TIMIT [30] for fine-tuning the Wav2Vec2 model [8].
- **Hardware:** The results from the paper are obtained on (1) a hardware platform equipped with a 12-core AMD Ryzen Threadripper 2920X CPU [3] (with 64 GB DDR4 [48] memory) and an NVIDIA RTX 3090 GPU [71], and (2) a GCP `a2-highgpu-1g` instance [34] that is equipped with 12 Intel Xeon 8273CL virtual CPUs [20] (with 85 GB RAM) and an NVIDIA A100 GPU [70].
- **Metrics:** Latencies (measured as seconds, lower is better) and GPU memory consumption (measured as MB, lower is better).
- **Experiments:** Use the provided scripts to run the experiments.
- **How much disk space required (approximately)?:** 100 GB

- **How much time is needed to prepare workflow (approximately)?:** It takes several minutes to compile the open GPU kernel module [75] and an hour to compile PyTorch [80].
- **How much time is needed to complete experiments (approximately)?:** It takes several minutes to run the runtime performance experiments of the GPT-2 [87] and the Wav2Vec2 [8] model, and an hour to run experiment of the GPT-J model [114]. The same also applies to the GPU memory experiments.
- **Publicly available?:** https://github.com/UofT-EcoSystem/Grape-MICRO56-Artifact
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.8322658

## A.3 Description

*A.3.1 How to access.* We open-source our artifacts in the GitHub repository at https://github.com/UofT-EcoSystem/Grape-MICRO56-Artifact.

*A.3.2 Hardware dependencies.* The results from the paper are obtained on (1) a hardware platform equipped with a 12-core AMD Ryzen Threadripper 2920X CPU [3] (with 64 GB DDR4 [48] memory) and an NVIDIA RTX 3090 GPU [71], and (2) a GCP `a2-highgpu-1g` instance [34] that is equipped with 12 Intel Xeon 8273CL virtual CPUs [20] (with 85 GB RAM) and an NVIDIA A100 GPU [70].

*A.3.3 Software dependencies.* The software dependencies are listed in the setup script of the GitHub repository: https://github.com/UofT-EcoSystem/Grape-MICRO56-Artifact/blob/main/scripts/0-install_build_essentials.sh

*A.3.4 Data sets.* We use the TIMIT dataset [30] for fine-tuning the Wav2Vec2 model [8].

*A.3.5 Models.* We use the GPT-2 [87] and the GPT-J [114] model (state-of-the-art text generation models, inference only) and the Wav2Vec2 model [8] (state-of-the-art speech recognition model, fine-tuned on the TIMIT dataset [30]).

## A.4 Installation

The installation steps are detailed in https://github.com/UofT-EcoSystem/Grape-MICRO56-Artifact/wiki#installation.

## A.5 Experiment workflow

The experiment workflow is detailed in https://github.com/UofT-EcoSystem/Grape-MICRO56-Artifact/wiki#experiment-workflow.

## A.6 Evaluation and expected results

The three scripts in the experiment workflow automatically run the experiments that correspond respectively to Figure 11, 12, and 13 in Section 5. After each experiment is completed, a CSV file will be dumped into the `experiments` folder. Illustrating those CSV files produces Figure 11 to Figure 13.

## A.7 Experiment customization

Not applicable.

## A.8 Notes

## A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

# REFERENCES


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

[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] AMD. 2022. AMD Ryzen™ Threadripper™ 2920X Drivers & Support. https://www.amd.com/en/support/cpu/amd-ryzen-processors/amd-ryzen-threadripper-processors/amd-ryzen-threadripper-2920x Accessed on April 28 2023.

[4] AMD. 2023. Graph Management. https://rocm.docs.amd.com/projects/HIP/en/docs-5.6.0/.doxygen/docBin/html/group___graph.html#graph-management

[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] Muralidhar Andoorveedu, Zhanda Zhu, Bojian Zheng, and Gennady Pekhimenko. 2022. Tempo: Accelerating Transformer-Based Model Training through Memory Footprint Reduction. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=xqyEG7EhTZ

[7] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler Transformations for High-Performance Computing. *Comput. Surveys* 26, 4 (1994), 345–420. https://doi.org/10.1145/197405.197406

[8] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12449–12460. https://proceedings.neurips.cc/paper_files/paper/2020/file/92d1e1eb1cd6f9fba3227870bb6d7f07-Paper.pdf

[9] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[10] BIZON. 2023. 2021 2020 Deep Learning Benchmarks Comparison: NVIDIA RTX 3090 vs NVIDIA A100 40 GB (PCIe). https://bizon-tech.com/gpu-benchmarks/NVIDIA-RTX-3090-vs-NVIDIA-A100-40-GB-(PCIe)/579vs592 Accessed on April 28 2023.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[12] CentML. 2023. CentML DeepView. https://docs.centml.ai/deepview/index.html Accessed on September 18th, 2023.

[13] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. 2018. Reversible Architectures for Arbitrarily Deep Residual Neural

Networks. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, Sheila A. McIlraith and Kilian Q. Weinberger (Eds.). AAAI Press, 2811–2818. https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16517

[14] 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

[15] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. *CoRR* abs/1604.06174 (2016). arXiv:1604.06174 http://arxiv.org/abs/1604.06174

[16] 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

[17] Esha Choukse, Michael B. Sullivan, Mike O'Connor, Mattan Erez, Jeff Pool, David W. Nellans, and Stephen W. Keckler. 2020. Buddy Compression: Enabling Larger Memory for Deep Learning and HPC Workloads on GPUs. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 926–939. https://doi.org/10.1109/ISCA45697.2020.00080

[18] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Christopher Ré, and Matei Zaharia. 2019. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Operating Systems Review* 53, 1 (2019), 14–25. https://doi.org/10.1145/3352020.3352024

[19] cppreference.com. 2022. C++ attribute: likely, unlikely (since C++20). https://en.cppreference.com/w/cpp/language/attributes/likely Accessed on April 28 2023.

[20] CPU World. 2023. Intel Xeon 8273CL specifications. https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%208273CL.html Accessed on April 28 2023.

[21] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 2023. Hidet: Task-Mapping Programming Paradigm for Deep Learning Tensor Programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 370–384. https://doi.org/10.1145/3575693.3575702

[22] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M. Aamodt. 2014. A scalable multi-path microarchitecture for efficient GPU control flow. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 248–259. https://doi.org/10.1109/HPCA.2014.6835936

[23] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN Computation Graph using Graph Substitutions. *Proceedings of the VLDB Endowment* 13, 11 (2020), 2734–2746. http://www.vldb.org/pvldb/vol13/p2734-fang.pdf

[24] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. 2021. Cortex: A Compiler for Recursive Deep Learning Models. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/182be0c5cdcd5072bb1864cdee4d3d6e-Abstract.html

[25] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. 2022. The CoRa Tensor Compiler: Compilation for Ragged Tensors with Minimal Padding. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2022/hash/d3d9446802a44259755d38e6d163e820-Abstract.html

[26] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 2023. TensorIR: An Abstraction for Automatic Tensorized Program Optimization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 804–817. https://doi.org/10.1145/3575693.3576933

[27] Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *17th International Conference on High-Performance Computer Architecture (HPCA-17 2011), February 12-16 2011, San Antonio, Texas, USA*. IEEE Computer Society, 25–36. https://doi.org/10.1109/HPCA.2011.5749714

[28] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. 2007. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-40 2007), 1-5 December 2007, Chicago, Illinois, USA*. IEEE Computer Society, 407–420. https://doi.org/10.1109/MICRO.2007.30

[29] Wilson W. L. Fung, Ivan Sham, George L. Yuan, and Tor M. Aamodt. 2009. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Transactions on Architecture and Code Optimization* 6, 2 (2009), 7:1–7:37. https://doi.org/10.1145/1543753.1543756

[30] John S. Garofolo, Lori F. Lamel, William M. Fisher, Jonathan G. Fiscus, David S. Pallett, Nancy L. Dahlgren, and Victor Zue. 1993. TIMIT Acoustic-Phonetic Continuous Speech Corpus. *Linguistic Data Consortium, Philadelphia*. https://doi.org/10.35111/17gk-bn40

[31] GeeksforGeeks. 2022. Run Length Encoding and Decoding. https://www.geeksforgeeks.org/run-length-encoding/ Accessed on April 28 2023.

[32] James Gleeson, Moshe Gabel, Gennady Pekhimenko, Eyal de Lara, Srivatsan Krishnan, and Vijay Janapa Reddi. 2021. RL-Scope: Cross-stack Profiling for Deep Reinforcement Learning Workloads. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2021/hash/676638b91bc90529e09b22e58abb01d6-Abstract.html

[33] Aidan N. Gomez, Mengye Ren, Raquel Urtasun, and Roger B. Grosse. 2017. The Reversible Residual Network: Backpropagation Without Storing Activations. 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.). 2214–2224. https://proceedings.neurips.cc/paper/2017/hash/f9be311e65d81a9ad8150a60844bb94c-Abstract.html

[34] Google Cloud. 2023. Accelerator-optimized machine family. Accessed on April 28 2023.

[35] Alan Gray. 2019. Getting Started with CUDA Graphs. https://developer.nvidia.com/blog/cuda-graphs/

[36] Anmol Gulati, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han, Shibo Wang, Zhengdong Zhang, Yonghui Wu, and Ruoming Pang. 2020. Conformer: Convolution-augmented Transformer for Speech Recognition. In *Interspeech 2020, 21st Annual Conference of the International Speech Communication Association, Virtual Event, Shanghai, China, 25-29 October 2020*, Helen Meng, Bo Xu, and Thomas Fang Zheng (Eds.). ISCA, 5036–5040. https://doi.org/10.21437/Interspeech.2020-3015

[37] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 2023. Graphene: An IR for Optimized Tensor Computations on GPUs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 302–313. https://doi.org/10.1145/3582016.3582018

[38] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[39] Mark Harris and Ian Buck. 2005. Chapter 34. GPU Flow-Control Idioms. https://developer.nvidia.com/gpugems/gpugems2/part-iv-general-purpose-computation-gpus-primer/chapter-34-gpu-flow-control-idioms

[40] Horace He. 2022. PyTorch 2.0: Dynamic Shapes Support. https://youtu.be/rnkJQ-7JmQ

[41] 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

[42] John L. Hennessy and David A. Patterson. 2017. *Computer Architecture, Sixth Edition: A Quantitative Approach* (6th ed.). Morgan Kaufmann Publishers Inc.

[43] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 2261–2269. https://doi.org/10.1109/CVPR.2017.243

[44] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research* 18 (2017), 187:1–187:30. http://jmlr.org/papers/v18/16-456.html

[45] Jerome C. Huck, Dale Morris, Jonathan Ross, Allan D. Knies, Hans M. Mulder, and Rumi Zahir. 2000. Introducing the IA-64 Architecture. *IEEE Micro* 20, 5 (2000), 12–23. https://doi.org/10.1109/40.877947

[46] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. 2018. Gist: Efficient Data Encoding for Deep Neural Network Training. In *Proceeding of the 45st Annual International Symposium on Computer Architecture (ISCA 2018)*. 776–789. https://doi.org/10.1109/ISCA.2018.00070

[47] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph Gonzalez. 2020. Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/320.pdf

[48] JEDEC. 2017. Main Memory: DDR4 and DDR5 SDRAM. https://www.jedec.org/category/technology-focus-area/main-memory-ddr3-ddr4-sdram

[49] JEDEC. 2021. GRAPHICS DOUBLE DATA RATE 6 (GDDR6) SGRAM STANDARD. https://www.jedec.org/standards-documents/docs/jesd250c

[50] JEDEC. 2021. HIGH BANDWIDTH MEMORY (HBM) DRAM. https://www.jedec.org/standards-documents/docs/jesd235a

[51] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 47–62. https://doi.org/10.1145/3341301.3359630

[52] Chenhao Jiang, Anand Jayarajan, Hao Lu, and Gennady Pekhimenko. 2023. Arbitor: A Numerically Accurate Hardware Emulation Tool for DNN Accelerators. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 519–536. https://www.usenix.org/conference/atc23/presentation/jiang-chenhao

[53] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the Future of Parallel Computing. *IEEE Micro* 31, 5 (2011), 7–17. https://doi.org/10.1109/MM.2011.89

[54] Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock. 2021. Dynamic Tensor Rematerialization. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Vfs_2RnOD0H

[55] Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding, Brian Pharris, Jared Casper, and Krste Asanovic. 2004. The Vector-Thread Architecture. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*. IEEE Computer Society, 52–63. https://doi.org/10.1109/ISCA.2004.1310763

[56] Ronny M. Krashinsky. 2011. Temporal SIMT execution optimization through elimination of redundant operations. https://patents.google.com/patent/US9830156B2

[57] Andrey Kuzmin, Mart Van Baalen, Yuwei Ren, Markus Nagel, Jorn Peters, and Tijmen Blankevoort. 2022. FP8 Quantization: The Power of the Exponent. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=H3Gv7XEGzYV

[58] 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. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[59] Ao Li, Bojian Zheng, Gennady Pekhimenko, and Fan Long. 2022. Automatic Horizontal Fusion for GPU Kernels. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2022, Seoul, Korea, Republic of, April 2-6, 2022*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 14–27. https://doi.org/10.1109/CGO53902.2022.9741270

[60] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph.* 37, 4 (2018), 139. https://doi.org/10.1145/3197517.3201383

[61] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context. In *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V (Lecture Notes in Computer Science, Vol. 8693)*, David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars (Eds.). Springer, 740–755. https://doi.org/10.1007/978-3-319-10602-1_48

[62] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 881–897. https://www.usenix.org/conference/osdi20/presentation/ma

[63] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. 1995. A Comparison of Full and Partial Predicated Execution Support for ILP Processors. *SIGARCH Comput. Archit. News* 23, 2 (May 1995), 138–150. https://doi.org/10.1145/225830.225965

[64] Peter Mattson, Christine Cheng, Gregory F. Diamos, Cody Coleman, Paulius Micikevicius, David A. Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim M. Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St. John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. 2020. MLPerf Training Benchmark. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/paper_files/paper/2020/hash/411e39b117e885341f25efb8912945f7-Abstract.html

[65] Jiayuan Meng, David Tarjan, and Kevin Skadron. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, André Seznec, Uri C. Weiser, and Ronny Ronen (Eds.). ACM, 235–246. https://doi.org/10.1145/1815961.1815992

[66] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1gs9JgRZ

[67] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, Naveen Mellempudi, Stuart F. Oberman, Mohammad Shoeybi, Michael Y. Siu, and Hao Wu. 2022. FP8 Formats for Deep Learning. *CoRR* abs/2209.05433 (2022). https://doi.org/10.48550/arXiv.2209.05433 arXiv:2209.05433

[68] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). arXiv:1906.00091 http://arxiv.org/abs/1906.00091

[69] Vinh Nguyen, Michael Carilli, Sukru Burc Eryilmaz, Vartika Singh, Michelle Lin, Natalia Gimelshein, Alban Desmaison, and Edward Yang. 2021. Accelerating PyTorch with CUDA Graphs. https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/

[70] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf

[71] NVIDIA. 2021. NVIDIA Ampere GA102 GPU Architecture. https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[72] NVIDIA. 2023. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/archive/12.0.0/cuda-c-programming-guide/index.html Accessed on April 28 2023.

[73] NVIDIA. 2023. Linux X64 (AMD64/EM64T) Display Driver. https://www.nvidia.com/download/driverResults.aspx/199656/en-us/

[74] NVIDIA. 2023. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core Accessed on July 8 2023.

[75] NVIDIA. 2023. NVIDIA Linux Open GPU Kernel Module Source. https://github.com/NVIDIA/open-gpu-kernel-modules/tree/525.89.02

[76] NVIDIA. 2023. NVIDIA/cutlass: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass Accessed on April 28 2023.

[77] ONNX. 2022. Open Neural Network Exchange. https://github.com/onnx/onnx/tree/v1.12.0

[78] OpenAI. 2023. GPT-4 Technical Report. *CoRR* abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774 arXiv:2303.08774

[79] David A. Padua and Michael Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* 29, 12 (1986), 1184–1201. https://doi.org/10.1145/7902.7904

[80] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, 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*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[81] David Patterson. 2022. A Decade of Machine Learning Accelerators: Lessons Learned and Carbon Footprint. https://chips-compilers-mlsys-22.github.io/assets/slides/10%20Lessons%204%20TPU%20gens%20+%20CO2e%2045%20minutes.pdf

[82] PCI-SIG. 2010. PCI Express Base Specification Revision 3.0. https://pcisig.com/specifications?field_technology_value%5B%5D=express&field_revision_value%5B%5D=3&field_document_type_value%5B%5D=specification&specLib=PCI+Express+Base+Specification+Revision+3.0

[83] Xuan Peng, Xuanhua Shi, Hulin Dai, Hai Jin, Weiliang Ma, Qian Xiong, Fan Yang, and Xuehai Qian. 2020. Capuchin: Tensor-based GPU Memory Management for Deep Learning. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 891–905. https://doi.org/10.1145/3373376.3378505

[84] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2022. Efficiently Scaling Transformer Inference. arXiv:2211.05102 [cs.LG]

[85] Python. 2023. Thread State and the Global Interpreter Lock. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock Accessed on April 28 2023.

[86] PyTorch. 2023. TorchDynamo Overview. https://pytorch.org/docs/stable/dynamo/index.html

[87] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019). https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[88] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, Christine Cuicchi, Irene Qualters, and William T. Kramer (Eds.). IEEE/ACM, 20. https://doi.org/10.1109/SC41405.2020.00024

[89] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-infinity: breaking the GPU memory wall for extreme scale deep learning. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 59. https://doi.org/10.1145/3458817.3476205

[90] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *KDD '20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, CA, USA, August 23-27, 2020*, Rajesh Gupta, Yan Liu, Jiliang Tang, and B. Aditya Prakash (Eds.). ACM, 3505–3506. https://doi.org/10.1145/3394486.3406703

[91] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2020. MLPerf Inference Benchmark. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[92] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. 2022. torch.fx: Practical Program Capture and Transformation for Deep Learning in Python. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2022/hash/ca46c1b9512a7a8315fa3c5a946e8265-Abstract.html

[93] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. ZeRO-Offload: Democratizing Billion-Scale Model Training. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 551–564. https://www.usenix.org/conference/atc21/presentation/ren-jie

[94] Minsoo Rhu and Mattan Erez. 2013. The dual-path execution model for efficient GPU control flow. In *19th IEEE International Symposium on High Performance Computer Architecture, HPCA 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 591–602. https://doi.org/10.1109/HPCA.2013.6522352

[95] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. 2016. vDNN: Virtualized Deep Neural Networks for Scalable, Memory-efficient Neural Network Design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, Article 18, 13 pages. http://dl.acm.org/citation.cfm?id=3195638.3195660

[96] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W. Keckler. 2018. Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks. In *IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, February 24-28, 2018*. IEEE Computer Society, 78–91. https://doi.org/10.1109/HPCA.2018.00017

[97] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature* 323, 6088 (1986), 533–536. https://doi.org/10.1038/323533a0

[98] Amit Sabne. 2020. XLA: Compiling Machine Learning for Peak Performance.

[99] Christian Sarofeen, Piotr Bialecki, Jie Jiang, Kevin Stephano, Masaki Kozuki, Neal Vaidya, and Stas Bekman. 2022. Introducing nvFuser, a deep learning compiler for PyTorch. https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/

[100] Roy Schwartz, Jesse Dodge, Noah A. Smith, and Oren Etzioni. 2020. Green AI. *Commun. ACM* 63, 12 (November 2020), 54–63. https://doi.org/10.1145/3381831

[101] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. 2022. Tensor Program Optimization with Probabilistic Programs. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=nyCr6-0hinG

[102] 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 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/4e732ced3463d06de0ca9a15b6153677-Abstract.html

[103] Sangeetha Siddegowda, Marios Fournarakis, Markus Nagel, Tijmen Blankevoort, Chirag Patel, and Abhijit Khobare. 2022. Neural Network Quantization with AI Model Efficiency Toolkit (AIMET). *CoRR* abs/2201.08442 (2022). arXiv:2201.08442 https://arxiv.org/abs/2201.08442

[104] Daniel Snider, Fanny Chevalier, and Gennady Pekhimenko. 2023. Hotline Profiler: Automatic Annotation and A Multi-Scale Timeline for Visualizing Time-Use in DNN Training. (2023). https://proceedings.mlsys.org/paper_files/paper/2023/hash/763d339d9e1b0caa2ea9ae01d4b9e0d0-Abstract-mlsys2023.html

[105] Sally Stevenson, Stephen Jones, and Fred Oh. 2022. Enabling Dynamic Control Flow in CUDA Graphs with Device Graph Launch. https://developer.nvidia.com/blog/enabling-dynamic-control-flow-in-cuda-graphs-with-device-graph-launch/

[106] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. 2020. Learning to summarize with human feedback. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 3008–3021. https://proceedings.neurips.cc/paper_files/paper/2020/file/1f89885d556929e98d3ef9b86448f951-Paper.pdf

[107] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 6105–6114. http://proceedings.mlr.press/v97/tan19a.html

[108] Xiaodan Tan, Paul Gilbert Meyer, Gennady Pekhimenko, and Randy Renfu Huang. 2023. Mixing sparsity compression. https://patents.google.com/patent/US20230100930A1

[109] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. 2022. FreeTensor: a free-form DSL with holistic optimizations for irregular tensor programs. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 872–887. https://doi.org/10.1145/3519939.3523448

[110] Philippe Tillet, Hsiang-Tsung Kung, and David D. Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Tim Mattson, Abdullah Muzahid, and Armando Solar-Lezama (Eds.). ACM, 10–19. https://doi.org/10.1145/3315508.3329973

[111] Han Vanholder. 2016. Efficient Inference with TensorRT. In *GPU Technology Conference*. https://on-demand.gputechconf.com/gtc-eu/2017/presentation/23425-han-vanholder-efficient-inference-with-tensorrt.pdf

[112] Patrick von Platen. 2020. How to generate text: using different decoding methods for language generation with Transformers. https://huggingface.co/blog/how-to-generate#beam-search

[113] Patrick von Platen. 2021. Fine-Tune Wav2Vec2 for English ASR with Transformers. https://huggingface.co/blog/fine-tune-wav2vec2-english

[114] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. https://github.com/kingoflolz/mesh-transformer-jax

[115] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 37–54. https://www.usenix.org/conference/osdi21/presentation/wang

[116] Shibo Wang and Pankaj Kanwar. 2019. BFloat16: The secret to high performance on Cloud TPUs. https://cloud.google.com/blog/products/ai-machine-

learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[117] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 599–623. https://proceedings.mlsys.org/paper/2021/file/a97da629b098b75c294dffdc3e463904-Paper.pdf

[118] Yao Wang. 2019. [RFC] Dynamic Shape Support - Graph Dispatching. https://github.com/apache/tvm/issues/4118

[119] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying Tensorized Instruction Compilation. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 77–89. https://doi.org/10.1109/CGO51591.2021.9370330

[120] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[121] Hao Wu, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. 2020. Integer Quantization for Deep Learning Inference: Principles and Empirical Evaluation. *CoRR* abs/2004.09602 (2020). arXiv:2004.09602 https://arxiv.org/abs/2004.09602

[122] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. 2022. Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance. In *Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, Diana Marculescu, Yuejie Chi, and Carole-Jean Wu (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2022/hash/38b3eff8baf56627478ec76a704e9b52-Abstract.html

[123] Yichen Yang, Phitchaya Mangpo Phothilimthana, Yisu Remy Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021*, Alex Smola, Alex Dimakis, and Ion Stoica (Eds.). mlsys.org. https://proceedings.mlsys.org/paper/2021/hash/65ded5353c5ee48d0b7d48c591b8f430-Abstract.html

[124] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 660–678. https://doi.org/10.1145/3582016.3582047

[125] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. Orca: A Distributed Serving System for Transformer-Based Generative Models. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 521–538. https://www.usenix.org/conference/osdi22/presentation/yu

[126] Geoffrey X. Yu, Tovi Grossman, and Gennady Pekhimenko. 2020. Skyline: Interactive In-Editor Computational Performance Profiling for Deep Neural Network Training. In *UIST '20: The 33rd Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 20-23, 2020*, Shamsi T. Iqbal, Karon E. MacLean, Fanny Chevalier, and Stefanie Mueller (Eds.). ACM, 126–139. https://doi.org/10.1145/3379337.3415890

[127] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparnda Das, and Scott Mahlke. 2017. Systems and devices for compressing neural network parameters. https://patents.google.com/patent/US11321604B2

[128] Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic Partition-based Operator Fusion through Layer by Layer Optimization. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 1–19. https://proceedings.mlsys.org/paper_files/paper/2022/file/069059b7ef840f0c74a814ec9237b6ec-Paper.pdf

[129] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, Peng Di, Kun Zhang, and Xuefeng Jin. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1233–1248. https://doi.org/10.1145/3453483.3454106

[130] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. 2022. DietCode: Automatic Optimization for Dynamic Tensor Programs. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 848–863. https://proceedings.mlsys.org/paper_files/paper/2022/file/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper.pdf

[131] Bojian Zheng, Nandita Vijaykumar, and Gennady Pekhimenko. 2020. Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 1089–1102. https://doi.org/10.1109/ISCA45697.2020.00092

[132] 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

[133] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 559–578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin

[134] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 874–887. https://doi.org/10.1145/3470496.3527440

[135] 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 *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, James R. Larus, Luis Ceze, and Karin Strauss (Eds.). ACM, 859–873. https://doi.org/10.1145/3373376.3378508

[136] Zhen Zheng, Xuanda Yang, Pengzhan Zhao, Guoping Long, Kai Zhu, Feiwen Zhu, Wenyi Zhao, Xiaoyong Liu, Jun Yang, Jidong Zhai, Shuaiwen Leon Song, and Wei Lin. 2022. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. (2022), 359–373. https://doi.org/10.1145/3503222.3507723

[137] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and Analyzing Deep Neural Network Training. In *2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018*. IEEE Computer Society, 88–100. https://doi.org/10.1109/IISWC.2018.8573476

[138] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. 2022. ROLLER: Fast and Efficient Tensor Compilation for Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 233–248. https://www.usenix.org/conference/osdi22/presentation/zhu

[139] Kai Zhu, Wenyi Zhao, Zhen Zheng, Tianyou Guo, Pengzhan Zhao, Junjie Bai, Jun Yang, Xiaoyong Liu, Lansong Diao, and Wei Lin. 2021. DISC: A Dynamic Shape Compiler for Machine Learning Workloads. In *EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021*, Eiko Yoneki and Paul Patras (Eds.). ACM, 89–95. https://doi.org/10.1145/3437984.3458838

Received 28 April 2023; revised 7 July 2023; accepted 24 July 2023