# PyGraph: Robust Compiler Support for CUDA Graphs in PyTorch

Abhishek Ghosh Indian Institute of Science Ajay Nayak Indian Institute of Science Ashish Panwar Microsoft Research India

Arkaprava Basu
Indian Institute of Science

## **Abstract**

CUDA Graphs — a recent hardware feature introduced for NVIDIA GPUs — aim to reduce CPU launch overhead by capturing and launching a series of GPU tasks (kernels) as a DAG. However, deploying CUDA Graphs faces several challenges today due to the static structure of a graph. It also incurs performance overhead due to data copy. In fact, we show a counter-intuitive result — deploying CUDA Graphs hurts performance in many cases.

We introduce PyGraph, a novel approach to automatically harness the power of CUDA Graphs within PyTorch2. Driven by three key observations, PyGraph embodies three novel optimizations: it enables wider deployment of CUDA Graphs, reduces GPU kernel parameter copy overheads, and selectively deploys CUDA Graphs based on a cost-benefit analysis. PyGraph seamlessly integrates with PyTorch2's compilation toolchain, enabling efficient use of CUDA Graphs without manual modifications to the code. We evaluate PyGraph across various machine learning benchmarks, demonstrating substantial performance improvements over PyTorch2.

# 1 Introduction

A significant portion of today's computing needs, particularly that of AI-driven software, is catered by Graphic Processing Units (GPUs) [2,7,8,26,28,51,53,54,68]. GPUs' massive data-parallel computing capability plays a key role in efficiently performing matrix and vector operations that are often at the heart of many ML algorithms [2,52].

While the demand for GPU computing continues to skyrocket, catering to this demand through hardware enhancement alone poses a significant challenge in the post-Moore era of slowing transistor technology [13]. Consequently, hardware vendors have been increasingly relying on hardware-software co-design to improve computing efficiency and extract more performance out of the silicon. For example, NVIDIA continues to introduce new performance features in the GPUs that need significant software enablement. Recent examples

include scoped synchronization [24, 25, 29] that allow programmers to selectively enable faster synchronization, and CUDA Graphs for improving GPU utilization.

In this work, we focus on CUDA Graphs [16,63]. An ML application can invoke hundreds to thousands of GPU kernels (routines) during its execution. Each kernel launched from the CPU adds 5-10 microseconds of latency [12], lowering GPU utilization. CUDA Graphs can improve GPU utilization by allowing software to *capture* a directed acyclic graph (DAG) of kernel launches (i.e., a CUDA Graph). A CUDA Graph can then be replayed repeatedly with the GPU hardware executing its consistent kernels following the DAG without involving the CPU in launching individual kernels. While useful, CUDA Graphs come with several restrictions. For example, the shape of a graph must remain unaltered, making it inconvenient for applications with conditional execution of kernels. It also hardcodes kernel parameters during the graph captures. If not carefully used by the software, CUDA Graphs can lead to erroneous execution.

The practical usefulness of such software-managed hardware features, including CUDA Graphs, is as good as their software enablement. This is made further challenging by the fact that most AI/ML programs are written in high-level programming frameworks such as PyTorch [40], TensorFlow [1] and are semantically distant from their execution on the hardware. It is thus the onus of these programming frameworks to bridge the gap between the high-level program semantics and the low-level idiosyncrasies of advanced hardware features.

Recently introduced PyTorch2 framework [4] provides a generational leap over the original PyTorch by automatically enabling significant performance optimizations while retaining ease of programmability. It introduces *compilation* (torch.compile) of models written in PyTorch that extracts computation graphs, also called FxGraph [47], while compiling the programs down to executables, including invocations of GPU kernels. The primary purpose of this FxGraphs is to help apply key optimizations, such as operator fusion, seamlessly. PyTorch2 further created the ability to potentially convert these graphs to CUDA Graphs and thereby harness

CUDA Graphs without programmer effort — the key to wider deployment of such software-managed hardware features.

We, however, discovered *three* key shortcomings of Py-Torch2's enablement of CUDA Graphs. We then introduce PyGraph, built atop PyTorch2 to address these shortcomings with three optimizations to effectively harness CUDA Graphs in modern NVIDIA GPUs. PyGraph does so without needing modifications to applications or manual alterations to PyTorch2's kernels that could otherwise limit its usefulness. ① Optimization Opportunities for Expanding CUDA

(1) Optimization Opportunities for Expanding CUDA **Graph Capture:** We found that key limitations of CUDA Graphs, coupled with subtle high-level programming idiosyncrasies in popular applications (e.g., speech transformer [11]), can hamstrung deployment of CUDA Graphs. During the capture of a CUDA Graph, it records parameters passed to GPU kernels by value. If a parameter is a scalar variable whose value changes across invocations, the kernel would receive stale values during graph replays. The value recorded during the graph's capture would be passed repeatedly during replays. PyTorch2 avoids deploying CUDA Graphs in such cases to ensure correctness but leaves performance on the table. If the PyTorch program was slightly modified such that the resultant kernels would instead receive *pointers* to scalar variables, such kernels could have been correctly captured in a CUDA Graph. The values of the parameter for a given kernel invocation could then be copied onto the address pointed to by the pointer (recorded during graph capture) upon the kernel's execution during CUDA Graph's replay. However, the challenge is detecting and automatically making relevant modifications to programs without programmers' intervention.

We found several variants of such subtleties in popular Py-Torch programs that impedes the use of CUDA Graphs. For example, it is common for applications to allocate a tensor on the CPU DRAM (say, address x) and then copy its content over to GPU memory during computation, e.g., in the forward pass. A CUDA Graph captured for such computation would have hard-coded the address of the CPU tensor (x) whose content could be copied over to GPU memory through asynchronous memcopy during replay. However, the host (CPU) program may de-allocate the CPU tensor after the graph capture. This could lead to intermittent crashes on deploying a CUDA Graph. PyTorch2 avoids CUDA Graphs in such cases. Instead, if the memcopy was hoisted before the computations, a long chain of kernels following the memcopy could be seamlessly captured in a CUDA Graph.

② Opportunities for Reducing CUDA Graph Replay Overheads: We noticed that significant overheads are added to the critical path during CUDA Graph replays due to the copying of GPU kernel's parameters to their placeholder (virtual addresses) recorded during the capture of the encompassing graphs [63]. PyGraph reduces the number of bytes copied by converting data (parameter) copy to pointer copy through the introduction of an additional layer of indirection. While parameters can be hundreds to thousands of bytes long, pointers

are only eight bytes. PyGraph converts the kernel parameters to *pointer-to-pointers* and then modifies the kernels to dereference the parameters. The key challenge is how to avoid manual modifications to kernels and maintain compatibility with the PyTorch2 framework. Towards this, we observe that the majority of the GPU kernels deployed by PyTorch2 are generated on the fly by OpenAI's Triton compilation framework [39]. PyGraph extends Tritron's compilation framework to automatically generate modified kernels. This optimization, however, is inapplicable to closed-source kernels, such as those from cuBLAS [31], cuDNN [35].

(3) CUDA Graphs May Hurt Performance: While CUDA Graphs is a performance feature, we find that the overheads of CUDA Graphs can sometimes outweigh their benefits. Upon analyzing over 400 CUDA Graphs from over 180 applications, we find that more than a third of the CUDA Graphs hurt performance. PyGraph augments the PyTorch2's compilation framework with automated profiling that decides if a CUDA Graph is beneficial or detrimental to the end-to-end application performance. The profiling happens along the compilation and graph capture phase (slow path) and does not add overhead during graph replays (fast path).

In summary, PyGraph extends PyTorch2 compilation framework to enable wider but judicious deployment of CUDA Graph while reducing its overheads where possible. Importantly, PyTorch2 is transparent to PyTorch applications and is fully compatible with the rest of PyTorch2 framework. It speeds up 20 diverse PyTorch applications by 12%, on average (geometric mean).

The key contributions of our work are as follows:

- We demonstrate subtle programming idiosyncrasies in popular PyTorch programs, coupled with limitations of CUDA Graphs can constrain a wider deployment of CUDA Graphs, leaving significant performance on the table.
- We demonstrate that overheads due to copying GPU kernel parameters can be limited by introducing an additional layer of indirection in passing the parameters to kernels.
- We demonstrate the need to *selectively* deploy CUDA Graphs to achieve better end-to-end application performance.
- We create PyGraph by extending PyTorch2's compilation framework that achieves the above-mentioned optimizations without needing any application modifications and manual alterations to PyTorch's kernels.

# 2 Background

# 2.1 CUDA Graphs

As GPUs become faster with every new generation, the time to launch a kernel from CPU to GPU can contribute significantly to overall application runtime. Moreover, in many applications (*e.g.*, DNN inference), kernels are launched iteratively on the GPU [2, 52], aggregating the launch overheads. The primary motivation behind CUDA Graphs is to reduce this

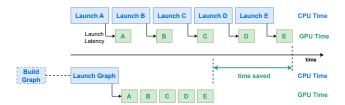


Figure 1: CUDA Graphs reducing CPU launch overheads.

CPU overhead and increase GPU utilization [30].

Traditionally, GPU tasks, such as GPU kernel, memcopy, memalloc, are launched individually from the CPU, incurring launch overhead for each GPU task. As shown in the top portion of Figure 1, this overhead creates noticeable gaps between task executions. CUDA Graphs allow developers to *capture* a series of GPU tasks into a computation graph. The graph, called CUDA Graph, can then be launched as a single entity [16]. The GPU hardware launches the tasks in the captured CUDA Graph without CPU intervention, as depicted in the bottom portion of Figure 1. This reduces the overhead of launching individual GPU tasks, improving GPU utilization. Note that there is an initial cost of building the graph for the first time. However, in iterative applications, a CUDA Graph can be captured once and can subsequently be *replayed* many times, aggregating its benefits.

#### 2.1.1 Creating CUDA Graphs

CUDA provides two ways to capture CUDA Graphs. (1) manual capture, and 2 stream capture. The developer must create a graph programmatically in the manual capture method. In comparison, stream capture requires minimal code changes by the developer. Thus, we focus on this method. Here, the developer wraps a region of code with GPU tasks within cudaStreamBeginCapture and cudaStreamEndCapture APIs. All GPU tasks launched within this region are captured (recorded) by the GPU runtime. In this phase, tasks are not launched on the GPU. The runtime records the tasks and the arguments, creating nodes and links between the tasks in the computation graph. Once the recording phase completes, the developer instantiates the graph structure created by the GPU runtime and launches it using cudaGraphInstantiate and cudaGraphLaunch APIs, respectively. The launched computation graph replays the captured tasks on the GPU.

#### 2.1.2 Challenges in adapting CUDA Graphs

The structure of the computation graph *i.e.*, GPU tasks and their ordering must remain the same throughout the graph's lifetime. Further, the captured values for each node of the computation graph (*e.g.*, kernel arguments) are recorded by value. If a program requires a change in the graph's structure (*e.g.*, control flow dependence) or the captured values (*e.g.*,

in a subsequent iteration), but the graph with unmodified information is replayed, the execution will be erroneous. These can cause silent data corruption and errors, besides application crashes. Thus, the developer must carefully analyze their applications while using CUDA Graphs.

# 2.2 PyTorch2

PyTorch [40] traditionally followed a define-by-run approach for ease of programmability and flexibility. However, this approach limits the visibility of a framework to a single operator at a time, limiting the scope of compiler optimizations. Py-Torch2 [4] overcomes these limitations by creating computation graphs, enabling compiler optimizations without compromising on developer convenience. It exposes this functionality via the torch.compile annotation. PyTorch2 first lowers the annotated Python function to an intermediate representation called TorchIR. TorchIR provides a global view of all operations in the Python function, enabling several compiler optimizations. Next, PyTorch2 generates highly optimized GPU kernels for the corresponding operations, and returns a compiled module. PyTorch2 also integrates with performance features such as CUDA Graphs, further improving performance of Python functions.

#### 2.2.1 PyTorch2 compile and Triton

The primary components involved in compiling a Python function are the frontend, Torch Dynamo [43] and the backend, Torch Inductor [20]. Torch Dynamo is a Python-level tracing tool that captures the operations in the Python function. Torch Inductor is responsible of generating efficient GPU operations that correspond to the Python function. When the function to be compiled is called for the first time, Torch Dynamo symbolically executes the Python function and creates a container called FxGraph for it. The FxGraph object holds a high-level intermediate representation (called TorchIR) for operations in the function. TorchIR tracks the information relevant to all the operations and the corresponding metadata in the function.

Torch Inductor then takes the FxGraph object and lowers it down to primitive operations, *i.e.*, it converts TorchIR to InductorIR. The visibility of all the operations within the FxGraph enables compiler optimizations such as fusion of two operations. The operations (even the fused ones) in InductorIR are compatible with operations supported by various code-generation backends integrated into Torch Inductor.

Torch Inductor is integrated with the Triton framework [39] for generating optimized GPU kernels on-the-fly. Triton is an open-source deep learning compiler and Python-based programming language developed by OpenAI. It provides abstractions for low-level GPU programming concepts while maintaining the flexibility of Python language. Along with Triton, Torch Inductor is also integrated with other highly optimized GPU libraries such as cuBLAS [31], CUTLASS [33].

Torch Inductor converts each primitive operation in InductorIR to GPU kernels by either ① generating optimized GPU kernels on-the-fly using the Triton framework, or ② calling library functions from cuBLAS, CUTLASS etc. These individual GPU kernels are then stitched together as a sequence of kernel calls to generate a compiled, optimized, callable module for the FxGraph object. Note that the kernels in this module can be a mix of Triton and non-triton (cuBLAS, CUTLASS) GPU kernels.

# 2.2.2 CUDA Graphs in PyTorch2

PyTorch2 enables CUDA Graphs with the reduce-overhead option to the torch.compile annotation. Torch Inductor evaluates various rules to check if the module can be *CUDA Graphified* or not. For example, if one of the operations in the FxGraph of the optimized module uses a CPU-resident scalar — the module is not graphified. Even if a single such offending operation exists, the *entire module is not graphified*. We provide more details on this in Section 3.

If the FxGraph can be graphified, Torch Inductor uses the stream capture method to record and replay the corresponding module. Torch Inductor encapsulates the callable module with PyTorch2 exposed APIs for stream capture. It then records the graph. This recorded graph is replayed to execute the kernels in the Python function. In every subsequent iteration, the recorded graph is replayed, getting benefits of CUDA Graph.

Recall that the kernel arguments are captured by value in the graph. Torch Inductor therefore modifies the module to avoid this scenario. It replaces all such arguments with *place-holder* tensor arguments. Torch Inductor then injects code to copy content from tensors to these place-holder arguments. During graph capture, these placeholders are recorded. We detail PyTorch2's integration of CUDA Graphs in Section 5.

#### 3 Motivation

CUDA Graph is a performance feature i.e., its goal is to improve the performance of GPU-accelerated applications. However, we find that overheads of CUDA Graphs can sometimes outweigh its benefits. This can be particularly true for graphs with smaller numbers of kernels.

Cost-benefit analysis of CUDA Graphs: To understand the cost-benefit of CUDA Graphs, we analyzed 416 CUDA Graphs generated by PyTorch2 for 183 applications. We perform two measurements to analyze the cost and benefit of deploying a CUDA Graph. We measure the time to launch and execute a given CUDA Graph (say, *x* sec). We then measure the time to individually launch and execute the kernels present in the given CUDA Graph. We add the times for these individual kernels when launched without CUDA Graph (say, *y* sec). The difference between the latter and the former (*y* - *x*) captures the benefit (cost) of deploying the given CUDA Graph. As illustrated in Figure 2, we found that 191 out of



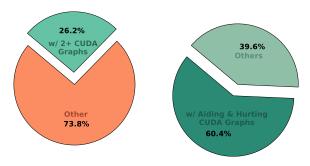
**Figure 2:** Cost-benefit statistics of CUDA Graphs over 183 applications.

416 CUDA Graphs improved performance by more than 2% while 82 CUDA Graphs delivered no speedup over individually launching kernels. Importantly, 143 CUDA Graphs hurts performance as the overheads of deploying them outweigh their benefits. In short, while CUDA Graphs are designed to be a performance feature, it can slow down applications if deployed universally.

We then empirically analyzed sources of overheads in CUDA Graphs. The primary source of overhead is the latency to copy a kernel's parameters (data) onto its static placeholder (address) that are baked into the kernels captured in a CUDA Graph (Section 2). This latency is incurred for many parameters of every kernel invocation during each replay of the encompassing CUDA Graph. This latency is in the critical path of a graph's replay. There are also overheads of bookkeeping and formatting the output of a CUDA Graph after each replay. However, nearly 60% of the overheads of a CUDA Graph can be attributed to the data (parameter) copy. This generally corroborates prior work [63]. To realize the full potential of CUDA Graphs, it is thus imperative to minimize the overheads of copying kernel parameters to its placeholder during replay. Importantly, to be practically useful, one must strive to do so while avoiding manual changes to GPU kernels, higher-level programming frameworks, and drivers.

# 3.1 Need for selective deployment of CUDA Graphs

Upon analyzing over 183 PyTorch applications from the TorchBench [18], HuggingFace [19] and TIMM [59] workload suites, we found that 48 applications (26%) generated two or more (up to 6) CUDA Graphs when compiled by PyTorch2. Importantly, 29 of these 48 applications contain CUDA Graphs that aid performance and also those that hurt performance. Figure 3 illustrates this distribution. Thus, the end-to-end performance of applications would benefit most if one could *selectively* deploy CUDA Graphs based on a cost-benefit analysis. Unfortunately, in PyTorch2, one can enable or disable CUDA Graphs for all graphs in a compiled module,



- Graphs. 48 out of 183 applications (26%) Graphs. 29 out of these 48 applications generate two or more CUDA Graphs when (60%) have CUDA Graphs that both aid compiled with PyTorch2.
- (a) Applications with multiple CUDA (b) Performance trade-off in CUDA and hurt performance.

Figure 3: Distribution of PyTorch applications based on CUDA Graph usage.

i.e., there is no way to pick and choose CUDA Graphs to deploy.

#### **Need for CUDA Graph-aware compilation** 3.2

As discussed in Section 2, the kernel parameters, e.g., variables, addresses, are recorded by value during graph capture. During the replay of the captured graph, the same values are passed as parameters. To allow a kernel to be correctly replayed as part of a CUDA Graph, every parameter, barring constants, must be an address (pointer or pointer-to-pointer) whose contents can then be altered across iterations of graph replays by PyTorch2.

We observed that subtle oversight while allocating variables in PyTorch programs could debar PyTorch2 from deploying CUDA Graphs. Let us take an example of a simplified Py-Torch code snippet from the speech transformer model [42] (Figure 4). In the constructor of MultiheadAttention (line 19), the variable temperature is assigned the value returned by a numpy method. Note temperature hosts a scalar value returned by the numpy routine and is allocated on the CPU DRAM. This variable is passed to the constructor of ScaledDotProductAttention (line 21).

Ultimately, the forward method of MultiheadAttention is invoked, which, in turn, invokes forward method of ScaledDotProductAttention. In line 13, temperature is used in calculating attn. These high-level functions are compiled down to optimized GPU kernels. For example, the computation of attn is performed by a GPU kernel (elementwise\_div\_kernel). The parameters of this kernel are attn and temperature. While the former is the address to a GPU tensor, the latter is a scalar variable. If this kernel is captured as part of a CUDA Graph, the scalar value of the parameter temperature will be hardcoded as a kernel parameter (copy by value). Later, if the captured graph is replayed, the kernel would execute with a stale value of the

```
import torch
     import numpy as np
     class ScaledDotProductAttention(nn.Module):
         def __init__(self, temperature, attn_dropout=0.1):
              self.temperature = temperature
         def forward(self, q, k, v, mask=None):
10
              attn = torch.bmm(q, k.transpose(1, 2))
11
              # Divide operation below with a numpy scaler on CPU
12
              attn = attn / self.temperature # offending line
13
14
15
     class MultiHeadAttention(nn.Module):
16
         def __init__(self, n_head, d_model, d_k, d_v, dropout=0.1):
17
18
19
             temp = np.power(d k, 0.5) # src of offending operand
             self.attn = ScaledDotProductAttention(
20
                 temperature=temp, attn_dropout=dropout)
21
22
23
24
         def forward(self, q, k, v, mask=None):
25
26
             output, attn = self.attn(q, k, v, mask=mask)
27
28
29
    module = Initialize the speech_tranformer model
30
    module = torch.compile(module, mode="reduce-overhead")
    result = module(inputs)
```

Figure 4: Simplified code snippet from speech transformer model with CPU-resident scalar (marked red).

temperature. Thus, PyTorch2 avoids creating CUDA Graph in the presence of such kernels.

In summary, a single scalar value as a kernel parameter can greatly limit the deployment of CUDA Graphs. However, the key challenge is to find such subtle reasons in PyTorch programs that limit the deployment of CUDA Graphs without manual inspection. Then, one must also fix such programs without the programmer's intervention to enable effective deployment of CUDA Graph.

Our analysis found other subtleties in popular PyTorch programs that limit deployment of CUDA Graphs. For example, it is not uncommon for applications to allocate a tensor on the CPU DRAM (say, address x) and then copy its content over to GPU memory at the start of the computation, e.g., during the forward pass. A CUDA Graph captured for such computation would have hard-coded the address of the CPU tensor (x) whose content can be copied to GPU using asynchronous memcopy during replay. However, the host (CPU) program is free to de-allocate the CPU tensor at address x at any point after the graph capture. During replay of the CUDA Graph, the address x on the DRAM may be invalid. Thus, PyTorch2 avoids creating CUDA Graphs in such cases, leaving performance on the table. In short, subtleties around the allocation and placement of tensors and their types can hinder one from fully harnessing the capabilities of CUDA Graphs.

Summary: 1) One must selectively deploy CUDA Graph in an application to achieve best end-to-end performance. (2) To harness the full benefits of CUDA Graphs, it is imperative to limit the overheads of copying kernel parameters during replay. ③ Subtle tensor allocation choices and tensor types can significantly impact the number of GPU kernels that execute via CUDA Graphs.

# 4 PyGraph: Goals and design principles

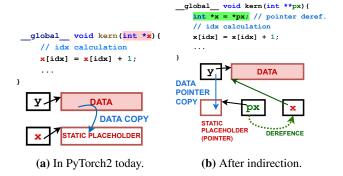
Driven by the observations in the previous section, we lay the goals of PyGraph, followed by a high-level design philosophy to achieve those goals.

Goals: ① PyGraph should find and mitigate factors that limit GPU kernels from being captured in a CUDA Graph. ② It should limit the overheads of copying kernel parameters (data) to their static placeholders during graph replay. ③ It should judiciously deploy CUDA Graphs such that the end-to-end application performance is maximized. ④ Most importantly, PyGraph must achieve the above three goals without any change to the PyTorch applications and without manual modifications to PyTorch's kernels that could limit PyGraph's applicability and forward compatibility.

CUDA Graph-aware Data Placement: Towards the first goal, PyGraph enhances PyTorch2's compilation toolchain to first track when the PyTorch2 fails to deploy CUDA Graph for a given FxGraph. It then ascertains if the reason for failure is due to CPU execution and/or access to CPU-resident variable (scalar) by GPU kernels. If so, it leverages debug information available in the PyTorch2's toolchain to backtrace to the offending allocation and/or function. PyGraph takes corrective action by allocating the offending tensor to GPU memory instead of CPU's DRAM (e.g., for the speech transformer example in the previous section). It may also hoist a CPU-to-GPU memory operation above a chain of kernels that could otherwise be encapsulated in a CUDA Graph. Most importantly, PyGraph performs all these transformations in the PyTorch2's IR (TorchIR) without needing modifications to the application source or manual changes to the PyTorch2's kernels.

Parameter Indirection: Next, we focus on limiting the kernel parameter copying during graph replay. We notice that complied PyTorch2 deploys two types of kernels. First are the ones that are generated on the fly through the Triton compilation. The second class of kernels is the closed-source ones, primarily from NVIDIA, e.g., cuBLAS. We focus on the former since closed-source kernels cannot be updated. Across 183 PyTorch applications employing more than 7000 kernels, more than 4000 kernels ( $\sim 58\%$ ) are generated by Triton during PyTorch2 compilation.

Our key idea to limit overheads of copying of parameter values (data) is to *replace data* (*parameter*) copy with pointer copies. The pointers are eight bytes, while the data can be hundreds to many thousands bytes. We achieve this by introducing an additional layer of indirection for the kernel parameters during the code generation of Triton kernels. Specifically, the parameters to the kernel are passed as *pointers-to-pointers*,



**Figure 5:** Converting data (parameter value) copy to pointer copy through indirection in Parameter Indirection.

which are then de-referenced in the generated kernel. The rest of the generated kernel code remains unaltered.

Figure 5 shows this pictorially. Figure 5(a) shows the data copy involved while replaying a CUDA Graph in the baseline. It also shows the kernel parameter (here, the address x) captured during the graph capture of CUDA Graph in red. During the replay of the captured graph, the data for the given invocation of the kernel, residing at address, say y must be copied to the memory location pointed to by x.

Figure 5(b) shows how PyGraph converts the kernel parameter to pointer-to-pointer. These pointer-to-pointers are then recorded during the graph capture. During the replay of such captured CUDA Graph, the pointers to the data corresponding to the given invocation of the kernel as part of the replay are copied onto the address (pointer-to-pointer) captured during the graph capture (here, px). The kernel is then modified to dereference these pointer-to-pointers before any computation (highlighted in green). The rest of the kernel remains unaltered.

To adhere to the goal ④, we extend Triton's code generation to generate the modified kernels automatically. Thus, this optimization also remains transparent to PyTorch programs and the rest of the PyTorch2 compilation framework.

Selective CUDA Graphs: As discussed earlier, CUDA Graphs can also hurt performance. To avoid blindly deploying CUDA Graphs, PyGraph employs automated profiling as part of the compilation and graph capture phase (slow path). Specifically, PyTorch2 measures the aggregate time to execute individual kernels encapsulated in a CUDA Graph. It also profiles the execution time of the corresponding CUDA Graph with and without Parameter Indirection. The profiler then automatically chooses the best-performing configuration for the given set of kernels in an application. Notice that this decision is made independently for each of the CUDA Graphs. Thus, the profiler decides to deploy or not deploy a CUDA Graph based on the profiled cost-benefit analysis. Notice that this profiling happens during compilation and CUDA Graph capture phase (slow path) and not during the graph replay

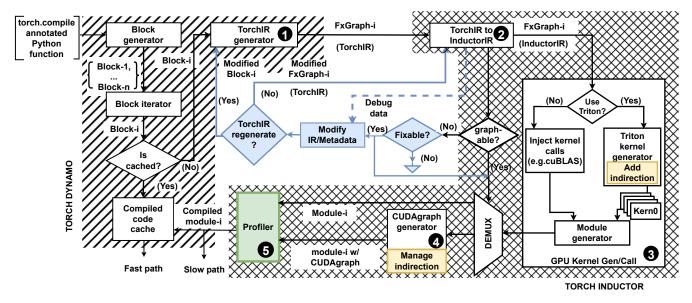


Figure 6: Implementation of PyGraph within PyTorch2's compilation framework.

(fast path).

In summary, the three above-mentioned optimizations, working together, ensure a larger number of kernels can be captured as part of CUDA Graphs, reduce data copy overheads, and selectively deploy CUDA Graphs where helpful. Importantly, all three optimizations are built into PyTorch2's compilation framework that does not involve any manual modifications to the kernels.

# 5 Implementing PyGraph

Figure 6 pictorially depicts a high-level view of PyGraph's implementation. The figure shows how it extends PyTorch2's compilation framework to implement the three optimizations. Major modifications are colored, while those *without* color show unmodified PyTorch2 [4]. The two gray patterns in the background delimit the key parts of PyTorch2 – Torch Dynamo and Torch Inductor. The modifications for CUDA Graph-aware Data Placement are in blue, while those in yellow and green show that for Parameter Indirection and Selective CUDA Graphs, respectively.

Before delving into the details, we provide a high-level overview. PyTorch2 breaks a Python function into multiple chunks of code called 'block.' For each block, it first checks if a compiled function is available in its software cache. If yes, PyTorch2 executes the compiled function from the cache (*fast path*). If not, PyTorch2 creates a compiled module for the block and adds it to the cache (*slow path*). PyGraph's modifications are limited to the slow path only. Its optimizations benefit the fast path, though.

PyTorch2's compilation frontend first lowers the blocks into an intermediate representation (IR) close to the Python source called 'TorchIR.' This frontend is broadly referred to

as Torch Dynamo (Dynamo in short). The IR is arranged in the form of FxGraph. An FxGraph captures the computation graph of the block to help apply optimizations such as operator fusion. The backend of PyTorch2, loosely referred to as Torch Inductor (Inductor), lowers the TorchIR of FxGraphs to InductorIR. The InductorIR consists of primitives that are supported by different code generators. A chosen code generator generates a compiled module from the InductorIR. PyTorch2's CUDA Graph generator deploys a CUDA Graph wherever it thinks possible.

PyGraph's extension: CUDA Graph-aware Data Placement, shown in blue in Figure 6 (Section 5.2), extends parts of both Torch Inductor and Torch Dynamo. It analyzes TorchIR and metadata to find reasons for PyTorch2's failure to deploy a CUDA Graph for a FxGraph. It then modifies TorchIR and/or associated metadata to alleviate those constraints, where possible and invoke Dynamo to generate new TorchIR (if needed). For Parameter Indirection (yellow), PyTorch2 extends Triton's GPU kernel generator to introduce a new layer of indirection (Section 5.3). It also extends CUDA Graph generator for managing the pointer allocations and copies. Finally, PyGraph implements Selective CUDA Graphs (green) by introducing a profiler in the slow path that decides whether to deploy a CUDA Graph based on a cost-benefit analysis (Section 5.4). Next, we discuss key parts of PyTorch2's compilation framework in detail, along with PyGraph's extensions or additions to them. PyGraph is implemented in over 2250 lines of code, distributed as 2100 lines in Python and 150 lines in LLVM.

# 5.1 Extending Dynamo for CUDA Graphaware Data Placement

Torch Dynamo, the frontend of PyTorch2's compiler, uses the CPython frame evaluation API [41] to dynamically modify Python bytecode. It rewrites bytecode to capture sequences of PyTorch operations in the form of computation graphs called FxGraphs containing TorchIR. This IR simplifies PyTorch and Python constructs for compiler optimizations.

Dynamo symbolically traces Python code to build an Fx-Graph. When it encounters a line that cannot be traced, such as input-dependent dynamic control flow, and calls to external libraries, it terminates the current FxGraph. It then compiles the FxGraph and injects the result back into the original source through bytecode rewriting. The offending operations are executed outside Dynamo after which it resumes tracing. This process, known as a *graph break*, divides the Python code into blocks. Dynamo processes each block, lowering them to TorchIR within a FxGraph (① in Figure 6)

**PyGraph's extension:** Dynamo is slightly extended to regenerate FxGraph and corresponding TorchIR from the modfiled code block. This is part of CUDA Graph-aware Data Placement to enable wider deployment of CUDA Graphs. The modified code block is provided by the extended Inductor described next.

# 5.2 Extending Inductor for CUDA Graphaware Data Placement

Inductor lowers the TorchIR generated by Dynamo into primitive operations, such as Aten/Prim operations, that correspond to high-level PyTorch constructs. This decomposes complex PyTorch operations into simpler ones, enabling easier analysis and fusion amongst a reduced set of operations.

Inductor then augments the IR with operand information. Specifically, it analyzes the inputs (e.g., tensors) for each operation and adds metadata for its operands. Metadata contains information such as the dimensions of the tensors, whether they reside in CPU DRAM or GPU memory, and also information on its mapping to TorchIR where available.

These steps convert the TorchIR to InductorIR (② in Figure 6), which then serves as input to the subsequent code generation phase (Section 5.3) of Inductor, where further optimizations and transformations occur.

Inductor also analyzes the InductorIR to decide whether a given FxGraph can be captured as a CUDA Graph. For example, if the metadata indicates that an operand of an operation (node of FxGraph) is a CPU scalar, then it cannot use CUDA Graph (Section 3). Similarly, it could be a memcopy operation involving a CPU-resident tensor that would disallow the use of CUDA Graph. Other reasons, such as the mutation of input, also disable CUDA Graph.

**PyGraph's extension:** PyTorch2 maps an entire FxGraph to a CUDA Graph. To increase the number of GPU kernels cap-

tured as part of CUDA Graphs, PyGraph strives to map more FxGraphs to CUDA Graphs (blue elements in the Figure 6).

Towards this, PyGraph first enhances PyTorch2's routine that decides if an FxGraph can be mapped to a CUDA Graph. Specifically, if the routine decides that mapping to a CUDA Graph is not possible, PyGraph inspects the reason for the failure. If the reason is the existence of a CPU operand and/or operation, PyGraph further tracks down the offending node in InductorIR. Upon analyzing the metadata of the identified InductorIR, if PyGraph finds it to be a scalar variable (e.g., a NumPy scalar as discussed in Section 3), it uses bytecode re-write to cast the variable as a GPU tensor. Inductor next updates the metadata of the concerned operand (node) as a GPU tensor. Importantly, the FxGraph could now map to a CUDA Graph.

Similarly, PyGraph may discover that a memcopy from a CPU tensor to GPU forced Inductor to avoid CUDA Graph. PyGraph then backtraces from InductorIR to the Python class in the model script to which the offending CPU tensors belong. It then copies that tensor from the CPU to the GPU. In effect, this alters the device type of the variable without modifying the source. Consequently, this effectively hoists the offending memcopy to the constructor of the class and thus precedes any computation involving the tensor. The modified block is fed to the Dynamo, which re-generates TorchIR and then proceeds to lower it to InductorIR. PyTorch2 will be able to map FxGraph onto a CUDA Graph thanks to the modification.

Finally, Inductor may fail to deploy a CUDA Graph if the output of a node (InductorIR) is a CPU tensor. PyGraph fixes this without needing to re-generate the TorchIR. It directly modifies the metadata corresponding to the offending InductorIR to make the output a GPU tensor (*i.e.*, add device='cuda').

In summary, PyGraph significantly extends Inductor analysis and fixes issues that typically force PyTorch2 to otherwise avoid deploying one or more CUDA Graphs.

# 5.3 Extension to GPU Kernel and CUDA Graph generators for Parameter Indirection

The code generator (③ in Figure 6) is responsible for generating an optimized module (executable) for the FxGraph. It uses different frameworks/libraries depending on the target hardware. Here, we focus on systems with NVIDIA GPUs.

The code generator is integrated with optimized external libraries (*e.g.*, cuBLAS) and OpenAI's Triton framework [39]. It maintains a list of InductorIR operations (*e.g.*, matrix-multiplication, convolution), which should leverage external libraries. For these operations, the code generator injects calls to relevant library functions. The rest of the operations are created on the fly using Triton. The code generator first creates a function in Triton DSL, which provides Python-level abstractions for low-level GPU operations. It adds the triton. jit

decorator to these functions. This decorator indicates the Triton compiler that it should treat these functions as Triton functions. It compiles them into binaries compatible with NVIDIA GPUs. Note that the module created for an FxGraph may contain both Triton and non-triton (external) kernels. Once all the operations in an FxGraph are resolved, the code generator stitches them together to create an optimized module.

CUDA Graph generator: The CUDA Graph generator (4) in Figure 6) captures a module as a CUDA Graph. It does so if prior analysis on InductorIR and/or CUDA Graph-aware Data Placement enabled an FxGraph to be mapped onto a CUDA Graph (blue elements in the figure). PyTorch2 captures CUDA Graphs at the granularity of an FxGraph, *i.e.*, either the entire FxGraph or none of it.

Recall that a CUDA Graph captures kernel parameters (e.g., virtual address in pointer parameters) by value. Thus, before a CUDA Graph is captured for the module, the generator creates static placeholders for such pointer parameters. These placeholders replace the original parameters in the kernels. They have the same lifetime as that of the captured CUDA Graph, thus, can be captured by value.

The generator must consider two types of kernel parameters: ① those passed by an external source, and ② those produced by another kernels in a CUDA Graph. In the former, the placeholders do not hold the actual data needed by the kernels during graph's replay (e.g., input to the application). Thus, the values of the parameters (data) must be copied to the placeholders pointed to by the kernel parameters (pointers). These data copies are device-to-device copy, i.e., happens on the GPU. In the second case, no data copy is required. The data would be produced and written to the address by the producer kernel in the CUDA Graph.

It is the responsibility of CUDA Graph generator to replace the original parameters with static placeholders. It must also inject code that would perform the necessary data copies onto the static placeholder during a kernel's invocation as part of a CUDA Graph replay. Finally, it uses the stream capture method (Section 2) to create a CUDA Graph for the module. **PyGraph's extensions:** To implement Parameter Indirection PyGraph enhances both Triton's code generator and CUDA Graph generator (yellow elements in Figure 6). Recall that Parameter Indirection's goal is to convert data copies to pointer copies by introducing an additional level of indirection in the kernel parameters.

PyGraph's enhanced kernel code generator must consider two cases. If PyTorch2 calls closed-source kernels, then it cannot apply Parameter Indirection since it must modify kernels to add an additional layer of indirection. Thus, this optimization is limited to the kernels generated by Triton. We found that these kernels are the majority across our applications.

PyGraph makes two enhancements to the kernels generated by Triton's compiler to introduce a layer of indirection. First, instead of passing a pointer to the kernel, we pass a *pointer* to pointer. It holds the pointer to the data region on which the kernel operates. Second, the kernels must dereference such pointers to get the pointers to the data region before using it.

PyGraph must also consider two subtleties. First, a CUDA Graph may contain both Triton and non-Triton kernels. Further, an external input passed to a kernel in the CUDA Graph may be used by both a Triton and a non-Triton kernel. If so, PyGraph must avoid applying indirection to those Triton kernels for correctness. For non-Triton kernels, data copy for parameters must be performed, and thus, if these kernels modify the parameter during computation, the updates would be reflected in the copy of the parameter and *not* to the original parameter. If indirection was applied to a Triton kernel that should operate on this modified parameter, it would receive stale data. This is because the Triton kernel would operate on the original parameter and not its copy, thanks to the indirection (pointer-copy instead of data-copy). Thus, PyGraph must avoid applying indirection to such Triton kernels.

The second subtlety is that not all kernel parameters need a data copy to its static placeholder. Only the parameters from an external source (*e.g.*, input to the applications) require indirection. PyGraph ensures that indirection is introduced for only these parameters.

To apply indirection to eligible Triton kernels without manual intervention, PyGraph enhances Triton's compilation framework [39]. It tracks the parameters that can support indirection in the code generation phase (③) in Figure 6). It adds this list to the torch.jit decorator. PyGraph's custom LLVM [9] pass, added to the Triton, takes this list of parameters from the decorator and modifies the PTX [38] code of the corresponding Triton kernels. It updates the signature of the kernels to incorporate 'pointer-to-pointer' for the relevant parameters. Importantly, it adds PTX code at the start of such kernels to dereference the parameters that were subjected to indirection. It obtains the data pointers from the pointer-to-pointers before using them in computation. The rest of the kernel remains unchanged as it operates on these data pointers.

PyGraph also modifies the CUDA Graph generator to support indirection. The enhanced generator receives the list of parameters of a Triton kernel that were subjected to indirection. It then treats such parameters differently as follows: ① It allocates static placeholders for *pointers* instead of data regions, and ② It injects code to perform pointer copies for these parameters instead of data copies. Note that these pointer copies are host-to-device copies as the list of pointer parameters originally resides on the CPU. The rest of the CUDA Graph generator remains unaltered.

# 5.4 Profiler for Selective CUDA Graphs

PyGraph introduces a profiler in the compilation path (slow-path) to judiciously decide whether to deploy a CUDA Graph based on a cost-benefit analysis (colored green, (\$\overline{\sigma}\$) in Figure 6). The final outcome of the compilation is a module

(executable) corresponding to a given block of code. It is cached for use in the fast path.

In the unmodified PyTorch2, if it is able to generate a module with CUDA Graph for a given block (FxGraph), it *always* caches (uses) it. PyGraph's profiler, however, considers up to three possible choices as follows: ① A module *without* CUDA Graph, ② a module with CUDA Graph but *without* Parameter Indirection, and ③ a module with CUDA Graph and Parameter Indirection. The profiler runs each of the available modules as part of the compilation pass, measures their execution time, and caches the one with the best performance.

As noted in Section 3, it is not uncommon for overheads of a CUDA Graph to outweigh its benefits. These overheads are due to the data copies for passing kernel parameters and other bookkeeping tasks in the CUDA Graph. While Parameter Indirection can reduce data copy for many Triton kernels in a CUDA Graph, bookkeeping overheads remain. Thus, in some cases, particularly for CUDA Graphs with closed-source non-Triton kernels, the use of CUDA Graphs can still hurt performance.

One may wonder why PyGraph may have to also choose between choices ② and ③. The subtle reason why Parameter Indirection may not always help is as follows. Parameter Indirection converts device-to-device (D2D) data copies to host-to-device (H2D) pointer copies. In some corner cases where a data copy operation happens on a small object, the overhead of crossing the PCIe for H2D pointer copies may outweigh the benefits. In such cases, PyGraph profiler may choose to cache the CUDA Graph without Parameter Indirection. Overall, PyGraph's profiler makes judicious choices in deploying CUDA Graph during compilation pass to improve end-to-end application execution time.

Summary: PyGraph extends Torch Dynamo and Torch Inductor to transparently update IRs for enabling more kernels to execute as part of CUDA Graph (CUDA Graph-aware Data Placement). It extends Triton's kernel generator and CUDA Graph generator to add indirections to convert data copies in CUDA Graph to pointer copies (Parameter Indirection). Finally, it adds a profiler in the slow-path to decide when it is beneficial to deploy CUDA Graph (Selective CUDA Graphs).

## 6 Evaluation

Our evaluation of PyGraph answers the following questions:

- How does PyGraph perform compared to PyTorch2?
- What are the effects of each of our optimizations on endto-end application performance?
- Where do the performance gains of PyGraph come from? Workloads and metric: We evaluate applications from three popular ML benchmark suites: TorchBench [18], Hugging-Face [19] and TIMM [59]. Table 1 lists these workloads. In total, these benchmarking suites contain 183 applications from image processing, natural language, and HPC, including

Suite	Application (Short Name, Type, Batch Size)
TorchBench	Equation of State (EOS, I 1048576), Speech Transformer
	(ST, I, 1), DALLE2 (DALLE, I, 1), Multi-modal Clip
	(MMC, I, 1), Turbulent Kinetic Energy (TKE, I, 1048576),
	Deep Recommender (DR-T, T, 4096), Deep Recommender
	(DR-I, I, 256), Vision MaskRCNN (VM, I, 1), Learning to
	Paint, (LP, T, 16), AlexNet (ANET, I, 128), Deep Learning
	Recommender (DLRM, I, 2048)
HuggingFace	XLNet LM-head Model (XLNet-I, I, 8), XLNet LM-head
	Model (XLNet-T, T, 8), Electra for Causal ML (ECML,
	T, 64), Camem BERT (CBERT, T, 64), Distilled GPT2
	(DGPT, T, 16), Deberta for Masked LM (DMLM, T,
	16), Google FNet (GLFNET, T, 16), Distilled BERT for
	Masked LM (DBMLM, T, 128)
TIMM	LCNet_050 (LCNET, I, 256)

**Table 1:** Details of the workloads (T: Training, I: Inference).

both training and inference. We focus on top 20 of these applications that are subject to benefiting from CUDA graphs. We report performance as an average of 50 runs for each application. Since ML applications are repetitive by nature, we use the runtime of a single iteration as the primary performance metric, i.e., the forward pass for inference and the forward and backward pass for training workloads.

**Environment:** We conduct all experiments on a single NVIDIA RTX A6000 GPU [37] with 48 GB of GDDR6 [21] memory. The host has a 64-core AMD EPYC 9554 [3] CPU with 512 GB DDR5 [22] memory. The software stack consists of PyTorch v2.4 [44] with CUDA v12.1 [32], cuDNN 8.9.2 [36], and NVIDIA driver version 535.216.01 [34].

**Baselines:** We evaluate PyGraph against two baselines. The first baseline is PyTorch2 *without* the use of CUDA Graphs and the second is PyTorch2 using CUDA Graphs. We refer to them as PyTorch2-NCG and PyTorch2-CG, respectively.

#### 6.1 Results

Figure 7 compares PyGraph's performance against that of PyTorch2-NCG and PyTorch2-CG. The figure shows two bars for each application wherein the first bar shows the speedup of PyTorch2-CG over PyTorch2-NCG, and the second reports the speedup of PyGraph over PyTorch2-NCG.

First, we find that PyTorch2-CG is not only suboptimal in deploying CUDA Graphs, it can also *hurt performance* for many applications. This is counter-intuitive since CUDA Graphs are introduced solely for improving performance. Overall, PyTorch2-CG degrades performance over PyTorch2-NCG by more than 3% for 8 out of the 20 applications, with the worst-case slowdown of up to 36% for EOS. TKE and DR-T slow down by 11%. This highlights the need to judiciously deploy CUDA Graphs. However, PyTorch2-CG blindly deploys CUDA Graphs. Unfortunately, this means that the onus of avoiding performance loss due to suboptimal use of CUDA Graphs lies on the end users today: the developers must profile each CUDA Graph candidate in their applications and

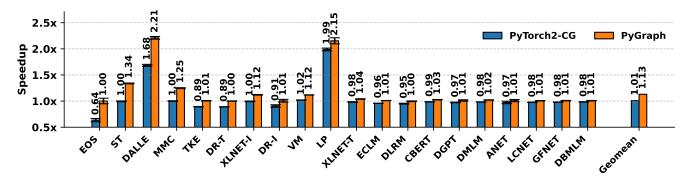


Figure 7: Performance improvement of PyTorch2-CG and PyGraph against PyTorch2-NCG.

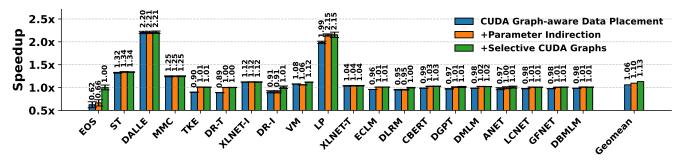


Figure 8: Ablation study on different optimizations of PyGraph (performance normalized against PyTorch2-NCG).

disable CUDA Graphs manually, wherever required.

Second, even in cases where PyTorch2-CG improves performance over PyTorch2, it still leaves performance on the table, as shown by the difference between PyTorch2-CG and PyGraph. PyGraph improves performance over PyTorch2-CG by up to 57% (12% on average). The best speedup of PyGraph over PyTorch2-NCG is 2.21× for DALLE and 2.15× for LP, whereas the best speedup over PyTorch2-CG is noted for EOS and ST (57% and 34%, respectively). In all cases, PyGraph is at least as good as the best of PyTorch2-NCG and PyTorch2-CG. This is an important result that shows that PyGraph can effectively leverage CUDA Graphs when there is an opportunity without hurting performance elsewhere. Importantly, PyGraph achieves this without manual intervention.

Figure 8 separately analyzes the impact of each of Py-Graph's optimizations. The figure shows three bars. The first bar shows the performance obtained by applying CUDA Graph-aware Data Placement alone. The second bar shows that with Parameter Indirection in tandem with CUDA Graph-aware Data Placement (+Parameter Indirection). The third bar shows the cumulative effect of all the optimizations applied together (+Selective CUDA Graphs). The height of each bar is normalized against the performance of PyTorch2-NCG. From Figure 8, we observe that each optimization is important to the overall performance of PyGraph.

CUDA Graph-aware Data Placement enables more kernels to be captured within CUDA Graphs, amortizing the overhead associated with individual kernel launches. As a result, applications such as ST and DALLE show significant performance gains of 34% and 31% over PyTorch2-CG, respectively.

Parameter Indirection reduces the number of bytes copied in passing parameters to GPU kernels while replaying CUDA Graphs. It does so by converting data copy to pointer copy through an additional layer of indirection. This significantly speeds up applications such as TKE, DR-T, and LP by 12.2%, 12.3%, and 8%, respectively, over and above CUDA Graphaware Data Placement. In the next section, we show that these gains come from reductions in the amount of bytes copied.

Selective CUDA Graphs judiciously deploys CUDA Graphs driven by a cost-benefit analysis. It identifies and disables CUDA Graphs where they could hurt performance, ensuring that PyGraph never degrades performance. This is particularly important for applications such as EOS, DR-I, DLRM where blind deployment of CUDA Graphs degrades performance by 5-36%. Overall, Figure 8 confirms that each optimization is important.

# 6.2 Analyzing sources of improvement

We here evaluate the secondary effect of each of the optimizations to provide a deeper insight into how PyGraph helps improve performance.

**CUDA Graph-aware Data Placement:** The goal of CUDA Graph-aware Data Placement is to increase the number of GPU kernels that execute as part of CUDA Graphs. Table 2 shows the percentage of kernels captured in CUDA Graphs before and after applying this optimization for a set of representative workloads. We notice a significant increase in

App.	% of Kernels in CUDA Graphs			
	PyTorch2-CG	CUDA Graph-aware Data Placement		
ST	4.66	78.08		
DALLE	76.51	99.14		
MMC	0.0	100.0		
XLNET-T	1.27	98.58		
XLNET-I	0.0	100.0		
VM	68.73	84.17		

**Table 2:** Enhanced coverage by CUDA Graphs due to CUDA Graph-aware Data Placement.

App.	% Reduction (# CUDA Graphs)	App.	% Reduction (# CUDA Graphs)
EOS	99 (0)	ST	1 (3)
DALLE	46 (2)	MMC	5 (1)
TKE	99 (1)	DR-T	99 (1)
XLNET-I	97 (0)	DR-I	0 (0)
VM	99 (4)	LP	99 (1)
XLNET-T	99 (2)	ECLM	99 (2)
DLRM	0 (0)	CBERT	99 (3)
DGPT	99 (3)	DMLM	99 (1)
ANET	99 (1)	LCNET	99 (1)
GFNET	99 (2)	DBMLM	99 (2)

**Table 3:** Reduction in data copy due to PyGraph.

the percentage of an application's kernels executed as part of CUDA Graphs. For instance, the percentage of kernels in CUDA Graphs for Speech Transformer (ST) increased from 4.66% to 78.08%. Similarly, the DALLE witnessed an increase from 76.51% to 99.14%, and the XLNET-T witnessed an increase from 1.27% to 98.58%. These highlight the substantial enhancement in CUDA Graphs coverage due to CUDA Graph-aware Data Placement.

Parameter Indirection: Table 3 shows the percentage reduction in the number of bytes copied for passing parameters to GPU kernels due to Parameter Indirection. We observe substantial reductions in bytes copied for most workloads. For instance, 14 out of the 20 workloads (e.g., Speech Transformer (ST), Distilled GPT (DGPT), AlexNET (ANET)) experienced a reduction of 99%, indicating that Parameter Indirection nearly eliminates all data copy related to kernel parameter copies in CUDA Graphs for these workloads. Some of the other workloads, e.g., DALLE and XLNET-I, also saw significant data copy reductions of 46% and 97%. In comparison, MMC, ST, and DLRM experienced no reduction in bytes copied. This is because these applications mostly rely on closed-source, non-Triton kernels, such as those from cuBLAS and cuDNN where our optimization does not apply.

Table 3 also shows the number of CUDA Graphs optimized by PyGraph via indirection (numbers in parenthesis). To explain, VM sees a 99% reduction in data copy due to Parameter Indirection. VM has a total of 21 CUDA Graphs (Table 4), of which 4 CUDA Graphs are deployed via pointer indirection

by PyGraph.

Overall, PyGraph optimized data copy overhead in 17 out of 20 applications via Parameter Indirection. We find that DGPT and CBERT exclusively used Parameter Indirection in all deployed CUDA Graphs, whereas GFNET and DBMLM benefited partially (some of their kernels are based on cuBLAS and cuDNN) (Table 4, discussed later).

Selective CUDA Graphs: Finally, we evaluate the impact of selectively deploying CUDA Graphs. Table 4 shows the total number of CUDA Graphs deployed by PyGraph without Selective CUDA Graphs (column "Total" in Table 4) and the full-fledged PyGraph with all the optimizations. It also shows the number of CUDA Graphs that were turned off by PyGraph's Selective CUDA Graphs (column "Disabled"). To illustrate a specific example of this, PyGraph can deploy a total of 21 CUDA Graphs for VM (column "Total" in Table 4). However, PyGraph with all optimizations, selectively enables only 11 of these CUDA Graphs (sub-column "Enabled" under PyGraph), disabling the remaining 10 due to unfavorable costbenefit analyses ("Disabled"). Of the 11 enabled graphs, 4 utilize pointer indirection ("Indirect"), while the remaining 7 are deployed without indirection.

From the results, we observe that PyGraph selectively deployed CUDA Graphs in 7 out of the 20 applications based on the cost-benefit analysis. In total, 47 CUDA Graphs were deployed by PyGraph across all the 20 workloads, and 20 were turned off by Selective CUDA Graphs. This selective deployment ensures that a CUDA Graph is deployed only when it is beneficial, helping end-to-end application performance. *Summary of results*: We show that PyTorch2 compiler fails to deploy CUDA Graphs in many cases while sometimes it deploys them even when they are counter-productive. PyGraph not only enables more kernels to execute in CUDA Graphs, it disables them when not helpful. Overall, PyGraph improves performance by up to 57% over PyTorch2-CG (geomean 12% for 20 applications), working transparently within PyTorch2.

#### 7 Related Work

ML compilers: Machine learning compilers such as Tensor-Flow [1], Caffe [23], Theano [57], and CNTK [48] extensively rely on graph-level optimizations such as fusion and scheduling across operator boundaries to extract maximum performance. Some compilers also aim to facilitate easy development of low-level tensor programs [5,6,10,15,17,27,39,49,58,60,62,64–67], and target efficient code generation for different hardware platforms as well as complex workloads such as those with irregular and dynamic shapes [14,50,56,61]. In general, such compilers are orthogonal to PyGraph since their primary focus is on improving the execution efficiency of different operators whereas our primary goal is to mitigate the overhead of launching GPU kernels.

The key contribution of PyTorch2 [4] was enabling compilation of graphs without sacrificing the flexibility and us-

App.	Number of CUDAGraphs				
	Total	Enabled	Indirect	Disabled	
EOS	1	0	0	1	
ST	7	4	3	3	
DALLE	6	4	2	2	
MMC	1	1	1	0	
TKE	1	1	1	0	
DR-T	3	1	1	2	
XLNET-I	1	1	0	0	
DR-I	1	0	0	1	
VM	21	11	4	10	
LP	2	2	1	0	
XLNET-T	3	3	2	0	
ECLM	2	2	2	0	
DLRM	1	0	0	1	
CBERT	3	3	3	0	
DGPT	3	3	3	0	
DMLM	3	3	1	0	
ANET	1	1	1	0	
LCNET	1	1	1	0	
GFNET	3	3	2	0	
DBMLM	3	3	2	0	

**Table 4:** CUDA Graphs disabled by PyGraph's Selective CUDA Graphs.

ability of Python. This requires allowing users to customize models with arbitrary code while still being able to map programs to fix graph structures. Prior approaches to capture graphs in PyTorch such as JIT tracing, JIT scripts [46], lazy tensors [45,55], symbolic tracing [47] faced several restrictions e.g., being unsound, unable to support all programs, high overheads, frequent recompilation etc. TorchDynamo instead rewrites Python bytecode to extract sequences of PyTorch operations into an FX graph [47]. In doing so, it gracefully falls back to partial graphs when needed. PyGraph relies on Torch-Dynamo's ability to generate CUDA graphs out of FX graphs and optimizes them further with several novel optimizations. **CUDA Graphs:** Grape [63] is the closest work to PyGraph, focusing on efficient usage of CUDA Graphs for dynamic DNNs via three key optimizations: a predication rewriter, a metadata compressor, and an alias predictor. While it is one of the earliest works for harnessing CUDA Graph, it requires significant manual code changes in GPU kernels and CUDA drivers which is at odds with PyGraphgoal of avoiding manual changes. For instance, Grape's predication rewriter necessitates adding prediction contexts within kernel code, leading to over 400 lines of code changes for optimizing beam search in GPT models. Additionally, its metadata compressor requires modifications to CUDA drivers and relies on a specific driver and CUDA version. Its alias predictor is incompatible with PyTorch2 due to its restriction on aliasing CUDA Graph's placeholders. In contrast, PyGraph is application-transparent, supporting over 180 applications out-of-the-box.

# 8 Conclusion

We present PyGraph — a compiler framework designed to enhance the deployment and performance of CUDA Graphs within PyTorch2. PyGraph addresses several key limitations of the PyTorch2 compiler, resulting in substantial performance improvement across various ML benchmarks without requiring manual modifications to applications.

#### References

- [1] 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 G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 265–283, USA, 2016. USENIX Association.
- [2] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming Throughput-Latency tradeoff in LLM inference with Sarathi-Serve. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 117–134, Santa Clara, CA, July 2024. USENIX Association.
- [3] AMD. Amd epyc 9554. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9554.html, 2024.
- [4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, 2024.
- [5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming

- Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 193–205, 2019.
- [6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: an automated endto-end optimizing compiler for deep learning. In *Pro*ceedings of the 13th USENIX Conference on Operating Systems Design and Implementation, OSDI'18, page 579–594, USA, 2018. USENIX Association.
- [7] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Serving heterogeneous machine learning models on Multi-GPU servers with Spatio-Temporal sharing. In 2022 USENIX Annual Technical Conference (USENIX ATC 22), pages 199–216, Carlsbad, CA, July 2022. USENIX Association.
- [8] Arnab Choudhury, Yang Wang, Tuomas Pelkonen, Kutta Srinivasan, Abha Jain, Shenghao Lin, Delia David, Siavash Soleimanifard, Michael Chen, Abhishek Yadav, Ritesh Tijoriwala, Denis Samoylov, and Chunqiang Tang. MAST: Global scheduling of ML training across Geo-Distributed datacenters at hyperscale. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 563–580, Santa Clara, CA, July 2024. USENIX Association.
- [9] LLVM developer group. The llvm compiler infrastructure. https://llvm.org/, 2024.
- [10] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. 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, page 370–384, New York, NY, USA, 2023. Association for Computing Machinery.
- [11] Linhao Dong, Shuang Xu, and Bo Xu. Speech-transformer: A no-recurrence sequence-to-sequence model for speech recognition. In 2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pages 5884–5888, 2018.
- [12] dscerutti and NVIDIA. Any way to measure the latency of a kernel launch? https://forums.developer.nvidia.com/t/any-way-to-measure-the-latency-of-a-kernel-launch/221413, 2022.

- [13] Babak Falsafi. Post-moore server architecture. In Proceedings of the 34th ACM International Conference on Supercomputing, ICS '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [14] Pratik Fegade, Tianqi Chen, Phillip B. Gibbons, and Todd C. Mowry. The cora tensor compiler: Compilation for ragged tensors with minimal padding, 2022.
- [15] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, and Tianqi Chen. 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, page 804–817, New York, NY, USA, 2023. Association for Computing Machinery.
- [16] Alan Gray. Getting started with cuda graphs. https://developer.nvidia.com/blog/cuda-graphs/, 2019.
- [17] Bastian Hagedorn, Bin Fan, Hanfeng Chen, Cris Cecka, Michael Garland, and Vinod Grover. 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, page 302–313, New York, NY, USA, 2023. Association for Computing Machinery.
- [18] Yueming Hao, Xu Zhao, Bin Bao, David Berard, Will Constable, Adnan Aziz, and Xu Liu. Torchbench: Benchmarking pytorch with high api surface coverage, 2023.
- [19] HuggingFace. Transformers. https://github.com/huggingface/transformers.
- [20] jansel. Torchinductor: a pytorch-native compiler with define-by-run ir and symbolic shapes. https://dev-discuss.pytorch.org/t/torchinductor-a-pytorch-native-compiler-with-define-by-run-ir-and-symbolic-shapes/747, 2024.
- [21] JEDEC. Graphics double data rate (gddr6) sgram standard. https://www.jedec.org/standards-documents/docs/jesd250d, 2023.
- [22] JEDEC. Ddr5 sdram. https://www.jedec.org/standards-documents/docs/jesd79-5c01, 2024.
- [23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding, 2014.

- [24] Aditya K. Kamath and Arkaprava Basu. iguard: Ingpu advanced race detection. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 49–65, New York, NY, USA, 2021. Association for Computing Machinery.
- [25] Aditya K. Kamath, Alvin A. George, and Arkaprava Basu. Scord: A scoped race detector for gpus. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pages 1036–1049, 2020.
- [26] Wonbeom Lee, Jungi Lee, Junghwan Seo, and Jaewoong Sim. InfiniGen: Efficient generative inference of large language models with dynamic KV cache management. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 155–172, Santa Clara, CA, July 2024. USENIX Association.
- [27] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Trans. Graph.*, 37(4), July 2018.
- [28] Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 929–945, Santa Clara, CA, July 2024. USENIX Association.
- [29] Ajay Nayak and Arkaprava Basu. Over-synchronization in gpu programs. In 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 795–809, 2024.
- [30] Vinh Nguyen, Michael Carilli, Sukru Burc Eryilmaz, Vartika Singh, Michelle Lin, Natalia Gimelshein, Alban Desmaison, and Edward Yang. Accelerating pytorch with cuda graphs. https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/, 2021.
- [31] NVIDIA. Basic linear algebra on nvidia gpus. https://developer.nvidia.com/cublas, 2024.
- [32] NVIDIA. Cuda toolkit 12.1. https://developer.nvidia.com/cuda-12-1-0-download-archive?target\_os=Linux&target\_arch=x86\_64&Distribution=Ubuntu&target\_version=22.04&target\_type=runfile\_local, 2024.
- [33] NVIDIA. Cutlass: Cuda c++ template abstractions for implementing high-performance matrix-matrix multiplication (gemm). https://github.com/NVIDIA/cutlass, 2024.

- [34] NVIDIA. Linux x64 (amd64/em64t) display driver 535.216.01 | linux 64-bit. https://www.nvidia.com/en-us/drivers/details/233000/, 2024.
- [35] NVIDIA. Nvidia cuda deep neural network library. https://developer.nvidia.com/cudnn, 2024.
- [36] NVIDIA. Nvidia cudnn documentation. https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-892/index.html, 2024.
- [37] NVIDIA. Nvidia rtx a6000 graphics card. https://www.nvidia.com/en-in/design-visualization/rtx-a6000/, 2024.
- [38] NVIDIA. Parallel thread execution isa https://docs.nvidia.com/cuda/parallel-thread-execution/, 2024.
- [39] OpenAI. Triton: Open-source gpu programming for neural networks. https://openai.com/index/triton/, 2024.
- [40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.
- [41] Python. Pep 523 adding a frame evaluation api to cpython. https://peps.python.org/pep-0523/, 2024.
- [42] PyTorch. speech transformer model in torchbenchmark. https://github.com/pytorch/benchmark/tree/main/torchbenchmark/models/speech\_transformer.
- [43] PyTorch. Dynamo overview. https://pytorch.org/docs/stable/torch.compiler\_dynamo\_overview.html, 2024.
- [44] PyTorch. Pytorch 2024-04-23 nightly release. https://github.com/pytorch/pytorch/commit/6dlc678c58d4fb9468aca3daac67ae7bcec8495d, 2024.
- [45] PyTorch. Pytorch/xla. https://github.com/pytorch/xla, 2024.
- [46] PyTorch. Torchscript. https://pytorch.org/docs/ main/jit.html, 2024.
- [47] James K. Reed, Zachary DeVito, Horace He, Ansley Ussery, and Jason Ansel. Torch.fx: Practical program capture and transformation for deep learning in python, 2022.

- [48] Frank Seide and Amit Agarwal. Cntk: Microsoft's opensource deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 2135, New York, NY, USA, 2016. Association for Computing Machinery.
- [49] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, Advances in Neural Information Processing Systems, 2022.
- [50] Haichen Shen, Jared Roesch, Zhi Chen, Wei Chen, Yong Wu, Mu Li, Vin Sharma, Zachary Tatlock, and Yida Wang. Nimble: Efficiently compiling dynamic neural networks for model inference. *ArXiv*, abs/2006.03031, 2020.
- [51] Sudipta Saha Shubha, Haiying Shen, and Anand Iyer. USHER: Holistic interference avoidance for resource optimized ML inference. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 947–964, Santa Clara, CA, July 2024. USENIX Association.
- [52] Muthian Sivathanu, Tapan Chugh, Sanjay S. Singapuram, and Lidong Zhou. Astra: Exploiting predictability to optimize deep learning. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 909–923, New York, NY, USA, 2019. Association for Computing Machinery.
- [53] Foteini Strati, Paul Elvinger, Tolga Kerimoglu, and Ana Klimovic. Ml training with cloud gpu shortages: Is crossregion the answer? In *Proceedings of the 4th Workshop* on Machine Learning and Systems, EuroMLSys '24, page 107–116, New York, NY, USA, 2024. Association for Computing Machinery.
- [54] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 1075–1092, New York, NY, USA, 2024. Association for Computing Machinery.
- [55] Alex Suhan, Davide Libenzi, Ailing Zhang, Parker Schuh, Brennan Saeta, Jie Young Sohn, and Denys Shabalin. Lazytensor: combining eager execution with domain-specific compilers, 2021.
- [56] Shizhi Tang, Jidong Zhai, Haojie Wang, Lin Jiang, Liyan Zheng, Zhenhao Yuan, and Chen Zhang. Freetensor: a

- free-form dsl with holistic optimizations for irregular tensor programs. In *Proceedings of the 43rd ACM SIG-PLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 872–887, New York, NY, USA, 2022. Association for Computing Machinery.
- [57] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre-Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Mélanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Zive Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian Goodfellow, Matt Graham, Caglar Gulcehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrancois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert T. McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. Theano: A python framework for fast computation of mathematical expressions, 2016.
- [58] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. Unit: Unifying tensorized instruction compilation. In 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 77–89, 2021.
- [59] Ross Wightman. Pytorch image models. https://github.com/rwightman/pytorch-image-models,

2019.

- [60] Jiarong Xing, Leyuan Wang, Shang Zhang, Jack Chen, Ang Chen, and Yibo Zhu. Bolt: Bridging the gap between auto-tuners and hardware-native performance. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceed*ings of Machine Learning and Systems, volume 4, pages 204–216, 2022.
- [61] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 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, page 660–678, New York, NY, USA, 2023. Association for Computing Machinery.
- [62] 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. Akg: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1233–1248, New York, NY, USA, 2021. Association for Computing Machinery.
- [63] Bojian Zheng, Cody Hao Yu, Jie Wang, Yaoyao Ding, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Grape: Practical and efficient graphed execution for dynamic deep neural networks on gpus. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '23, page 1364–1380, New York, NY, USA, 2023. Association for Computing Machinery.
- [64] 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. Ansor: generating high-performance tensor programs for deep learning. In *Proceedings of the* 14th USENIX Conference on Operating Systems Design and Implementation, OSDI'20, USA, 2020. USENIX Association.
- [65] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. Amos: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 874–887, New York, NY, USA, 2022. Association for Computing Machinery.
- [66] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule ex-

- ploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 859–873, New York, NY, USA, 2020. Association for Computing Machinery.
- [67] 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. ROLLER: Fast and efficient tensor compilation for deep learning. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 233–248, Carlsbad, CA, July 2022. USENIX Association.
- [68] Donglin Zhuang, Zhen Zheng, Haojun Xia, Xiafei Qiu, Junjie Bai, Wei Lin, and Shuaiwen Leon Song. MonoNN: Enabling a new monolithic optimization space for neural network inference tasks on modern GPU-Centric architectures. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 989–1005, Santa Clara, CA, July 2024. USENIX Association.