# Evaluating the PyTorch Compiler in the Vision Domain

*Aidan Goldfarb*

Department of Computer Science

McGill University

Montréal, Québec, Canada

August 15, 2024

A thesis presented for the degree of

Computer Science

# Contents

# List of Figures

# List of Tables

# Abstract

Machine learning (ML) is becoming pervasive, with applications spanning virtually every domain of human activity, from art to autonomous driving. Several tools are available to develop ML models, with Python being a popular choice due to its ease of use. As datasets grow ever larger, the need for speed in processing and training models becomes more critical, particularly in the context of vision models. These models, which include convolutional neural networks (CNNs), are essential for tasks such as image recognition, object detection, and autonomous driving. Training these models can take months on multiple GPUs, so even minor speed improvements can significantly reduce overall computation time and energy cost. Compilation is the natural solution to improving performance of long running programs.

Despite their flexibility and ease of use, in-Python compilers, which are designed to speed execution time, can slow some operations. This thesis explores the performance of the PyTorch compiler, TorchDynamo, starting with the fundamental operations of matrix multiplication, then convolutions, and finally evaluating full vision models. It also has the goal of understanding where and why the compiler excels or fails in optimizing performance.

The primary objective of this thesis is to understand the behavior of the PyTorch compiler in the context of vision models. These investigations aim to identify bottlenecks and optimization opportunities, ultimately enhancing the efficiency of vision models in practical applications. Specifically, six key vision model architectures are examined and tested. Evaluating the PyTorch compiler's performance on these models and their fundamental operations, such as matrix multiplication and convolutions, provides insights into its effectiveness as well as potential areas for improvement. By exploring these building blocks, this thesis aims to contribute to the optimization of vision models, augmenting their efficiency and applicability in real-world scenarios. Additionally, the thesis explores potential paths for improving compiler performance, such as selective compilation strategies.

# Abrégé

L'apprentissage machine (AM) devient omniprésent, avec des applications couvrant pratiquement tous les domaines de l'activité humaine, de l'art à la conduite autonome. Plusieurs outils sont disponibles pour développer des modèles d'apprentissage automatique, Python étant un choix populaire en raison de sa facilité d'utilisation. Les ensembles de données devenant de plus en plus volumineux, le besoin de rapidité dans le traitement et l'entraînement des modèles devient de plus en plus critique, en particulier dans le contexte des modèles de vision. Ces modèles, qui comprennent les réseaux neuronaux convolutifs (RNC), sont essentiels pour des tâches telles que la reconnaissance d'images, la détection d'objets et la conduite autonome. L'entraînement de ces modèles peut prendre des mois sur plusieurs GPU, de sorte que même des améliorations mineures de la vitesse peuvent réduire de manière significative le temps de calcul global et le coût énergétique. La compilation est la solution naturelle pour améliorer les performances des programmes de longue durée.

Malgré leur flexibilité et leur facilité d'utilisation, les compilateurs in-Python, qui sont conçus pour accélérer le temps d'exécution, peuvent ralentir certaines opérations. Cette thèse

explore les performances du compilateur PyTorch, TorchDynamo, en commençant par les opérations fondamentales de multiplication de matrices, en progressant vers les convolutions, et enfin en évaluant des modèles de vision complets. Elle a aussi pour but de comprendre où et pourquoi le compilateur excelle ou échoue dans l'optimisation des performances.

L'objectif principal de cette thèse est de comprendre le comportement du compilateur PyTorch dans le contexte des modèles de vision. Ces recherches visent à identifier les goulots d'étranglement et les opportunités d'optimisation, pour finalement améliorer l'efficacité des modèles de vision dans les applications pratiques. Plus précisément, six architectures clés de modèles de vision sont examinées et testées. L'évaluation des performances du compilateur PyTorch sur ces modèles et leurs opérations fondamentales, telles que la multiplication matricielle et les convolutions, fournit des indications sur son efficacité ainsi que sur les domaines potentiels d'amélioration. En explorant ces blocs de construction, cette thèse vise à contribuer à l'optimisation des modèles de vision, en augmentant leur efficacité et leur applicabilité dans les scénarios du monde réel. En outre, la thèse explore des voies potentielles pour améliorer la performance des compilateurs, telles que les stratégies de compilation sélective.

# Acknowledgements

I would like to express my deepest gratitude to those who have supported and guided me throughout the journey of this thesis.

First and foremost, I am profoundly grateful to my advisors, Oana and Christophe, whose insightful guidance, encouragement, and support have been instrumental in the completion of this work.

I also wish to thank my labmates, Loïc and Sebastian, for their invaluable support and guidance as I navigated a new department, city, and country.

To my friends, without you this thesis would not be possible. Brennan, your serious dedication to LaTeX has been a source of strength for me in my darkest moments. Arinze, you always knew when to not take something too seriously. Nic, you were a good TA and a better friend.

Most importantly, I owe an immeasurable debt of gratitude to my parents, and sister. Your unwavering support, boundless encouragement, and steadfast belief in me have been the foundation upon which this achievement is built. Thank you for always having my back.

# Chapter 1

# Introduction

## 1.1 Deep Learning in the Vision domain

Since 2010, the field of machine learning (ML) has seen a significant shift towards deeper models, enabled by increased computing capabilities. However, the amount of data needing to be processed is growing at a faster rate than available processors can handle. This necessitates the use of parallelization and accelerators to maintain performance. This is more evident in the computer vision field.

The vision field encompasses tasks for acquiring, processing, analyzing and understanding digital images, and extraction of high-dimensional data from the real world to produce numerical or symbolic information [1]. Data growth is particularly pronounced in this field, driven by the proliferation of high-definition images and videos, which can now be easily

produced and stored. Deep learning models in the vision field are traditionally implemented in Python due to its ease of entry and extensive community development.

Python's interpreted nature means it is easy to write and debug, but slow to execute. Developers employ compilation to long running programs in hopes that the more efficient machine code will offset the compilation time. With the growing size of datasets, this translation from easy-to-write Python code to fast machine-executable code is becoming the industry norm. PyTorch, a widely used framework in Python for ML and deep learning, benefits from the contributions of a large community. This includes a Just-in-Time (JIT) deep learning compiler, TorchDynamo, that accelerates Python code by generating C++/OpenMP [2] code for the CPU and Triton code for the GPU. However, this compiler is not always faster; in some cases, the generated machine code can be slower than interpreted execution.

## 1.2 Contributions

This thesis explores the predictability and stability of Python's state-of-the-art deep learning compiler, TorchDynamo, on models in the vision domain. To this end, the building blocks of these complex models are tested. Initially, an analysis of the compiler's performance on matrix multiplication (MM), the fundamental operation in almost all machine learning, is conducted. Then convolutions, the building block of all image processing models, are tested. Finally, a deep analysis of six vision models is done. Throughout, the methods for timing

and evaluation are critically vetted, as timing both accelerated operations are non-trivial.

This thesis demonstrates:

- Examinations of the CUDA and Triton matrix multiplication algorithms.
- Which convolutions perform well under compilation, and the effectiveness of convolutional fusion in the PyTorch compiler.
- Vision model features which perform well under compilation. Namely 1x1 dimension reducing convolutions.

## 1.3 Thesis structure

The thesis is partitioned into six chapters. Following this introductory section, Chapter 2 discusses the higher-level concepts required to understand the work done. Chapter 3 introduces the relevant literature and work that has made this study possible. Chapter 4 describes in detail the methodology of each of the experiments conducted. All experiments and code can be reproduced with information in this section. Chapter 5 presents and discusses the results and implications of each of the experiments. Finally, Chapter 6 provides a summary of contributions, a critical analysis of the work, and avenues for future exploration and implementation.

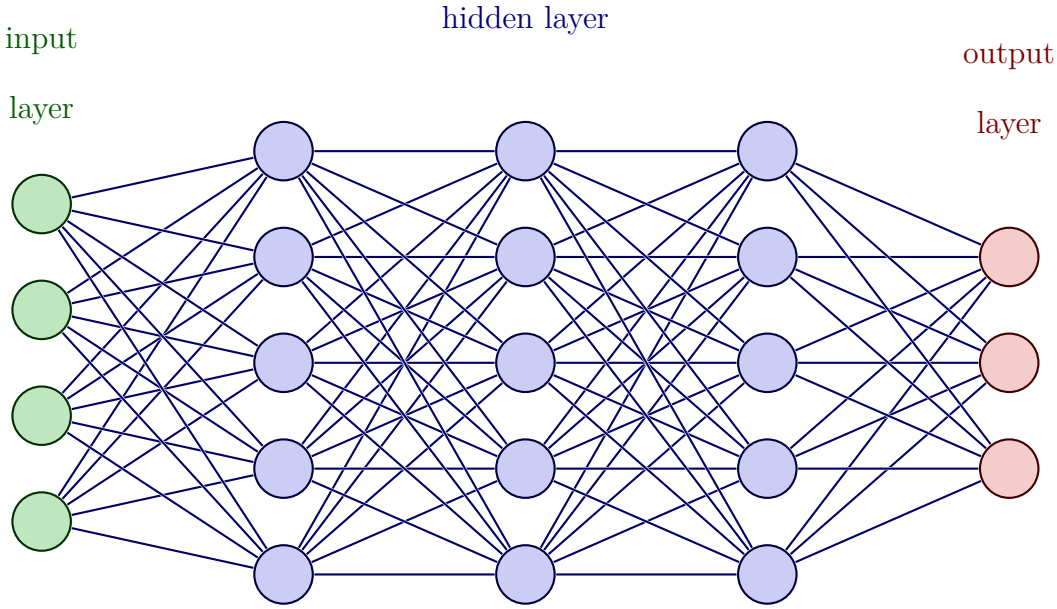
# Chapter 2

# Background

## 2.1 Neural networks

Neural networks are a class of machine learning models based on the structure and function of the human brain. They consist of interconnected layers of nodes–or neurons–wherein each connection represents a weighted path through which data flows. The primary utility of neural networks is their capacity to learn patterns from data. The learning is done through a process called training. Most neural networks are trained with labeled data, data which is tagged with identifying properties, enabling it to be compared to a "prediction" made in the final layer of the network. During supervised training, the network passes data forward through the network to generate predictions, a process called inference, or the forward pass. The network adjusts the weights of its connections based on the input data

and the error of its predictions through a process known as backwards propagation of errors, backpropagation, or the backward pass. The diagram below represents a five layer neural network. The forward pass consists of passing data from left to right along the edges and through the nodes, and the backwards pass from right to left.




**Five layer neural network**

Backpropagation [3] involves the application of the chain rule to compute gradients in a neural network. This is typically done using an optimization algorithm like gradient descent, facilitated by a technique called auto-differentiation [4], which to automatically computes gradients. Due to their ability to model non-linear relationships and capture intricate data dependencies, neural networks have proven effective in various tasks, including image and speech recognition, natural language processing, and game playing. Their versatility and

pattern-finding ability make them an essential feature of modern artificial intelligence and machine learning algorithms.

## 2.2 Modern workloads

Modern workloads in machine learning and artificial intelligence involve complex operations and models. The experiments described in this thesis focus on vision models and their principal components.

### 2.2.1 Matrix multiplication

Matrix multiplication is a core operation found in most vision models. It involves the multiplication of two matrices to produce a third matrix. This operation is crucial for various tasks, including neural network layer computations, and many other numerical algorithms. Efficient execution of matrix multiplication can significantly impact the performance of machine learning models, especially when dealing with large datasets and high-dimensional data.

### 2.2.2 Convolutions

Convolutions are operations that are virtually always employed in the field of computer vision and are the building blocks of Convolutional Neural Networks (CNNs). A convolution

involves sliding a filter (kernel) over an input image or feature map to produce a new feature map that captures specific patterns. The general formula for a convolution is described on the following page, adapted from [5].

$$\mathbf{g(x,\ y)} = \omega * f(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} \omega(i,j) f(x-i, y-j)$$

$g(x,y)$: The output image after applying the convolution.
$\omega$: The convolution kernel or filter.
$*$: The convolution operation.
$f(x,y)$: The input image.
$i,j$: Indices that iterate over the kernel dimensions.
$a,b$: Half-width and half-height of the kernel.

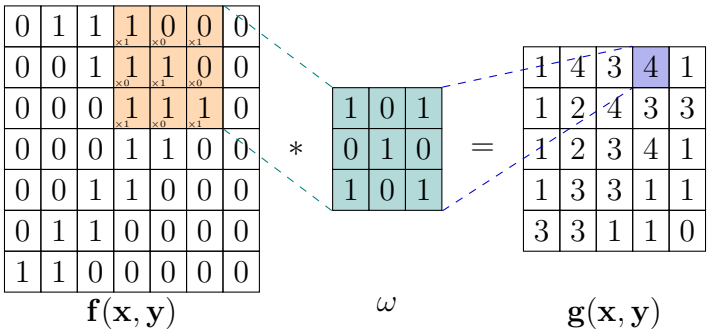
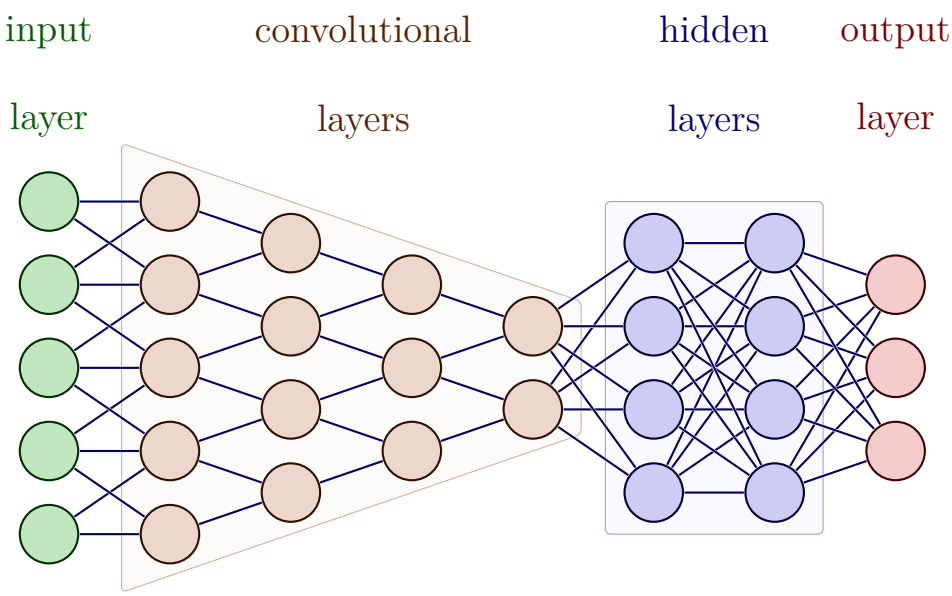



**Figure 2.1:** Convolution definition

Convolutions help reduce the spatial dimensions of the data while preserving important features needed for tasks like image classification, object detection, and segmentation. With the growing size of datasets, optimizing convolution operations is crucial for improving the efficiency and speed of CNNs.

### 2.2.3 Vision models

Vision models, such as CNNs, are designed to process and analyze visual data, including images and videos. These models have revolutionized the field of computer vision, enabling significant advancements in areas like image recognition, facial detection, and autonomous driving. Vision models typically consist of multiple layers, including convolutional, pooling, and fully connected layers, all of which work together to extract and interpret visual features. The performance of these models depends heavily on the efficiency of underlying operations, such as matrix multiplication and convolutions. Enhancements in compilers and hardware acceleration have been instrumental in scaling vision models to handle large and complex datasets, making them more powerful and practical for real-world applications. An example of a CNN is presented below, adapted from [5]. Like the five layer NN described above, the forwards and backwards passes function with the same mechanics. This network contains special convolutional nodes just after the input layer. The convolutional layers are generally the computational bottleneck of the forwards and backwards passes.




**Convolutional Neural Network**

## 2.3 Compilers

A compiler is a computer program that functions effectively as a language translator. In its most usual form, a compiler translates code written in a high-level language, such as Python, into a set of machine instructions that can be understood by a CPU or a GPU. The time required for this process is referred to as compilation time. Compilers come in several flavors, the most common two being Ahead-of-Time (AOT) and Just-in-Time (JIT). In both types of compilation, extensive code optimization is possible.

### 2.3.1 AOT compilation

Ahead-of-Time (AOT) compilers translate code before the program is executed. This process happens during the build or compile time, producing a binary executable that can be run on the target machine. The primary advantage of AOT is rapid execution time. Once a program has been compiled, the code being executed can run directly on the target platform, whether it be a CPU or a GPU, whenever the user desires. However, AOT compilers are known to face challenges with dynamic language features and runtime optimizations that require knowledge of the program's behavior during execution. These limitations can restrict the flexibility and adaptability of the compiled code compared to code produced from Just-in-Time (JIT) compilation.

### 2.3.2 JIT compilation

Just-in-Time (JIT) compilers compile code at runtime, translating high-level code into machine code immediately before the program is executed on the target platform. This approach allows the compiler to optimize the code based on runtime information. This can lead to significant performance improvements for programs that benefit from dynamic optimizations. JIT compilation provides greater flexibility, allowing for optimizations that adapt to the program's execution patterns, such as in-lining, frequently called functions or optimizing hot paths. However, JIT compilers introduce additional overhead during execution, which can lead to slower startup times and less predictable performance,

especially for short-lived applications. Despite these trade-offs, JIT compilation is particularly well-suited for environments where code needs to be highly dynamic, as in Python environments. Specifically, CNNs stand to benefit heavily from JIT compilation. This is due to optimizations such as convolutional fusion wherein the primary performance bottleneck of vision models is improved. This optimization can only be maximized with runtime information or a statically defined model, the latter being very unpythonic and not preferred by Python developers.

## 2.4 Pytorch

PyTorch [6] is an open-source deep learning framework developed by Facebook's AI Research lab (FAIR). Introduced in 2016, PyTorch has become one of the most popular tools for building and training neural networks due to its ease of use and flexibility. The framework is designed to provide a seamless path from research prototyping to production deployment, featuring a dynamic computation graph that allows developers to change training and inference behavior through a variety of tools. These tools include JIT and AOT compilers, custom backends, automatic differentiation, and a library of relevant linear algebra implementations.

One of the standout features of PyTorch is its intuitive API, which readily integrates with the Python programming language and its vast ecosystem of libraries. PyTorch supports distributed training, enabling the scaling of model training across multiple GPUs

and nodes. Additionally, PyTorch's strong community support and documentation facilitate collaboration and rapid development. The framework also offers interoperability with other popular machine learning tools and libraries, such as NumPy [7] and TensorFlow [8], further enhancing its versatility and ease of adoption for both academic research and industry applications.

State of the art machine learning models can be executed in just a few lines of code:

```
1  import torch
2  import torchvision.models as models
3  input_data = torch.rand(1, 3, 224, 224) #random input data
4  model = models.resnet50()
5  output = model(input_data) #execute the forward pass (inference)
```

**Figure 2.2:** ResNet50 inference using PyTorch in Python

## 2.5 Compilers in Python

Compilers in modern Python machine learning frameworks fall into one of two categories: *eager mode* or *graph mode*. Eager mode frameworks are noted for their imperative define-by-run methodology [9], where models are represented, like all other language features, in code which is interpreted each run. On the other hand, graph mode frameworks use a define-and-run [9], where a graph is constructed prior to execution and called each time the model is invoked. Eager mode frameworks benefit from being easier to understand and debug. Unfortunately, eager mode frameworks suffer from a lack of operation level optimization. Optimizations such as operator fusion and scheduling are not possible with basic eager mode frameworks. Advancements in eager mode frameworks [10, 11] have shown promise, but at the cost of flexibility and ease of programming, a cornerstone of Python and a deal-breaker for many developers. The Pytorch compiler is an extension to PyTorch which aims to harness the flexibility of basic eager mode frameworks with the optimizations of graph mode frameworks.

### 2.5.1 The PyTorch compiler

The Pytorch compile is engaged simply by wrapping any Python function with `torch.compile()`, which incorporates two technologies, TorchDynamo and TorchInductor.

### Usage

The function signature of the torch compiler is:

```
def compile(model: Optional[Callable] = None, *,
        fullgraph: builtins.bool = False,
        dynamic: Optional[builtins.bool] = None,
        backend: Union[str, Callable] = "inductor",
        mode: Union[str, None] = None,
        options: Optional[Dict[str, Union[str, builtins.int,
           builtins.bool]]] = None,
        disable: builtins.bool = False) -> Callable:
```

**Model** The function you wish to compile. This variable is optional due to the presence of a function decorator mode, i.e. `@torch.compile(...)`

**Fullgraph** This flag disallows graph breaks or unsupported features. If enabled, it will raise error alerts when it recognizes functions that cannot be captured in a single FX graph.

**Dynamic** This flag enables dynamic shape tracing. This is a feature that generates maximally dynamic kernels, a feature that precludes recompilation. Recompilation may occur when input sizes change, as is common in matrix multiplication for example.

**Backend** As described above, TorchDynamo requires backends to produce code from captured FX graphs. The default 'inductor' is designed to strike a balance between performance and overhead.

**Mode** Modes of compilation are described below:

| Backend | Description |
|---|---|
| Default | The default mode, aimed to be a good balance between performance and overhead. |
| Reduce-overhead | Caches the workspace memory required for the invocation so that it is not reallocated in subsequent runs. This flag is ignored if the CUDA graphs generated mutate input. |
| Max-autotune | Leverages Triton based matrix multiplications and convolutions It enables CUDA graphs by default. |
| Max-autotune-no-cudagraphs | Same as above, without CUDA graphs. |

**Table 2.1:** Compilation modes

**Options** Various other options, available only through invoking *torch._inductor.list_options*() or reading the source code found in [12].

**Disable** This feature disables all internal operations during testing.

Most simply, it can be invoked as:

```
    import torch
    def foo(a,b):
        return a+b
    foo_compiled = torch.compile(foo)
    result = foo_compiled(5,6) #result will be an int of value 11
```

### TorchDynamo

TorchDynamo is a Python bytecode to Python bytecode JIT compiler. The details are described precisely in Ansel et al. [13], but the basic concept is shown in Figure 2.3, from the original paper. The figure compares the execution pipeline of a function `foo()` in default Python and under compilation. The key concepts and terms are described in Table 2.2.

The default Python behavior consists of a function `foo()` being converted two Python bytecode and then executed by the interpreter. In Figure 2.3, we see the bytecode representation as `PyCodeObject`. The interpreter, `_PyEval_EvalFrameDefault()` uses this bytecode, and stack frame information stored in the `PyFrameObject` to execute the code.

Code compiled by TorchDynamo has a similar execution pipeline. The function `foo()` is converted into a `PyFrameObject` and a `PyCodeObject`. Instead of immediately executing the bytecode using the frame data, TorchDynamo compiles the `PyCodeObject`. As we saw earlier, TorchDynamo is a bytecode to bytecode compiler. In this compilation, dynamic analysis is performed to extract FX graphs. This FX graph is then be passed into the backend of the compiler, TorchInductor by default, which is described in the following section. This backend performs graph level optimizations on the captured FX graphs. Finally, these changes are patched into the original `PyCodeObject` and the FX graphs are cached in working memory, as they may be called again. As these optimizations are happening during execution, there is a chance that behavior of the function could change due to a new control flow branch being taken. To ensure correct results, each function is protected by a `guard`. These guards verify

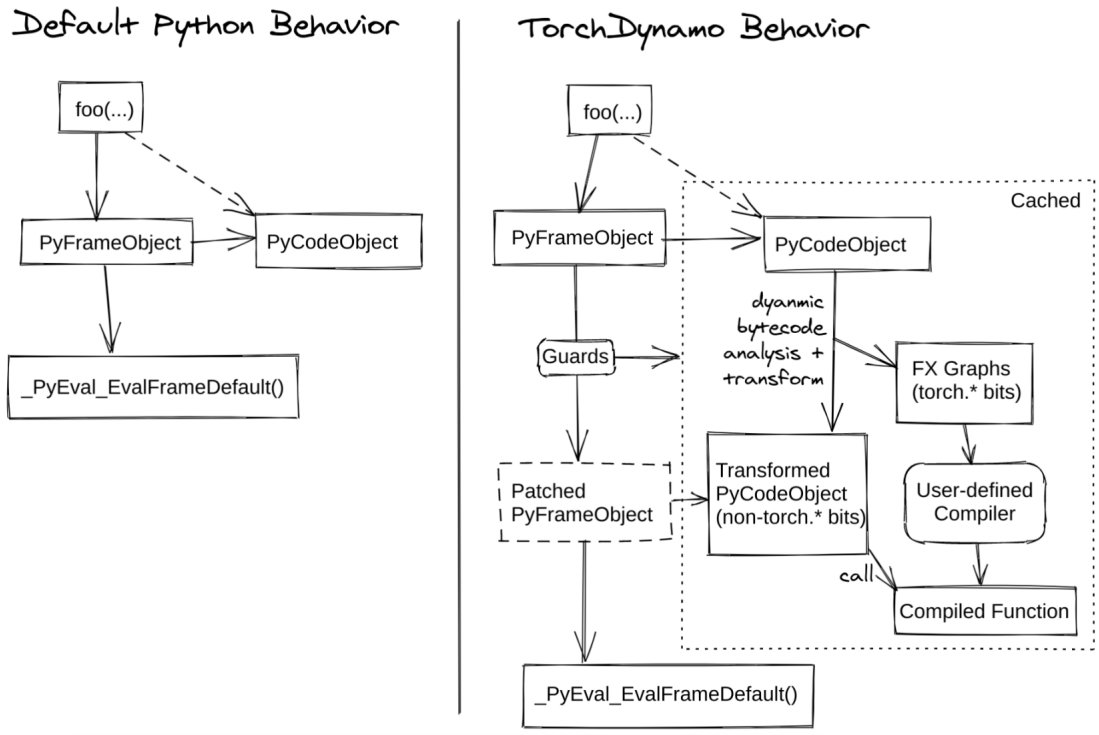



**Figure 2.3:** TorchDynamo overview of the execution of function foo(). Default Python behavior on the left, and torch.compile(foo) on the right. All compilation takes place on the CPU, although Triton operations are dispatched to the GPU during execution.

the previously applied optimizations are still valid given the current execution path. The now patched `PyCodeObject` can now be interpreted by the `_PyEval_EvalFrameDefault()`, which makes calls to cached functions or artifacts.

### TorchInductor

After the Python code has been captured by TorchDynamo and transformed into an FX graph, a backend is needed to apply optimizations and generate fast code. TorchInductor is

the primary backend to the torch compiler. More detailed information about TorchInductor is provided by Ansel et al. [13], but it is effectively designed on four principles, PyTorch Native, Python First, Breadth First, and Reuse State-Of-The-Art Languages.

**PyTorch Native** The principle of remaining faithful to the design choices of PyTorch, such as in-place mutation of data and data structures. The designers emphasize that to best represent Pytorch programs in optimized FX graphs, the backend must share as many Pytorch abstractions as possible

**Python First** TorchInductor is written in Python. This choice more easily engages the extensive Pytorch and Deep learning communities, a feature which enables more activism in community driven development.

**Breadth First** TorchInductor is designed to be a general-purpose backend. Users wishing to achieve high performance in a specific domain usually need to write their own backends.

**Reuse State-Of-The-Art Languages** Many high-performance applications are written in domain specific languages (DSLs), such as Triton for the GPU and C++/OpenMP for the CPU. Thus, the TorchInductor designers chose to leverage these DSLs as output languages for many compiled artifacts.

| Term | Definition |
|---|---|
| **foo(...)** | The function being called. |
| **PyFrameObject** | Represents the execution frame for the function call. It includes local variables, the execution stack, and other state information necessary for function execution. |
| **PyCodeObject** | Represents the compiled bytecode for the function. It is the lower-level representation of the function that can be executed by the Python interpreter. |
| **_PyEval_EvalFrameDefault()** | The core function of Python's interpreter, responsible for executing the bytecode contained in a `PyCodeObject`. It manages the interpretation and execution flow of Python code. |
| **Guards** | Conditions or checks that ensure the validity of the transformations applied by TorchDynamo. If the guards are satisfied, the optimized execution proceeds; otherwise, it falls back to the default execution. |
| **Dynamic Bytecode Analysis** | The process performed by TorchDynamo to analyze the bytecode dynamically and transform it for optimization, particularly targeting PyTorch operations. |
| **Transformed PyCodeObject** | The bytecode after it has been transformed by TorchDynamo, excluding PyTorch-specific operations. These non-PyTorch operations are modified for more efficient execution. |
| **FX Graphs** | A symbolic representation of the PyTorch operations in the function. FX graphs [14] are used for further analysis, transformation, and optimization by a user-defined compiler. |
| **User-defined Compiler** | A compiler or backend defined by the user that takes FX graphs (containing PyTorch operations) and further optimizes or compiles them for specific execution backends, such as GPUs or other specialized hardware. |
| **Compiled Function** | The final output of the TorchDynamo process. This function is a compiled version of the original function, incorporating all the optimizations, and can be executed directly for improved performance. |
| **Patched PyFrameObject** | The original `PyFrameObject` that has been modified to include the transformed code. This patched frame can then be executed by the Python interpreter. |

**Table 2.2:** Definitions of terms in Figure 2.3

# Chapter 3

# Related Work

This chapter presents relevant research to this thesis. Initially, section 3.1 will introduce how the field of machine learning has taken shape in the realm of Python. Then, section 3.2 will discuss the innovation to AOT and JIT compilation that have advanced in Python.

## 3.1 ML in Python

As of 2023, Python was by far the most popular language for ML development [15]. In Python there exists several popular frameworks for programming in the ML space. PyTorch [6] and Tensorflow [8] have emerged as the two most popular general-purpose frameworks [16]. Since 2017, the Pytorch framework in Python has grown in complexity and popularity [16]. The field has seen a growing demand for high performance machine learning frameworks with the growing size of datasets and the increased use of accelerators

in ML training. This has led to all popular frameworks to integrate accelerators, GPUs, TPUs, FPGAs, to parallelize and improve training and inference performance. The designers and maintainers of Pytorch have been responsive to these demands. The current state of compilers in Python will be discussed in section 3.2.

## 3.2 Compilers in Python

Compilation from interpreted Python to an accelerator DSL is the most natural way to achieve high performance computation from Python. As discussed in section 2.5, ML compilers in Python are either *eager-mode* or *graph-mode*. Compilers of the latter mode are better suited to high performance computing, as they enable a declarative style of model construction in a DSL, followed by repeated model execution on an accelerator. Unfortunately, the flexibility of Python and PyTorch does not translate well to fixed graph constructions. Several attempts have been made to bridge this deficiency in translation, described below.

### 3.2.1 TorchScript

TorchScript [10] is a framework that allows for the serialization and optimization of models from PyTorch code into a standalone graph which can be executed on an accelerator. TorchScript contains two technologies which can be used to accelerate Python code: Torch JIT trace and Torch JIT script.

#### Torch JIT trace

Torch JIT Trace is an approach where the PyTorch model is first run once with sample inputs to record the operations executed. Subsequently, this trace is then converted into a TorchScript graph that can be optimized and run on accelerators. While tracing captures the dynamic nature of Python, it has limitations with control flow operations like loops and conditionals, which can lead to incomplete or incorrect traces if the sample inputs do not cover all execution paths.

#### Torch JIT script

Torch JIT Script allows for explicitly defining a model in a subset of Python that is statically analyzable. This approach ensures that the entire model, including control flow operations, is accurately captured in a TorchScript graph. However, it requires developers to write their models in a more constrained, static manner, sacrificing some of the flexibility and expressiveness of standard Python and Pytorch.

### 3.2.2 Lazy tensors

Lazy tensors [11], initially designed for Google TPUs, is a graph capturing mechanism in C++. Lazy tensors delay computation, accumulating iterations a graph using the XLA compiler [17]. This approach allows for optimization opportunities by analyzing the entire computation graph before execution. However, lazy tensors incur a lot of superfluous data

movement. Serialization and execution occur at different points, and both require prior loading of the graph into the CPU. In practice, lazy tensors often incur noticeable overheads due to recompilation as soon as guards fail. The benefits of lazy tensors, including improved scheduling optimization due to delayed execution, have been included in a backend of the Pytorch compiler studied in this thesis, in the form of a hybrid eager/lazy backend to TorchDynamo.

### 3.2.3 Torch FX symbolic trace

Torch symbolic trace [14] is a technique whereby an FX graph is constructed symbolically when the model is defined, rather than during execution. It uses similar tracing methods to that which was described in section 3.2.1 above, except at the Python bytecode level, as opposed to the Pytorch/C++ level. This approach manages dynamic control flow and is more robust to changes in the model's execution paths. It aims to combine the best aspects of tracing and scripting but can still be unsound and can produce incorrect results, see the example below from [13].

```
1    def example(x):
2        return torch.rand(10) + x
```

which when run through the `torch.fx.symbolic_trace()` produces a graph equivalent to:

```
1    def example_incorrect_capture(x):
2        return _tensor_constant0 + x
```

The call to `torch.rand()` has been removed and replaced with `_tensor_constant0`. In the

original code, each call to `example()` would result in a unique result due to the call to the pseudo-random number generator (PRNG) `torch.rand()`. The replacement of a PRNG with a constant is incorrect behavior.

### 3.2.4 JAX

JAX [18] is a machine learning framework developed by Google. It combines NumPy-like syntax with automatic differentiation and GPU/TPU acceleration. It allows for composable function transformations, including just-in-time (JIT) compilation, automatic vectorization, and automatic differentiation. JAX's approach is well-suited for high performance computing, but it requires users to adopt its constrained programming style. Just as TorchScript in subsection 3.2.1, JAX does not capture data dependent control flow.

## 3.3 Summary

In this section we have seen an overview of the state of high-performance deep learning. We have seen the various methods for improving performance of existing machine learning code. These include both eager and graph mode compilers. These compilers make concessions based on ease of use and performance. We can see the niche that TorchDynamo fulfills, and how it strives to minimize these concessions.

# Chapter 4

# Methodology

The goal of thesis thesis is to examine the performance of the PyTorch compiler on vision models. To this end, the building blocks of these models, matrix multiplication and convolutions, are first examined. After this examination, theoretical and practical techniques to optimize performance under compilation are proposed and tested.

The examination of runtime is not as simple as running a stopwatch alongside a program. Although this timing method, described in detail in section 4.2.3, is the most important for end users, it does not offer any granularity at the operation level. Without this level of timing, bottlenecks and expensive features within models and operations cannot be pinpointed and optimized. Therefore, a timing methodology encompassing multiple levels of granularity is described and rigorously tested.

## 4.1 Examination of core operations

The basic strategy was to divide complex operations into manageable pieces that could be separately interrogated. Vision models, specifically convolutional neural networks, can be separated into convolutions and pooling operations, the former being the computational and data bottleneck. Convolutions can then in turn be broken down into matrix multiplications. All three of these operations will be examined in detail in increasing complexity, with the goal of understanding vision models in their entirety.

### 4.1.1 Matrix multiplication

Matrix multiplication is the most fundamental operation of vision models. Experiments concerning this operation are sourced from the PyTorch and Numpy implementations. These are the most popular and well optimized Python libraries for these operations. Matrix multiplication is relatively efficient in the context of the operations in this thesis. Thus, a large range of inputs can be explored, beyond sizes that commonly occur in vision models. Square matrices up to size 17,500 are examined with the rapid increased scaling of data sets, these ranges may become normal.

### 4.1.2 Convolutions

Convolutions, the core operations of vision models, are in practice a series of matrix multiplication. Convolutions within vision models appear with a myriad of parameters.

Thus, the experiments are designed to capture the variety of configurations of convolutions found in real world models. For the analysis, the parameters that vary most frequently in real vision models will be tested, include kernel size, stride, padding. These experiments are sourced from the PyTorch neural network library [12].

**Convolutional ablation** For experiments altering convolutional parameters, the kernel size, stride, and padding values are taken from sets [1,3,5], [1,2], and [1,2,3]. These values are representative of convolutions parameters that occur in real vision models.

**Convolution blocks** Vision models rarely contain standalone convolutions. Instead, multiple layers of convolutions are executed sequentially. To more accurately reflect patterns found in full vision models, the compiler is benchmarked on convolutions in blocks. A block is a series of convolutions, defined by size, i.e. the number of convolutions. A simplified implementation of a convolution block is shown below.

```
def convblock(input: Tensor, conv: nn.Conv, size: int):
    for _ in range(size):
        _ = conv(input)
```

**Listing 4.1:** Python convolution block

### 4.1.3 Vision models

The six vision models, AlexNet [19], ResNet50 [20], MobileNetV2 [21], GoogLeNet [22], SqueezeNet1.1 [23], and DenseNet121 [24] are sourced from the Pytorch vision repository [12]. These models were selected to represent the breadth of the vision field.

**AlexNet** AlexNet is simple compared to the other five networks assessed in this study. These other networks are based on AlexNet.

**ResNet** A key contribution to the vision domain, ResNet is the first successful application of residual connections. This approach is selected in order to test the compiler on a relatively simple network that includes a feature, residual connections, that is present in more complex networks.

**MobileNetV2** MobileNet was optimized by Google to be lightweight for mobile and embedded environments. It is selected for study to test the Pytorch compiler on memory focused optimizations, such as bottleneck layers and inverted residual structures.

**GoogLeNet** GoogLeNet is a network that has been optimized for increased depth at a constant compute cost. This network tests the compiler on GPU compute optimizations and the exploitation of the data parallelism of GPUs.

**SqueezeNet** SqueezeNet aims to supplant AlexNet. It includes some convolutional-specific operations mainly the 1x1 squeeze convolutions, which are readily fusible and, as we will see later, optimized by the PyTorch compiler.

**DenseNet** DenseNet focuses on data reuse with its fully connected components. It tests the compiler's memory optimization on hot memory.

## 4.2 Timing

Quantifying timing and benchmarking of operations on the GPU is non-trivial [25]. GPUs are inherently data parallel and present many opportunities for parallel and distributed optimizations. These optimization may be prevented or disrupted by naive timing methods. Careful examination of the effects of timers on these optimizations is presented in Figure 5.7.

Ultimately, the most important metric is end-to-end (E2E) time. That is, the time between the start and end of the operation. Therefore, the important evaluation metric of this thesis is the E2E timing of vision models. However, E2E timing offers no granularity at the operation level. On the other hand, fine-grain timers, those surrounding each core operation in a complex model, may pose issues for optimization and yield misleading timings. Several timing methods are presented and analyzed in this thesis. The evaluation of several different timing methods described below can be found in Figure 5.7, where techniques are compared side-by-side for individual experiments.

The timing methodology for all experiments is provided by either Python's nanosecond timer [26] or the Pytorch profiler [27]. The former is a simple timer that affords practical advantages to understanding the compiler's behavior. The latter is a context manager that provides profiling at the operation level. Any timers used on the GPU, unless otherwise specified, as in Figure 5.7, are executed with synchronization to ensure correctness. That is, all timed operations are instructed to complete before the next one can begin.

### 4.2.1 Matrix multiplication timing

E2E timers are used exclusively for the simplest operations. As described later, many of the optimizations discussed and proposed in this thesis are data-bound. A data movement analysis of each of the operations, such as was done in Smith et al. [28] for matrix multiplication, could be relevant to minimizing the overhead demonstrated in Figure 5.10.

### 4.2.2 Convolution timing

Like matrix multiplication, convolutions in this thesis are not timed at a granularity finer than a full kernel pass, i.e. an E2E timing of the equation shown in Figure 2.1. The key optimization for convolutional blocks, as described in section 4.1.2, is fusion. Therefore, these blocks are timed with E2E timers without the presence of internal, fine-grain timers.

### 4.2.3 Vision model timing

A single run consists of a single forward pass through a network, also known as inference. The four timing methods used will be E2E, layer timing, layer subtraction, and profiler data.

**End to end** In E2E timing, a timer is started before the call to the forward pass of a model and terminated upon its `return`. This is the most relevant benchmark for vision practitioners, although it is of the coarsest granularity.

**Layer timing** Layer timing consists of placing timers on either side of each layer of a network. These times may be summed, resulting in a metric referred to as *Arr_Sum*. This metric provides finer granularity but may interfere with inter-layer optimization.

**Layer subtraction** This timing method is used to capture data movement and operations between layers, while estimating the duration of each layer. Its use may best leverage the granularity of layer timing with the noninterference of E2E. It is performed by running an E2E timer and halting the execution of the network at desired layers. The procedure used to obtain the time for layer N is as follows: the network is timed until layer N-1 is complete, then the execution is halted and the time recorded. The network is then rerun until layer N is complete, and, as before, execution is halted and the time recorded. The difference between the latter and the former yields an estimation of the duration of layer N.

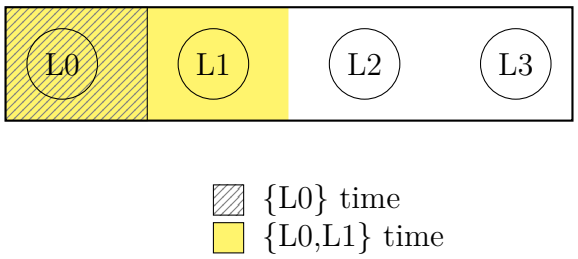



**Figure 4.1:** Layer subtraction of a four layer (L0-L3) network. Estimated execution time of L1 is {L0,L1} - {L0}

**Profiler data** Profiler data is the most fine-grained available method for assessing performance. It is performed by running the operation in question inside of a context manager, which allows every operation to be traced. Tracing profilers, which are different from sampling profiles, allow finer grained details to be observed, albeit at the cost of greater overhead due to program interrupts and data logging. The PyTorch profiler used the Intel VTune [29] tracer under the hood.

Profiler data comes in three types: *CUDA_total*, *CUDA_self*, and *CPU_self*. These values are presented for each operation which appears in the context manager. *CUDA_total* represents the total time spent on GPU operations, including both kernel execution and memory operations. *CUDA_Self* measures the time spent on GPU operations excluding time spent on nested operations. *CPU_Self* indicates the time spent on CPU operations, excluding the time consumed by any child operations or functions. These values may be summed for a particular run to provide insight into the total time spent in a model's forward pass.

| Timing Method | Description |
|---|---|
| **End-to-end (E2E)** | A timer on either side of the operation or model call |
| **Layer Timing (arr_sum)** | Placing timers on either side of each layer in a network. |
| **Profiler Data** | CUDA total, CUDA self, and CPU self. |
| **Layer Subtraction** | Running an E2E timer, halting execution at desired layers, and recording the time. The difference between two successive timings provides an estimate of the duration of the specified layer. |

**Table 4.1:** Summary of Timing Methods

## 4.3 The oracle

After measurements are taken, an oracle, or a theoretic optimal runtime of a model, is constructed. The basic idea is the selective compilation of layers. Compiling only layers in which speedups were observed. The details are discussed below, with the implementation of this theory described in subsection 4.3.2.

### 4.3.1 Construction of the theoretical oracles

#### Layer timed oracle

Construction of an oracle from a layer timed run is done with both naive layer timers and layer subtraction. This construction is done by timing two configurations of each model, one interpreted and one compiled. For a network of N layers, each run produces an N-sized array of times. The oracle is constructed as follows:

Given two arrays, **a** and **b**, of layer times for a network with $n$ layers, $\mathbf{a} = [a_1, a_2, \ldots, a_n]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_n]$, the oracle run time (ORT) is the sum of the minimum values for each index is given by:

$$ORT = \sum_{i=1}^{n} \min(a_i, b_i) \tag{4.1}$$

**Profile timed oracle**

Construction of an oracle from a profiler-timed run is like that of a layer timed oracle. Two configurations are run, one interpreted and the other compiled. The ORT is taken by summing the operation times for each operation present in the profiler trace.

### 4.3.2 Oracle implementation

The implementation of an efficient configuration is nontrivial. Data movement and data transformation between CUDA and Triton are not free and are not represented in the theoretic runtime as described above. The technique of compiling some layers of a model while leaving others untouched will be called selective compilation. A configuration with selective compilation will be referred to as a *custom* configuration. Two different approaches to custom configurations are considered and implemented.

### Best of two runs

In the best-of-two-runs (BTR) solution, the fastest choice is made for each layer from two unique runs of a model. This avenue is the most direct implementation of the layer timed oracle. This approach considers two runs of a layer-timed model, one interpreted one compiled. For each layer, as in the theoretic layer-timed oracle, the fastest mode is chosen. The forward pass of the model is modified such that layers that performed best under compilation are compiled, and those who performed worse are left interpreted.

### Dynamic programming

The best of two runs technique for oracle implementation does not consider the cost of alternating between CUDA and Triton execution runtimes. To address this, a dynamic programming solution is implemented that optimally decides which layers of a neural network to compile, minimizing the total execution time.

A dynamic programming approach tracks the minimum execution times for both compiled and non-compiled states of each layer. By maintaining an array of execution times and the corresponding paths that lead to these times, an estimation of the optimal configuration of compiled and non-compiled layers is found.

Let $T(i)$ be the shortest time to run up to layer $i$, and let $dp[i][0]$ and $dp[i][1]$ represent the minimum execution times up to layer $i$ when the $i^{th}$ layer is not compiled and compiled, respectively. The recurrence relations are defined as:

$$\mathbf{T(i)} = \min(dp[i][0], dp[i][1])$$

$$\mathbf{dp[i][0]} = \min(dp[i-1][0], dp[i-1][1]) + \text{time\_layer\_not\_compiled}(i)$$

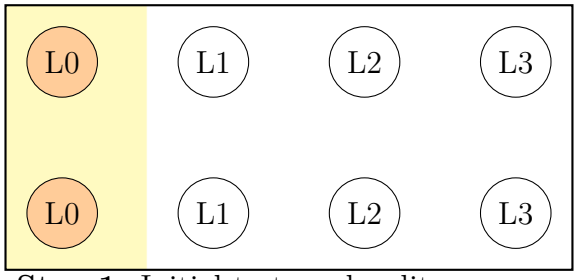
$$\mathbf{dp[i][1]} = \min(dp[i-1][0], dp[i-1][1]) + \text{time\_layer\_compiled}(i)$$

where:

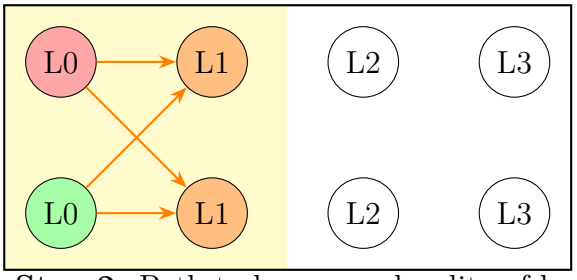
- $T(i)$ is the minimum execution time up to layer $i$.
- $dp[i][0]$ is the minimum execution time up to layer $i$ with the $i^{th}$ layer not compiled.
- $dp[i][1]$ is the minimum execution time up to layer $i$ with the $i^{th}$ layer compiled.
- time_layer_not_compiled($i$) is the execution time of layer $i$ without compilation.
- time_layer_compiled($i$) is the execution time of layer $i$ with compilation.

**Figure 4.2:** Dynamic programming recurrence relation

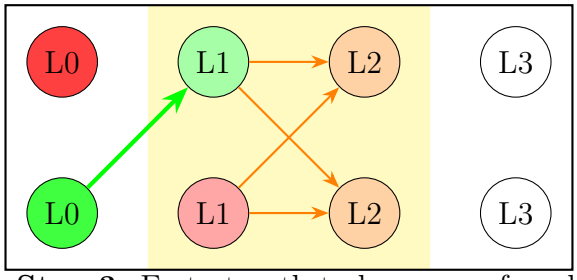
This solution has a locality of one layer. That is, it considers the path to each of the two previous nodes to be optimal, and such that a change to nodes more than one node away from the current node will have no effect. An exhaustive solution to this problem was explored but is intractable. Figure 4.3 on the following page shows walk-through of this algorithm on a four layer network.



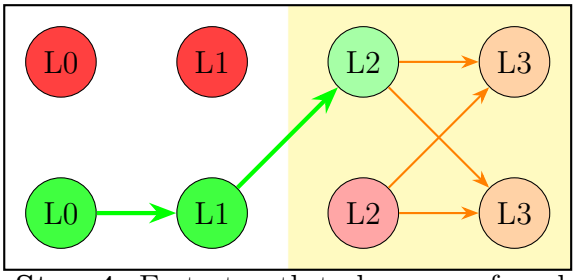

**Step 1:** Initial test, no locality



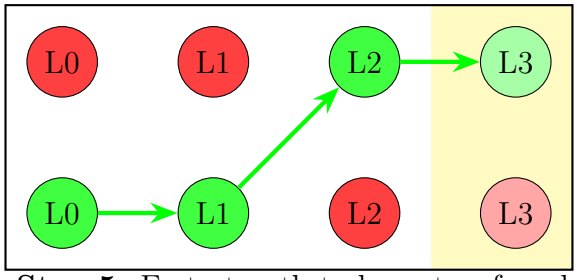

**Step 2:** Path to layer one, locality of layer 0



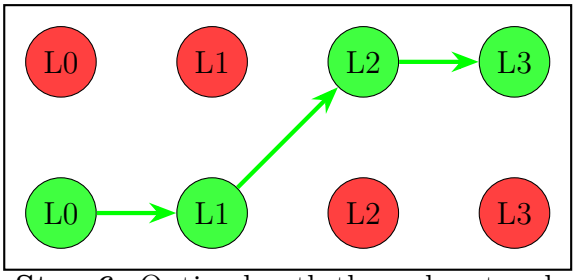

**Step 3:** Fastest path to layer zero found



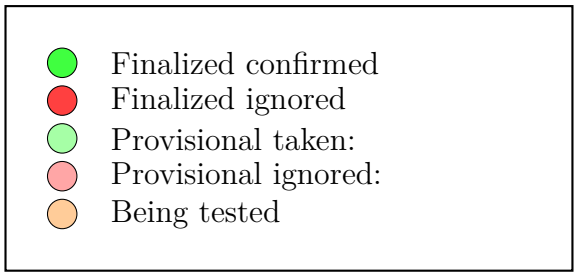

**Step 4:** Fastest path to layer one found




**Step 5:** Fastest path to layer two found.




**Step 6:** Optimal path through network




Note that in **step 3**, *L1* initially performs better when uncompiled. However, in **step 4** we see the fastest path to layer one is by compiling *L1*, minimizing data movement..

**Figure 4.3:** Illustration of the optimal path for a four layer network via the DP algorithm described in section 4.3.2

# Chapter 5

# Evaluation

This chapter describes experiments designed to provide insights into the behavior of the PyTorch compiler. Beginning with basic benchmarks and progressively moving to more complex scenarios that evaluate performance in real-world models. The chapter is divided into three sections: matrix multiplication, convolutions, and vision models.

## 5.1 Research questions

This evaluation aims to answer the following questions:

1. Are the timing techniques described in section 4.2 valid? Do they correctly capture program behavior and can they be used as a performance measurement?

2. What does MM runtime look like under compilation? Are there speedups to be had

at this level of granularity?

3. How do convolutions perform under compilation? Do these results align with those found for MM?

4. How do full vision models perform under compilation? Do these results align with those found for both MM and convolutions?

5. Can runtime be optimized for vision models using selective compilation?

## 5.2 Experiment setup

This section contains the information to reproduce all experiments described in section 5.3, section 5.4, and section 5.5.

### 5.2.1 System configuration

The system CPU is an Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz, with a 51200 KB cache. The GPU is a NVIDIA Tesla V100-SXM2-32GB. Python 3.10.12 is used with Pytorch 2.3.0+cu121. There are no other intensive operations running during experimentation.

### 5.2.2 Experiment conditions

Unless stated, vision model experiments are run inside the PyTorch profiler's context manager. All data values are the medians of multiple runs. Multiple runs consist of twenty

runs, consisting of ten rounds of the target code segment, each executed twice. All timed experiments run on the GPU are synchronized, that is, all scheduled operations are completed and offloaded from the GPU before another run is initiated.

As no backpropagation is involved in any experiments, eval() mode [30] and torch no_grad() mode [31] are used wherever applicable.

Because this study is focused on Pytorch and its compiler, Pytorch libraries are selected over competitors such as Tensorflow [8]. This includes convolutions and vision models.

## 5.3 Matrix multiplication

To begin we focus on evaluating PyTorch performance on the fundamental operation of matrix multiplication (MM), the core operation in most machine learning models [32]. Understanding the compiler's behavior on the simple operation establishes a foundation for assessing behavior in more complex scenarios.

To this end, use the PyTorch implementation of MM. Although NumPy shows similar results to PyTorch on the CPU, it does not support GPU execution and is therefore ignored in this thesis.

These initial tests aim to determine if the compiler increases the operational speed of simple algorithms. Thus, performance is initially tested on an extremely optimized implementation; specifically, the most employed implementation of MM. Figure 5.1 shows the performance of the PyTorch compiler on a simple PyTorch [6] dot product of 2D

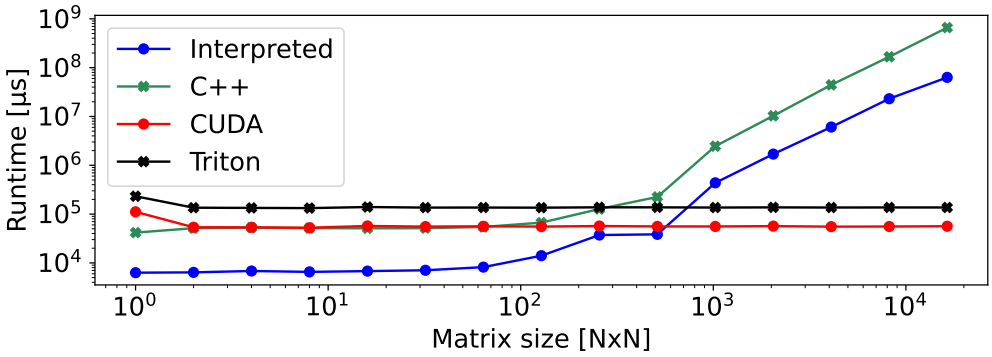



**Figure 5.1:** PyTorch matrix multiplication on 2D matrices. Interpreted and C++ are run on the CPU, CUDA and Triton on the GPU.

arrays, compared against the uncompiled version, on the CPU and GPU. On both devices, the uncompiled version (Interpreted and CUDA) slightly outperforms the compiled version (C++ and Triton). For matrices smaller than $10^3$, execution on the CPU is faster. On the CPU, the conversion to C++ is simply another layer of abstraction to the same library call. On the GPU, the overhead of constructing and launching of a custom Triton kernel is not offset by any optimizations present. Examining the source code of PyTorch [6] and Triton [33], the two different approaches to optimization can be seen. CUDA uses a traditional scalar program with blocked threads model [34], while Triton blocks the algorithm at the program level, using scalar threads [33]. An adaptation of the implementation can be seen below in Figure 5.2. Per the Triton designers, this approach allows for more data movement optimizations. Although the Triton version is slowing on standalone MM, perhaps it will gain traction once data movement patterns become more complicated.

```
        #pragma parallel
        for(int m = 0; m < M; m++)
            #pragma parallel
            for(int n = 0; n < N; n++){
              float acc = 0;
              for(int k = 0; k < K; k++)
                acc += A[m, k] * B[k, n];
              C[m, n] = acc;
            }
```

**(a)** CUDA MM Model

```
        #pragma parallel
        for(int m = 0; m < M; m += MB)
            #pragma parallel
            for(int n = 0; n < N; n += NB){
              float acc[MB, NB] = 0;
              for(int k = 0; k < K; k += KB)
                acc +=  A[m:m+MB, k:k+KB]
                      @ B[k:k+KB, n:n+NB];
              C[m:m+MB, n:n+NB] = acc;
            }
```

**(b)** Triton MM Model

**Figure 5.2:** Comparison of CUDA and Triton MM.

**Takeaways** The constant compilation overhead is expected, as the function body and input type remain the same across different input sizes. If a guard [13] fails for something as simple as a large input size, a single recompilation should suffice. A matrix multiplication may be non-blocked or tiled for certain sizes, which should only incur a partial artifact substitution. However, practitioners leveraging the PyTorch API for matrix multiplication have nothing to gain from compilation and can avoid the compilation overhead altogether.

Now that it has been established that the compiler cannot improve optimized MM, we

extend the analysis by increasing the algorithm's complexity in search of speedup.

## 5.4 Convolutions

The next set of experiments is the next natural step towards full vision models. The direct application of the previous experiments, matrix multiplication, in vision models, is the 2D convolution. The 2D convolution can be broken down into a series of matrix multiplications transformation into the frequency domain via the convolutional theorem [35]. CUDA and Triton libraries both perform this optimization, among many others. A common and extremely beneficial convolutional optimization is fusion [36], which can merge consecutive kernel operations into a single call. The high-performance library BLAS [37], used under the hood of PyTorch, invests significant effort towards optimizing convolutions in this way. These optimizations include a myriad of theoretical optimizations, as described in the documentation [38], as well as architecture-specific, handwritten assembly code. PyTorch defaults to CUDA implementations, while leveraging the PyTorch compiler will yield Triton kernels. It remains to be seen whether the compilation to Triton is fruitful.

Convolutions have been shown to be series of matrix multiplication, an operation which, as seen in Figure 5.1, performed poorly on the CPU. Thus, for this section only convolutional results on the GPU are presented.

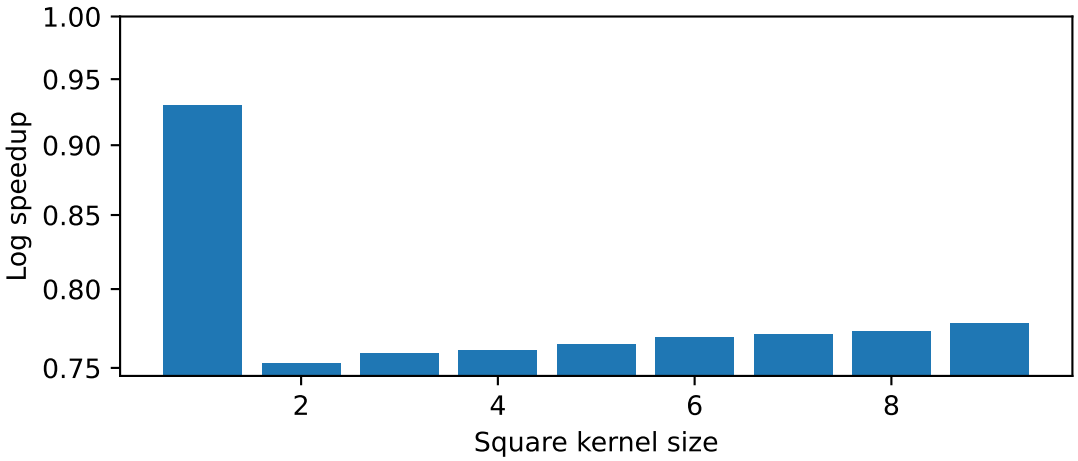



**Figure 5.3:** Single convolution speedup on the GPU, square kernel size vs the log of speedup. Speedup values <1 indicates that compilation slows execution.

### 5.4.1 Different square kernels

Convolutions are found with a variety of parameters in real world vision models. The practical aims of the model designer strongly influence both the shape and order of convolutions. To learn how the compiler performs on a variety of different convolutions, an ablation benchmark is run. Notable results are shown in Figure 5.3, which illustrates an ablation study on kernel size. Here, all convolutions executed in isolation perform worse under compilation. The 1x1 kernel convolution is the only kernel that achieves near CUDA-level performance.

**Takeaways** The overall poor performance of convolutions in Triton compared to CUDA can be attributed to, contrary to the results seen in section 5.3, to unbeatable implementations and underdeveloped Triton optimizations. There have been several

investigations into the subject of Triton's poor convolutional performance [39–41]. However, there is a remarkable drop-off in performance once the kernel size increased beyond one. As we will see in section 5.5, Triton's performance on the 1x1 convolution contributes heavily to the overall runtime of vision models.

NVIDIA's guide on optimizing convolutions [42] has several clues to explain the performance of 1x1 convolutions compared to other kernel sizes. A major reason is memory layout. High dimensional data, like images, are traditionally stored in two orders, NHWC (Number of samples, height, width, channel) or NCHW (Number of samples, channel, height, width). PyTorch tensors use the NCHW format. CUDA prefers the NHWC format. As stated in their convolution guide [42], data given in NCHW format will have to undergo one or more transposes during computation. Triton has a flag which enforces data format throughout computation, which CUDA does not have. Triton is often able to avoid multiple transposes on data.

Secondly, NVIDIA has optimized their implementation for larger kernels. Their guide [42] presents several figures demonstrating the high performance of kernels larger than 1x1. As a general-purpose library, the optimization of large kernels saves more compute in more common cases, such as in graphics rendering and general algorithm acceleration with their general purpose memory optimizations. On the other hand, Triton is a DSL written exclusively for deep learning, where 1x1 convolutions are much more common.

Finally, the fusion of 1x1 kernels is far less computationally expensive than the fusion of larger kernels [36]. Having seen the performance of standalone convolutions, the focus can now be shifted to a less contrived example, convolutions in a real vision model.

### 5.4.2 Convolutions of ResNet

We will look at replications of the convolutions in ResNet. Figure 5.4 presents the convolutions of ResNet timed in two manners, using the PyTorch profiler and E2E timers. The former involves monitoring and recording each convolutional operation under the run of a forward pass of ResNet using the profiler. Multiple data points for each size of convolution are seen due to identical convolutions being run multiple times in an execution of ResNet. The latter involves duplicated each unique convolution of ResNet and running it with an E2E timer. Timers were not inserted into ResNet for this experiment, only a single value is shown for each size convolution, as each type of convolution was run in isolation. As shown, the isolated convolutions, as in Figure 5.3, perform poorly. According to the PyTorch profiler, the convolutions achieved noticeable speedup during the execution of the forward pass.

**Takeaways** These two results are somewhat conflicting. One one hand, this demonstrates that Triton cannot cope with standalone convolutions, whether contrived or ubiquitous. However, it demonstrates that, at least according to the profiler, convolutions being executed in a model can be sped up. Although there is no linear pattern to be found in this relationship, an MLP was trained to predict the speedups of a given convolution,

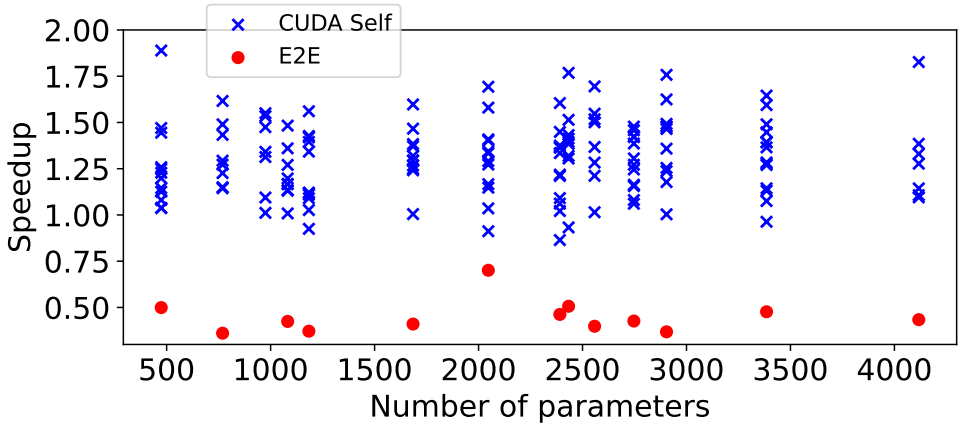



**Figure 5.4:** Speedup achieved under compilation for all of the convolutions in ResNet. Profiler data and E2E timing. There is no correlation between parameters and speedup

within five percent, with 95% accuracy. This indicates that there is a quantifiable relationship Further investigation is warranted into the difference between convolutions executed in isolation and in series, or blocks, as described in section 4.1.2. The next section explores this key difference by evaluated the compiler of convolutional blocks.

### 5.4.3 Convolution blocks

Although Figure 5.3 shows poor results, this scenario is artificial. In real vision models, convolutions are not executed in isolation. Instead, they are chained and executed consecutively. Thus, it is asked whether the compiler's poor performance is limited to single convolutions. Perhaps the compiler gains "traction" and fuse operations once longer and still under-optimized workloads are executed. The complexity of convolutional operations is further explored in Figure 5.5, where multiple convolutions are executed

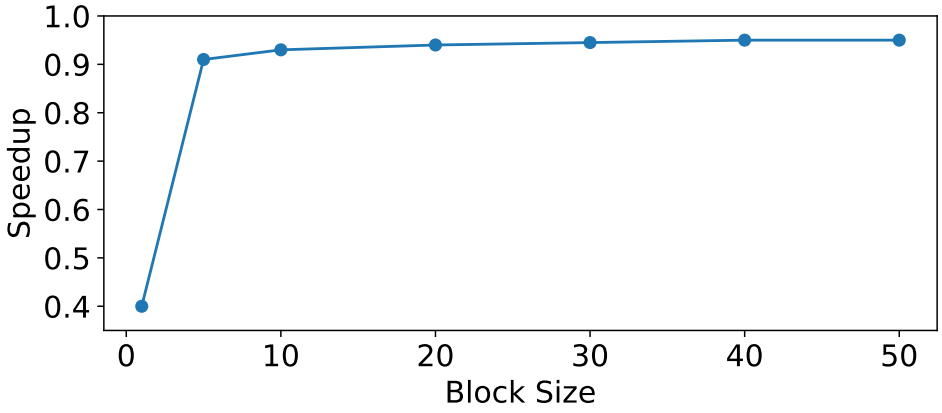



**Figure 5.5:** Speedup achieved under compilation for convolutions executed in blocks. The number of convolutions in each block is the X axis.

sequentially, as they would be in a real-world vision network. It can be seen that performance converges to a speedup around 0.95. These results show that once sufficient convolutions are present, the compiler can optimize to a near-CUDA level of performance.

**Takeaways** The speedup converges around a speedup of 0.95, so practitioners should still avoid the compilation of convolutions, whether in isolation or in series. The strength of Triton and its usage of CUDA graphs is its operation fusion and performance on small, 1x1 kernels. Although an individual convolutional kernel may not be sufficiently tuned to a specific hardware device, convolutions executed in series are fused successfully.

As shown, the compiler does not improve the performance of the standalone convolutions that are tested. However, the compiler's performance improves when a more realistic workload, involving consecutive convolutions, is tested. The foundation is now set for the jump to full vision models. In these cases, Triton's strengths, fusion and small

kernel optimization will be allowed to shine.

## 5.5 Vision models

Full vision models are now tackled, having established the performance of the compiler on their building blocks, convolutions. Six architectures, MobileNetV2 [21], ResNet [20], SqueezeNet [23], AlexNet [19], DenseNet [24], and GoogLeNet [22], are tested from the Pytorch repository [12]. As full vision models are the most complex piece of code analysed in this work, we briefly revisit the CPU in this section when comparing layer performance across devices in section 5.5.3. CPU performance is considered in all optimal configuration construction, as described in section 4.3 and tested in subsection 5.5.4.

### 5.5.1 E2E tests

Initially, the out-of-box performance of the compilation is questioned. If performances like those seen in the previous two sections are observed, there is not much to be said. Figure 5.6 shows the end-to-end results of each of the six models in Triton and CUDA. GoogLeNet, MobileNetV2, and ResNet all exhibit significant speedups with compilation. In contrast, DenseNet and SqueezeNet show minimal improvement, and AlexNet experiences a noticeable slowdown. These experiments demonstrate the potential for substantial speedups when running full vision models under compilation. Further investigation, as we will see in this chapter, is warranted to optimize performance. It is essential to identify and mitigate

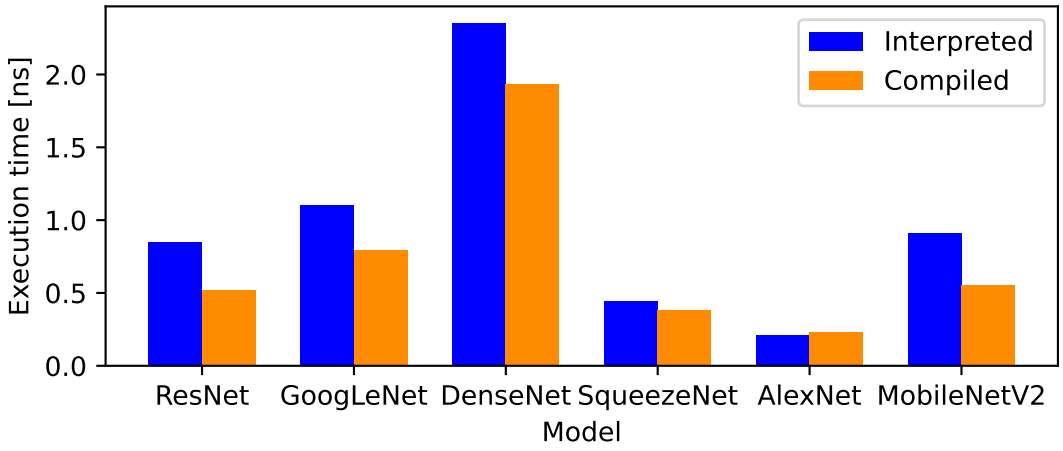



**Figure 5.6:** End-to-end time for the six vision models. In each graph, the CUDA version is on the left, and the Triton on the right.

the factors contributing to slowdowns while preserving the elements which perform well.

**Takeaways** GoogLeNet, MobileNetV2, and ResNet all benefit significantly from out-of-the-box compilation, largely due to their extensive use of dimension-reducing 1x1 convolutions, which perform very well under Triton's optimizations. All three of these networks use these dimension reducing layers combined with dimension expanding layers, as was innovated by ResNet [20]. This is done with a goal of reducing parameters prior to expensive computation, such as 3x3 or 5x5 convolutions, and a restoration of data following expensive operations.

In contrast, SqueezeNet and DenseNet see only marginal benefits. The SqueezeNet designers formatted their architecture in accordance with the Caffe format [23]. Caffe constraints [43] imposed by hardware limitations prevent the implementation of combined

1x1 and 3x3 convolutions in a single layer. This results in extraneous data movement following each of the 1x1 dimension reducing convolutions. SqueezeNet does not see the same quality of speedup as seen in other networks which performed well in Triton.

DenseNet's performance is hindered by its memory-bound architecture, which restricts the compiler's ability to improve efficiency [44]. As we will see later, work [45–48] has been done to mitigate this, but the current version PyTorch version has not resolved this issue.

AlexNet, being an older and smaller model, offers limited opportunities for optimization, resulting in negligible speedups. The computation that exists is not shown to be data bound. Simple compute bound programs are difficult to continue to optimize, as seen with the matrix multiplication experiments in this thesis.

Before moving to more fine grain analyses of the model performance, it is necessary to validate the timing methods described in section 4.2.

### 5.5.2 Timing validation

As discussed in the section 4.2, the various timing methods have potential side effects on the performance, and in general are designed to capture different behaviors. To understand the side effects of these modifications, the models are run with a variety of timers, as shown for AlexNet in Figure 5.7. All runs are performed in either CUDA or Triton, as denoted by the suffix on the Y axis labels. *Pure* refers to unmodified code, as the model will appear out of the box to the user. There are no timers nor synchronization. *Sync* denote runs which contain

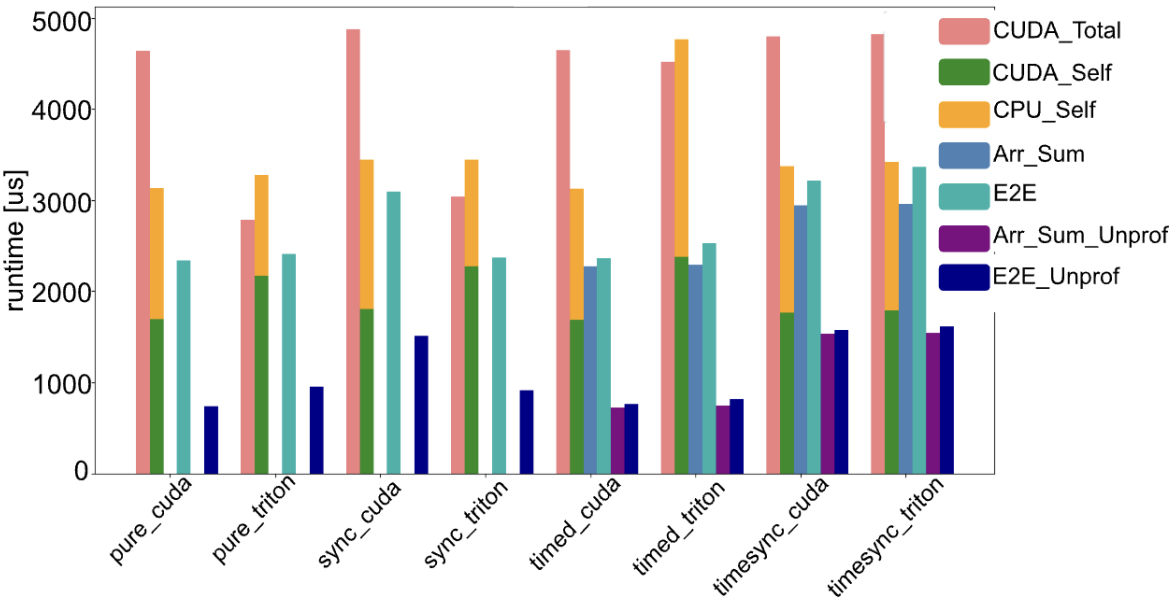



**Figure 5.7:** AlexNet benchmark. Along the X axis, the different configurations of the models are shown, as summarized in Table 4.1. The different bars, as per the legend, describe the timing methods.

only GPU synchronization, to demonstrate the effects of reduced GPU parallelism. *Timed* runs timers on either side of each of the layers of the forward pass, exactly as described by section 4.2.3. It follows that runs with *timed_sync* contain both timers and synchronization. The legend of Figure 5.7 contains the type of data being displayed as described in section 4.2. *Arrsum_Unprof* and *E2E_Unprof* are the same as *Arr_Sum* and E2E from section 4.2, except they are unprofiled, run outside of the context of the Pytorch profiler.

**Takeaways** The most apparent takeaway from Figure 5.7 is the impact of the PyTorch profiler. Experiments conducted within the profiler's context manager experience a twofold

slowdown, as is expected with tracing profilers. However, this reduced performance remains consistent across different configurations, as evidenced by the scaling of unprofiled results. Comparing pure_cuda, pure_triton, and sync_cuda, we observe that profiled E2E times increase in line with unprofiled E2E times. Synchronization and timers have minimal impact independently, as shown by the similar results between pure experiments and those with timing and synchronization. The combination of timing and synchronization (*timed_sync*) results in a notable slowdown compared to timed experiments. In general, CUDA generally suffers more from synchronization than Triton.

Combining the results from Figure 5.4 and Figure 5.7, the choice to abandon the PyTorch profiler in favor of simple timers is made. The lack of correlation between the profiler figures and the wall clock runtime make this an unreliable and deceptive metric for the goals of this thesis. As it has been shown above that timers and synchronization have minimal and consistent effects, all experiments from here, including the oracles described subsection 5.5.4, will be using timers. The previous experiments run with the profiler demonstrate that the profiler overhead is consistent within each experiment, even though the magnitude of the overhead varies depending on program complexity.

### 5.5.3 Layer level timing

Now that it has been shown that timers have little impact on the performance of models, layer level timing can be performed. Initially, an investigation across both the CPU and

GPU is conducted. Additionally, all four compilation modes are revisited for completeness, although no differences are observed in section 5.3 or section 5.4.

### Comparing layers across devices

Just as in matrix multiplication and convolutions, the compilation modes had almost no effect on performance. A visualization of the layer timers under the different modes of compilation for all six models can be found in the Appendix.

Figure 5.11 shows the four different configurations for ResNet in the default compilation mode. By referring to the final graph of overlapping curves, strengths and weaknesses of each configuration can be seen. On the CPU, the presence of green throughout shows the poor performance of ResNet compiled for the CPU. CUDA (Red) performs poorly in the first third of the network but is then roughly equivalent to Triton (Black). The weaknesses in different parts of the models points towards the opportunity of an optimal combination of Interpreted and C++ for the CPU and CUDA and Triton for the GPU.

**Takeaways** For all six models, there exist artifacts such as the ones seen at layer 50, 100, and 175 of Figure 5.11. These artifacts show memory bottlenecks in the CUDA performance early in the model's execution. The presence of red in the final, overlapping, graph shows that these slowdowns are uniquely present in the CUDA version. Previously, the Triton kernels have shown superior ability to fuse similar operations. Now, they are showing their ability to manage the loading of the large, unreduced (inputs are reduced by the 1x1 convolutions as

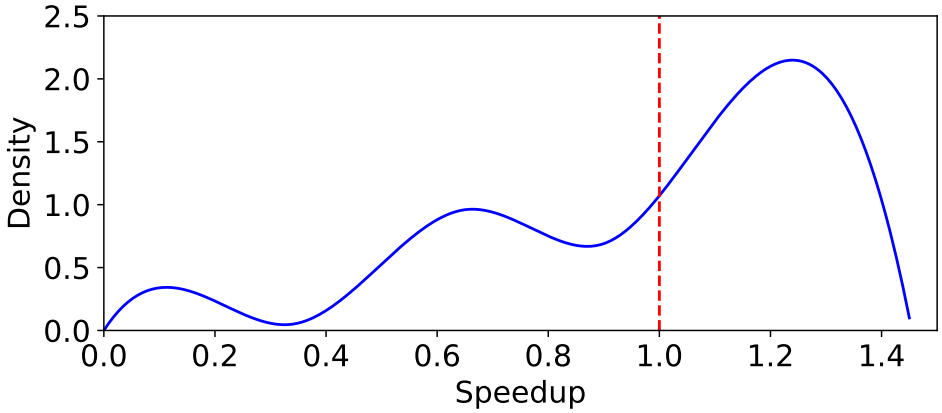



**Figure 5.8:** Frequency (density) plot of GoogLeNet's layers. Layers to the right of 1.0 speedup are accelerated by compilation.

the data flows through the network) inputs onto the GPU more efficiently than CUDA. These artifacts are more significant in the three accelerated models, GoogLeNet, MobileNetV2, and ResNet. AlexNet, the only network significantly slowed by the compiler, shows very minor artifacts of this nature, again due to its simplicity. The attention is now turned deeper to the models, in search of optimal performance. First, it needs to be shown that speedups are possible in each of these models.

### Finding slow layers

We now turn our attention to the problem of finding the types who perform well under compilation, and those which do not. An investigation into which models contained positively affectable layers conducted by observing the density of sped-up layers. The ratio of layers which performed better under compilation is shown in Figure 5.8 for GoogLeNet. The plot

for all six models can be found in the Appendix. The X axis is speedup, the runtime of CUDA divided by the runtime of Triton, and the Y axis is the density. The red dashed vertical line shows a speedup of 1, or unaffected by compilation.

**Takeaways** All models contain layers which fall on either side of the red line, meaning they have both layers which were accelerated by compilation as well those whose performance was reduced. It stands to reason that each of these models could benefit from selective compilation. The next step, having established room for speedup with Figure 5.8, is to find these speedups in practice. A technique for optimizing performance when slow layers are present as described in section 4.3, selective compilation.

### 5.5.4 Implementation of the oracle

Two oracle implementations are described in subsection 4.3.2, a best of two runs solution (BTR), and a DP solution. We first visit the BTR solution. It is the simplest, most agnostic to data movement. Next, the DP solution, designed to account for data movement and fusion optimizations tested.

#### The BTR oracle

Taking these results, it becomes possible to construct an Oracle. This Oracle represents the theoretical optimal runtime of each model. This is done by taking the fastest layers from each of the previous configurations. These theoretic results are shown alongside the practical

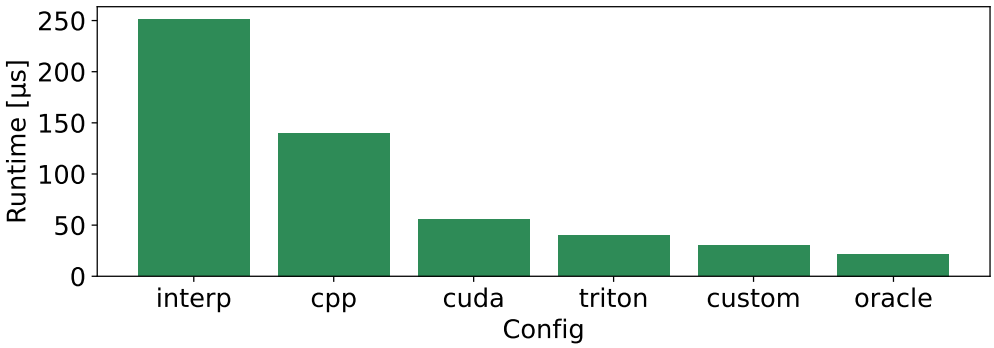



**Figure 5.9:** Layer timed oracle for different configurations of ResNet. Interpreted, C++, CUDA, Triton, and theoretical oracle are shown.

results in Figure 5.9 for ResNet. The remaining five models can be found in the Appendix.

**Takeaways** By construction, the theoretical oracle is always faster than either practical implementation. The theoretical speedups are shown in the Table 5.1 below. This result demonstrates that speedups are possible. Even marginal speedups will amortize greatly over the lifetime of long-running operations. Although an interesting theoretical result, this oracle construct is far from reality, assuming zero data movement and optimal communication and translation between CUDA and Triton. We now move to a much more practical design using layer subtraction, described in section 4.2.3.

### The DP oracle

The implementation of the DP oracle, described as described insection 4.3.2, is tested and evaluated. This timing method, which is not agnostic to data movement and optimizations between layers, is shown in Figure 5.10 for SqueezeNet. Graphs for each of the six networks

| Model | Speedup |
|---|---|
| AlexNet | 1.043 |
| DenseNet | 1.037 |
| GoogleNet | 1.021 |
| MobileNet | 1.045 |
| ResNet | 1.032 |
| SqueezeNet | 1.046 |

**Table 5.1:** Theoretical Model Speedups using the Best of Two Runs oracle.

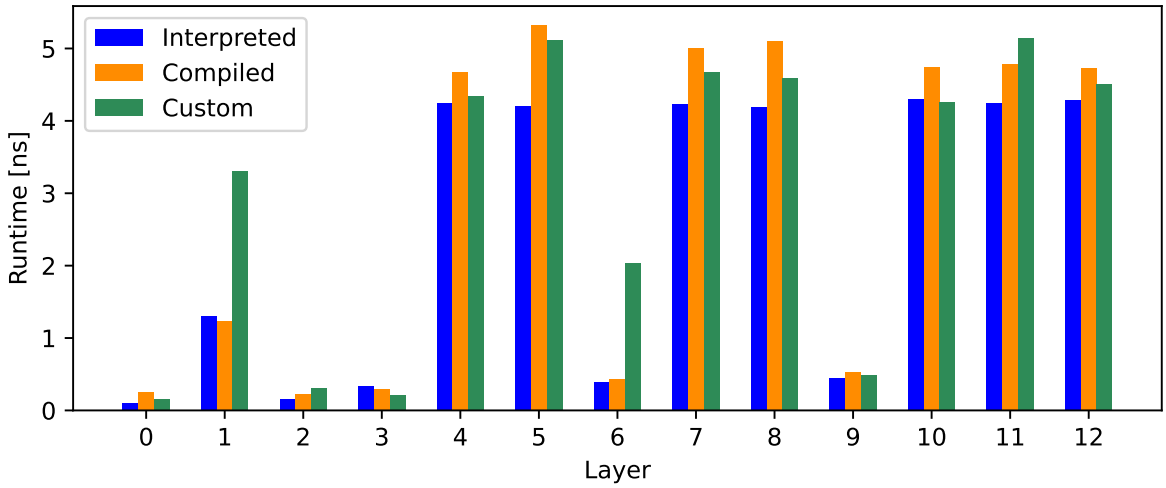



**Figure 5.10:** SqueezeNet layer time by layer subtraction. Interpreted, compiled, and custom configuration compared. Custom is the Layer Subtraction Oracle implementation.

can be found in the Appendix. A custom configuration using dynamic programming, as described in section 4.3.2, is shown in green.

**Takeaways** Overall, the hybrid compilation incurs too much data movement to outperform compilation of the entire model. However, there are many instances of improvements, demonstrating that hybrid compilation has potential.

**SqueezeNet** was a network that showed very marginal speedups in Triton. Although designed to be used in computationally restricted environments, the Caffe format restrictions prevent the implementation of a standalone dimension reducing layers. Consequently, we see the so-called Fire [23] layers with marginal slowdowns from Triton.

**AlexNet** has slowdowns in layers four and nine, with marginal slowdowns in layer seven. All three of these layers are large kernel (5x5, 3x3,3x3 respectively), dimension expanding convolutions. As shown in Figure 5.3, these larger kernels do not perform well in Triton

**DenseNet** has been shown to be heavily data bound across all layers [44], has consistent but not severe slowdowns at each of the denseblocks. Compiler struggles are not apparent in the early convolutional layers, as their runtime pales in comparison to the time spent in the denseblocks. Exploring optimized versions of DenseNet [44] would be a value extension to this work.

**GoogLeNet** performed very well under compilation, seeing a speedup of 1.1 in the compute intensive layers (nine-thirteen). These inception blocks contain, among other things, dimension reducing components. As we have seen in Figure 5.3, these dimension reducing components (1x1 convolutions) perform the best of any configuration.

**MobileNetV2** is packed full of inverted residual layers, with 1x1 convolutions for dimension reductions. Each of these layers performs well under compilation, leading to

significant model speedups with Triton.

**ResNet** was one of the first networks to employ 1x1 convolutions in layers. The simpler architecture lends itself well to being compute bound, resulting in accelerated convolutions benefiting the overall runtime greatly.

Small benefits can be seen from the custom configuration over the compiled version in layers {0,3,4,5,7,8,10,12}. As seen in Figure 5.6, the Triton version performs better than the CUDA version. These experiments further stress the presence of potential optimizations. Layers {0,2,6,7,8,9,11,12} as perform best when uncompiled, and except for the very fast layers {2,3,6,9}, and worst when compiled to CUDA. This means that if data movement can be minimized, many parts of these vision models stand to gain performance.

## 5.6 Summary

This chapter has evaluated the PyTorch compiler on matrix multiplication (MM), convolutions, and full vision models. The compiler shows poor performance on MM and convolutions, but promising results for full vision models.

### 5.6.1 Matrix multiplication

NumPy matrix multiplication are explored in depth. Results are shown on the CPU and GPU with E2E timers. Differences in the MM implementation between CUDA and

Triton are demonstrated. CUDA has been optimized for all round performance and has seen more optimization and influence from state-of-the-art libraries such as BLAS [37]. Triton, as an deep learning DSL, has been optimized for data movement in complex deep learning scenarios. It's advantages are not seen until more complex scenarios

### 5.6.2 Convolutions

Convolutions from the PyTorch library are tested. As a computationally intensive task, only GPU results are presented. Square kernels of size one heavily outperform other shapes and configurations under compilation. Convolutions replicated from ResNet show good performance under the compiler, but poor performance with E2E timers. Under compilation, convolutions executed in series are shown to outperform those executed in isolation.

### 5.6.3 Vision models

Full vision models from PyTorch and the timing methods from section 4.2 are evaluated. E2E tests of models show the compiler can improve performance on some vision models. Before diving deeper, timing methods are evaluated, and the results of the PyTorch profiler [27] are no longer considered. Synchronized timers are shown to be reliable tools for evaluation. An examination into each model shows that all models have layers whose performance is improved and hindered by compilation. This sets the stage for the optimal configurations, as described in section 4.3. Initially, theoretic speedups are

shown in the construction of theoretical oracles, as described in subsection 4.3.1. Practical constructions of the oracle, as described in subsection 4.3.2 are not able to overcome the side effects of data movement and language translation, but some optimal cases are shown for certain layers. All models demonstrated, in the oracle configuration, speedups in certain layers. SqueezeNet has four layers with >1NS saved per layer per forward pass.

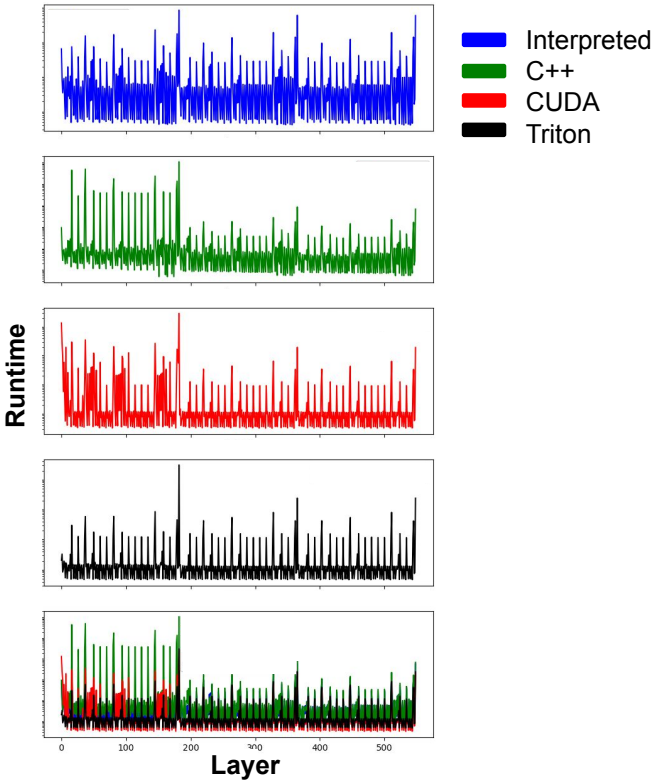



**Figure 5.11:** ResNet layer times, different devices and configurations. Final graph is all curves overlapped.

# Chapter 6

# Conclusion

## 6.1 Summary of contributions

In this work, the behavior of the PyTorch compiler on six vision models and their components was evaluated. Various timing methods, including wall clock timers and profilers, were explored to assess their viability and correctness. The profiler examined was shown to produce consistent results, however with a drastic impact on overall runtime. It was shown that synchronization and inserted timers have minimal and consistent effects on the overall runtime of programs on the GPU. The building blocks of vision models, convolutions and matrix multiplication, were evaluated separately. Standalone matrix multiplication was shown to not perform well with the Triton algorithm. Standalone convolutions performed poorly. However, it was shown that standalone 1x1 convolutions

achieve near CUDA performance in Triton. This result paired with the exceptional ability of the compiler to fuse consecutive operations, as shown in block convolution experiments, explained the performance on vision models. Models that employed 1x1 convolutions as dimension reduction operations just before more expensive convolutions saw dramatic increases in performance in Triton. ResNet, GoogLeNet and MobileNetV2 are the three networks which most heavily employed these 1x1 convolutions. Under Triton out-of-the-box, ResNet saw a 1.61 speedup, GoogLeNet a 1.43, and MobileNetV2 1.63. Under the custom oracle configuration, these speedups were to 1.23, 1.15, and 1.19.

## 6.2 Analysis

### 6.2.1 Choice of Domain

The execution and compilation of large deep learning models are resource-intensive operations. To ensure the feasibility of this study, certain compromises were made. Six vision models were selected to represent the current trends in the field. This selection includes both older models that have defined key optimizations and newer models optimized for various purposes. With additional resources, it would be worthwhile to expand this research to include a broader range of models. Numerous vision models, as well as models from other domains within deep learning, could provide valuable insights into the field of deep learning acceleration.

### 6.2.2 Hardware selection

CUDA frameworks have been hand-tuned to achieve high performance on a wide variety of systems and hardware. Triton is a more specialized DSL, and therefore will not achieve competitive performance on as wide a variety of systems. The choice of hardware was kept constant for this thesis, this may have been unfair to either CUDA or Triton. Providing multiple hardware platforms for each test greatly increases the robustness of a study.

## 6.3 Future Work

### 6.3.1 Choice of Compiler

While PyTorch is a popular and widely used framework, it is not the only one available. Other deep learning frameworks, such as TensorFlow [8], Caffe [43], and several others, also offer unique compiler technologies. A comprehensive exploration of these compilers could yield significant insights, particularly in selective compilation. Investigating the potential synergy between the TensorFlow and PyTorch compilers, for example, might lead to optimal performance results.

### 6.3.2 Implementation of a hybrid compiler

Although selective compilation was explored in this thesis, it was done via the combination of the existing compiler and interpreter. A potentially fruitful avenue of

research would be a lower-level solution to the idea of mixing compilation and interpretation. A lower-level solution would have more control over data movement and transformation, the major pitfall of the techniques explored in this work.

### 6.3.3 Data movement analysis

Many of the advantage of the compilation of vision models are due to the Triton's optimization of data movement between layers. This data movement was not exclusively quantified in this work. The analysis of these same models and building blocks using data movement aware methods could lead to the development of a faster, hybrid compiler.

# Bibliography

[1] R. Klette, *Concise computer vision*, vol. 233. Springer, 2014.

[2] R. Chandra, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[3] S. Linnainmaa, *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors*. PhD thesis, Master's Thesis (in Finnish), Univ. Helsinki, 1970.

[4] L. B. Rall, *Automatic differentiation: Techniques and applications*. Springer, 1981.

[5] TikZ, "Neural networks," 2024.

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[7] C. contributors, "Numpy," 2024.

[8] A. Martín, B. Paul, C. Jianmin, C. Zhifeng, D. Andy, D. Jeffrey, D. Matthieu, G. Sanjay, I. Geoffrey, I. Michael, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

[9] S. Tokui, R. Okuta, T. Akiba, Y. Niitani, T. Ogawa, S. Saito, S. Suzuki, K. Uenishi, B. Vogel, and H. Y. Vincent, "Chainer: A deep learning framework for accelerating the research cycle," 2019.

[10] Z. DeVito, "Torchscript: Optimized execution of pytorch programs," *Retrieved January*, 2022.

[11] A. Suhan, D. Libenzi, A. Zhang, P. Schuh, B. Saeta, J. Y. Sohn, and D. Shabalin, "Lazytensor: combining eager execution with domain-specific compilers," *arXiv preprint arXiv:2102.13267*, 2021.

[12] T. maintainers and contributors, "TorchVision: PyTorch's Computer Vision library," Nov. 2016.

[13] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, *et al.*, "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*, pp. 929–947, 2024.

[14] J. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel, "torch. fx: Practical program capture and transformation for deep learning in python," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 638–651, 2022.

[15] O. Community, "State of machine learning survey results part one," *ODSC*, 2023.

[16] H. He, "The state of machine learning frameworks in 2019," *The Gradient*, 2019.

[17] A. Sabne, "Xla: Compiling machine learning for peak performance," *Google Res*, 2020.

[18] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang, "JAX: composable transformations of Python+NumPy programs," 2018.

[19] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[21] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," 2014.

[23] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size," 2016.

[24] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," 2018.

[25] L. Aceto, D. P. Attard, A. Francalanza, and A. Ingólfsdóttir, "On benchmarking for concurrent runtime verification," in *International Conference on Fundamental Approaches to Software Engineering*, pp. 3–23, Springer International Publishing Cham, 2021.

[26] Python, "Python documentation, time," 2024.

[27] Python, "Python documentation, profiler," 2024.

[28] W. Smith, A. Goldfarb, and C. Ding, "Beyond time complexity: data movement complexity analysis for matrix multiplication," in *Proceedings of the 36th ACM International Conference on Supercomputing*, ICS '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[29] Intel, "Intel® vtune™ profiler release notes and new features," 2024.

[30] PyTorch, "Pytorch module documentation," 2024.

[31] PyTorch, "Pytorch documentation," 2024.

[32] D. Blalock and J. Guttag, "Multiplying matrices without multiplying," 2021.

[33] P. Tillet, H.-T. Kung, and D. Cox, "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*, pp. 10–19, 2019.

[34] M. Auguin and F. Larbey, "Opsila: an advanced simd for numerical analysis and signal processing," in *Microcomputers: developments in industry, business, and education, Ninth EUROMICRO Symposium on Microprocessing and Microprogramming, Madrid, September 13*, vol. 16, pp. 311–318, 1983.

[35] C. D. McGillem and G. R. Cooper, "Continuous and discrete signal and system analysis," *(No Title)*, 1991.

[36] L. Mao, "Neural network 1 x 1 convolution horizontal fusion," 2021.

[37] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, *et al.*, "An updated set of basic linear algebra subprograms (blas)," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[38] C. contributors, "Blas (basic linear algebra subprograms)," 2022.

[39] RocBLAS, "Optimizing triton kernels," 2024.

[40] TritonTeam, "Convolution sample 1," 2022.

[41] TritonTeam, "Convolution sample 2," 2023.

[42] NVIDIA Corporation, *Optimizing Convolutional Layers User's Guide*. NVIDIA Corporation, Feb. 2023. Version DU-09795-001_v001.

[43] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

[44] G. Pleiss, D. Chen, G. Huang, T. Li, L. van der Maaten, and K. Q. Weinberger, "Memory-efficient implementation of densenets," 2017.

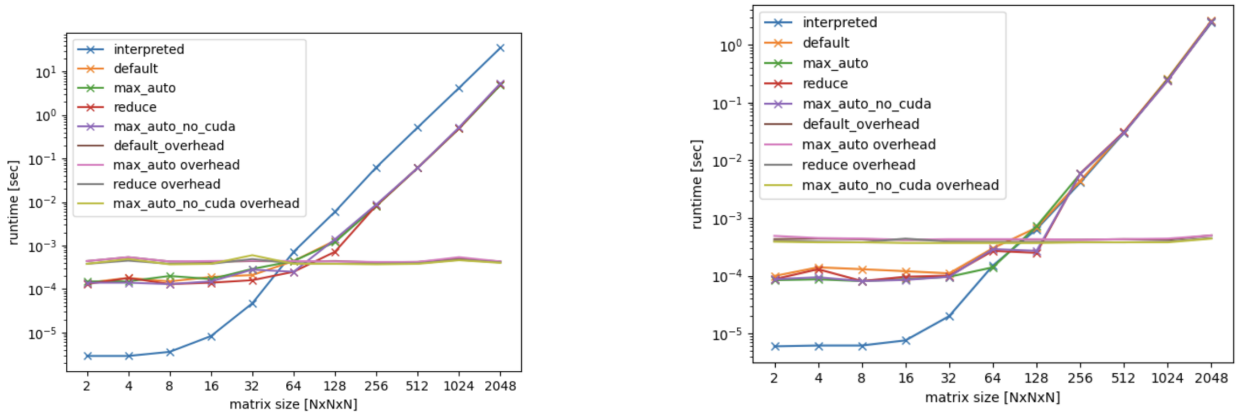
[45] G. Pleiss, D. Chen, G. Huang, T. Li, L. van der Maaten, and K. Q. Weinberger, "Memory-efficient implementation of densenets," *arXiv preprint arXiv:1707.06990*, 2017.

[46] Z. Liu, "Memory efficient implementation of densenets," 2018.

[47] J. Yearsley, "efficient_densenet_tensorflow," 2019.

[48] T. Li, "Densenet space efficient implementation in caffe," 2018.

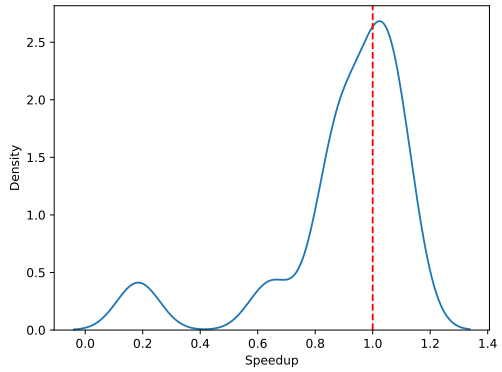
# Appendices



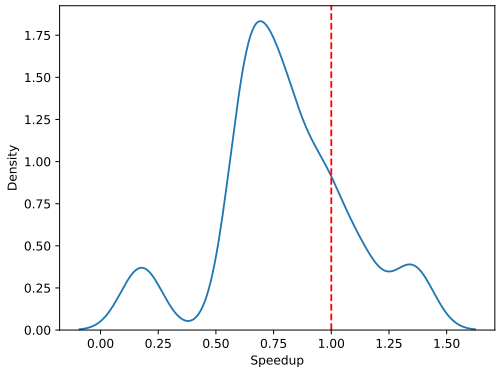

**(a)** NumPy CPU MM

**(b)** PyTorch CPU MM

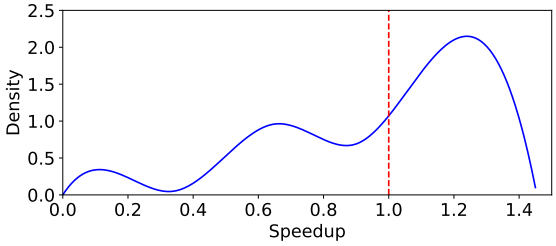
**Figure 1:** CPU multiplication, with compilation time, for four different compile modes in NumPy and PyTorch



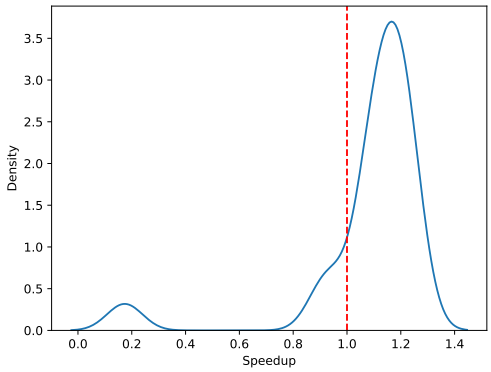

**(a)** AlexNet



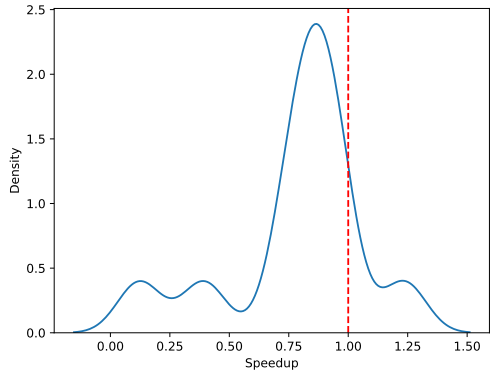

**(b)** DenseNet



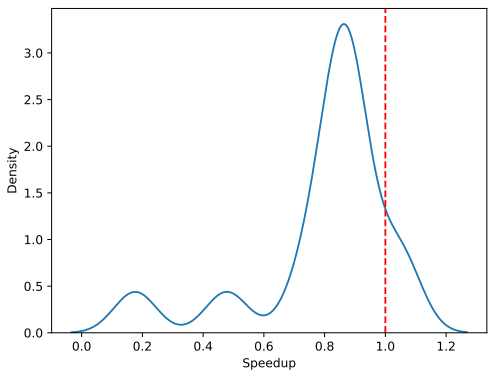

**(c)** GoogLeNet




**(d)** MobileNetV2

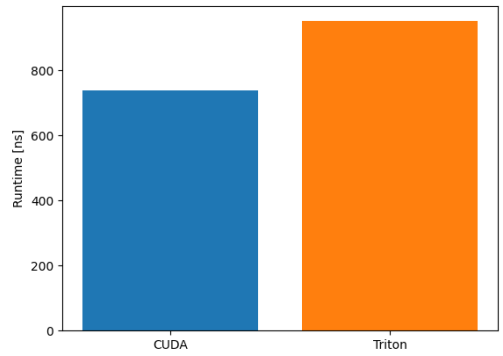



**(e)** ResNet

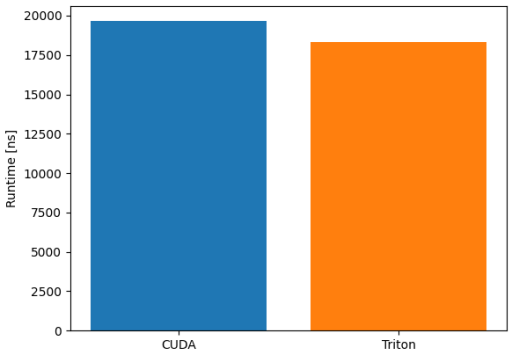



**(f)** SqueezeNet

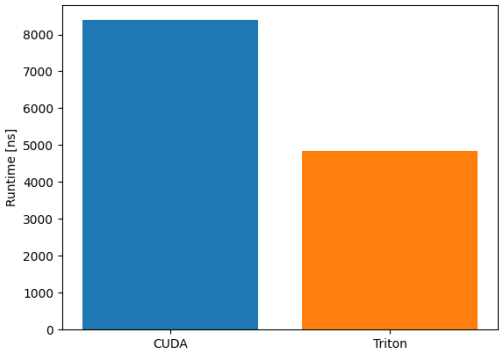
**Figure 2:** Density plots of all models on the GPU, measured with E2E timers



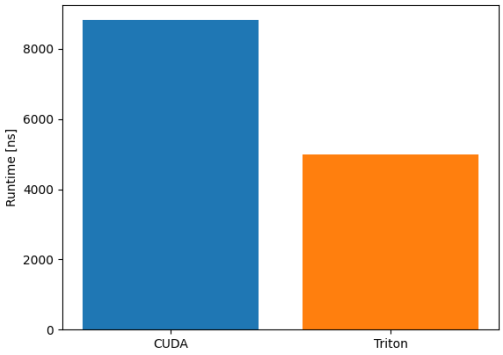

**(a)** AlexNet



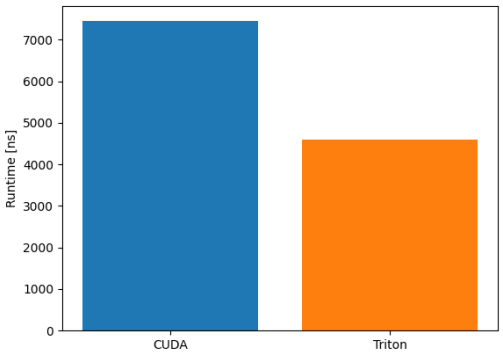

**(b)** DenseNet



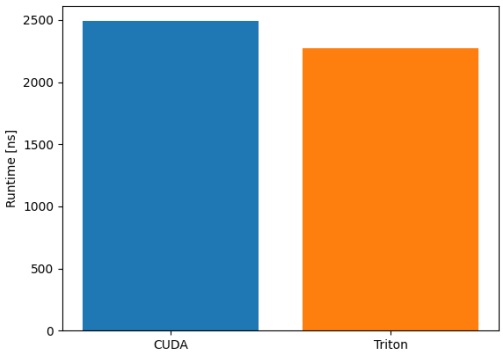

**(c)** GoogLeNet




**(d)** MobileNetV2

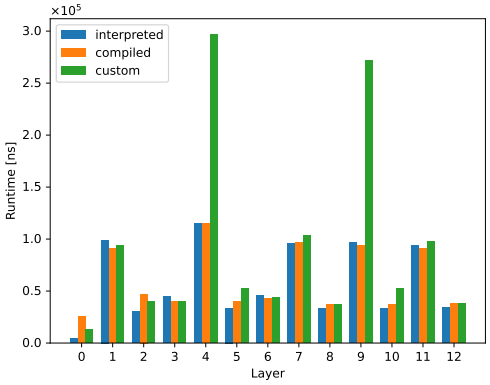



**(e)** ResNet

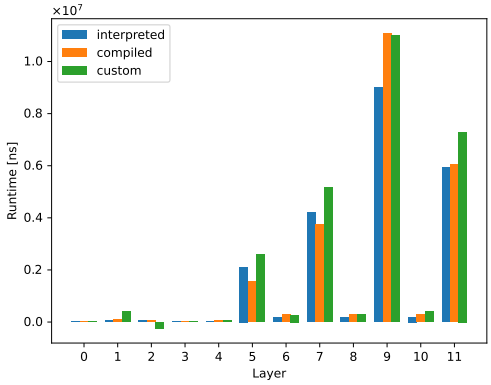



**(f)** SqueezeNet

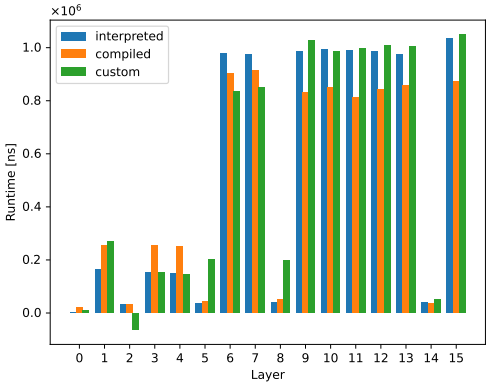
**Figure 3:** E2E runs of all models on the GPU



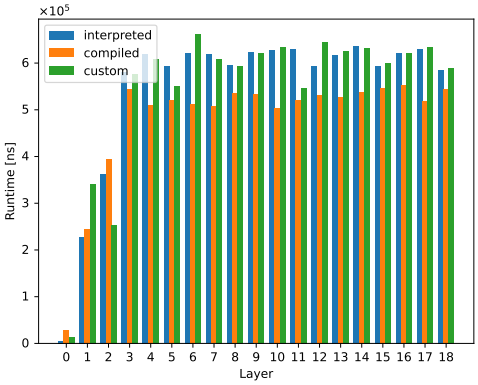

**(a)** AlexNet



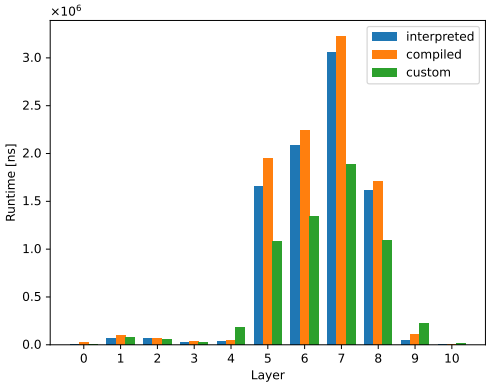

**(b)** DenseNet



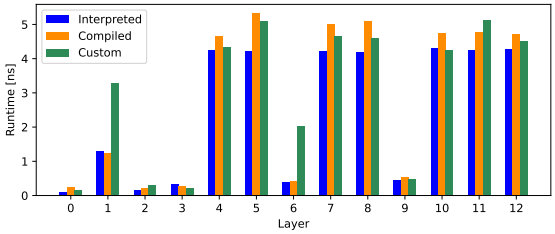

**(c)** GoogLeNet




**(d)** MobileNetV2




**(e)** ResNet




**(f)** SqueezeNet

**Figure 4:** DP oracle runs for each model, on the GPU. Compared to CUDA and Triton.

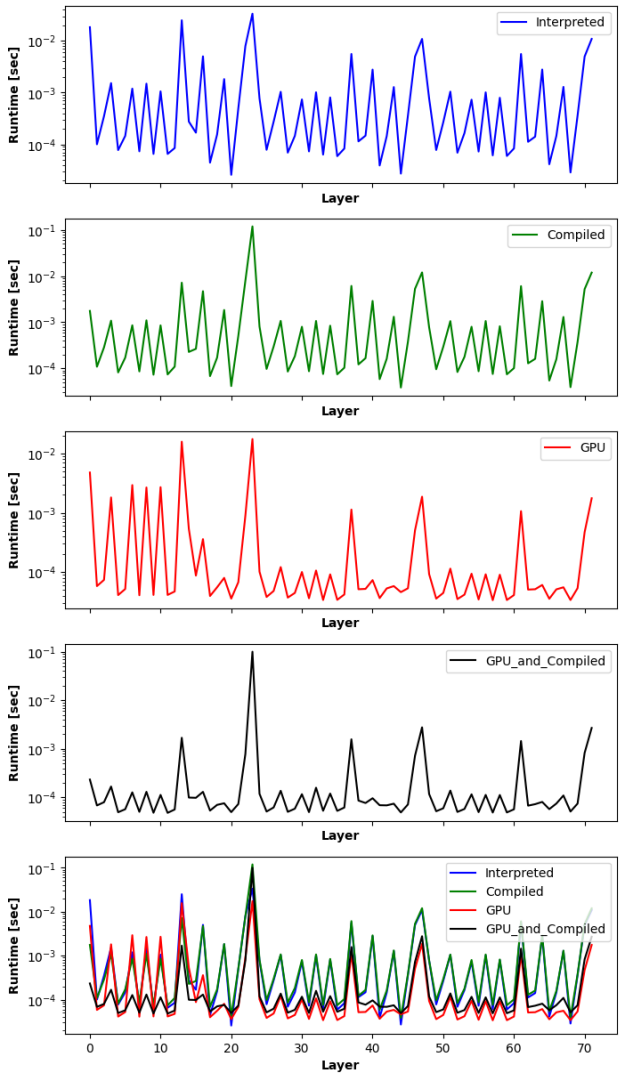



**Figure 5:** AlexNet, default compile mode, layer times with E2E timers.

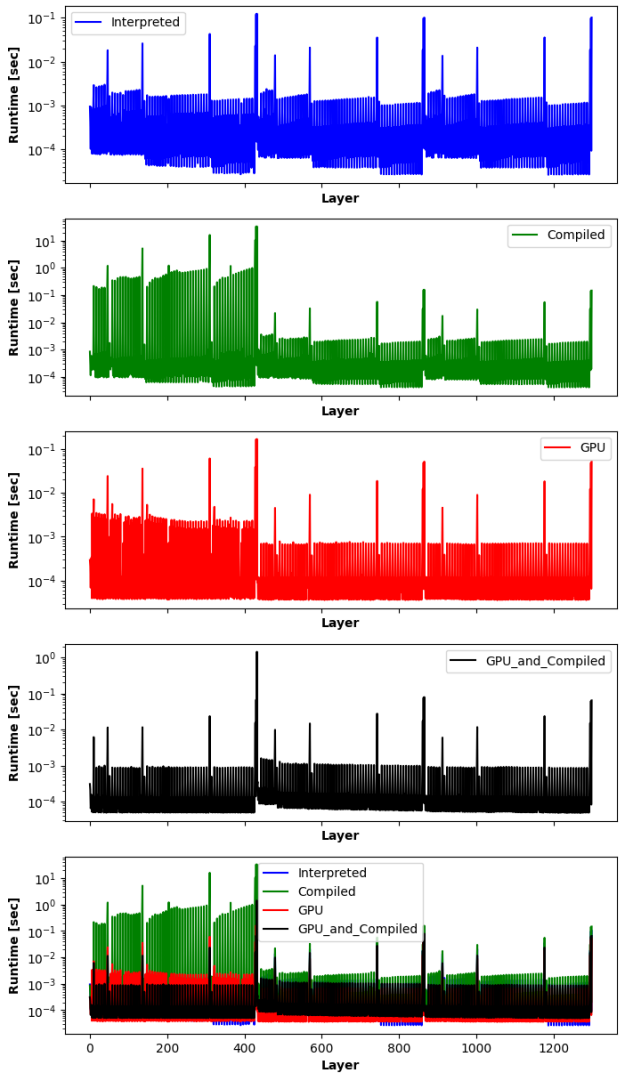



**Figure 6:** DenseNet, default compile mode, layer times with E2E timers.

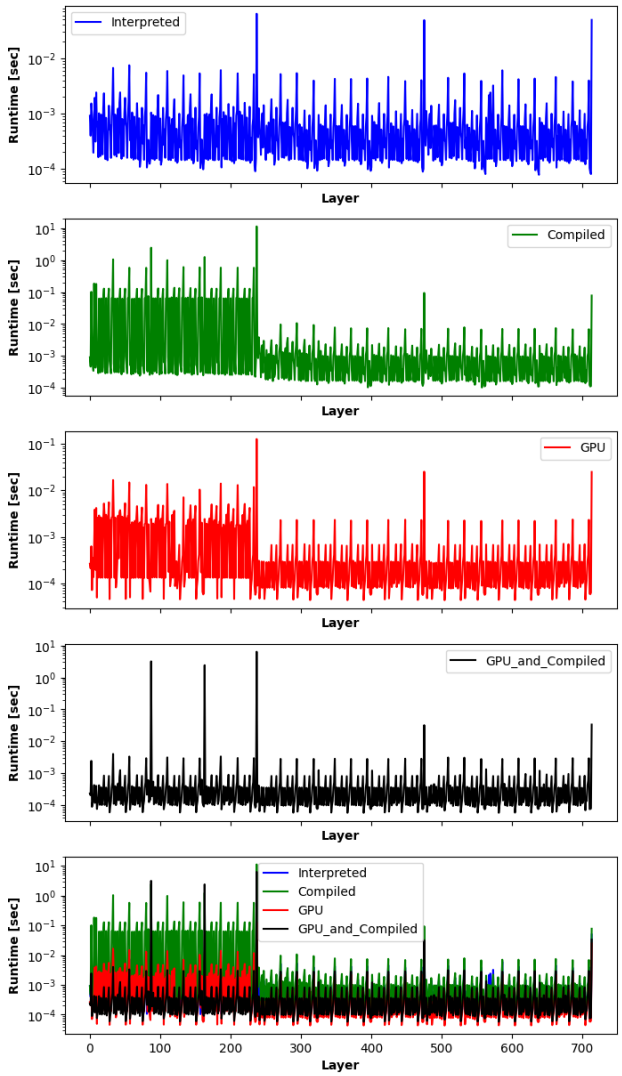



**Figure 7:** GoogLeNet, default compile mode, layer times with E2E timers.

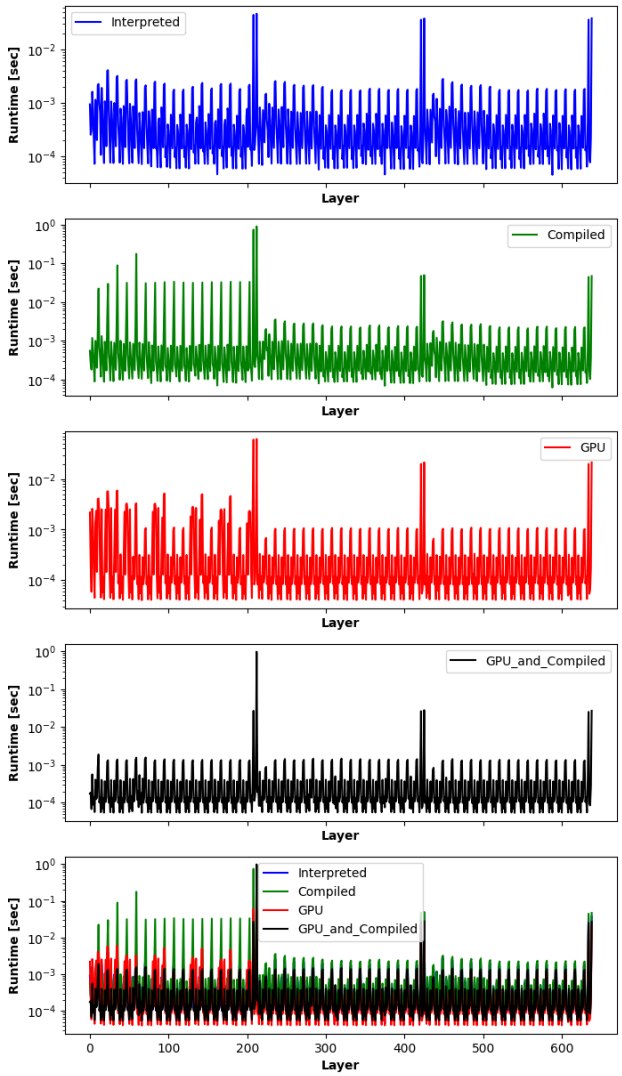



**Figure 8:** MobileNetV2, default compile mode, layer times with E2E timers.

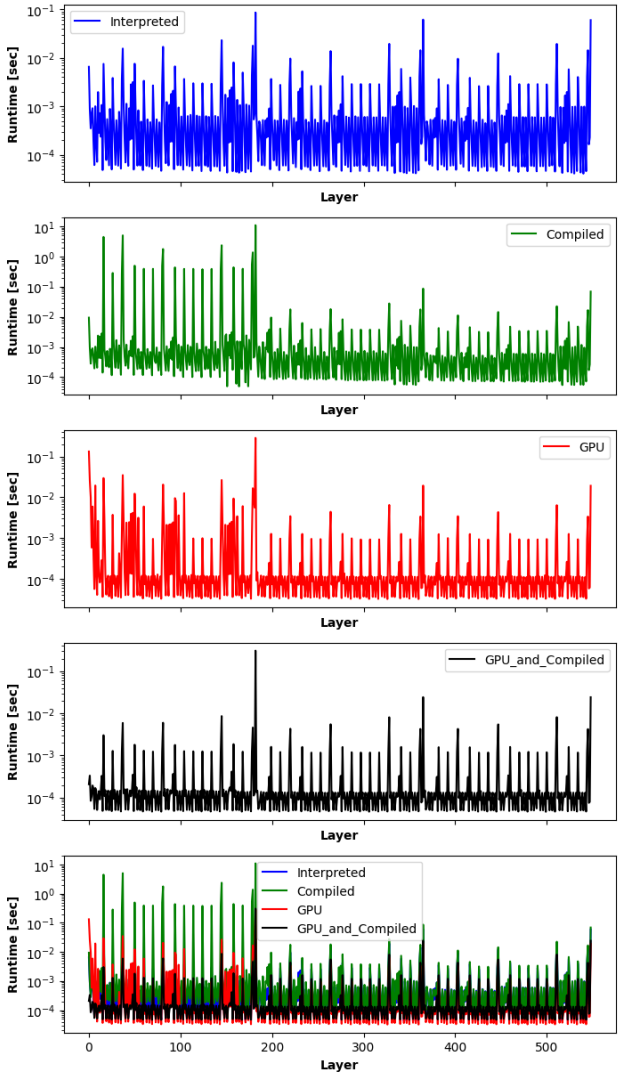



**Figure 9:** ResNet, default compile mode, layer times with E2E timers.

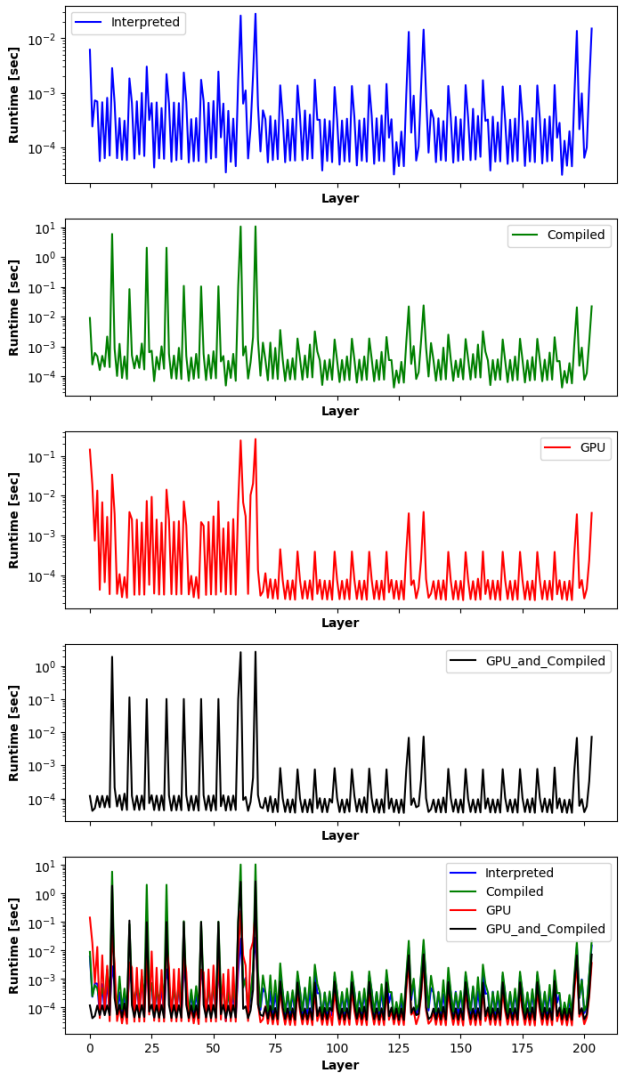



**Figure 10:** SqueezeNet, default compile mode, layer times with E2E timers.

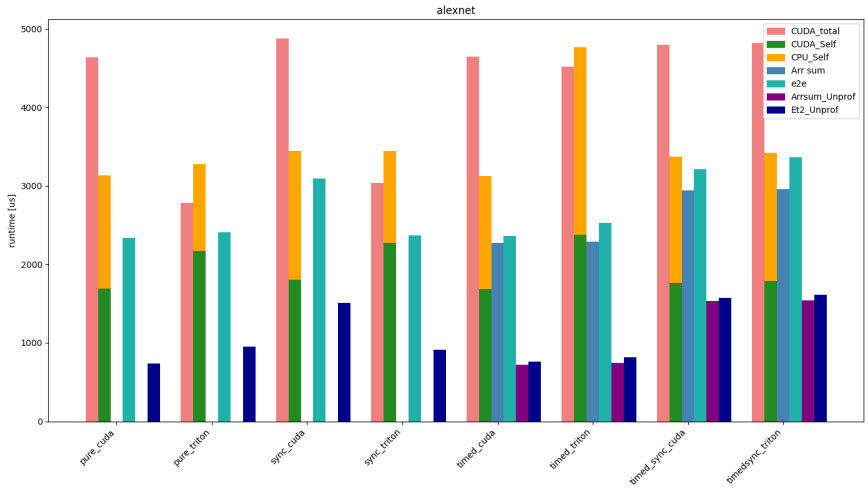



**Figure 11:** AlexNet, timer verification.

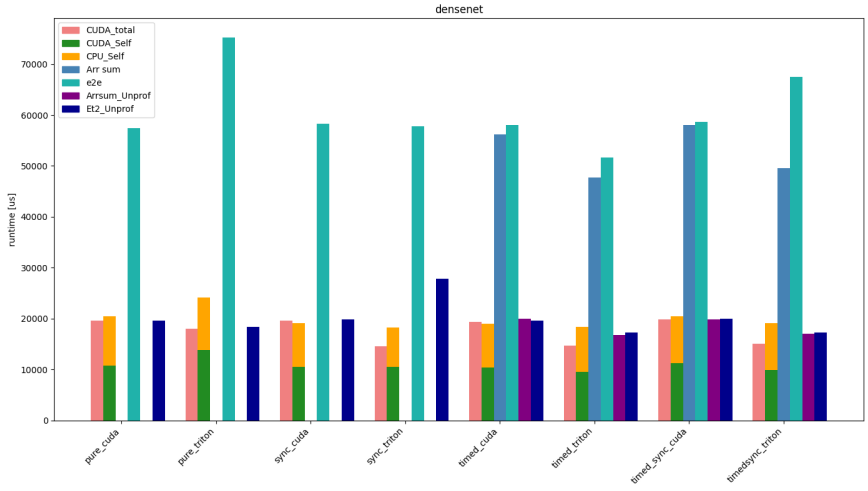



**Figure 12:** DenseNet, timer verification.

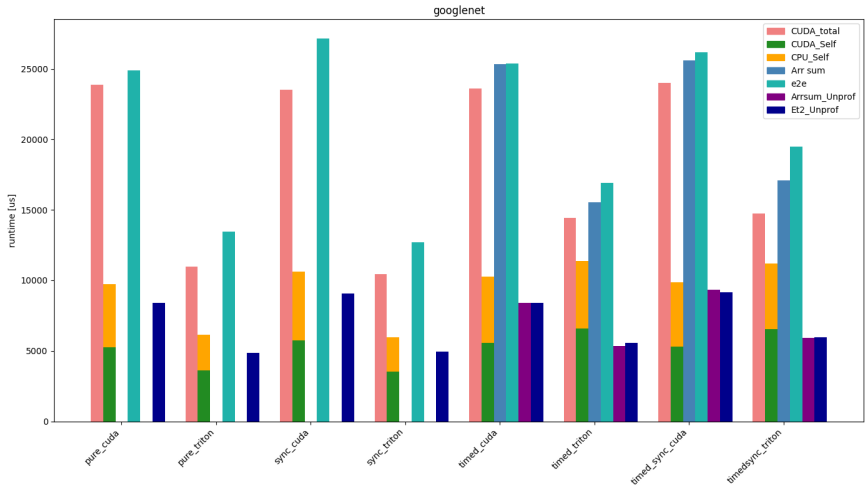



**Figure 13:** GoogLeNet, timer verification.

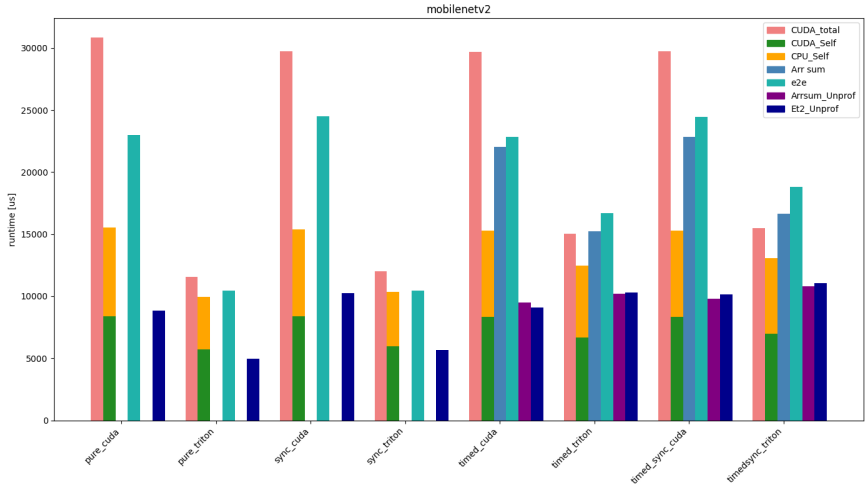



**Figure 14:** MobileNetV2, timer verification.

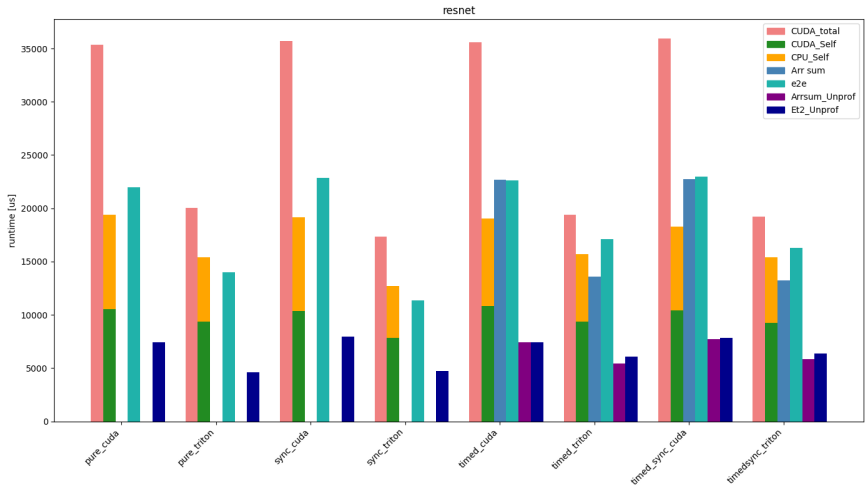



**Figure 15:** ResNet, timer verification.

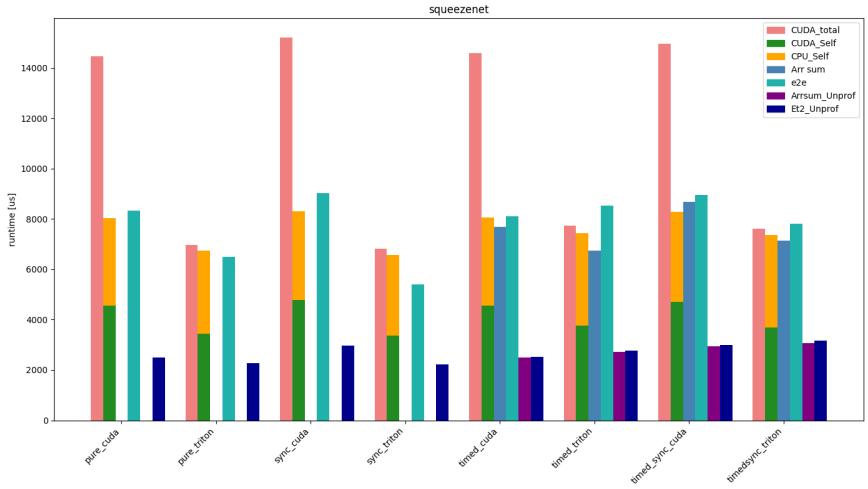



**Figure 16:** SqueezeNet, timer verification.