Characterization of Machine Learning Compilers for LLM inference on NVIDIA GPUs

Alejandro Carmona
Libelium Lab
Gregorio Bernabé
gbernabe@um.es
University of Murcia
José M. García
University of Murcia

Research Article

Keywords: LLMs, Machine Learning Compilation, Inference, NVIDIA, GPU, PyTorch

Posted Date: October 10th, 2025

DOI: https://doi.org/10.21203/rs.3.rs-7652970/v1

Additional Declarations: No competing interests reported.

Characterization of Machine Learning Compilers for LLM inference on NVIDIA GPUs

Alejandro Carmona-Martínez[1,2*], Gregorio Bernabé[1†] and José M. García[1†]

[1*]Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, C. Campus Universitario, 11, Murcia, 30010, Murcia, Spain.
[2]Libelium Lab, C. Luis Buñuel, 6, Murcia, 30562, Murcia, Spain.

*Corresponding author(s). E-mail(s): alejandro.carmonam@um.es;
Contributing authors: gbernabe@um.es; jmgarcia@um.es;
†These authors contributed equally to this work.

Abstract

AI inference is conflicted between Performance, developer Productivity, and device Portability—the P3 problem. Machine Learning Compilers (MLCs) aim to resolve this, but their ecosystem is fragmented, with tools each prioritizing one issue. This paper evaluates deploying trade-offs of PyTorch-based LLMs on NVIDIA GPUs using four intertwined prominent MLC tools: torch.compile, TensorRT, XLA, and ONNX Runtime. A dual methodology is used, leveraging synthetic PyTorch models to isolate optimizations and end-to-end benchmarks with State-of-The-Art (SoTA) models (TinyLlama-1.1B, Llama-2-7B) to measure real-world performance. Findings reveal that Ahead-Of-Time (AOT) compilation's peak performance requires architecture-specific tools like TensorRT-LLM, necessary for SoTA LLMs but unusable for PyTorch models. As for Just-In-Time (JIT) solutions like torch.compile and its backends, they prove flexible and portable, compatible with all tested models but unable to accelerate LLMs consistently, therefore, the choice of MLC depends on P3 considerations and model architecture.

Keywords: LLMs, Machine Learning Compilation, Inference, NVIDIA, GPU, PyTorch

1 Introduction

Large Language Models (LLMs) remarkable capabilities in natural language under- standing, generation, and reasoning are deeply reliant on General Matrix Multiplication (GEMM) as the Transformer architecture used by LLMs implements its two main sub-layers, the Multi-Head Self-Attention mechanism and the Feed-Forward Network (FFN) with GEMMs which account for billions of Floating-Point Operations (FLOPs).

GPUs, with their design for massive parallelism, excel at tasks with high arithmetic intensity like GEMMs due its high throughput (FLOPs/s) making them the de-facto accelerators for LLM and AI inference as a whole. NVIDIA GPUs, in particular, have risen to prominence, capable of performing a vast number of arithmetic operations with their CUDA cores, and offering even more specialized sub-accelerators for AI: the Tensor Cores. Tensor Cores are Application-Specific Integrated Circuits (ASICs) specifically designed to execute matrix multiplications. They offer significantly superior performance compared to the more generalized CUDA Cores and vastly outperform the vector units found in CPUs, specially for the *shorter* datatypes as this allows to increase throughput. Tensor Cores support the traditional float (FP32) through a casting to an internal datatype, TF32, which only utilizes 19 bits, to balance FP32's higher precision with FP16 (half floating precision) faster

performance. BF16 precision takes the compromise further, as it does have the same range as FP32 with less precision (mantissa bits).

Capitalizing on the raw power of this specialized hardware, however, introduces us to the P3 issue. Developers require a *Productive* and *Portable* programming model, a function provided by frameworks like PyTorch, which abstracts away the low-level complexities of CUDA programming through its eager model, using CUDA libraries to implement ML operations. Yet, this abstraction comes at a cost, it leaves significant *Performance* on the table by interpreting operations one by one, failing to see the larger computational graph that could be run as a single program instead of performing multiple library calls with their associated synchronization costs. This tension exposes the core P3 Problem: the fundamental trade-off between *Productivity*, *Portability* (the ease of PyTorch and usability across different devices or multiple GPUs), and the theoretical hardware *Performance*. It is precisely this gap that Machine Learning Compilers (MLCs) aim to bridge by translating high-level, framework-defined ML models into optimized, hardware-specific code (Figure 1).

MLCs transform high-level primitive-based graph representations of neural networks into device-specific executables by tailoring them for specific hardware targets with a variety of optimizations, aiming to exceed the capabilities of the high-level frameworks in exchange for a compilation pass that can be performed either Just-In-Time (JIT) or Ahead-Of-Time (AOT).

The MLC landscape, however, is not a unified solution but a fragmented ecosystem of competing tools. In 2020, Li et al. [1] provided the first survey specifically dissecting the State-of-The-Art (SoTA) Deep Learning compilers. But, since 2020, new players like torch.compile, onnxruntime and TensorRT have appeared and, moreover, LLMs and NVIDA GPUs have taken over as the main model and hardware on the AI landscape.

This work directly addresses the trade-offs specifics to each solution when deploying PyTorch models and SoTA LLMs on NVIDIA GPUs, utilizing the current, and a bit intertwined, SoTA MLC solutions. As front-ends, PyTorch's native torch.compile (JIT) and ONNX Runtime (AOT) and as backends, NVIDIA's proprietary TensorRT (AOT via its own front-end and ONNX and JIT via torch.compile), the general-purpose XLA (via torch.compile).

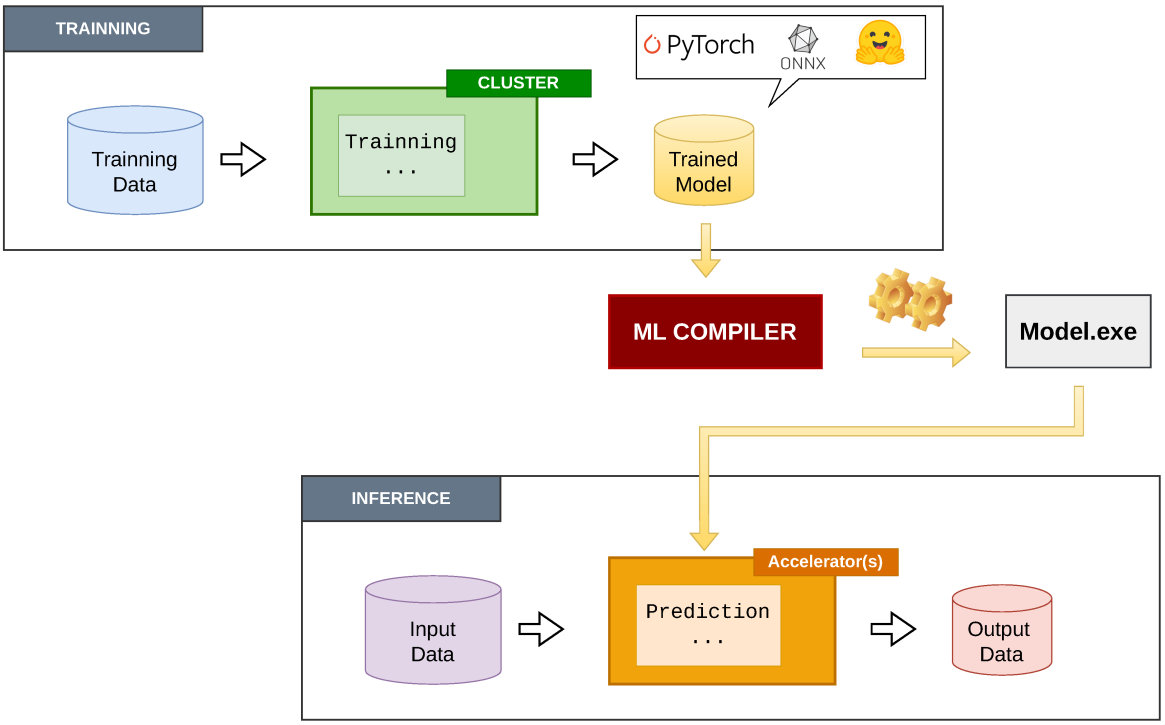

Fig. 1: Machine Learning Compiler in the Training/Inference Pipeline

To this end, the primary objective of this work is to conduct a thorough examination of these MLC workflows, addressing the P3 problem within the context of deploying models with PyTorch on NVIDIA GPUs, systematically evaluating the trade-offs between performance, productivity, and portability for each compiler.

To do so, benchmarked models will include both State-of-the-Art (SoTA) LLMs, `TinyLlama/TinyLlama-1.1B-Chat-v1.0` and `meta-llama/Llama-2-7b-chat-hf`, to assess end-to-end performance and synthetic Feed-Forward Network (FFN)-based models to isolate specific optimizations like GEMM acceleration for the SoTA models.

The scope is on inference optimization for models on NVIDIA hardware using PyTorch, excluding training optimization and model-altering compression techniques such as quantization or pruning. Ultimately, this research aims to synthesize the findings into practical guidelines for developers, helping them navigate and understand the fragmented MLC ecosystem and select the optimal compilation strategy for their own specific requirements.

1.1 Main Contributions

This work makes three core contributions to the understanding and practical deployment of Machine Learning Compilers (MLCs) for PyTorch-based Large Language Model (LLM) inference on NVIDIA GPUs.

1. **Crafting a comparative analysis of four major (and intertwined) MLC tools**—PyTorch's JIT-based `torch.compile`, NVIDIA TensorRT (including TensorRT-LLM), Google's XLA, and the ONNX format with ONNX Runtime—evaluating their Productivity, Portability, and Performance (P3) characteristics. The study quantifies performance gains, compilation overheads, integration complexity, and P3 tradeoffs.
2. **Characterizing MLC benefits through systematic synthetic and SoTA LLM benchmarking.** It presents MLC gains when it comes to reduce GPU synchronization—and the relative impact of precisions when using Tensor Cores. It demonstrates that AOT TensorRT workflows consistently yield the highest throughput for low-precision formats, while multi-target approaches like PyTorch's `torch.compile` prioritize portability at the expense of raw performance, which may yield no speedUp for LLM inference.
3. **Synthesizing general guidelines to help developers to select an MLC** strategy aligned with specific P3 priorities and deployment constraints as there is no universally optimal machine learning compilation workflow; rather, the choice of MLC depends on context: consistent performance gains for any architecture and LLMs in particular (TensorRT/TensorRT-LLM), device-portability (ONNX), or development agility out-of-the-box (torch.compile). This tension defines the P3 trade-off that any team deploying ML models on GPU must navigate.

1.2 Structure of the Document

The rest of the document is as follows: Chapter 2 reviews the state of the art, covering a survey of the compilation frameworks and workflows (torch.compile, TensorRT, XLA, ONNX Runtime and others) to be reviewed. Chapter 3 describes the model selection based on SoTA LLMs as well as synthetic FFN-based PyTorch Models. Chapter 4 presents the performance evaluation for both SoTA and synthetic models and concludes with a discussion about the MLC workflows within the P3 Paradigm and their respective trade-offs. Finally, Chapter 5 recaps the previous contributions and proposes directions for future research.

2 Background & State Of The Art

2.1 JIT: PyTorch's torch.compile

PyTorch 2.0 introduced `torch.compile` as a JIT compilation tool to streamline the process of optimizing PyTorch programs [2] with the primary gain coming from using a single decorator to allow device and code agnostic compilation.

This compilation mechanism operates by first employing its front-end, *TorchDynamo*, to safely capture Python bytecode from model definitions or functions, translating these into a PyTorch-specific Intermediate Representation (IR) known as FX graphs. These graphs subsequently go through a series of optimization passes (operator fusion, algebraic simplification, and memory layout transformations). Finally, a backend compiler, with PyTorch in-house solution, *Inductor*, serving as the default, takes this optimized IR and generates optimized kernels for the specified device, often in C++ or specialized kernel languages like Triton for GPU [2].

Multiple backends extend the core compilation pipeline to diverse hardware targets. `torch.compile` also offers support for TensorRT, TVM and XLA backend compatibility which

enables cross-platform execution on CPUs, GPUs, and TPUs. Although each backend comes with its own set of options, they adhere to a common IR interface, thus facilitating seamless switching.

In case torch.compile cannot optimize an operation or block of operations, it defaults to the PyTorch eager implementation.

2.2 AOT: ONNX & onnxruntime

The Open Neural Network Exchange (ONNX) is a Microsoft-developed open-source format to represent machine learning models in a fully agnostic manner across frameworks, tools, runtimes, and compilers. ONNX defines a common set of operators, which serve as the fundamental building blocks for machine learning and deep learning models, and a standardized file format [3]. This enables developers to train a model in one framework (PyTorch, TensorFlow, ...) and then export it into an ONNX model for inference on a dedicated inference engine, onnxruntime.

onnxruntime works as the inference engine designed to accelerate ONNX models with support for multiple devices [4], as it acts as an interface to integrate various hardware-specific libraries as it may leverage hardware-specific backends for each supported device, denominated Execution Providers (EPs) [4]. When an ONNX model is loaded, ONNX Runtime queries the registered EPs to determine which parts of the model graph each EP can handle. An EP can take over entire graphs, subgraphs or specific nodes, as requested by the end-user, which can submit a priority list for the execution of the model with the preferred order for the EPs, executing them using optimized kernels for the target hardware. Execution Providers include TensorRT, Intel OpenVINO EP, ARM Compute Library or the default CPU EP.

2.3 TensorRT & TensorRT-LLM

NVIDIA TensorRT is a software ecosystem within CUDA to accelerate ML inference on NVIDIA-based devices [5]. As an integral part of the NVIDIA AI platform, TensorRT provides a suite of tools, including an inference optimizer and a high-performance runtime, designed to maximize production ML apps [6]. After a model has been built and trained using a framework such as PyTorch or TensorFlow, TensorRT takes this model as input to create a highly optimized *engine* specifically tailored for the target NVIDIA hardware.

On their 2022 study, Zhou and Yang [7] summarized the key workflows to leverage TensorRT. In our paper, an updated workflow classification also including JIT/AOT considerations is presented.

Two main AOT workflows exist to leverage TensorRT. The first, Torch-TensorRT, is a dedicated library that compiles compatible PyTorch models directly into TensorRT engines. This approach offers a direct, in-framework optimization path. The second AOT workflow leverages the Open Neural Network Exchange (ONNX) format. A PyTorch model is first exported to ONNX, and then ONNX Runtime executes it using the TensorRT Execution Provider (EP) to take advantage of ONNX's interoperability while delegating optimized execution to TensorRT.

For a more flexible approach, a JIT workflow is offered via torch.compile with the `torch_tensorrt` backend. This approach streamlines the process, allowing developers to apply TensorRT optimizations with a single function call, as discussed earlier. This method abstracts away the complexities of engine building, triggering compilation on the first model invocation.

Across these workflows, developers can control key parameters to balance performance and resource usage. Critical customizations include precision control (e.g., enabling FP16 or BF16) to leverage Tensor Cores and workspace size allocation, which provides memory for TensorRT's builder to find optimal kernels [6]. These options allow fine-tuning the compilation process to meet specific deployment constraints.

Distinct from the general-purpose TensorRT, NVIDIA TensorRT-LLM is a specialized tool engineered exclusively for accelerating Large Language Model (LLM) inference [8]. It packages a compiler and runtime armed with LLM-centric optimizations, including custom attention kernels and efficient Key-Value (KV) cache management [9]. It does allow for a set of operands to deploy custom-built LLMs, but does not allow for conversion from PyTorch. However, it facilitates the use of some specific SoTA models as optimized engines via a Python or PyTorch-integrated API. [10].

2.4 OpenXLA

Google's Accelerated Linear Algebra (XLA) is an open-source domain-specific compiler able to accelerate machine learning models from popular frameworks like PyTorch, TensorFlow, and JAX across

an array of hardware platforms [11, 12] whose motivation is to provide a unified compilation target while maximizing performance and portability for ML workloads. However, OpenXLA is not a single compiler, but rather an ecosystem of ML Infrastructure modular components that can be assembled to form an end-to-end stack.

A key architectural aspect is its use of an MLIR-based Intermediate Representation (IR). MLIR is a compiler toolchain [13] that allows common optimizations from its device-agnostic representation with multiple levels of lowering until the machine-specific code is generated, which supports multiple backends. This process begins with a series of high-level, target-independent optimizations on the input graph, which use the StableHLO MLIR dialect. Following this initial stage, a hardware-specific backend performs its own optimizations tailored to the target device. Finally, this backend emits LLVM IR and invokes the LLVM compiler to perform low-level optimizations and generate the final machine code.

XLA supports only JIT compilation, and consequently, to leverage it in PyTorch, it is used through `torch.compile`. PyTorch/XLA offers a backend for TorchDynamo, enabling both inference and training acceleration. This integration is activated by specifying `backend='openxla'` when calling `torch.compile`. The underlying mechanism involves TorchDynamo providing a TorchFX graph of the model to PyTorch/XLA, which then compiles this graph into an optimized function for XLA devices [14] which may include CPU, GPU and TPUs.

2.5 IREE

IREE (Intermediate Representation Execution Environment) [15] is an MLIR-based end-to-end compiler and runtime that lowers ML models to a unified IR that scales up to meet the needs of the datacenter and down to satisfy the constraints and special considerations of mobile and edge deployments [16, 17]. Its approach is mainly AOT compilation, producing a self-contained binary artifact that is executed by its own runtime. This makes IREE well-suited for deployment scenarios where the model execution is decoupled from the high-level framework, such as edge devices. While IREE was once part of a unified OpenXLA project, it has since transitioned into an independent open-source initiative with strong support by AMD. Despite this formal separation, the two ecosystems remain linked through their shared foundation in MLIR and their mutual leveraging of StableHLO as a common front-end input format. For the scope of this work, we will leverage MLIR and StableHLO through the OpenXLA project via torch.compile, as it presents an in-framework PyTorch workflow and a broader popularity across developers.

2.6 Apache TVM

Apache TVM is an open source machine learning compiler framework for CPUs, GPUs, and other less popular machine learning accelerators. It aims to enable machine learning engineers to optimize and run computations efficiently on any hardware backend [18]. It predates almost every previous ML compilation effort as it was introduced in 2018 [19] in an effort to bridge the gap between the productivity found in high level frameworks and specialized backends by providing an end-to-end compilation pipeline that transforms high-level model descriptions into optimized, deployable code [20, 21] for any supported platform without exclusively defaulting to vendor libraries. It employs its own IRs, Relax IR for high-level graph optimizations and TensorIR for detailed, hardware-specific tuning. The main workflow involves importing a model, applying graph-level optimizations, and then using Ahead-Of-Time (AOT) compilation to generate an optimized model. However, TVM's current support for PyTorch and GPU presents significant challenges, as outdated and conflicting documentation across versions result in a lack of usability. Due to these practical key limitations within our scope, we concluded that a fair comparison was not feasible and therefore excluded TVM from our benchmark analysis.

3 Model Selection

3.1 SoTA Models

To provide a comprehensive view of the SoTA LLM landscape, we look for different models to cover the dimensions' spectrum so that we cover compact and more substantial models that also may fit within our testbed configuration (Table 1) even when accounting for potential compilation overhead.

Furthermore, a second critical point for this evaluation is the model's native data type. The choice of data type profoundly impacts performance, as it directly determines which hardware execution units and compiler optimizations can be utilized on modern GPUs. Therefore, our selection of models must allow us to directly test how MLC workflows handle these different key computation characteristics through leveraging hardware and compiler optimization.

3.1.1 TinyLlama/TinyLlama-1.1B-Chat-v1.0

The first model, `TinyLlama-1.1B-Chat-v1.0`, is a compact LLM with 1.1 billion parameters [22, 23] with BF16 weights. This presents a specific test case for compiler performance and compatibility with a data format that is increasingly prevalent in modern training and inference pipelines. Due to its relatively small memory footprint, TinyLlama serves as an excellent benchmark for evaluating baseline compiler efficiency without running out of memory.

3.1.2 meta-llama/Llama-2-7b-chat-hf

The second model chosen is `meta-llama/Llama-2-7b-chat-hf`, a popular model in the open-source community with 7 billion parameters [24, 25]. From a compilation standpoint, this model is particularly relevant for several reasons. First, its weights are stored in FP16 format, the standard for achieving maximum throughput on NVIDIA GPUs by leveraging specialized `Tensor Cores`. Second, its 7B parameter size presents a more significant challenge in terms of memory consumption and computational load compared to TinyLlama. And finally, it does present some layer with FP32 precision accumulation, which presents the opportunity to evaluate Mixed Precision performance. This allows for an evaluation of how each compiler scales and manages resources under greater pressure, without being prohibitively large for typical research and deployment environments.

3.2 Synthetic Models

To better understand how each MLC workflow deals with the key GEMM operation, four synthetic PyTorch models are brought up: Matmul, Matmul+ReLU, FFN-1-layer, and FFN-multilayer. These models allow observation of the behaviour of the standalone matmul op with and without an activation layer and within an isolated FFN block and with multiple ones:

1. **SyntheticMatmulBlock:** A single linear layer to establish a baseline for General Matrix Multiplication (GEMM) performance.
2. **Matmul+ReLU:** A linear layer followed by a ReLU activation to specifically test the compilers' operator fusion capabilities.
3. **FFN_Layer:** A complete Feed-Forward Network block (Linear + ReLU + Linear), which constitutes a significant portion of a Transformer's computational load.
4. **FFN_multilayer:** A stack of multiple FFN blocks to simulate model depth and analyze performance on deeper computational graphs.

All four models are parametrized so that they support multiple dimensions and datatypes, which can accommodate similarly sized operations and equal precision setup to those in state-of-the-art models. This allows us to replicate the dimensions of the GEMM ops on FFN layers coming from the SoTA LLMs that we will bench.

4 Results

4.1 Testbed

The chosen hardware to execute the benchmark suite is the NVIDIA GeForce RTX 4090. A powerful consumer-grade GPU, its 4th generation Tensor Cores allow support for BF16 and TF32 and its 24 GiB of VRAM are sufficient to deploy and compile the selected *smaller* LLMs.

With regard to the software, as we have deployed two different environments under different Docker images. *Triton Env*, based on the NVIDIA NGC Triton Image *25.03-trtllm-python-py3* [26], comes with support for TensorRT and TensorRT-LLM out-of-the-box as well as onnxruntime, while PyTorch can easily be installed. As OpenXLA requires specific CUDA and PyTorch versions which are incompatible with the Triton versions, openXLA has been deployed on its own docker image, *OpenXLA Env* (Table 1). This modular approach allows creating new docker images to support further MLC workflows.

Table 1: Software Versions: Triton vs. OpenXLA Environments

Software	Versions	
	Triton Env	**OpenXLA Env**
Ubuntu	24.04	22.04
Python	3.12	3.11
NVIDIA CUDA	12.9.0	12.4.1
NVIDIA cuBLAS	12.9.0.2	12.4.5.8-1
NVIDIA cuDNN	9.9.0.52	9.1.0.70-1
Nsight Systems	2025.2.1.130-1	2025.2.1.130-1
NVIDIA TensorRT	10.9.0.34	—
Torch-TensorRT	2.7.0	—
ONNX	1.17.0	—
ONNX Runtime	1.21.0	—
TensorRT-LLM	0.18.2	—
transformers	4.48.3	4.51.3
accelerate	1.6.0	1.8.0
flash-atten	2.7.4.post1	—
PyTorch	2.7.0	2.5.0
TorchVision	0.22.0a0	0.20.0
Torch XLA	—	2.5.0
xgboost	—	3.0.1

4.2 SoTA LLM Evaluation

To accurately evaluate the end-to-end SoTA performance of the compiler workflows compared to the PyTorch Eager baseline, for each of the two selected SoTA models we will both evaluate how many tokens/s an end-to-end LLM produces after an inference pass on a batch of prompts (throughput). LLM prompts will be 32 tokens long and its total count will be determined by the batch size, with batch size = 1, representing a sole 32 token prompt (a short sentence) as input while batch size = 64 translates into a batched input of 64 prompts of 32 tokens each. Then the LLM generates a fixed 32 tokens as output per prompt.

As we know the total number of output tokens and its total inference time, we then calculate the throughput like this:

$$\text{Throughput (tokens/s)} = \frac{\text{batch\_size} \times \text{output\_sequence\_length}}{\text{runtime\_in\_seconds}} \tag{1}$$

4.2.1 TinyLlama-1.1B-Chat-v1.0 (BF16)

Table 2: Supported MLC Workflows by TinyLlama-1.1B-Chat-v1.0

LLM	**torch.compile**			**TorchTensorRT**	**onnxruntime-tensorrt**	**TensorRT-LLM**
	Inductor	**XLA**	**TensorRT**			
TinyLlama-1.1B	✓	✓	✓	×	✓	×

Starting with the lighter `TinyLlama/TinyLlama-1.1B-Chat-v1.0` on the default BF16 weights, the used workflows for the end-to-end LLM JIT evaluation are torch.compile with the inductor, tensorrt and xla backends. Regarding the AOT workflows for the LLM evaluation (Table 2), TorchTensorRT and onnxruntime with TensorRTExecutionProvider cannot compile down some model's operations dependent on the transformers package. This is a common occurrence which happens with some of the latest SoTA models and, therefore, cannot generate a TensorRT Engine. However, TensorRT-LLM with PyTorch as the front-end or TorchTensorRT-LLM does come with an out-of-the box implementation of a TensorRT Engine for TinyLlama/TinyLlama-1.1B-Chat-v1.0.

Results (Figure 2) indicate that JIT compilation via torch.compile cannot improve on the baseline with any of the utilized backends (Inductor with Triton kernels, XLA with MLIR or TensorRT with low level CUDA) which results on these workflows defaulting to the PyTorch Eager implementations

relying on vendor libraries and therefore, achieving similar results. This can be attributed to two factors, either subgraphs cannot be captured or they cannot be optimized.

Back with TensorRT-LLM, the speedup is consistent across all batch sizes, but happens to peak for the lower batch sizes (1 through 16) with a ˜60% speedUp becoming a 37% one as the batch size increases. When batch sizes become increasingly bigger, so do the GEMM operations' dimensions which cause them to dominate the time in the overall graph execution, therefore, diminishing other performance improvements for other lesser important operations, so in the end, performance improvement may start to dwindle when LLMs pass a certain dimension threshold which depends on the model's complexity and layer count.

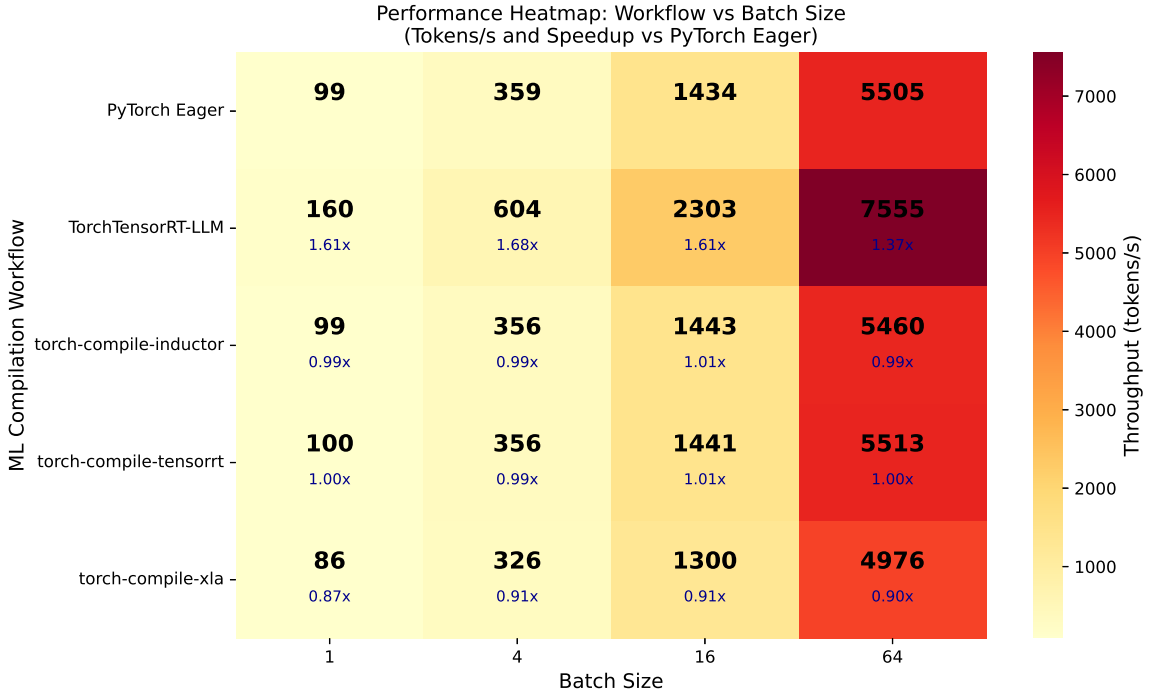

Fig. 2: `TinyLlama/TinyLlama-1.1B-Chat-v1.0` BF16 throughput across different precisions and MLC workflows through batch sizes =1, 4, 16, 64

4.2.2 Llama-2-7b-chat-hf (FP16)

Table 3: Supported MLC Workflows by Llama-2-7b-chat-hf

LLM	torch.compile			TorchTensorRT	onnxruntime-tensorrt	TensorRT-LLM
	Inductor	XLA	TensorRT			
Llama-2-7b	✓	✓	✓	✓	×	✓

For the performance evaluation of the larger `meta-llama/Llama-2-7b-chat-hf`, which operates using FP16 precision, all the previous workflows are used in addition to TorchTensorRT which can compile the full LLM this time around. onnxruntime with TensorRT cannot perform the compilation of the full `meta-llama/Llama-2-7b-chat-hf` LLM within the onnxruntime inference session as the process runs out of memory (see Table 3).

Back with the results, throughput is considerably lower than for `TinyLlama/TinyLlama-1.1B-Chat-v1.0` despite the change to the more Tensor Core friendly FP16 as `meta-llama/Llama-2-7b-chat-hf` is a far more complex model with greater dimensions and weights. The results, detailed in Figure 3, again position TensorRT-LLM as the superior compilation framework, but with a revealing trend reversal in its scaling behavior. Unlike with TinyLlama, the

speedUp delivered by TensorRT-LLM over the PyTorch Eager baseline increases with the batch size. It begins with a modest 1.15x improvement at a batch size of one and grows to a more significant 1.29x advantage at a batch size of 64. First, this indicates that the speedUp is lower than for its BF16 counterpart. This can be attributed to the fact that Llama-2's architecture forces some layers to FP32 precision, which benefits the least when compiled with TorchTensorRT. However, the growth in speedup from the lower to the bigger batch size indicates that for a model of this scale and complexity, the overheads that TensorRT-LLM mitigates—such as kernel launch latency and inter-operation synchronization—become an even more dominant bottleneck in the eager execution path as the workload scales. By aggressively fusing the deeper computational graph, TensorRT-LLM's optimizations become progressively more impactful, while PyTorch Eager generates more synchronization calls in between kernels, which causes warp occupancy to be lower overall as synchronization forces to empty the GPU pipeline before starting the next kernels, therefore, causing speedUp to still increase for batch size = 64 (see *Carmona-Martínez, A. (2025): Evaluation and Characterization of Machine Learning Compilers for AI inference on GPU. Master Thesis, Universidad de Murcia*).

The JIT compilation landscape continue to show negligible impact for on LLM performance. The JIT workflows performance for the end-to-end LLM remains almost identical to the PyTorch Eager baseline across nearly all configurations because of the same reasons, too. The only improvement comes from a minor 1.07x speedup via `torch_compile_xla` at the largest batch size of 64, suggesting it finds some optimization potential once the computational load is sufficiently high. However, this gain is too small to be considered a competitive advantage.

Finally, the generic Torch-TensorRT AOT workflow, which this time around is able to be compiled, exhibits a clearly worse performance than that of TensorRT due to the fact that the standalone TensorRT implementation does not support KV caching [9] while TensorRT-LLM, does. As the batch size increases, the size of the Key-Value (KV) cache grows proportionally. The generic Torch-TensorRT compiler, lacking specialized LLM optimizations, manages this large cache inefficiently. This flawed memory management creates a severe bottleneck that compounds with scale. The cost of moving and accessing this cache grows faster than the computational workload itself. In conclusion, TensorRT is not to be solely used, but is to always be complemented by TensorRT-LLM when SoTA LLM compilation is needed. However, in that case, the end-user is limited to the current supported LLM pool, which is an important issue to take into account before selecting said model as we'd comment more in the next section.

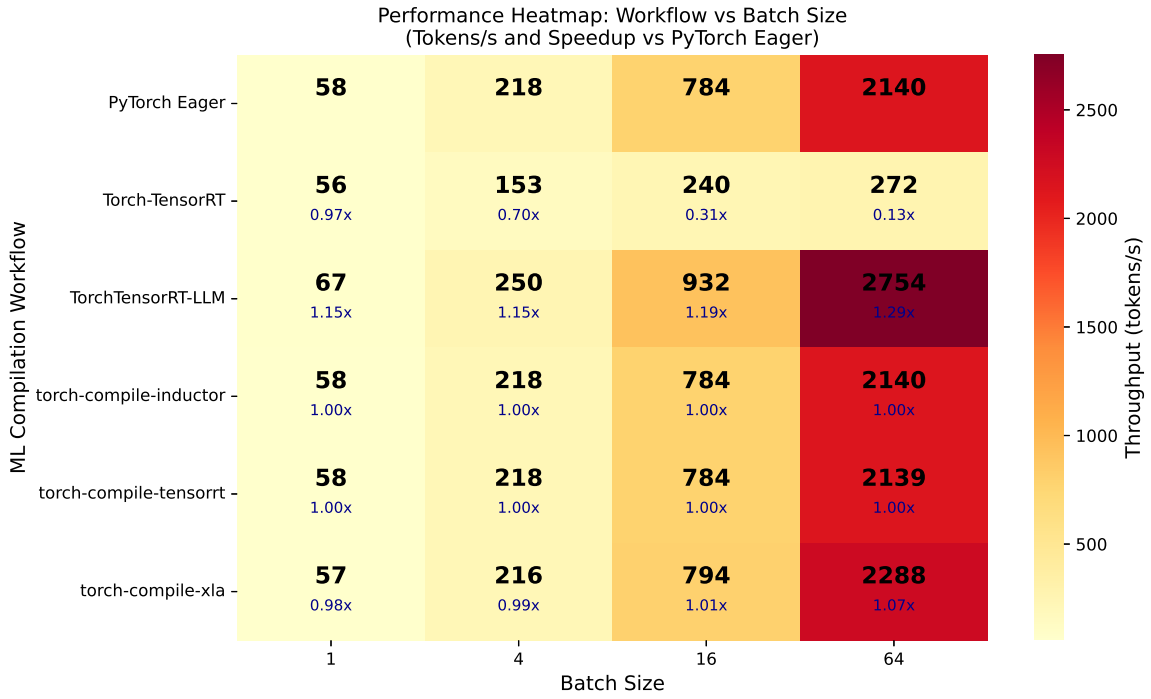

Fig. 3: `meta-llama/Llama-2-7b-chat-hf` FP16 throughput across different precisions and MLC workflows through batch sizes =1, 4, 16, 64

4.3 Synthetic Model Evaluation

The evaluation of synthetic PyTorch models allows to both isolate the FFN computational pattern (GEMM+activation+GEMM) of the previous LLMs while also testing the MLC behavior on vanilla PyTorch models. To that end, the synthetic models are adapted to the dimensions and datatype used on each LLM (Table 4) so that we can measure how many TFLOPS/s the compiled synthetic models achieve with equivalent synthetic 32-token inputs and outputs. The switch to TFLOPS/s allows to more easily compare the performance achieved with the peak performance figures for a specific datatype in the NVIDIA whitepaper [27] for our GPU.

All previous workflows used for the full LLMs have been used on their synthetic models' counterpart (Table 5) except for TorchTensorRT-LLM, as it cannot compile PyTorch models and has been replaced by the default TorchTensorRT. onnxruntime could not be used with BF16 alongside numpy, therefore it only was used for Llama-2.

Table 4: Specifications for Synthetic Models

Reference Architecture	Model Name	Num layers	M = batch_size × seq_len	d_model (N)	d_ff (K)	Precision
TinyLlama-1.1B	FFN_Matmul_1layer	1	batch_size x 32	2048	5632	BF16
	FFN_MatmulRelu_1layer	1				
	FFN_1layer	1				
	FFN_multilayer	22				
Llama-2-7B	FFN_Matmul_1layer	1	batch_size x 32	4096	11008	FP16
	FFN_MatmulRelu_1layer	1				
	FFN_1layer	1				
	FFN_multilayer	32				

Table 5: Supported MLC Workflows by Reference Architecture

Reference Architecture	torch.compile			TorchTensorRT	onnxruntime-tensorrt	TensorRT-LLM
	Inductor	XLA	TensorRT			
TinyLlama-1.1B	✓	✓	✓	✓	×	×
Llama-2-7B	✓	✓	✓	✓	✓	×

Moving on to the actual synthetic results for the TinyLlama-1.1B-based model in Figure 4, although, for the simpler models and smaller batch sizes, PyTorch Eager often outperforms compiled solutions. This is attributed to the initial compilation overhead that Machine Learning Compilers (MLCs) introduce, which is not sufficiently amortized by the simple workload. When model `FFN_multilayer_tinyllama` is evaluated, torch.compile does not improve on the baseline neither via Inductor nor TensorRT, the TensorRT backend in particular, is casting kernels to accumulate on FP32 even though, the model's using BF16 natively as further evaluated in *Carmona-Martínez, A. (2025): Evaluation and Characterization of Machine Learning Compilers for AI inference on GPU. Master Thesis, Universidad de Murcia* via NVIDIA nsight systems. This simplified execution allows us to observe how for Inductor, the graph can be captured, but the optimized implementation, in this case Triton kernels generated by the compiler, may not improve on the baseline, defaulting to Eager mode. On the other hand, the compiler may be able to improve specific computaional blocks, in this case the FFN as is the case for XLA which can accelerate its performance visibly, however if the greater computational graph cannot be captured, optimizations cannot be leveraged. Continuing on the FFN-multilayer, as was the case in the LLM evaluation, when compilation is AOT via TorchTensorRT, performance improves greatly with the batch increases over both PyTorch Eager and JIT.

On the Llama-2 front (Figure 5), for AOT compilation, it is also remarkable the close to 2x speedUp achieved by Torch-TensorRT on FP16 data over PyTorch Eager and torch.compile once the overhead is compensated due to PyTorch (and torch.compile) always forcing FP32 accumulation for

FP16 data, a useful technique to maximize the achieved accuracy during training and some layer execution, which also halves FP16 Tensor Core performance from 330.4 to 165.2 TFLOPS [27]. This technique may not always be necessary in inference, and when TensorRT evaluates the FP16 GEMM, it defaults to native FP16 GEMM without casting to FP32 midprocess when possible, which allows it to double PyTorch's performance [27]. Furthermore, it is also visible when comparing against the performance for BF16. FP16 GEMM outperforms BF16's by approximately 100 TFLOPS, which comes down to the fact that Tensor Cores forces FP32 accumulation on BF16 opposed to FP16, which also limits its performance to 165.2 TFLOPS.

Continuing with Llama-2, when multiple FFN layers are combined and the overhead is at its lowest, onnxruntime with TensorRT outweighs its more pronounced early execution overhead for the Matmul and Matmul+ReLU models, finishing just below TorchTensorRT, which has a less pronounced execution overhead. This overhead comes from Host-to-Device and Device-to-Host memory transfers for the inputs as the default way of using onnxruntime inference session is to store the inputs in CPU and let the inference session manager perform the transfers to any devices that will be used by the session as it is only known at runtime by the manager opposed to TorchTensorRT which allows easier binding through PyTorch (*Carmona-Martínez, A. (2025): Evaluation and Characterization of Machine Learning Compilers for AI inference on GPU. Master Thesis, Universidad de Murcia*). This, of course, will come with an overhead for smaller models in which inputs are reused often. For that scenario, onnxruntime also supports I/O bindings of the input to a specific device which greatly reduces overhead, and onnxruntime TRT performance matches and even may surpass that of TorchTensorRT. As a trade-off, the user cannot leverage onnxruntime out-of-the-box portability as they have to deal with input-loading specific to the device, so it becomes a matter of compromises and priorities; however, as seen with large enough models, the performance difference is not as steep and the portability benefits may be worth it.

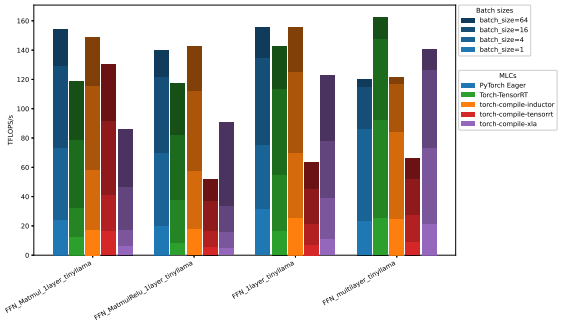

Fig. 4: Every synthetic Tinyllama-based BF16 model (Table 4) for every MLC workflow with batch sizes = 1, 4, 16, 64

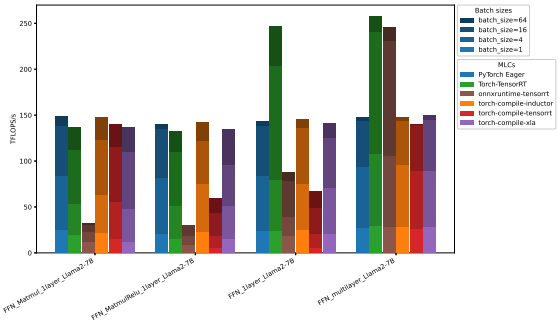

Fig. 5: Every synthetic Llama2-based FP16 model (Table 4) for every MLC workflow with batch sizes = 1, 4, 16, 64

5 Discussion

Performance-wise, it is clear that the AOT TensorRT/TensorRT-LLM workflows outperform the competition (see Figure 6). When dealing with regular PyTorch models, TensorRT suffices, but it can be limited when it comes to input flexibility as it performs the best when the input is fixed. It does allow for both the onnxruntime and TorchTensorRT approaches, and while the onnxruntime approach can favour portability, with easy to switch Execution Providers for different devices, in certain scenarios, its performance may not match that of TorchTensorRT in addition with running out of memory more easily as models have to be converted into ONNX first and then optimized once again with TensorRT. For SoTA LLMs, TensorRT-LLM covers that other part of the model spectrum, offering support for SoTA LLMs along with more input flexibility and ease of use as the model is already built in TensoRT-LLM without the user explicit conversion. While both TensorRT and TensorRT-LLM can work in tandem to work with what the other cannot, the portability aspect remains the main concern while the necessary deployment and framework knowledge to know which models can be successfully deployed with TensorRT and TensorRT-LLM alongside options setup also hampers the Productivity aspect.

When reviewing the performance gains of TensorRT, we can also see that the performance advantage does not only come from optimized kernels with but also reducing operation launches as this drastically reduces the synchronization overhead, a key bottleneck that plagues the PyTorch Eager runtime as it dispatches a multitude of smaller, un-fused operations. The profiling traces (see *Carmona-Martínez, A. (2025): Evaluation and Characterization of Machine Learning Compilers for AI inference on GPU. Master Thesis, Universidad de Murcia*) can confirm this, showing the replacement of several aten calls with a handful of TensorRT-generated kernels. While AOT compilation is always beneficial from a performance standpoint, its relative impact can be nuanced as the workload scales to larger batch sizes and becomes more compute-bound and less complex, with the baseline PyTorch Eager implementation becoming more efficient as kernel optimization start to matter less on very compute-bounded kernels, and synchronization costs become negligible.

PyTorch moved in the other direction with its JIT-based torch.compile. This prioritizes ease of use and compatibility with PyTorch over performance and while Inductor can make out-of-the box gains in other model architectures, in particular CNNs [28], it's clearly not adequate for LLMs in its current form even when utilizing the Nvidia-specific TensorRT backend, which can cause issues with accumulation precision.

Finally, the OpenXLA approach is a compelling one as it tries to leverage a unique MLC for all devices, however, it is clear that only offering it in PyTorch through a JIT workflow in torch.compile cannot make it compete with the Nvidia GPU-based alternatives. Although its JIT performance on BF16 (a Google impulsed datatype) for PyTorch models may be of interest to many, overall its lack of peak performance and lack of support for LLMs render it second-best.

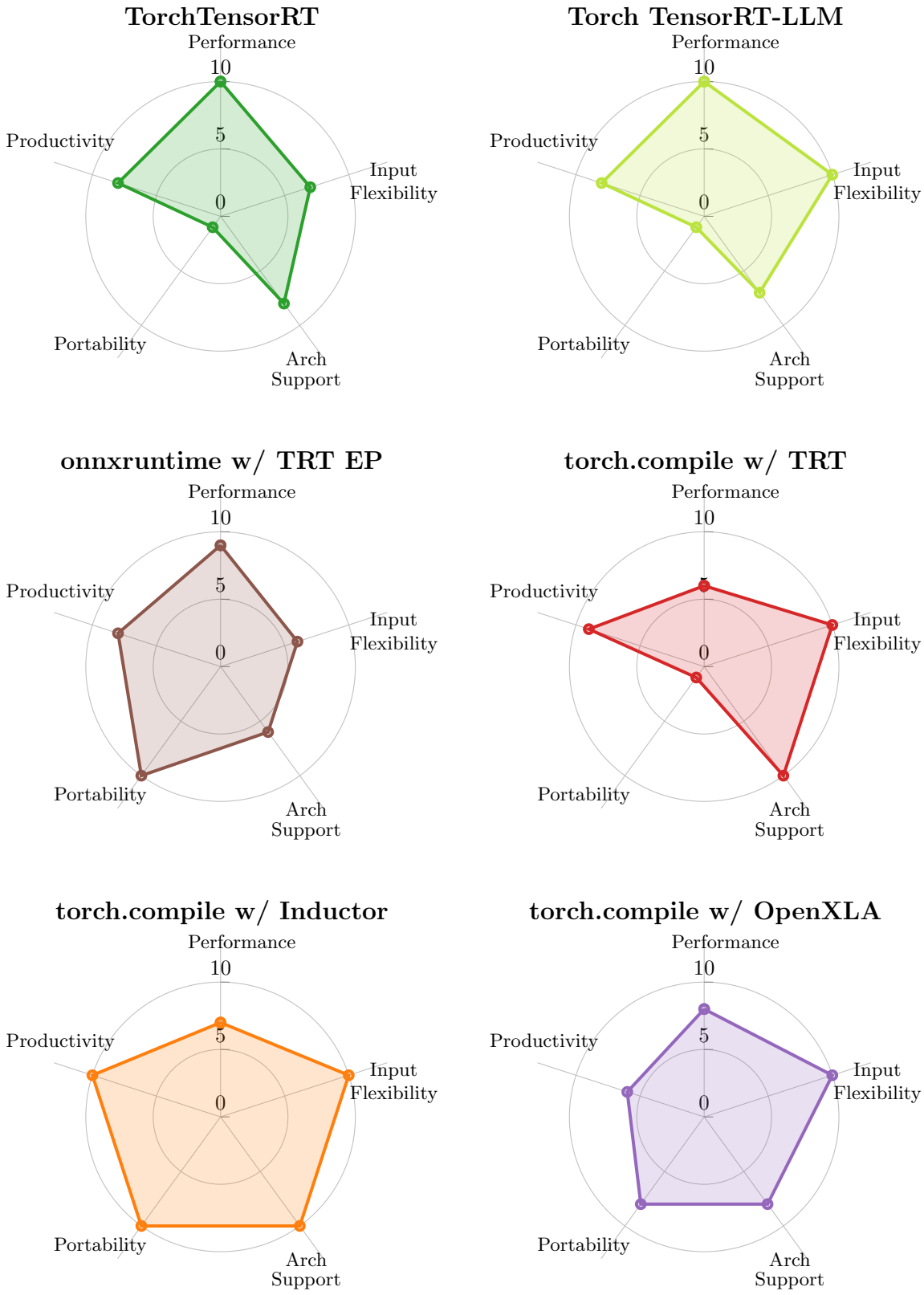

Fig. 6: Review of the MLC frameworks evaluated across key criteria

6 Conclusions

This work meets its three main goals towards MLC study and usage for LLM inference on NVIDIA GPUs within a PyTorch-centric environment. First, a review of four major (and intertwined) MLC tools—PyTorch's JIT-based `torch.compile`, NVIDIA TensorRT (including TensorRT-LLM), Google's XLA, and the ONNX format with ONNX Runtime—evaluating their Productivity, Portability, and Performance (P3) characteristics is performed. Benchmarking using SoTA LLMs as well as PyTorch synthetic allows to depict the MLC benefits—particularly in reducing GPU synchronization— or lack thereof which explains how AOT TensorRT workflows consistently yield the highest throughput while multi-target approaches like PyTorch's `torch.compile` prioritize portability at the expense of raw performance, which may yield no speedUp for LLM inference. Finally, it comments on general guidelines to enable developers to select an MLC strategy aligned with specific P3 priorities and deployment constraints, showcasing the benefits of each approach and how that impacts on the P3 trade-offs.

Further research would focus on evaluating the new Modular platform against the incumbents evaluated in this work. Modular's MAX engine leverages its own programming language based on MLIR, Mojo, to create a unified hardware-agnostic deployment stack, from the programming language to the serving layer. First, NVIDIA GPU support was added to the already existent CPU and even more recently, full support for both NVIDIA and AMD GPUs within a single container was released [29]. The proposed research would benchmark MAX as an offline compiler, directly comparing its performance and P3 capabilities against the established workflows. To test MAX's portability claims, the hardware testbed would be expanded. Alongside the NVIDIA RTX 4090, the study would incorporate a somewhat equivalent AMD Radeon RX 7900 XTX. This would enable a direct comparison of MAX's performance against native PyTorch on AMD hardware as well as NVIDIA's and its CUDA ecosystem, assessing performance, productivity as well as opening new avenues with the cost-per-token metric for each hardware and software combination. The evaluation would mainly leverage LLM SoTA models, `TinyLlama/TinyLlama-1.1B-Chat-v1.0` (BF16) and `meta-llama/Llama-2-7b-chat-hf` (FP16) would be selected for continuity and joined by the more recent `meta-llama/Llama-3.1-8B-Instruct` (BF16) to include a model from the latest generation of LLMs. By analyzing performance on both NVIDIA and AMD hardware, the study would offer new insights into the practical viability of the Modular platform's MAX as a hardware-agnostic MLC and its implications on the current SoTA for the P3 problem.

Furthermore, the newly released NVIDIA TensorRT for RTX (TensorRT-RTX) is presented as a specialized version of NVIDIA TensorRT, specifically designed for the RTX product line. A distinguishing feature of TensorRT-RTX is its hybrid AOT and JIT compilation. The AOT compilation does not require access to a GPU because after the TensorRT-RTX engine file is built, one-shot JIT compilation is performed to select the exact GPU kernels for running on the target machine. The whole compilation process is expected to be faster than the TensorRT baseline. At the moment, it's available with its own Python API, as well as an onnxruntime backend [30]. Further research would include onnxruntime with TensorRT-RTX and its Python API as two distinct workflows in order to evaluate whether there are performance improvemnts to be found on RTX machines or its main advantage is to offer a more lightweight environment for RTX machines. TensorRT-LLM is not supported by TensorRT-RTX at the moment.

Acknowledgements. Grant PID2022-136315OB-I00 funded by MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe", EU

References

[1] Li, M., Liu, Y., Liu, X., Sun, Q., You, X., Yang, H., Luan, Z., Gan, L., Yang, G., Qian, D.: The deep learning compiler: A comprehensive survey. IEEE Transactions on Parallel and Distributed Systems **32**(3), 708–727 (2021) https://doi.org/10.1109/TPDS.2020.3030548

[2] Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., Hirsh, B., Huang, S., Kalambarkar, K., Kirsch, L., Lazos, M., Lezcano, M., Liang, Y., Liang, J., Lu, Y., Luk, C.K., Maher, B., Pan, Y., Puhrsch, C., Reso, M., Saroufim, M., Siraichi, M.Y., Suk, H., Zhang, S., Suo, M., Tillet, P., Zhao, X., Wang, E., Zhou, K., Zou, R., Wang, X., Mathews, A., Wen, W., Chanan, G., Wu, P., Chintala, S.: 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. ASPLOS '24, pp. 929–947. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3620665.3640366 . https://doi.org/10.1145/3620665.3640366

[3] ONNX Community: Open Neural Network Exchange (ONNX). https://onnx.ai/. Accessed: 08/06/2025 (2025)

[4] ONNX Runtime Team: ONNX Runtime Documentation. https://onnxruntime.ai/docs/. Accessed: 08/06/2025 (2025)

[5] NVIDIA Corporation: NVIDIA TensorRT SDK. https://developer.nvidia.com/tensorrt. Accessed: 07/06/2025 (2025)

[6] NVIDIA Corporation: NVIDIA TensorRT Documentation. https://docs.nvidia.com/deeplearning/tensorrt/latest/index.html. Accessed: 07/06/2025 (2025)

[7] Zhou, Y., Yang, K.: Exploring TensorRT to Improve Real-Time Inference for Deep Learning. In: 2022 IEEE 24th Int Conf on High Performance Computing Communications; 8th Int Conf on Data Science Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud Big Data Systems Application (HPCC/DSS/SmartCity/DependSys), pp. 2011–2018 (2022). https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00299

[8] NVIDIA Corporation: NVIDIA/TensorRT-LLM GitHub Repository. https://github.com/NVIDIA/TensorRT-LLM. Accessed: 09/06/2025 (2025)

[9] Elmeleegy, A., Comly, N., Johnsen, T.: 5x Faster Time to First Token with NVIDIA TensorRT-LLM KV Cache Early Reuse. https://developer.nvidia.com/blog/5x-faster-time-to-first-token-with-nvidia-tensorrt-llm-kv-cache-early-reuse/. Accessed: June 23, 2025

[10] NVIDIA Corporation: PyTorch Backend — TensorRT-LLM. https://nvidia.github.io/TensorRT-LLM/torch.html. Accessed: 09/06/2025 (2025)

[11] Sabne, A.: XLA : Compiling Machine Learning for Peak Performance (2020)

[12] OpenXLA Community: XLA - OpenXLA Project. https://openxla.org/xla. Accessed: 09/06/2025

[13] Lattner, Chris and Amini, Mehdi and Bondhugula, Uday and Cohen, Albert and Davis, Andy and Pienaar, Jacques and Riddle, River and Shpeisman, Tatiana and Vasilache, Nicolas and Zinenko, Oleksandr: MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 2–14 (2021). https://doi.org/10.1109/CGO51591.2021.9370308

[14] PyTorch Foundation: TorchDynamo(torch.compile) integration in PyTorch XLA — PyTorch/XLA master documentation. https://docs.pytorch.org/xla/master/torch_compile.html. Accessed: 09/06/2025

[15] Liu, H.-I.C., Brehler, M., Ravishankar, M., Vasilache, N., Vanik, B., Laurenzo, S.: TinyIREE: An ML Execution Environment for Embedded Systems From Compilation to Deployment. IEEE Micro **42**(5), 9–16 (2022) https://doi.org/10.1109/MM.2022.3178068

[16] The IREE Authors: IREE Home Page. https://iree.dev/guides/deployment-configurations/gpu-cuda/. Accessed: 09/06/2025

[17] The IREE Authors: IREE GitHub Repository (2019). https://github.com/iree-org/iree

[18] Apache Software Foundation: Apache TVM Official Website. https://tvm.apache.org/. Accessed: 09/06/2025

[19] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A.: TVM: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. OSDI'18, pp. 579–594. USENIX Association, USA (2018)

[20] Apache Software Foundation: Apache TVM Overview. https://tvm.apache.org/docs/get_started/overview.html. Accessed: 09/06/2025

[21] mlc.ai: Introduction to Machine Learning Compilation. https://mlc.ai/chapter_introduction/index.html. Accessed: 09/06/2025

[22] Zhang, P., Zeng, G., Wang, T., Lu, W.: TinyLlama: An Open-Source Small Language Model (2024). https://arxiv.org/abs/2401.02385

[23] TinyLlama: TinyLlama/TinyLlama-1.1B-Chat-v1.0 Hugging Face page. https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0. Accessed: 15/06/2025 (2024)

[24] Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C.C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P.S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E.M., Subramanian, R., Tan, X.E., Tang, B., Taylor, R., Williams, A., Kuan, J.X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., Scialom, T.: Llama 2: Open Foundation and Fine-Tuned Chat Models (2023). https://arxiv.org/abs/2307.09288

[25] Meta: meta-llama/Llama-2-7b-chat-hf Hugging Face page. https://huggingface.co/meta-llama/Llama-2-7b-chat-hf. Accessed: 15/06/2025 (2023)

[26] NVIDIA Corporation: Triton Inference Server. https://catalog.ngc.nvidia.com/orgs/nvidia/containers/tritonserver. Accessed: 10/06/2025 (2025)

[27] NVIDIA Corporation: NVIDIA ADA GPU Architecture. https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf. Accessed: 16/06/2025

[28] Hugging Face Team: Optimize inference using torch.compile(). https://huggingface.co/docs/transformers/v4.38.1/en/perf_torch_compile. Accessed: 24/06/2025

[29] Modular Team: Modular 25.4: One Container, AMD and NVIDIA GPUs, No Lock-In. https://www.modular.com/blog/modular-25-4-one-container-amd-and-nvidia-gpus-no-lock-in. Accessed: 24/06/2025 (2025)

[30] NVIDIA Corporation: Example Deployment Using ONNX. NVIDIA TensorRT for RTX Documentation. https://docs.nvidia.com/deeplearning/tensorrt-rtx/latest/installing-tensorrt-rtx/example-deployment.html (2025)