The Journal of Supercomputing (2026) 82:420
https://doi.org/10.1007/s11227-026-08559-6

Characterization of machine learning compilers for LLM inference on NVIDIA GPUs

Alejandro Carmona-Martínez[1,2] · Gregorio Bernabé[1] · José M. García[1]

Received: 18 September 2025 / Accepted: 27 April 2026

Abstract
AI inference is conflicted between Performance, developer Productivity, and device Portability–the P3 problem. Machine learning compilers (MLCs) aim to address this, but their ecosystem is fragmented, with tools that each prioritize a different issue. This paper evaluates the deployment 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 the peak performance of Ahead-Of-Time (AOT) compilation requires architecture-specific tools such as TensorRT-LLM, which are necessary for SOTA LLMs but are unusable for PyTorch models. As for Just-In-Time (JIT) solutions such as `torch.compile` and its backends, they are flexible and portable, compatible with all tested models, but they do not consistently accelerate LLMs; therefore, the choice of MLC depends on P3 considerations and model architecture.

Keywords LLMs · Machine learning compilation · Inference · NVIDIA · GPU · PyTorch

Gregorio Bernabé and José M. García have contributed equally to this work.

✉ Alejandro Carmona-Martínez
alejandro.carmonam@um.es

Gregorio Bernabé
gbernabe@um.es

José M. García
jmgarcia@um.es

[1] Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, C. Campus Universitario, 11, 30010 Murcia, Murcia, Spain

[2] Libelium Comunicaciones Distribuidas S.L., Avda. María Zambrano 31 Edificio WTCZ, Torre Este Planta 7, 50018 Zaragoza, Aragon, Spain

Published online: 15 May 2026

1 Introduction

Large language models (LLMs) have remarkable capabilities in natural language understanding, generation, and reasoning that 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 FLOPs.

GPUs, designed for massive parallelism, excel at tasks with high arithmetic intensity, such as GEMMs, due to their high throughput (FLOPs/s), making them the de facto accelerators for LLMs and AI inference. 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 general-purpose CUDA Cores and vastly outperform the vector units found in CPUs, especially for the *shorter* datatypes, thereby enabling increased throughput. Tensor Cores support the traditional float (FP32) via a cast to an internal datatype, TF32, which uses only 19 bits to balance FP32's higher precision with FP16 (half-precision) performance. BF16 precision takes the compromise further, as it has 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 capability provided by frameworks such as PyTorch that 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 executed as a single program rather than issuing multiple library calls with their associated synchronization costs. This tension exposes the core P3 Problem: the fundamental trade-off among *Productivity*, *Portability* (the ease of PyTorch and usability across devices or multiple GPUs), and 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 (Fig. 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. Since 2020, new players such as `torch.compile`, `onnxruntime`, and TensorRT have emerged, and LLMs and NVIDIA GPUs have taken over as the main models and hardware in the AI landscape.

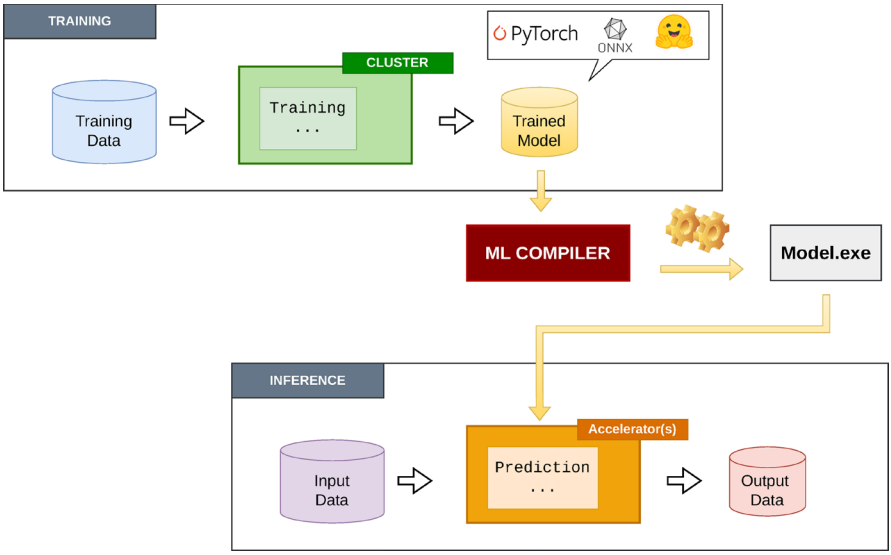

Fig. 1 MLC in the training/inference pipeline

This work directly addresses the trade-offs specific to each solution when deploying PyTorch models and SOTA LLMs on NVIDIA GPUs, leveraging the current, and somewhat 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`).

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

To do so, benchmarked models will include both 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 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 the fragmented MLC ecosystem and select the optimal compilation strategy for their specific requirements.

1.1 Main contributions

This work makes three core contributions to the understanding and practical deployment of MLCs for PyTorch-based 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 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 reducing 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: Sect. 2 reviews the state of the art, covering a survey of compilation frameworks and workflows (`torch.compile`, TensorRT, XLA, ONNX Runtime, and others). Section 3 describes model selection based on SOTA LLMs and synthetic FFN-based PyTorch Models. Section 4 presents the performance evaluation of both SOTA and synthetic models and concludes with a discussion of MLC workflows within the P3 Paradigm and their respective trade-offs. Finally, Sect. 5 recaps the previous contributions and proposes directions for future research.

2 Background and state of the art

2.1 JIT: PyTorch's `torch.compile`

PyTorch 2.0 introduced `torch.compile`, a JIT compilation tool to streamline the optimization of PyTorch programs [2], with the primary benefit of a single decorator that enables device- and code-agnostic compilation.

This compilation mechanism operates by first using its front-end, *TorchDynamo*, to safely capture Python bytecode from model definitions or functions, and then translating it 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, using PyTorch's in-house solution, *Inductor*, 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 GPUs [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 has its own set of options, they all adhere to a common IR interface, 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 that serve as the fundamental building blocks for machine learning and deep learning models, along with 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, who can submit a priority list for model execution with the preferred order of EPs, using optimized kernels for the target hardware. EPs 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 the performance of production ML applications [6]. After a model has been built and trained using a framework such as PyTorch or TensorFlow, TensorRT takes the model as input to create a highly optimized *engine* tailored to the target NVIDIA hardware.

In their 2022 study, Zhou and Yang [7] summarized the key workflows to leverage TensorRT. In our paper, we present an updated workflow classification that also includes JIT/AOT considerations.

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 ONNX format. A PyTorch model is first exported to ONNX, and then ONNX Runtime executes it using the TensorRT EP to leverage 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.

Unlike the general-purpose TensorRT, NVIDIA TensorRT-LLM is a specialized tool engineered exclusively to accelerate LLM inference [8]. It packages a compiler and runtime equipped with LLM-centric optimizations, including custom attention kernels and efficient Key-Value (KV) cache management [9]. It allows a set of operands to deploy custom-built LLMs, but does not support 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 that accelerates machine learning models from popular frameworks such as PyTorch, TensorFlow, and JAX across a range of hardware platforms [11, 12], aiming 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 modular ML infrastructure components that can be assembled into an end-to-end stack.

A key architectural aspect is its use of an MLIR-based IR. MLIR is a compiler toolchain [13] that enables common optimizations through its device-agnostic representation, with multiple levels of lowering that generate machine-specific code, supporting multiple backends. This process begins with a series of high-level, target-independent optimizations of the input graph that 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; consequently, to leverage it in PyTorch, it is used via `torch.compile`. PyTorch/XLA provides a backend for TorchDynamo, enabling acceleration for both inference and training. 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 the graph into an optimized function for XLA devices [14], which may run on CPU, GPU, or 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 in which model execution is decoupled from the high-level framework, such as on 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 from AMD. Despite this formal separation, the two ecosystems remain linked through their shared foundation in MLIR and their mutual leveraging of StableHLO [18] as a common front-end input format. For the scope of this work, we will leverage MLIR and StableHLO via the OpenXLA project via `torch.compile`, as it provides an in-framework PyTorch workflow and broader developer adoption.

2.6 Apache TVM

"Apache TVM is an open source MLC 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 [19]". It predates almost every previous ML compilation effort as it was introduced in 2018 [20] 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 [21, 22] 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 AOT compilation to generate an optimized model. However, TVM's current support for PyTorch and GPU poses significant challenges, as outdated and conflicting documentation across versions leads to reduced 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 consider models across a range of dimensions, including compact and more substantial models that may also fit within our testbed configuration (Table 1), even 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

Table 1 Software versions: triton vs. OpenXLA environments

Software	Versions	
	Triton-Env-2504	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

determines which hardware execution units and compiler optimizations can be utilized on modern GPUs. Therefore, our selection of models must enable direct testing of how MLC workflows handle these key computational characteristics by leveraging hardware and compiler optimizations.

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 [23, 24] with BF16 weights. While the model's tensors are stored and loaded in BF16, modern NVIDIA GPUs internally accumulate the results of Matrix Multiplications (GEMMs) in FP32 within the Tensor Cores to prevent numerical overflow and ensure stability. 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 [25, 26]. 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`. Similar to the BF16 processing in TinyLlama, the Tensor Cores process these FP16 inputs by internally accumulating the GEMM results in FP32. Second, its 7B parameter size poses a greater memory and computational load challenge than TinyLlama. Therefore, it provides an opportunity to assess Mixed FP16/FP32 Precision performance, allowing evaluation on 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 handles the key GEMM operation, four synthetic PyTorch models are introduced: Matmul, Matmul+ReLU, FFN-1-layer, and FFN-multilayer. These models allow observation of the behavior of the standalone matmul op[1] 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 GEMM performance.

[1] Throughout this paper, the terms "matmul", "GEMM" and "linear layer" are used interchangeably.

2. **Matmul+ReLU:** A linear layer followed by a ReLU activation to specifically test the compilers' operator fusion capabilities.
3. **FFN_Layer:** A complete FFN 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 to support multiple dimensions and data types, which can accommodate similarly sized operations and an equal-precision setup to those in state-of-the-art models. This allows us to replicate the dimensions of GEMM ops in FFN layers from the SOTA LLMs we will benchmark.

4 Results

4.1 Testbed

The hardware chosen to run the primary evaluation 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. Regarding the software, we have deployed two environments using separate Docker images. *Triton-Env-2504*, based on the NVIDIA NGC Triton Image *25.04-trtllm-python-py3* [27], comes with support for TensorRT and TensorRT-LLM out-of-the-box as well as `onnxruntime`, while a compatible PyTorch/TorchTensorRT pair can easily be installed. As OpenXLA requires specific CUDA and PyTorch versions that are incompatible with Triton versions, OpenXLA has been deployed in its own Docker image, *OpenXLA Env* (Table 1). This modular approach allows the creation of new Docker images to support additional MLC workflows. Furthermore, to evaluate the cross-architectural generalizability of our findings, we introduce a secondary hardware platform. The NVIDIA GeForce RTX 5090, featuring next-generation architecture and 5th-generation Tensor Cores, is utilized specifically for the comparative evaluation detailed in Sect. 4.5

Regarding performance metrics, we utilize two distinct units depending on the workload type. For the end-to-end Large Language Models 4.2, we report throughput in *tokens per second (tokens/s)* to reflect the user-perceived generation speed, including all system overheads. Conversely, for the synthetic micro-benchmarks 4.3, we utilize *TFLOPS* to measure raw computational efficiency and compare directly against the hardware's theoretical peak specifications.

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 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

Table 2 Supported MLC workflows: TinyLlama-1.1B-Chat-v1.0 on RTX 4090

GPU	`torch.compile`			TorchTensorRT	`onnxruntime-tensorrt`	TensorRT-LLM
	Inductor	XLA	TensorRT			
RTX 4090	✓	✓	✓	×	×	✓

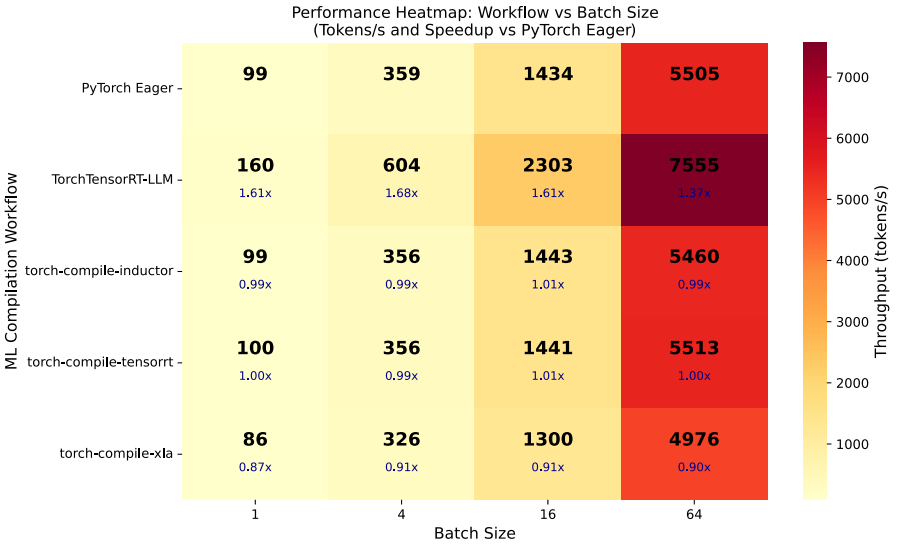

Fig. 2 `RTX 4090/TinyLlama/TinyLlama-1.1B-Chat-v1.0` BF16 throughput across different precisions and MLC workflows through batch sizes = 1, 4, 16, 64

long, and their 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. The LLM then generates a fixed output of 32 tokens per prompt.

As we know the total number of output tokens and their 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)

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 some models' operations that depend on the transformers package. This is a common issue with some of the latest SOTA models and, therefore, cannot generate a TensorRT Engine. However, TensorRT-LLM with PyTorch as the front-end (TorchTensorRT-LLM) includes an out-of-the-box implementation of a TensorRT Engine for TinyLlama/TinyLlama-1.1B-Chat-v1.0.

On the NVIDIA RTX 4090, results (Fig. 2) indicate that JIT compilation via `torch.compile` cannot improve on the baseline with any of the backends used (Inductor with Triton kernels, XLA with MLIR, or TensorRT with low-level CUDA), resulting in these workflows defaulting to the PyTorch Eager implementations that rely 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. This will be delved into at Sects. 4.4 and 5. With TensorRT-LLM, the speedup is consistent across all batch sizes, but peaks at 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 less 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.

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

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. As before, TensorRT-LLM can also use this large model. However, `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` as `meta-llama/Llama-2-7b-chat-hf` is a far more complex model with greater dimensions and weights. The results, detailed in Fig. 3, again position TensorRT-LLM as the superior compilation framework, but reveal a 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

Table 3 Supported MLC workflows: Llama-2-7b-chat-hf on RTX 4090

	`torch.compile`			TorchTensorRT	`onnxruntime`-tensorrt	TensorRT-LLM
GPU	Inductor	XLA	TensorRT			
RTX 4090	✓	✓	✓	✓	×	✓

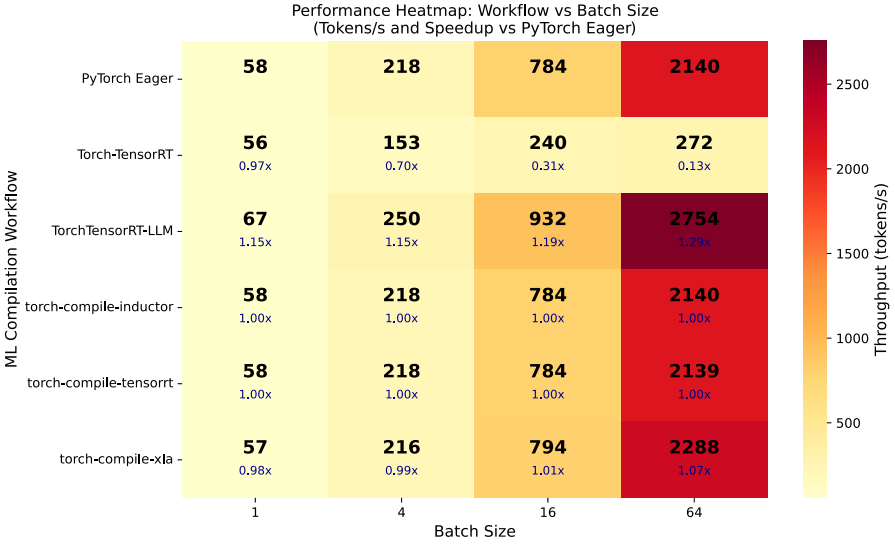

Fig. 3 `RTX 4090/meta-llama/Llama-2-7b-chat-hf` FP16 throughput across different precisions and MLC workflows through batch sizes = 1, 4, 16, 64

one and grows to a more significant 1.29x advantage at a batch size of 64. Speedup from the lower to the larger batch size indicates that, for a model of this scale and complexity, the overheads that TensorRT-LLM mitigates–such as kernel launch latency and interoperation synchronization–become an even more dominant bottleneck on 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 [28]).

The JIT compilation landscape continues to show negligible impact on LLM performance. The JIT workflows' performance for the end-to-end LLM is nearly identical to the PyTorch Eager baseline across nearly all configurations. The only improvement comes from a minor 1.07x speedup with `torch_compile_xla` at the largest batch size of 64.

Finally, the generic Torch-TensorRT AOT workflow, which can now be compiled, exhibits clearly worse performance than TensorRT-LLM because the standalone TensorRT implementation does not support KV caching [9], whereas TensorRT-LLM does. As the batch size increases, the KV cache size grows proportionally. The generic Torch-TensorRT compiler, lacking specialized LLM optimizations, inefficiently manages this large cache. This flawed memory management creates a severe bottleneck that compounds as scale increases. The cost of moving and accessing this cache grows faster than the computational workload itself. In conclusion, TensorRT should not be used in isolation; it should always be complemented by TensorRT-LLM when SOTA LLM compilation is needed. However, in that case, the end-user

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				

is limited to the current supported LLM pool, which is an important consideration when selecting the model, as we discuss further in the next section.

4.3 Synthetic model evaluation

The evaluation of synthetic PyTorch models allows us to both isolate the FFN computational pattern (GEMM+activation+GEMM) of previous LLMs and test 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 the compiled synthetic models achieve with equivalent synthetic 32-token inputs and outputs. The switch to TFLOPS allows us to more

Table 5 Supported MLC Workflows by reference architecture for synthetic models on RTX 4090

GPU	Reference Architecture	`torch.compile`			Torch TensorRT	`onnxruntime-tensorrt`	TensorRT LLM
		Inductor	XLA	TensorRT			
RTX 4090	TinyLlama-1.1B	✓	✓	✓	✓	×	×
	Llama-2-7B	✓	✓	✓	✓	✓	×

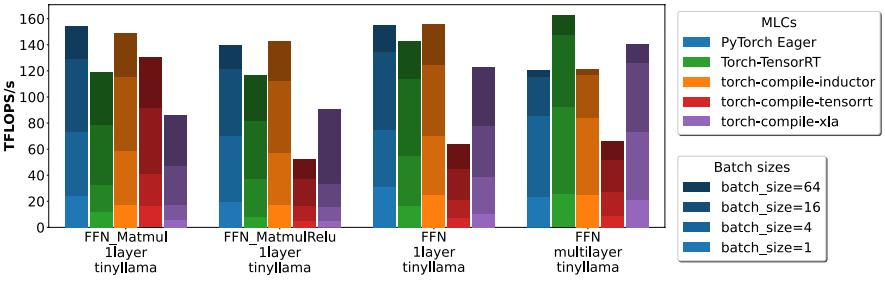

Fig. 4 `NVIDIA 4090`: Every synthetic Tinyllama-based BF16 model (Table 4) for every MLC workflow with batch sizes = {1, 4, 16, 64}

easily compare the performance achieved with the peak performance figures for a specific datatype in the NVIDIA whitepaper [29] for our GPU.

All previous workflows used for the full LLMs have been applied to their synthetic models' counterparts except for TorchTensorRT-LLM, which 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 5).

Moving on with the actual synthetic results for the TinyLlama-1.1B-based model in Fig. 4, PyTorch Eager often outperforms compiled solutions for the simpler models and smaller batch sizes. This is attributed to the initial compilation overhead introduced by MLCs, which is not sufficiently amortized by the simple workload.

In the `FFN_multilayer_tinyllama` model, `torch.compile` does not improve the baseline, neither with Inductor nor with TensorRT. 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 computational blocks, such as the FFN, as is the case with XLA, which can visibly accelerate performance; however, if the larger computational graph cannot be captured, optimizations cannot be leveraged. Continuing with the FFN-multilayer, as in the LLM evaluation, when compilation is AOT via TorchTensorRT, performance improves substantially with increasing batch size over both PyTorch Eager and JIT. A further evaluation featuring NVIDIA nsight systems traces breakdown can be found in [28].

Turning to the Llama-2 front (Fig. 5) with FP16 data, we can check that PyTorch (and `torch.compile`) always force FP32 accumulation for FP16 data, a useful technique to maximize the achieved accuracy during training and some layer execution, which also halves theoretical FP16 Tensor Core performance from 330.4 to 165.2 TFLOPS [29]. As this casting to FP32 may not always be necessary in inference, TensorRT defaults to native FP16 GEMM, which allows it to double PyTorch's performance [29]. Furthermore, it is also evident when compared with the performance of BF16. FP16 GEMM outperforms BF16 by approximately 100 TFLOPS, which comes down to the fact that Tensor Cores force FP32 accumulation on BF16 datatype, opposed to FP16, which also limits its performance to 165.2 TFLOPS.

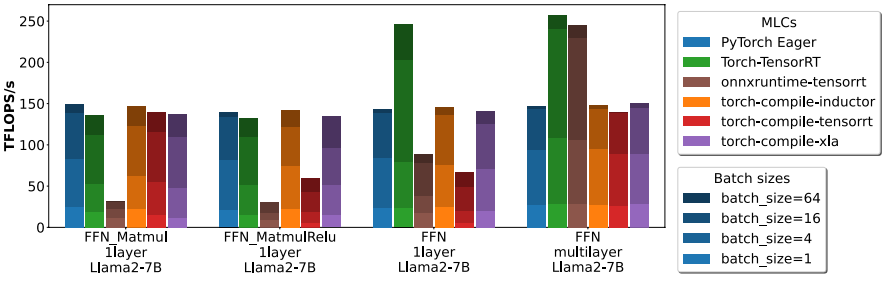

Fig. 5 `NVIDIA 4090`: Every synthetic Llama2-based FP16 model (Table 4) for every MLC workflow with batch sizes = {1, 4, 16, 64}

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 [28]. This, of course, will incur overhead for smaller models that reuse inputs frequently. For that scenario, `onnxruntime` also supports I/O bindings for the input to a specific device, which greatly reduces overhead, and `onnxruntime` TRT performance matches and may even 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.

4.4 Compiler costs (time and memory)

While the evaluation focuses primarily on inference throughput, for real deployment scenarios, especially those involving Ahead-Of-Time workflows such as TensorRT and TensorRT-LLM, compilation overhead can be a decisive factor in the P3 trade-off. Therefore, this subsection reports the compilation time (in seconds) and memory usage (in GiB) for all available MLC workflows across SOTA and synthetic models. The time value for each workflow depends on the workflow's JIT/AOT nature. For JIT, it is the sum of the time to load the model and the time to run it the first time. For AOT, it is the time required for AOT compilation, or, for TorchTensorRT-LLM, the time to load the already compiled model. The time taken for PyTorch Eager to load the model is also logged. Models have already been downloaded, so we discard

Table 6 Compilation overhead: synthetic TinyLlama FFN models & SOTA (Batch Size 16, BF16)

MLC	TinyLlama 1.1B		FFN Multilayer		FFN 1-Layer		FFN Matmul +ReLU		FFN Matmul	
	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)
PyTorch Eager	1.497	2.051	1.500	0.946	0.070	0.043	0.050	0.021	0.068	0.021
Torch-compile-inductor	1.493	2.087	4.579	1.037	1.229	0.134	1.129	0.112	0.685	0.038
Torch-compile-tensorrt	1.297	2.087	25.023	1.917	3.099	0.111	2.922	0.068	0.213	0.047
Torch-TensorRT	N/A	N/A	48.681	0.953	17.840	0.051	0.324	0.029	0.278	0.029
Torch-compile-xla	1.396	2.238	5.443	0.950	1.867	0.047	0.665	0.029	0.710	0.029
TorchTensorRT-LLM	8.912	3.775	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 7 Compilation overhead: synthetic Llama2-7B FFN Models & SOTA (Batch Size 16, FP16)

MLC	Llama2 7B		FFN Multilayer		FFN 1-Layer		FFN Matmul +ReLU		FFN Matmul	
	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)	Time (s)	Mem (GiB)
PyTorch Eager	2.334	12.55	12.45	5.376	0.346	0.168	0.192	0.084	0.208	0.084
Torch-compile-inductor	3.472	13.00	13.75	5.479	1.395	0.271	1.193	0.187	0.766	0.107
Torch-compile-tensorrt	3.021	13.00	37.14	10.84	2.566	0.428	2.264	0.260	0.244	0.176
Torch-TensorRT	141.5	13.90	94.18	5.392	34.19	0.184	0.450	0.100	0.382	0.100
Torch-compile-xla	4.651	14.22	11.90	5.384	1.470	0.176	0.750	0.098	0.896	0.098
Onnxruntime-tensorrt	N/A	N/A	46.62	5.933	6.185	0.278	4.043	0.190	3.946	0.190
TorchTensorRT-LLM	8.783	14.70	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

that download time. The recorded memory footprint is the peak GPU memory used during the process.

To simplify the information displayed, the batch size has been set to 16 for all models, yielding balanced inputs (roughly 512 tokens). Nonetheless, except for TorchTensorRT, which processes the input size beforehand, the input size is not a critical factor; according to our benchmarking, the time and memory peaks are consistent throughout the batch size increases.

Tables 6 and 7 cover the costs for the SOTA TinyLlama-1.1B and Llama2-7B and their synthetic siblings, respectively. The first striking insight for both is that, while memory costs increase with model complexity, peaking for the SOTA models, time increases only up to FFN Multilayer. Upon closer inspection, we observe that when the SOTA LLM is evaluated, the compilation overhead for all JIT-based PyTorch workflows on the SOTA models often collapses to the same scale as PyTorch Eager. To obtain the LLM's output, PyTorch's generate is used while the synthetic models use forward. Under the default Hugging Face `generate()` path, execution is frequently not captured in a stable, compilable graph, and therefore, the run does not actually enter a sustained compiled regime. Therefore, the performance gains in synthetic benchmarks are consistently not being translated into the LLMs, save for a few exceptions. `generate()` is a Python-level decoding procedure with distinct *prefill* and *decode* phases, iterative state updates, and dynamic tensor growth (e.g., input ids and masks expanding token-by-token). These characteristics are exactly the conditions that tend to introduce graph breaks and/or repeated recompilations in JIT compilers that rely on graph capture, thereby leading to partial capture or a fallback to eager execution. In contrast, our synthetic FFN benchmarks, invoking `forward()` directly, provide a more stable compute graph that is straightforward to capture and optimize. This is why, despite MLC workflows such as XLA or TorchTensorRT significantly improving performance on synthetic benchmarks, this is not reflected in SOTA LLMs. It is also noteworthy that TensorRT-LLM has its own `generate()` method associated with the optimized TensorRT engine rather than using PyTorch's `generate()`.

Across both model families, peak memory usage is primarily driven by the baseline model weights (TinyLlama: ≈ 2 GiB; Llama2-7B: ≈ 12.55 GiB in PyTorch Eager), with additional compiler/runtime overhead depending on the workflow. For the SOTA models, this appears as a slight uplift for most JIT/AOT stacks (e.g., Llama2-7B: 12.55 GiB → 13.00 GiB for `torch-compile-inductor`/`torch-compile-tensorrt`, +3.6%; 13.90 GiB for Torch-TensorRT, +10.8%; 14.22 GiB for XLA, +13.3%), consistent with extra compilation context, workspace buffers, and allocator reservations. Two notable outliers are `TorchTensorRT-LLM` and `torch-compile-tensorrt` on the larger graphs: TorchTensorRT-LLM raises the TinyLlama footprint to 3.775 GiB (almost 2x the 2.051 GiB Eager baseline), while for Llama2-7B the increase is smaller but still visible (14.70 GiB vs 12.55 GiB, +17.1%). Similarly, `torch-compile-tensorrt` shows disproportionately high memory peaks on the largest synthetic graphs (e.g., FFN Multilayer: 1.917 GiB vs ≈ 0.95–1.04 GiB for Inductor/XLA on TinyLlama; 10.84 GiB vs ≈ 5.38–5.48 GiB for Inductor/XLA on Llama2), indicating a substantially larger compilation/execution workspace than its JIT/AOT peers. This indicates that, while there might be an ≈ 5% always-present overhead, if the graph is actually captured and there is sufficient margin for compilation strategies, this can increase to nearly 100%. Moreover, during the compilation process for `torch-compile-tensorrt` and `Torch-TensorRT`, some compilation strategies may be skipped if VRAM is insufficient. This is consistent with the relatively small memory usage of Torch-TensorRT despite its long compilation times.

For both TinyLlama and Llama2 synthetic siblings, Torch-TensorRT (AOT) is consistently the slowest path, reaching 48.681 s (Tiny FFN Multilayer) and 94.18 s (Llama2 FFN Multilayer), and peaking at 141.5 s for the full Llama2-7B SOTA model as it is evaluating different compilation strategies. With this occurring offline before the inference process, it can also reduce its total memory footprint, but on hardware with larger VRAM capacities, its memory consumption may be even higher. For reference, the Llama2-7B SOTA model weighs 12.5 GiB, and the 24 GiB of the 4090's VRAM were not enough for compilation strategies that, according to the compilation logs, could peak at 24.85 GiB, twice the size of the original model at its intended 16-bit precision.

Among JIT options, `torch-compile-tensorrt` is typically the most expensive time-wise, taking a substantial fraction of its AOT counterpart (e.g., Llama2 FFN Multilayer: 37.14 s vs 94.18 s), while Inductor and XLA remain comparatively close on these FFN graphs (Tiny FFN Multilayer: 4.579 s vs 5.443 s; Llama2 FFN Multilayer: 13.75 s vs 11.90 s). Finally, `onnxruntime-tensorrt` (AOT) shows relatively high overhead for the smallest FFN variants (e.g., 3.946–4.043 s on Llama2 Matmul/Matmul+ReLU, where other backends are sub-second), and then converges and even halves TorchTensorRT's time while just 7 s off torch-compile-tensorrt's JIT pace.

4.5 Cross-generational hardware scaling

While Sects. 4.2 and 4.3 establish a comprehensive performance baseline on the RTX 4090, relying on a single GPU architecture naturally raises questions about the broader applicability of these conclusions. A critical consideration is whether

the observed performance bottlenecks—most notably, the limited optimization gains from JIT-based workflows like `torch.compile`—are tied to the Ada Lovelace architecture, or if they represent inherent challenges in modern LLM computational graphs.

To ensure a fair comparison, the software stack on the RTX 5090 also employs the previously discussed *Triton-Env-2504*. However, necessary environmental updates and changes must be noted due to the bleeding-edge nature of the hardware/software pairing. First, the RTX 5090 requires CUDA 12.8 or later to function. Because OpenXLA is strictly bound to CUDA 12.4 and PyTorch 2.5.0, the *OpenXLA Env* cannot be deployed on this newer GPU, consequently, XLA workflows are excluded from this cross-generational evaluation. Second, with the release of CUDA 13.0, TensorRT-LLM introduced updated kernel mappings that significantly altered the performance metrics. Therefore, an updated Docker environment (*Triton-Env-2602*, based on the newer *26.02-trtllm-python-py3* image) featuring CUDA 13.0 and TensorRT-LLM 1.1.0 has been included, as it better reflects the current TensorRT-LLM SOTA. Finally, the usage of CUDA 13.0 and TensorRT-LLM 1.1.0 causes two incompatibilities: i) onnxruntime-gpu with the TensorRT-EP cannot be used since the latest version requires CUDA 12.8, and ii) Torch-TensorRT cannot either, as this TensorRT-LLM requires the transformers 4.56.0, but TorchTensorRT usage needs 4.52.4 or earlier to work with Llama-2-7B.

Table 8 Software versions: triton environments

Software	Versions	
	Triton-Env-2504	Triton-Env-2602
Ubuntu	24.04	24.04
Python	3.12	3.12.3
NVIDIA CUDA	12.9.0	13.0
NVIDIA cuBLAS	12.9.0.2	13.1
NVIDIA cuDNN	9.9.0.52	9.14
Nsight Systems	2025.2.1.130-1	2026.1.1
NVIDIA TensorRT	10.9.0.34	10.13.3
Torch-TensorRT	2.7.0	2.9.0
ONNX	1.17.0	–
ONNX Runtime	1.21.0	–
TensorRT-LLM	0.18.2	1.1.0
Transformers	4.48.3	4.56.0
Accelerate	1.6.0	1.12.0
Flash-atten	2.7.4.post1	–
PyTorch	2.7.0	2.9.0
TorchVision	0.22.0a0	0.24.0
Torch XLA	–	–
xgboost	–	–

Table 9 Supported MLC workflows: TinyLlama-1.1B-Chat-v1.0 and Llama-2-7b-chat-hf on RTX 5090

GPU/Env	SOTA LLM	`torch.compile`			Torch TensorRT	`onnxruntime-tensorrt`	TensorRT LLM
		Inductor	XLA	TensorRT			
RTX 5090/ Triton-Env-2504	TinyLlama-1.1B	✓	×	✓	×	×	✓
	Llama-2-7B	✓	×	✓	✓	×	✓
RTX 5090/ Triton-Env-2602	TinyLlama-1.1B	✓	×	✓	×	×	✓
	Llama-2-7B	✓	×	✓	×	×	✓

Table 10 Supported MLC workflows by reference architecture for synthetic models on RTX 5090

GPU/Env	Reference Architecture	`torch.compile`			Torch TensorRT	`onnxruntime-tensorrt`	TensorRT LLM
		Inductor	XLA	TensorRT			
RTX 5090 Triton-Env-2504	TinyLlama-1.1B	✓	×	✓	✓	×	×
	Llama-2-7B	✓	×	✓	✓	✓	×
5090/ Triton-Env-2602	TinyLlama-1.1B	✓	×	✓	×	×	×
	Llama-2-7B	✓	×	✓	×	×	×

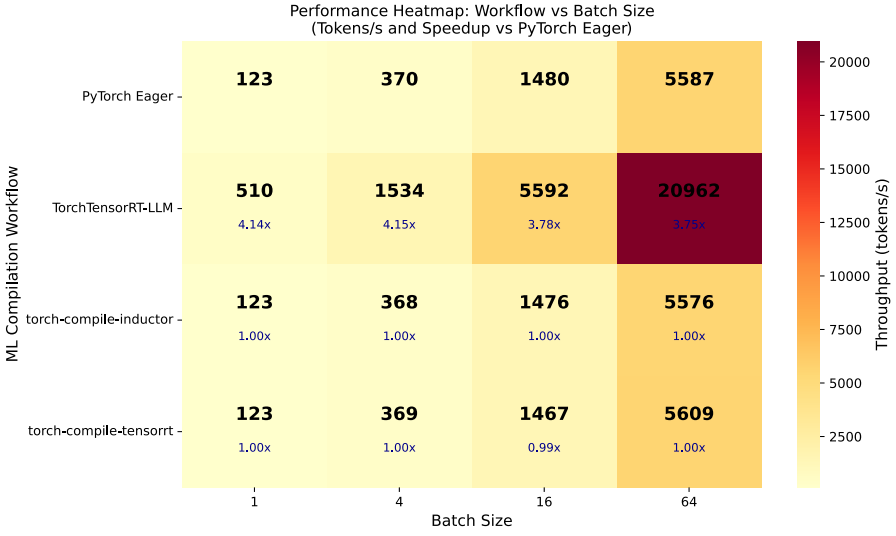

Fig. 6 `RTX 5090/Triton-Env-2602/TinyLlama/TinyLlama-1.1B-Chat-v1.0` BF16 throughput across different precisions and MLC workflows through batch sizes = 1, 4, 16, 64

Table 8 gives the software details of the Triton Environments used with the NVIDIA RTX 5090. Additionally, Tables 9 and 10 give the supported MLC workflows for SOTA LLMs and Synthetic Models.

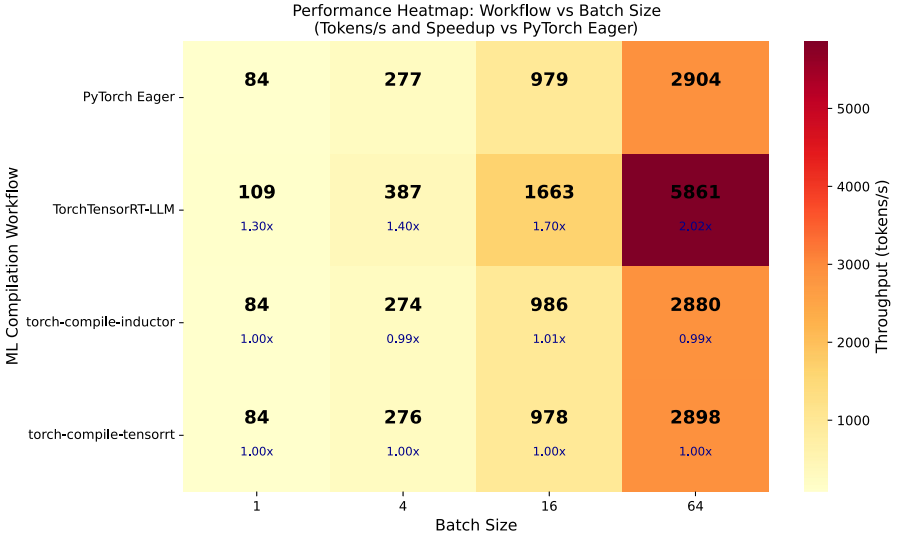

Fig. 7 `{RTX 5090/Triton-Env-2602/meta-llama/Llama-2-7b-chat-hf` FP16 throughput across different precisions and MLC workflows through batch sizes = 1, 4, 16, 64.

4.5.1 SOTA LLM and synthetic model evaluation

The RTX 5090 measurements reinforce two of the central conclusions of the RTX 4090 study. First, for Hugging Face-style autoregressive inference (i.e.,

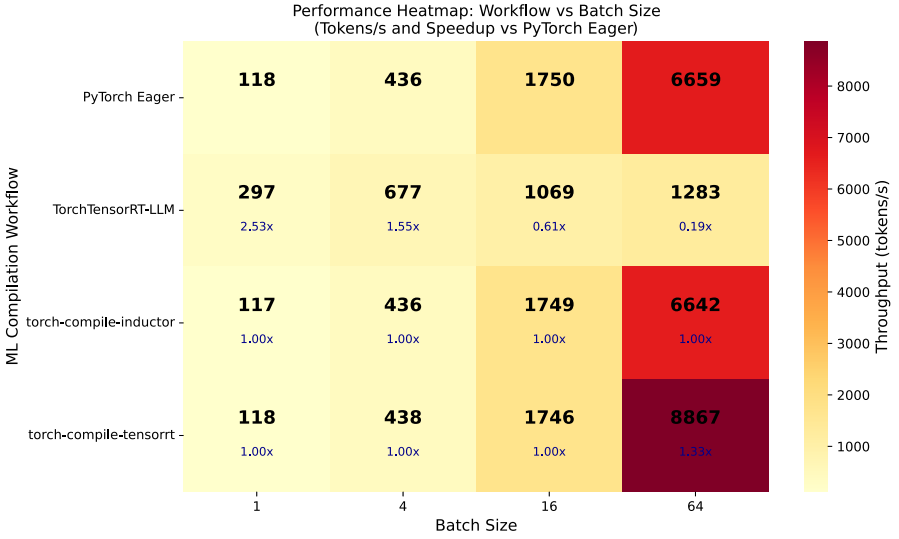

Fig. 8 `RTX 5090/Triton-Env-2504/TinyLlama/TinyLlama-1.1B-Chat-v1.0` BF16 throughput across different precisions and MLC workflows through batch sizes = 1, 4, 16, 64

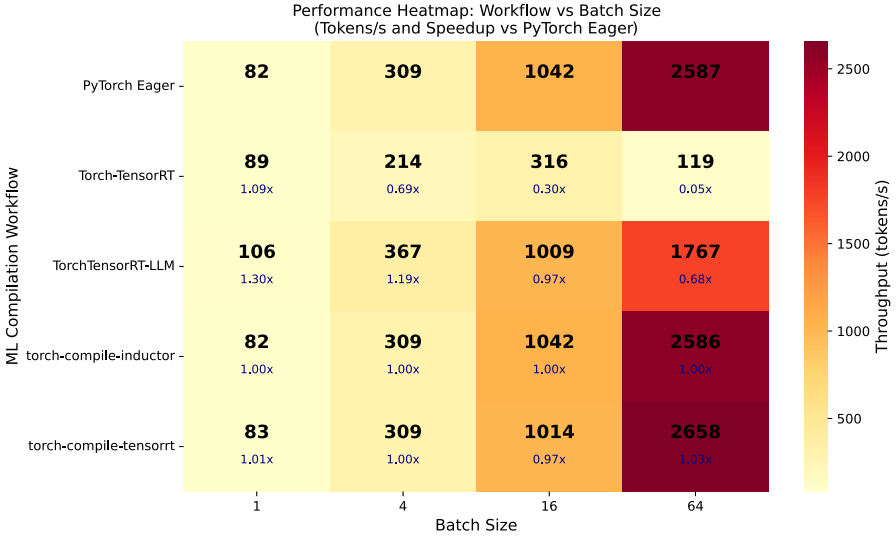

Fig. 9 `RTX 5090/Triton-Env-2504/meta-llama/Llama-2-7b-chat-hf` FP16 throughput across different precisions and MLC workflows through batch sizes = 1, 4, 16, 64

`generate()`), JIT-based PyTorch workflows remain tightly coupled to the PyTorch Eager baseline across batch sizes, consistent with the limited and unstable graph capture discussed in Sect. 4.4. This is clearly observed in the *Triton-Env-2602* runs (Figs. 6 and 7): for TinyLlama, PyTorch Eager and `torch-compile-inductor`/`torch-compile-tensorrt` are effectively identical (e.g., batch size 64: 5587 vs 5576–5609 tokens/s), and the same pattern holds for Llama2-7B (batch size 64: 2904 vs 2880–2898). Similar results are recorded for *Triton-Env-2504* (Figs. 8 and 9). In other words, despite moving from Ada Lovelace to Blackwell, the status quo remains.

Second, the cross-generational results indicate that TensorRT-LLM is far more sensitive to the maturity of the software stack than the PyTorch baselines. Under *Triton-Env-2504*, the TorchTensorRT-LLM engines exhibit an anomalous scaling behavior on both TinyLlama and Llama2. In particular, for TinyLlama, while it improves upon PyTorch Eager for batch size equals 1 (297 vs 118 tokens/s) and remains more than competitive at batch size 4 with 1.55x speedup, it then degrades sharply at larger batch sizes, falling below Eager when batch size rises to 16 to 0.6x and collapsing to approximately 0.2x of that baseline at batch size 64. This inversion is aligned with the execution traces, which strongly suggest that, for the larger GEMM (general matrix multiplication) shapes induced by higher batch sizes, the engine's selected kernels in this environment/driver/compiler combination are suboptimal or unoptimized for Blackwell, plausibly reflecting the early nature of CUDA 12.8 as the first supporting release for the architecture. By contrast, under *Triton-Env-2602* (CUDA 13.0 and TensorRT-LLM 1.1.0), TensorRT-LLM performance becomes consistently superior and exhibits the expected scaling while clearly surpassing the speed ups achieved with the 4090: on TinyLlama it goes from

roughly 4.1x speedUp on Eager to 3.8 for 64, while peaking at 20963 tokens/s; and on Llama2-7B it improves from 1.3x to 2x with 5861 tokens against Eager's 2904. These shifts instead indicate a genuine change in kernel mapping and/or scheduling within the TensorRT-LLM stack, consistent with the recorded higher Tensor Core usage.

A noteworthy exception concerns `torch-compile-tensorrt` on TinyLlama under *Triton-Env-2504* (Fig. 8), there is a clear gain of 33% for batch size = 64. Similar to their `torch.compile` pairs, upon close trace inspection, the kernel list and count were identical. However, some of these same kernels in this scenario

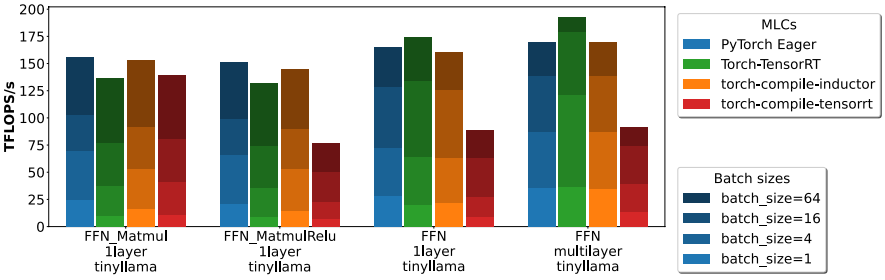

Fig. 10 `NVIDIA 5090/Triton-Env-2504`: every synthetic Tinyllama-based BF16 model (Table 4) for every MLC workflow with batch sizes = {1, 4, 16, 64}

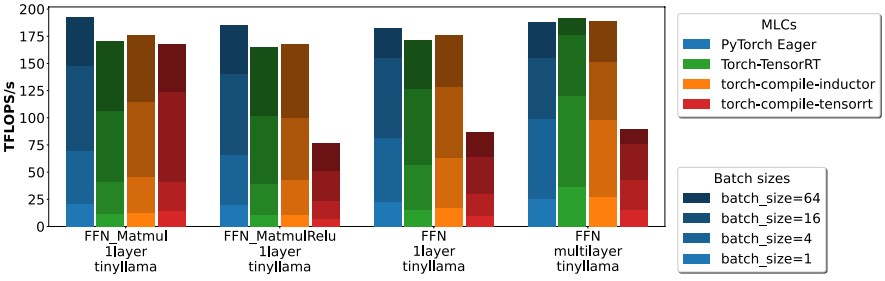

Fig. 11 `NVIDIA 5090/Triton-Env-2602`: every synthetic Tinyllama-based BF16 model (Table 4) for every MLC workflow with batch sizes = {1, 4, 16, 64}

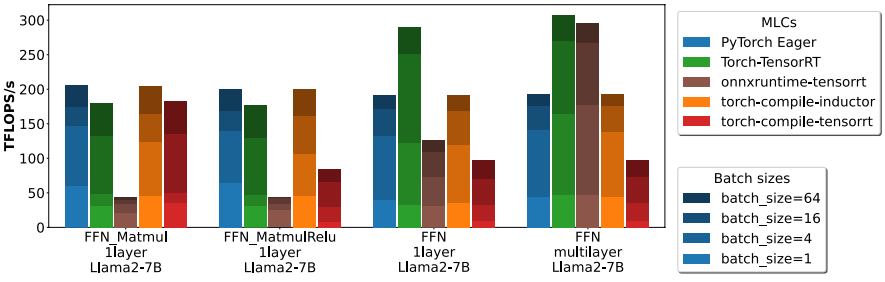

Fig. 12 `NVIDIA 5090/Triton-Env-2504`: every synthetic Llama2-based FP16 model (Table 4) for every MLC workflow with batch sizes = {1, 4, 16, 64}

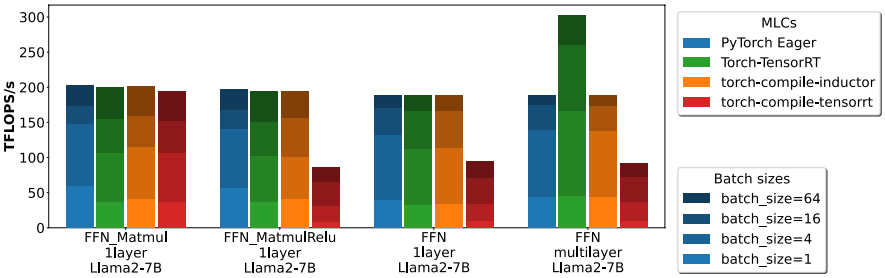

Fig. 13 `NVIDIA 5090/Triton-Env-2602`: every synthetic Llama2-based FP16 model (Table 4) for every MLC workflow with batch sizes = {1, 4, 16, 64}

reported considerably high Tensor Core usage while their `torch.compile` equivalents kernels did not report any at all. This indicates that `torch-compile-tensorrt` path can activate some kernels to use TC when possible. This did not occur for *Triton-Env-2602*, where the kernel list and count remained the same, and no differences in TC use were reported, nor in Llama2 for either *Triton-Env-2504*, *Triton-Env-2602*, or in the RTX 4090.

For the synthetic benchmarks (see Figs. 10, 11, 12 and 13), the previous trends are somewhat maintained. PyTorch eager remains fastest for the smaller models and batch sizes, then TensorRT-AOT based MLCs surpass both PyTorch and the `torch.compile` workflows which are now missing OpenXLA. For TinyLlama, while TorchTensorRT still records the highest TFLOPS peak, results are now much closer, especially for *Triton-Env-2602*, where performance figures are consistently higher than for *Triton-Env-2504*, while TorchTensorRT's FFN-multilayer remains fairly similar, which indicates improvements for the more memory-dependent models, which were not as visible for the larger LLMs from the older to the newest environment. It is a similar story on the Llama2 side: *Triton-Env-2504*'s MLC workflows' results resemble what is seen on the RTX 4090, with the high peak-performance difference on TorchTensorRT that also translates to *Triton-Env-2602*. In this last environment, however, TorchTensorRT's optimizations do not take effect until FFN-Multilayer, and the performance is identical to that of PyTorch for the other FFN models

5 Discussion

Performance-wise, it is clear that the AOT TensorRT/TensorRT-LLM workflows outperform the competition (see Fig. 14). When working with standard PyTorch models, TensorRT suffices, but it can be limited in input flexibility, as it performs best when the input is fixed. It supports both the `onnxruntime` and TorchTensorRT approaches. While the `onnxruntime` approach can favor portability, with easy-to-switch EPs across devices, in certain scenarios its performance may not match that of TorchTensorRT. For SOTA LLMs, TensorRT-LLM covers the other part of the model spectrum, offering support for SOTA LLMs with greater input

flexibility and ease of use, as the model is already built into TensorRT-LLM without the user's explicit conversion. While both TensorRT and TensorRT-LLM can work in tandem to leverage what the other cannot, portability remains the primary concern. The necessary deployment and framework knowledge to determine which models can be successfully deployed with TensorRT and TensorRT-LLM, along with options setup, also hampers productivity. This is taken further when considering inter-hardware software-stack compatibility. As we have seen, a stack may work well on one platform for all involved MLCs, but that might change on another, even if there is official compatibility.

When reviewing the performance gains of TensorRT, we can also see that the performance advantage does not only come from optimized kernels but also from 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, unfused operations. The profiling traces (see [28]) confirm this, showing that several aten calls are replaced 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 starts to matter less on very compute-bound 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 deliver out-of-the-box gains with other model architectures, in particular CNNs [30], it is clearly not adequate for LLMs in its current form, even when using the NVIDIA-specific TensorRT backend, which can cause issues with accumulation precision.

Regarding OpenXLA, its approach is compelling, as it aims to leverage a unique MLC across all devices; however, it is clear that offering it only in PyTorch via a JIT workflow in `torch.compile` cannot make it compete with the NVIDIA GPU-based alternatives. Moreover, as of early 2026, OpenXLA has stopped receiving direct support for CUDA, remaining tied to ever older CUDA versions as their developers have chosen to focus on TPUs.

Compiler costs may also play a role in choosing a strategy, as results indicate that compilation costs are not considerable in time for either JIT or AOT workflows, but they can be in memory, as reflected by the TensorRT-based workflows. Therefore, it is necessary to consider that the memory of the device on which the compilation occurs may hamper the final performance of the compiled model and limit the gains from these tools, with the compilation of the TinyLlama SOTA LLM and synthetic models requiring approximately twice the baseline model's total peak allocated memory.

Overall, the results suggest a practical deployment guideline aligned with the P3 criteria. For research, prototyping, and rapid iteration on custom models, `torch.compile` (typically Inductor) is the recommended default: it offers the highest Productivity through a near drop-in workflow, preserves PyTorch-level flexibility, and provides a convenient baseline, even if end-to-end LLM gains can be limited by `generate()` graph-capture constraints. In contrast, for production

and edge scenarios where cost-per-token dominates and models are static, `TensorRT-LLM` is the preferred option when the target LLM and GPU are supported: it trades Portability and some Input Flexibility for consistently higher Performance and better efficiency via an engine-native execution path. Finally, the compilation time, memory and performance highs (and lows) observed in TensorRT-based workflows indicate that the build environment (VRAM capacity and software-stack versioning) is part of the deployment decision and must be studied both theoretically and in practice.

Table 11 Evaluation rubric for machine learning compiler characterization (0–10 scale)

Criterion	Scoring definition
Performance	**10 (High):** Achieves significant speedup (>1.25x) over the PyTorch Eager baseline by leveraging aggressive operator fusion and specialized hardware units like Tensor Cores (FP16/BF16). **5 (Medium):** Performance effectively matches the baseline ($1.0 \times \pm 5\%$) in the majority of scenarios as overheads negate gains or the compiler defaults to standard library calls. **0 (Low):** Suffers performance regression (<0.9x) or high latency due to excessive compilation overhead, lack of kernel optimization, or inefficient memory management
Arch. Support	**10 (High):** Demonstrates broad compatibility, successfully compiling and executing all tested Synthetic and SOTA Large Language Model architectures out-of-the-box without intervention. **5 (Medium):** Offers partial support; successfully processes simpler synthetic blocks but fails on specific complex SOTA layers (e.g., Transformers) or vice versa, necessitating graph breaks or partial execution. **0 (Low):** Incompatible with tested architectures; fails to compile valid graphs, resulting in frequent crashes or an inability to generate executable kernels for the target model
Portability	**10 (High):** Fully device- and framework-agnostic; enables execution across all hardware without lock-in to proprietary drivers or specific training frameworks. **5 (Medium):** Hardware-agnostic (supports CPU/TPU/GPU) but framework-constrained; strictly tied to specific workflows (e.g., PyTorch JIT), preventing independent deployment. It also applies to software tied to specific tool versions, which hampers compatibility with newer software such as CUDA versions. **0 (Low):** Hardware locked; relies exclusively on proprietary libraries tailored for specific hardware architecture, such as NVIDIA's, offering zero portability to other hardware vendors
Input Flexibility	**10 (High):** Seamlessly handles dynamic input shapes via automatic fallbacks to eager execution or specialized pre-built engines, requiring no user intervention. **5 (Medium):** Requires manual constraint management; users must explicitly handle host-to-device memory bindings or perform model conversions to accommodate inputs. **0 (Low):** Rigid and static; performance or execution capability is strictly limited to fixed input dimensions defined ahead of time, struggling with dynamic variations
Productivity	**10 (High):** *Out-of-the-box* agility; integrates via a single decorator/function with a model-agnostic approach and native fallbacks, requiring no more source code modification. **5 (Medium):** Effective for pre-supported models but rigid, lacking the flexibility to support arbitrary models or code without setup, requiring explicit export steps or engine building. **0 (Low):** Necessitates complex manual kernel writing, build systems, or manual graph rewriting and hardware-specific deployment knowledge

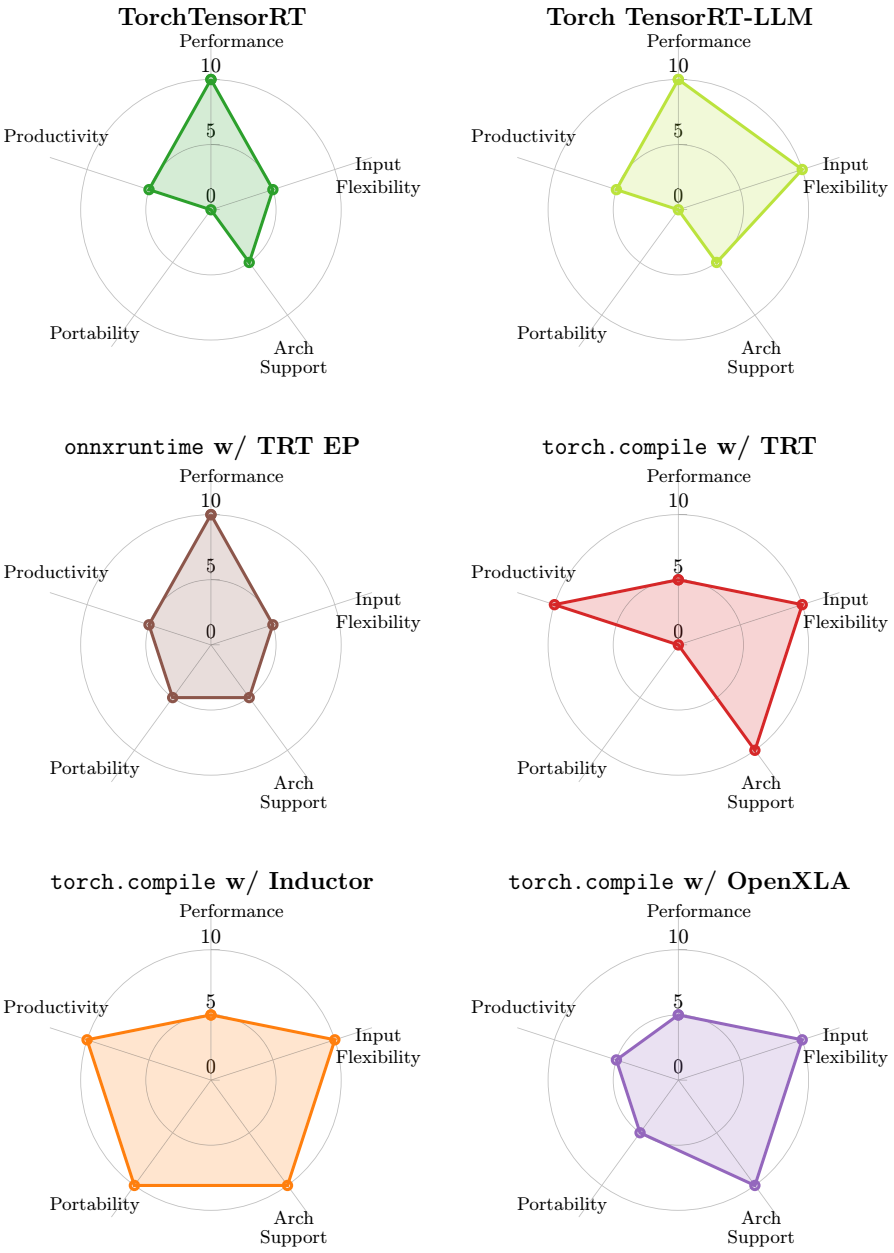

Fig. 14 Review of the MLC frameworks evaluated across key criteria

5.1 MLC evaluation summary

To synthesize the empirical findings into a summarized infographic (Fig. 14), we have rated each MLC workflow across five key criteria: Performance, Productivity, Portability, Input Flexibility, and Architecture Support. A quantitative rubric defining the scoring scale for each dimension (see Table 11) has also been provided.

It is important to note that this discretized scoring condenses complex performance behaviors into three distinct tiers, which inevitably involves some simplification. We assigned scores based on the presence of substantial performance gains relative to the PyTorch baseline, acknowledging that overhead may cause regressions on smaller workloads. As this high-level summary cannot capture every nuance, we refer readers to the quantitative plots across Sect. 4 for granular performance data across specific scenarios.

6 Conclusions

This work meets its three main goals toward 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–P3 characteristics are evaluated.

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 provides general guidelines to help developers select an MLC strategy aligned with specific P3 priorities and deployment constraints, and it showcases the benefits of each approach and how they affect the P3 trade-offs.

While relevant on their own right, the exclusive usage of the NVIDIA RTX 4090 and 5090 limits the conclusions to consumer-grade GPUs. While this choice reflects a common and cost-effective deployment setting, data center accelerators (e.g., NVIDIA A/H series) differ materially in VRAM capacity, memory subsystem behavior, and, in multi-GPU configurations, interconnect and partitioning features. Consequently, the reported P3 evaluation trade-offs should be interpreted as representative of RTX-class deployments.

Further research will also evaluate the new Modular platform against the incumbents studied in this work. Modular's MAX engine leverages its own programming language, Mojo, based on MLIR, to create a unified, hardware-agnostic deployment stack from the programming language to the serving layer. First, NVIDIA GPU support was added to the existing CPU, and, more recently, full support for both NVIDIA and AMD GPUs within a single container was released [31]. The proposed research would benchmark MAX as an offline compiler by directly comparing its performance and P3 capabilities against established workflows. To test MAX's portability claims, the hardware testbed would be expanded. In addition to

the current NVIDIA RTX 4090/5090 pairing, the study would also incorporate a similarly powerful AMD Radeon RX 7900 XTX. This would enable a direct comparison of MAX's performance against native PyTorch on AMD hardware, as well as on NVIDIA hardware and its CUDA ecosystem, assessing performance, productivity, and opening new avenues using 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 a GPU because, after the TensorRT-RTX engine file is built, one-shot JIT compilation selects the appropriate GPU kernels for the target machine. The whole compilation process is expected to be faster than the TensorRT baseline. Currently, it is available via its own Python API and an `onnxruntime` backend [32]. Further research would include using `onnxruntime` with TensorRT-RTX and its Python API as two distinct workflows to evaluate whether there are performance improvements on RTX machines, or whether its main advantage is a lighter environment for RTX machines. TensorRT-LLM is not supported by TensorRT-RTX at the moment.

Acknowledgements This work was partly supported by Grant PID2022-136315OB-I00 funded by MCIN/AEI/10.13039/501100011033/ and by "ERDF A way of making Europe", EU, and also partly by the action 23039/GERM/25, funded by FSRM/10.13039/100007801.

Author Contributions Conceptualization, A. C. and J.M.G.; Methodology, G.B. and J.M.G.; Software, A.C.; Validation, A.C., G.B. and J.M.G.; Investigation, A.C., G.B. and J.M.G.; Writing—original draft, A.C.; Writing—review & editing, A.C., G.B. and J.M.G.; Supervision, G.B. and J.M.G.; Project administration, G.B. and J.M.G.; Funding acquisition, G.B. and J.M.G.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Data Availability No datasets were generated or analyzed during the current study.

Declarations

Conflict of interest The authors declare no conflict of interest.

References

1. Li M, Liu Y, Liu X, Sun Q, You X, Yang H, Luan Z, Gan L, Yang G, Qian D (2021) The deep learning compiler: a comprehensive survey. IEEE Trans Parallel Distrib Syst 32(3):708–727. 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 CK, Maher B, Pan Y, Puhrsch C, Reso M, Saroufim M, Siraichi MY, 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 (2024) 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. https://doi.org/10.1145/3620665.3640366
3. ONNX Community: Open Neural Network Exchange (ONNX) (2025). https://onnx.ai/. Accessed 08 June 2025
4. ONNX Runtime Team: ONNX Runtime Documentation. https://onnxruntime.ai/docs/. Accessed 08 June 2025 (2025)
5. NVIDIA Corporation: NVIDIA TensorRT SDK (2025). https://developer.nvidia.com/tensorrt. Accessed 07 June 2025
6. NVIDIA Corporation: NVIDIA TensorRT Documentation (2025). https://docs.nvidia.com/deeplearning/tensorrt/latest/index.html. Accessed 07 June 2025
7. Zhou Y, Yang K (2022) Exploring TensorRT to Improve Real-Time Inference for Deep Learning. In: 2022 IEEE 24th International Conference on High Performance Computing & Communications; 8th International Conference on Data Science & Systems; 20th International Conference on Smart City; 8th International Conference on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys), pp. 2011–2018. https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00299
8. NVIDIA Corporation: NVIDIA/TensorRT-LLM GitHub Repository (2025). https://github.com/NVIDIA/TensorRT-LLM. Accessed 09 June 2025
9. Elmeleegy A, Comly N, Johnsen T (2025) 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 23 June 2025
10. NVIDIA Corporation: PyTorch Backend—TensorRT-LLM. https://nvidia.github.io/TensorRT-LLM/torch.html. Accessed 09 June 2025 (2025)
11. Sabne A (2020) XLA: Compiling Machine Learning for Peak Performance
12. OpenXLA Community: XLA - OpenXLA Project. https://openxla.org/xla. Accessed 09 June 2025
13. Lattner C, Amini, M, Bondhugula U, Cohen A, Davis A, Pienaar J, Riddle R, Shpeisman, Vasilache N, Zinenko O (2021) MLIR: scaling compiler infrastructure for domain specific computation. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp 2–14. 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 June 2025
15. Liu H-IC, Brehler M, Ravishankar M, Vasilache N, Vanik B, Laurenzo S (2022) TinyIREE: an ML execution environment for embedded systems from compilation to deployment. IEEE Micro 42(5):9–16. 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 June 2025
17. The IREE Authors: IREE GitHub Repository (2019). https://github.com/iree-org/iree

18. OpenXLA: OpenXLA/StableHLO GitHub Repository. https://github.com/openxla/stablehlo. Accessed 30 March 2026 (2026)
19. Apache Software Foundation: Apache TVM Official Website. https://tvm.apache.org/. Accessed 09 June 2025
20. Chen T, Moreau T, Jiang Z, Zheng L, Yan E, Cowan M, Shen H, Wang L, Hu Y, Ceze L, Guestrin C, Krishnamurthy A (2018) 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
21. Apache Software Foundation: Apache TVM Overview. https://tvm.apache.org/docs/get_started/overview.html. Accessed 09 June 2025
22. mlc.ai: Introduction to Machine Learning Compilation. https://mlc.ai/chapter_introduction/index.html. Accessed 09 June 2025
23. Zhang P, Zeng G, Wang T, Lu W (2024) TinyLlama: An Open-Source Small Language Model. https://arxiv.org/abs/2401.02385
24. TinyLlama: TinyLlama/TinyLlama-1.1B-Chat-v1.0 Hugging Face page (2024). https://huggingface.co/TinyLlama/TinyLlama-1.1B-Chat-v1.0. Accessed 15 June 2025
25. 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 CC, 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 PS, 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 EM, Subramanian R, Tan XE, Tang B, Taylor R, Williams A, Kuan JX, Xu P, Yan Z, Zarov I, Zhang Y, Fan A, Kambadur M, Narang S, Rodriguez A, Stojnic R, Edunov S, Scialom T (2023) Llama 2: Open Foundation and Fine-Tuned Chat Models. https://arxiv.org/abs/2307.09288
26. Meta: meta-llama/Llama-2-7b-chat-hf Hugging Face page (2023). https://huggingface.co/meta-llama/Llama-2-7b-chat-hf. Accessed 15 June 2025
27. NVIDIA Corporation: Triton Inference Server (2025). https://catalog.ngc.nvidia.com/orgs/nvidia/containers/tritonserver. Accessed 10 June 2025
28. Carmona-Martínez A.: Evaluation and characterization of machine learning compilers for AI inference on GPU. Master's thesis, Universidad de Murcia (2025)
29. 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 June 2025
30. Hugging Face Team: Optimize inference using torch.compile(). https://huggingface.co/docs/transformers/v4.38.1/en/perf_torch_compile. Accessed 24 June 2025
31. 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 June 2025 (2025)
32. NVIDIA Corporation: Example Deployment Using ONNX. NVIDIA TensorRT for RTX Documentation (2025). https://docs.nvidia.com/deeplearning/tensorrt-rtx/latest/installing-tensorrt-rtx/example-deployment.html