Studying the Overhead of `torch.compile()` in Large Language Model Inference

Andrzej Szablewski [1]

Abstract

This works investigates how the compilation toolkit in PyTorch can be applied to optimise Large Language Model inference. We begin with a brief description of the elements of the compilation suite, and their application to an multi-layer perceptron forward pass in an LLM. Furthermore, we conduct experiments assuming a scenario of inference on spot instances. Finally, we discuss how subtle compilation settings may contribute to additional generation latency. Our code is publicly available at https://github.com/TheRootOf3/torch-compile-benchmarks.

1. Introduction

Given the widespread use of Large Language Models and the costs related to their development and usage, it is essential to provide tools that optimise their computations. Those models usually process data in two distinct paradigms: training and inference. The optimisation goal for model training is maximising throughput, leading to more samples seen by the model in a given time unit, and hence quicker learning. On the other hand, optimising inference is equivalent to minimising the generation latency. Unlike model training, inference cannot be easily parallelised given the sequential nature of the decoding process from the transformer. In this work, we focus on the stack of `torch.compile()`, which allows optimisations applicable to arbitrary tensor programs implemented in PyTorch (Ansel et al., 2024).

Our main contributions are as follows:

- We study the overhead of `torch.compile()` for the specific case of LLM inference and its common optimisations (e.g. key-value cache).
- We share experimental results showing what workloads benefit from program compilation.
- We provide a brief explanation of different elements of the `torch.compile()` stack.

2. Background – LLM Inference

One of the key elements in decreasing the latency of LLM generations is key-value cache (KV cache) (Pope et al., 2022). To decrease the latency of a decoding step of a sequence of n tokens, the autoregressive property of decoder-based LLMs allows to re-use the previously computed key and value projections of the past $n-1$ tokens. Hence, this requires only computing the query-key-value projections of the nth token and its attention scores with the previous tokens.

LLM inference is usually divided into a *prefill* and a *decode* stage (Pope et al., 2022). The former involves processing the user prompt, by progressively computing and caching corresponding key-value pairs. The latency of a single step of prefill depends on the user prompt length. While there exist approaches which re-use common prompt prefixes (e.g. system messages in chatbots like in Hydragen (Juravsky et al., 2024)), the stage is difficult to speed-up in an arbitrary way. Furthermore, the decoding stage decodes tokens sequentially, one by one following the autoregressive nature of LLMs, making it difficult to optimise. Importantly, both workloads differ in the shape of data that is passed through the model. While the prefill involves operations on the inputs which can be batched and accelerated in hardware, decode always processes a single input token. However, the self-attention mechanism in decode computes over to the entirety of cached keys and values, which can result in a large number of computations (Vaswani et al., 2023).

Finally, one way to reduce the cost of LLM serving is through using spot instances, usually available at much lower price (Miao et al., 2023). Nevertheless, these compute platforms are preemptible, which results in more frequent initialisations of LLM servers including the compilation of computation graphs. Frequent optimisations incur an overhead, which in some scenarios may outweigh the benefits of the compilation.

Word count: 2491, excluding abstract, tables and captions (using `texcount`). [1]Department of Computer Science and Technology, University of Cambridge, Cambridge, UK. Correspondence to: Andrzej Szablewski <as3623@cam.ac.uk>.

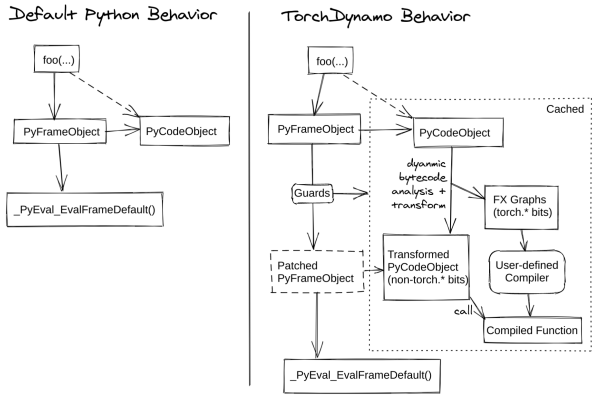


Figure 1. Diagram showing the internal workings of TorchDynamo. Note the dynamic inspection of the Python bytecode, extraction of the FX graph, its further compilation and call from the transformed bytecode. Diagram from the official PyTorch documentation.

3. The `torch.compile()` Stack

In a default, *eager* execution mode, PyTorch dynamically executes corresponding operations and, if needed, creates a backwards computation graph. While this provides developers and researchers with a significant amount of freedom during designing and debugging their programs, it significantly misses on potential speed-ups coming from algebraic optimisation, operator fusion or reordering. Hence, a *graph* mode has been introduced, allowing graph representation of PyTorch programs and subsequent optimisations for target hardware.

The stack of `torch.compile()` tools allows just-in-time (JIT) compilation of arbitrary Python code into optimised computation kernels. A significant size and scope of the project is dictated by the number of challenges it needs to address. Firstly, it can trace arbitrary Python code, which is a large improvement when compared to TensorFlow (Abadi et al., 2016) and the previous attempts of optimising PyTorch programs through TorchScript[1]. Those approaches use domain specific languages (DSL) with static typing and further limitations, making it difficult to translate programs from the eager computation model. Furthermore, the compilation suite also supports views and handles mutations, as opposed to JAX (Frostig et al.). Finally, `torch.compile()` supports a vast range of hardware accelerators through multiple layers of intermediate representations and a decoupled optimisation backend.

The JIT compilation occurs only after data is fed into a PyTorch program, allowing data-specific optimisations such as those dependent on tensor shapes. Initially, a Python tracer captures PyTorch operations during runtime. It utilizes the CPython Frame Evaluation API[2] to dynamically convert Python bytecode into a PyTorch FX graph intermediate representation, enabling further optimisation. Next, another tool is used to trace both the forward and backward passes of a model. This allows further lowering of the graph into the native ATen IR, and architecture agnostic joint optimisations. Finally, the program is fed into a backend compiler, which turns the incoming instructions into hardware-specific IR and compiles it into optimised code. It supports multiple backend targets, including GPUs and CPUs, by generating optimised C++ and CUDA kernels with Triton.

3.1. TorchDynamo – Python Code Tracer

TorchDynamo is the key factor contributing to making `torch.compile()` model- and application-agnostic. By Just-In-Time evaluating the bytecode in a CPython frame, it builds an FX graph of supported PyTorch operations, and substitutes the original bytecode with calls to functions representing version of the graph compiled by TorchInductor. The optimised code resulting from the extracted graph is cached and re-used to speed up the execution process. However, the optimisations often depend on tensor shapes (e.g., leveraging alignment with GPU tile dimensions in matrix multiplication operations). Furthermore, TorchDynamo traces linearly. However, if the traced code has branches, the execution path can change with the progression of the program. To provide soundness, TorchDynamo uses the concept of guards. This mechanism automatically binds the compiled functions with a set of conditions. If they are not met, the code will be re-traced and recompiled. Hence, it is important to note that efficient PyTorch programs should minimise the number of possible recompilations by eliminating redundant branching. On the other hand, if tensor shapes are often changing dynamically, TorchDynamo will capture the graph for the corresponding operations and populate it with symbolic shapes. This however, may lead to diminishing returns of compilation due to the need for more abstract code.

Although TorchDynamo can trace arbitrary Python code, some of the Python operations such as `print` are not supposed to be added into FX graphs. TorchDynamo realises completeness by implementing graph breaks whenever it encounters unsupported operations. A graph break results in finishing the tracing of the first graph, executing the bytecode operations, and initiating capturing the remaining instructions into another graph. Graph breaks are detrimental to optimisation gains because they do not allow joint compilation of the captured graphs. While in some cases it is impossible to eliminate graph breaks, they should be avoided.

[1] Link to TorchScript documentation.

[2] Link to PEP 523.

3.2. AoT Autograd – Forward-Backward Graph Tracer

The Ahead-of-Time (AoT) Autograd is another tracer, which is responsible for lowering of FX graphs, generation of the backwards graphs and initial graph transformations. To decrease the compilation complexity, it converts FX graphs representing all available PyTorch operations into a limited set of $\sim$ 200 ATen operations, which constitute the native PyTorch Intermediate Representation (IR).

3.3. TorchInductor – Compilation backend

The resulting ATen IR of a compiled function of module is fed into TorchInductor, which is the compilation backend. TorchInductor optimisations are mainly grouped into: 1) operator fusion (vertical and horizontal), 2) out of order execution, and 3) automatic work placement when numerous hardware is used. PyTorch programs usually involve consecutive operations on propagated operands. In the eager mode of execution, each operation involves sequentially loading operands, performing the computation and storing the result. Vertical operator fusion allows to load the operands once, and perform a number of computations, hence eliminating the need for continuous access a relatively slower memory. On the other hand, horizontal fusion batches computations for better hardware use (e.g. turning batch matrix multiplications into grouped GEMM operations). In this work, we focus on compilations to C++ OpenMP for CPU architectures.

4. Methodology

To gain deeper understanding of the usefulness of `torch.compile()` for LLM inference, we initially investigate the properties of resulting FX Graphs (TorchDynamo) and their compilations (TorchInductor) separately. First, we profile the execution of eager and compiled instances of an LLM to understand how different ATen operations contribute to the computation time. Furthermore, we study the case of compilation of the Llama3 multi-layer perceptron (MLP). In the main part of experiments, we investigate the latency of `torch.compile()` in a particular setting, when a model is initialised on a spot instance. We study how the workload differences between prefill and decode affect the compilation process and its efficiency. The measured compilation properties include the number of graph breaks, number of recompilations, compilation time and execution time.

In the prefill+decode experiments we feed the model with a constant number of prefill tokens and always generate a constant number of tokens. To decrease the decoding latency, we use a a KV Cache, which is initialised and memory-allocated before the prefill stage. We ensure that that the cache is always sufficiently large for all generated tokens.

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
aten::addmm	71.13%	236.799ms	74.79%	248.970ms	5.187ms	48
aten::copy_	6.69%	22.273ms	6.69%	22.273ms	168.732us	132
ProfilerStep*	4.38%	14.591ms	100.00%	332.889ms	332.889ms	1
aten::bmm	2.87%	9.553ms	2.88%	9.574ms	398.919us	24
aten::gelu	2.50%	8.330ms	2.50%	8.330ms	694.156us	12
aten::mm	2.15%	7.146ms	2.15%	7.147ms	7.147ms	1
aten::add	1.63%	5.417ms	1.63%	5.417ms	216.681us	25
aten::_softmax	1.54%	5.125ms	1.54%	5.125ms	427.048us	12
aten::mul	1.31%	4.358ms	1.47%	4.886ms	407.193us	12
aten::native_layer_norm	0.95%	3.151ms	1.06%	3.542ms	141.694us	25
aten::view	0.60%	1.981ms	0.60%	1.981ms	9.434us	210
aten::linear	0.45%	1.507ms	78.15%	260.160ms	5.309ms	49
aten::masked_fill_	0.42%	1.403ms	0.42%	1.403ms	116.912us	12
aten::eq	0.36%	1.196ms	0.42%	1.415ms	117.892us	12
aten::expand	0.28%	921.331us	0.36%	1.203ms	10.025us	120
aten::clone	0.25%	840.407us	3.61%	12.028ms	200.473us	60
aten::empty	0.25%	825.954us	0.25%	825.954us	5.942us	139
aten::transpose	0.25%	819.778us	0.33%	1.113ms	10.214us	109
aten::matmul	0.24%	795.440us	7.91%	26.340ms	1.054ms	25
aten::as_strided	0.22%	734.778us	0.22%	734.778us	2.296us	320

Self CPU time total: 332.889ms

Figure 2. Profiler output after running GPT-2 in eager mode.

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls
aten::addmm	63.44%	234.079ms	68.88%	254.173ms	5.295ms	48
aten::copy_	5.21%	19.223ms	5.21%	19.223ms	400.477us	48
CompiledFunction	3.27%	12.053ms	98.87%	364.808ms	364.808ms	1
aten::bmm	3.03%	11.173ms	3.03%	11.199ms	466.611us	24
aten::mm	2.36%	8.708ms	2.36%	8.709ms	8.709ms	1
graph_0_cpp_fused_gelu_23	0.68%	2.500ms	0.68%	2.500ms	2.500ms	1
ProfilerStep*	0.65%	2.389ms	100.00%	368.908ms	368.908ms	1
graph_0_cpp_fused_gelu_29	0.58%	2.128ms	0.58%	2.128ms	2.128ms	1
graph_0_cpp_fused_add_native_layer_norm_native_layer...	0.55%	2.034ms	0.55%	2.034ms	2.034ms	1
graph_0_cpp_fused__softmax_clone_eq_masked_fill_mul_...	0.54%	1.994ms	0.54%	1.994ms	1.994ms	1
graph_0_cpp_fused_gelu_17	0.51%	1.882ms	0.51%	1.882ms	1.882ms	1
graph_0_cpp_fused__softmax_clone_eq_masked_fill_mul_...	0.50%	1.846ms	0.50%	1.846ms	1.846ms	1
graph_0_cpp_fused__softmax_clone_eq_masked_fill_mul_...	0.49%	1.806ms	0.49%	1.806ms	1.806ms	1
graph_0_cpp_fused__softmax_clone_eq_masked_fill_mul_...	0.47%	1.743ms	0.47%	1.743ms	1.743ms	1
graph_0_cpp_fused_gelu_47	0.47%	1.740ms	0.47%	1.740ms	1.740ms	1
graph_0_cpp_fused_clone_43	0.46%	1.712ms	0.46%	1.712ms	1.712ms	1
graph_0_cpp_fused__softmax_clone_eq_masked_fill_mul_...	0.46%	1.709ms	0.46%	1.709ms	1.709ms	1
graph_0_cpp_fused_gelu_35	0.44%	1.626ms	0.44%	1.626ms	1.626ms	1
graph_0_cpp_fused_gelu_11	0.44%	1.621ms	0.44%	1.621ms	1.621ms	1
graph_0_cpp_fused__softmax_clone_eq_masked_fill_mul_...	0.44%	1.620ms	0.44%	1.620ms	1.620ms	1

Self CPU time total: 368.908ms

Figure 3. Profiler output after running a compiled GPT-2. Note the compilation took place during warmup runs executed beforehand.

In all experiments, we study a 1B version of the Llama3.2 model (Grattafiori et al., 2024) implemented in the *transformers*[3] library. In addition, we use the GPT2 (Radford et al.) model for profiling PyTorch operations (Figures 2 and 3). As sample model inputs we use a manually-crafted list of prompts and longer, automatically-generated paragraphs of text. To mitigate the inconsistency of results caused by potential caching of program data, we follow the official benchmarking suggestions from PyTorch documentation, including running PyTorch programs with the `TORCHINDUCTOR_FORCE_DISABLE_CACHES=1` flag.

5. Results

5.1. Eager vs Compiled Profiling

In the first experiment, we profile single-token computations through eager and compiled instances of GPT-2. As shown in Figure 2, more than 71% of time is spent in add-multiply operations. The remaining time is spent across numerous ATen dynamically executed operations such as `gelu` (12 calls), other matmul operations (`bmm` – 24 calls, `mm` – 1 call, `mul` – 12 calls). On the other hand, the compiled version (Figure 3) contains a number of dynamically optimised functions, such as `graph_0_cpp_fused_gelu_22`. Interestingly, `torch.compile()` results in several copies of the same fused operation, each per layer. In addition, the compiled version of the model uses less function calls, since

[3] Link to the GitHub repository of the Transformers library.

```python
def forward(self, x):
    down_proj = self.down_proj(self.act_fn(self.gate_proj(x)) * self.up_proj(x))
    return down_proj
```

Figure 4. PyTorch code of Llama MLP forward pass. Note three linear projection layers, an activation function (SiLU) and an element-wise multiplication.

```
permute_9: "f32[2048, 8192]" = torch.ops.aten.permute.default(arg10_1, [1, 0]);  arg10_1 = None
view_19: "f32[8, 2048]" = torch.ops.aten.view.default(mul_10, [8, 2048])
mm_4: "f32[8, 8192]" = torch.ops.aten.mm.default(view_19, permute_9);  view_19 = permute_9 = None
view_20: "f32[1, 8, 8192]" = torch.ops.aten.view.default(mm_4, [1, 8, 8192]);  mm_4 = None
sigmoid: "f32[1, 8, 8192]" = torch.ops.aten.sigmoid.default(view_20)
(variable) permute_10: Any torch.ops.aten.mul.Tensor(view_20, sigmoid);  view_20 = sigmoid = None
permute_10: "f32[2048, 8192]" = torch.ops.aten.permute.default(arg11_1, [1, 0]);  arg11_1 = None
view_21: "f32[8, 2048]" = torch.ops.aten.view.default(mul_10, [8, 2048]);  mul_10 = None
mm_5: "f32[8, 8192]" = torch.ops.aten.mm.default(view_21, permute_10);  view_21 = permute_10 = None
view_22: "f32[1, 8, 8192]" = torch.ops.aten.view.default(mm_5, [1, 8, 8192]);  mm_5 = None
mul_12: "f32[1, 8, 8192]" = torch.ops.aten.mul.Tensor(mul_11, view_22);  mul_11 = view_22 = None
permute_11: "f32[8192, 2048]" = torch.ops.aten.permute.default(arg12_1, [1, 0]);  arg12_1 = None
view_23: "f32[8, 8192]" = torch.ops.aten.view.default(mul_12, [8, 8192]);  mul_12 = None
mm_6: "f32[8, 2048]" = torch.ops.aten.mm.default(view_23, permute_11);  view_23 = permute_11 = None
view_24: "f32[1, 8, 2048]" = torch.ops.aten.view.default(mm_6, [1, 8, 2048]);  mm_6 = None
```

Figure 5. FX Graph representation of the MLP forward pass using ATen IR operations. Note the three matrix multiplication operations (`aten.mm`) corresponding to 3 linear projections.

it replaces many of the ATen operations with the compiled subroutines.

5.2. Case-Study of MLP Compilation

We investigate how `torch.compile()` processes a multi-layer perceptron. The MLP in the Llama3.2-1B model uses 3 linear layers and a SiLU activation function ($silu(x) = x * \sigma(x)$, where $\sigma(x)$ is a logistic sigmoid function) (Figure 4). The linear layers in the model are as follows: *up, gate* – upscaling from 2048 into 8192, and *down* – downscaling back to 2048. Initially, TorchDynamo represents this single line of PyTorch code as a FX Graph in the ATen IR, presented in Figure 5. For example, the SiLU activation is performed in the following way: `mm4` represents the output of the *gate* projection. It is followed by the sigmoid operation, which computes $\sigma(x)$, and finally the element-wise multiplication of $\sigma(x) * x$. Further operations involve the *up* projection (`mm5`), the element-wise multiplication (`mul12`) and the final projection back to 2048 dimensional vector (`mm6`).

Such representation is passed to TorchInductor, which implements vertical fusion of the SiLU computation. The modified, readable Python representation of the MLP is presented in Figure 6. While most of the operations involve accelerator-specific matrix multiplication (`extern_kernels.mm`), TorchInductor reduces the number of memory reads through fusing the computation of SiLU and reordering the element-wise multiplication between SiLU outputs and the *up* projections. The operations are replaced with a call to the `cpp_fused_mul_silu_7` C++ OpenMP kernel. The kernel is implemented directly in the generated C++ code, presented in Figure 7. For each group of elements the CPU can process in parallel, it loads previously computed projections of *gate* and *up*

```
# Topologically Sorted Source Nodes: [linear_4], Original ATen: [aten.mm]
extern_kernels.mm(reinterpret_tensor(buf24, (8, 2048), (2048, 1), 0), reinterpret_tensor(arg10_1, (2048, 8192), (1, 2048), 0), out=buf25)
del arg10_1
buf26 = empty_strided_cpu((8, 8192), (8192, 1), torch.float32)
# Topologically Sorted Source Nodes: [linear_5], Original ATen: [aten.mm]
extern_kernels.mm(reinterpret_tensor(buf24, (8, 2048), (2048, 1), 0), reinterpret_tensor(arg11_1, (2048, 8192), (1, 2048), 0), out=buf26)
del arg11_1
buf27 = reinterpret_tensor(buf25, (1, 8, 8192), (65536, 8192, 1), 0); del buf25  # reuse
cpp_fused_mul_silu_7(buf27, buf26)
buf28 = reinterpret_tensor(buf24, (8, 2048), (2048, 1), 0); del buf24  # reuse
# Topologically Sorted Source Nodes: [down_proj], Original ATen: [aten.mm]
extern_kernels.mm(reinterpret_tensor(buf27, (8, 8192), (8192, 1), 0), reinterpret_tensor(arg12_1, (8192, 2048), (1, 8192), 0), out=buf28)
```

Figure 6. Fragment of the readable compiled Python code corresponding to the MLP forward pass. Note the use of the C++ kernel `cpp_fused_mul_silu_7`.

```
cpp_fused_mul_silu_7 = async_compile.cpp_pybinding(['float*', 'const float*'], '''
#include "/var/folders/zy/tf5h417916z3zcm160nrcxvc0000gn/T/torchinductor_aszab/tmpluswp7_8/vu/cvuvp4i7roujum4xemrfwnb3t4c5t3r3mi
extern "C"  void kernel(float* in_out_ptr0,
                        const float* in_ptr0)
{
    #pragma omp parallel num_threads(8)
    {
        int tid = omp_get_thread_num();
        {
            #pragma omp for
            for(int64_t x0=static_cast<int64_t>(0LL); x0<static_cast<int64_t>(65536LL); x0+=static_cast<int64_t>(8LL))
            {
                auto tmp0 = at::vec::Vectorized<float>::loadu(in_out_ptr0 + static_cast<int64_t>(x0), static_cast<int64_t>(8));
                auto tmp3 = at::vec::Vectorized<float>::loadu(in_ptr0 + static_cast<int64_t>(x0), static_cast<int64_t>(8));
                auto tmp1 = decltype(tmp0)(1)/(decltype(tmp0)(1) + tmp0.neg().exp());
                auto tmp2 = tmp0 * tmp1;
                auto tmp4 = tmp2 * tmp3;
                tmp4.store(in_out_ptr0 + static_cast<int64_t>(x0));
            }
        }
    }
}
''')
```

Figure 7. C++ OpenMP kernel for the fused multiplication and SiLU activation computation.

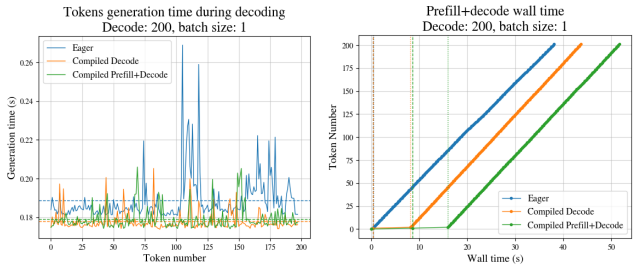
(lines 13-14), computes the sigmoid on the former (line 15) and performs the first element-wise multiplication of SiLU (line 16) and the second one with the *up* projection (line 17). Hence, it reuses the already loaded output of the sigmoig and SiLU in computing the outcomes of the element-wise operations and stores them only once.

5.3. Prefill+Decode: Eager vs Compiled per Token Latency

In this experiment we compare three scenarios: 1) full model in eager mode, 2) prefill stage inference in eager and compiling for decode, and finally 3) compiling before the prefill stage. We present detailed results in table 1, where we specifically focus on the prefill and first token latency, as well as the total generation duration. Notably, we observe that compiling before prefill results in two compilation periods: one for prefill and a recompilation for decode. This is clearly visible as the prefill+decode mode has overhead in both the prefill mode and the decode mode compared to others. While the overheads lead to longer generations for smaller batch sizes (1 and 4), the compilation brings the total time down when more prompts are combined. Furthermore, we observe a progressive gap in the eager and compiled decoding latency per token in Figures 8a, 9a and 10a. Moreover, Figures 9b and 10b show after how many tokens the compiled versions compensate the compilation time. Hence, we conclude it is beneficial to compile a model for the decode stage given large batches are used. On the other hand, a nuanced difference of when in the program the model is compiled may result in additional latency.

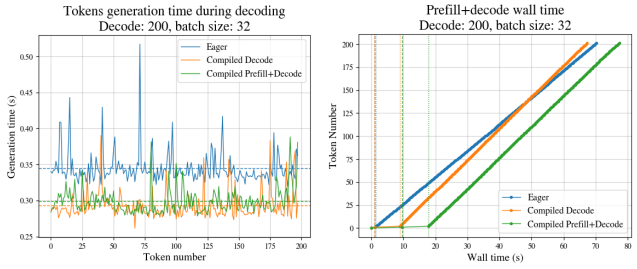
Batch Size	Mode	Prefill	Decode (first)	Decode (rest)	Decode/token (rest)	Total
1	Eager	**0.29**	**0.18**	37.58	0.18	**38.06**
	Compile (Decode)	0.30	7.88	35.44	0.17	43.63
	Compile (Prefill+Decode)	8.55	7.40	35.71	0.17	51.67
4	Eager	0.38	**0.34**	72.48	0.36	**73.22**
	Compile (Decode)	**0.34**	8.18	74.83	0.37	83.36
	Compile (Prefill+Decode)	8.39	7.72	71.50	0.35	87.62
32	Eager	1.11	**0.34**	68.68	0.34	70.14
	Compile (Decode)	**1.10**	7.81	58.38	0.29	**67.31**
	Compile (Prefill+Decode)	9.57	8.22	59.56	0.29	77.36
128	Eager	**4.78**	**0.99**	157.73	0.78	163.52
	Compile (Decode)	5.06	10.32	115.12	0.57	**130.51**
	Compile (Prefill+Decode)	12.78	8.41	113.69	0.56	134.89
256	Eager	11.12	**1.41**	269.44	1.34	281.98
	Compile (Decode)	**9.70**	8.75	177.98	0.88	**196.44**
	Compile (Prefill+Decode)	16.81	8.49	199.72	0.99	225.04

Table 1. Comparison between spot instance LLM initialisations: eager, compiled after prefill, and compiled before prefill. Time in seconds. We intentionally split the decode into the time taken to decode the first token (which results in additional recompilations), and the rest of tokens.



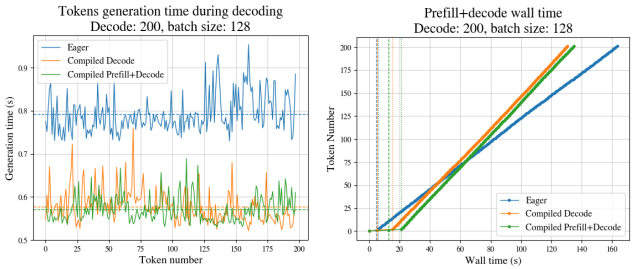
(a) (b)

Figure 8. Comparison of studied compilation modes with batch size 1. Horizontal lines in the left plot represent mean generation times. Vertical dashed lines in right plot represent the end of prefill stage, while dotted lines represent the end of first token generation.



(a) (b)

Figure 9. Comparison of studied compilation modes with batch size 32.



(a) (b)

Figure 10. Comparison of studied compilation modes with batch size 128.

5.4. Prefill+Decode: Compiling for Dynamic Prefill Shapes

While compiling to static shapes may result in better optimisations, it is not possible when using a model for prefill and decoding due to variable-length user inputs. Hence, as we observed in the previous experiment, models compiled before the prefill stage suffer two recompilation overheads: the first one during prefill and the second one for the first decoded token because the tensor shapes are changing. Therefore, we studied the causes of those recompilations in a scenario where initially the model is prefilled with 5 tokens, decodes one by one, then it is prefilled with 10 tokens for another round of decoding. Due to the presence of KV cache, the original model is initially already split into 3 subgraphs. Without the flag `dynamic=True`, there are 6 recompilations in total. The first compilation

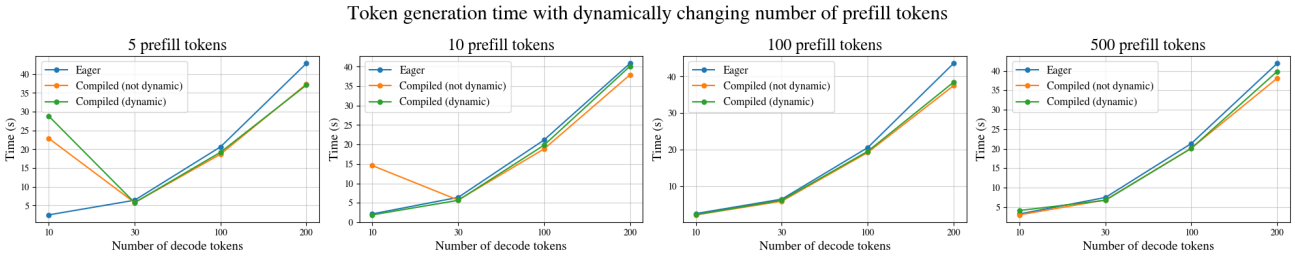


Figure 11. Token generation times under increasing prefill sequence size. Importantly, this process constitutes a single experiment, and hence the compiled model is not manually reset between generations.

"bakes in" constant tensor sizes for prefill of 5 tokens. When the model starts decoding, `torch.compile()` triggers 3 recompilations for different elements. Surprisingly, in the next prefill stage with 10 tokens, there are additional 3 recompilations triggered to introduce symbolically traced length of input token sequence. On the other hand, when the flag `dynamic=True` is used, there are only 3 recompilations during the decode step, resulting from the presence of previously unpopulated KV cache.

Figure 11 shows a scenario when the model generates 300 tokens given increasing number of prefill tokens. We show that when the model is compiled with the explicit instruction to trace shapes symbolically (`dynamic=True`), it does not recompile when the shape of prefill tensor changes. At the same time, the generation performance is barely affected.

6. Conclusions

In this work, we focused on the application of `torch.compile()` stack to optimising LLM inference. The initial examples providing a brief overview of the toolkit allowed better understanding of different system components. Further case study showed how elements of an LLM are converted into optimised code. Finally, the study of inference on spot instances, provided useful insights into how to ensure minimal overheads related to compilation. One of the future direction involves investigating why commonly used computation patterns across the transformer get compiled into different functions (e.g. activations in MLP layers). While we managed to design initial experiments and pose a hypothesis, the limited time resources did not allow us to analyse and present the full results. Furthermore, we studied the inference using a single implementation. To provide a more comprehensive view of the field it is necessary to understand how commonly used inference servers such as vLLM (Kwon et al., 2023) or llama.cpp[4] handle compilation.

[4]Link to the Github repository of llama.cpp.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: A system for large-scale machine learning, May 2016. URL http://arxiv.org/abs/1605.08695. arXiv:1605.08695 [cs].

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 929–947, La Jolla CA USA, April 2024. ACM. ISBN 9798400703850. doi: 10.1145/3620665.3640366. URL https://dl.acm.org/doi/10.1145/3620665.3640366.

Frostig, R., Johnson, M. J., and Leary, C. Compiling machine learning programs via high-level tracing.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., et al. The Llama 3 Herd of Models, November 2024. URL http://arxiv.org/abs/2407.21783. arXiv:2407.21783 [cs].

Juravsky, J., Brown, B., Ehrlich, R., Fu, D. Y., Ré, C., and Mirhoseini, A. Hydragen: High-Throughput LLM Inference with Shared Prefixes, May 2024. URL http://arxiv.org/abs/2402.05099. arXiv:2402.05099 [cs].

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient Memory Management for Large Language Model Serving with PagedAttention, September 2023. URL http://arxiv.org/abs/2309.06180. arXiv:2309.06180 [cs].

Miao, X., Shi, C., Duan, J., Xi, X., Lin, D., Cui, B., and Jia, Z. SpotServe: Serving Generative Large

Language Models on Preemptible Instances, November 2023. URL http://arxiv.org/abs/2311.15566. arXiv:2311.15566 [cs].

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently Scaling Transformer Inference, November 2022. URL http://arxiv.org/abs/2211.05102. arXiv:2211.05102 [cs].

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language Models are Unsupervised Multitask Learners.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention Is All You Need, August 2023. URL http://arxiv.org/abs/1706.03762. arXiv:1706.03762 [cs].