CIDAS
Campus-Institut Data Science

Constantin Dalinghaus

What’s new with PyTorch?

Supercharging at the compiler level with PyTorch 2.0

Table of contents

What's PyTorch again?

- Framework for deep learning in python developed at Meta AI
- Based on and named after the Torch framework for lua
- Initially released in September 2016

What makes PyTorch so popular?

- PyTorch design philosophy
  - Principle 1: Usability over Performance
  - Principle 2: Simple Over Easy
  - Principle 3: Python First with Best In Class Language Interoperability
- Low barrier of entry and focus on user experience arguably what makes PyTorch this popular

PyTorch 2.0

- Seminal new release of PyTorch, released in March of 2023 (*PyTorch 2.0 — pytorch.org*)
- Major overhaul to the backend of PyTorch, while keeping the frontend undisturbed
- Added torch.compile(), a mechanism for ahead-of-time compilation
  - Allows for optimizations to computations at graph and operator level
  - Backend agnostic, introduces a smaller operator set to make backend development more accessible
  - Usually works out of the box with large performance benefits

Usability over performance?

- PyTorch 2.0 is focused on usability
- Very complex compilation features can be used with a single line of code

```python
# Define your model as usual
model = MyCoolModel()

# Compile the model
torch.compile(model)

# Train your model as usual
for X, y in train_data_loader:
  y_pred = model(X)
  # ...
```

Figure: Using torch.compile() is simple!

PyTorch 2.0: Backend overview

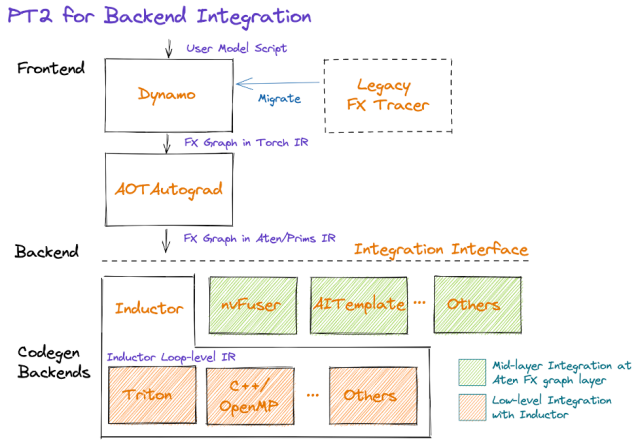


Figure: Source *PyTorch 2.0 — pytorch.org*

Torch Dynamo

- Python-level Just-In-Time (JIT) compiler
- Extracts an FX Graph, which can then be optimized with a custom backend
- Performs operator lowering

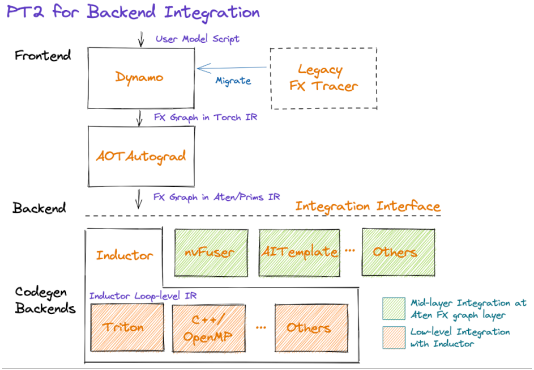


Figure: Source: *PyTorch 2.0 — pytorch.org*

AOT Autograd

- Autograd? Automatically determine the gradient of your model!
  - Model at this stage represented as graph
  - Autograd traverses the directed acyclic graph starting at the root node
  - Leaf nodes are processed according to the chain rule

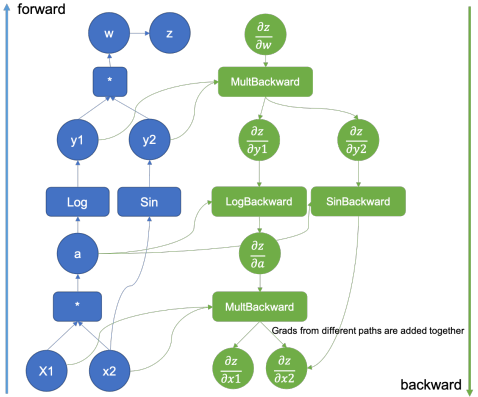


Figure: Source: *PyTorch 2.0 — pytorch.org*

AOT Autograd

- With AOT Autograd: Graph is compiled *ahead of time*
  - Only need to build the graph *once*
  - Graph structure and operations can be optimized

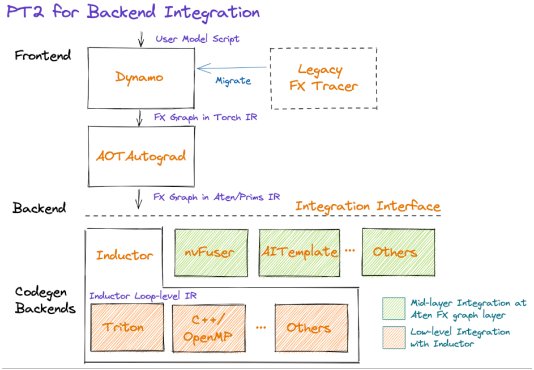


Figure: Source: *PyTorch 2.0 — pytorch.org*

Torch Inductor

- Compiles the low level graph to hardware specific kernel code
- For NVIDIA / AMD GPUs: OpenAI Triton
  - Triton is a DSL (domain specific language) for cuda kernel programming
  - Python-like, higher level but powerful cuda abstraction

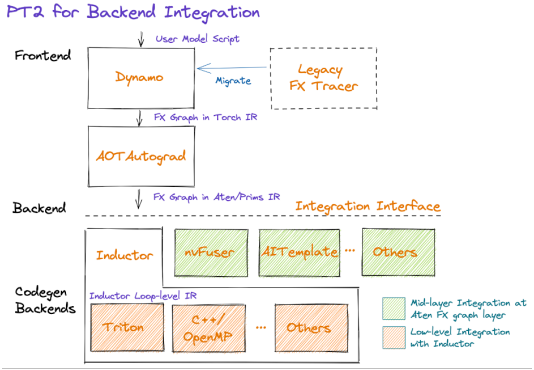


Figure: Source: *PyTorch 2.0 — pytorch.org*

PrimTorch

- A canonicalization of all ∼2000 PyTorch operators to a small set of ∼250 primitive operators
- Makes developing a custom backend for PyTorch much easier as only the smaller set of primitives needs to be implemented

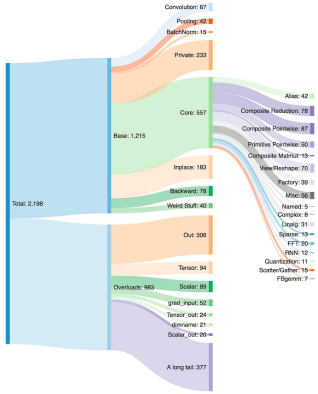


Figure: Source: *PyTorch 2.0 — pytorch.org*

Why benchmarking?

- Pytorch 2.0 is about improving performance => Performance benchmarking
  - Goal: Verify claims about performance speedups made by the PyTorch foundation

What do we compare?

- torch.compile() allows compilation to Inductor and Cudagraphs backends
  - Evaluate compiled performance vs performance in eager execution mode
  - Use the university HPC system for evaluation

```
# Don't compile - use eager execution
torch.compile(model, disable=True)

# Compile with cudagraphs backend
torch.compile(model, backend="cudagraphs")

# Compile with torch Inductor backend
torch.compile(model, backend="inductor")
```

Figure: Using torch.compile() is simple!

What do we measure?

- Train model for 1000 iterations
- Track metrics for these 1000 iterations
  - Time per iteration
  - Mean Flop Utilization (MFU)
    - Ratio of FLOPS actually performed and total number of FLOPS that the GPU die could have performed

How about model sizes?

- Measure LLM training performance at multiple scales
  - Small
    - 10.65M parameters
    - Trained on Shakespeare corpus
  - Medium
    - 127M parameters / "GPT2-small"
    - Trained on OpenWebText [Gokaslan and Cohen, *OpenWebText Corpus*]
  - Large
    - At least 175B parameters
    - **Out of reach for us due to hardware constraints!**

Raw results

- Execution time in ms, per iteration

	μ_{gpt2}	σ_{gpt2}	$\mu_{shakespeare}$	$\sigma_{shakespeare}$
Eager execution	3107,334	2,762	16,029	2,666
Compiled (Cudagraphs backend)	3098,297	0,956	16,828	3,935
Compiled (Inductor backend)	2640,227	5,773	12,191	1,556

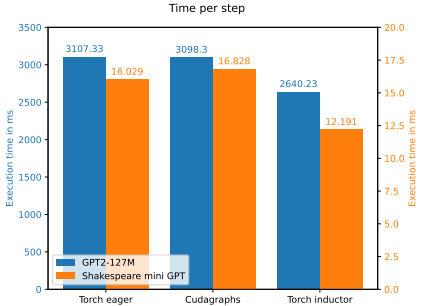
Raw results

- Mean Flop Utilization (mfu) in percent

	μ_{gpt2}	σ_{gpt2}	$\mu_{shakespeare}$	$\sigma_{shakespeare}$
Eager execution	43,337	0,000	22,951	0,853
Compiled (Cudagraphs backend)	43,463	0,000	21,916	0,653
Compiled (Inductor backend)	51,023	0,004	29,920	0,952

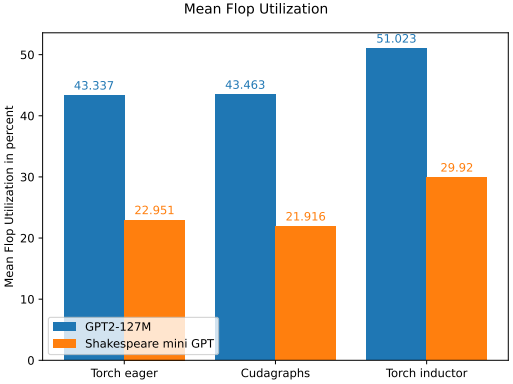
Interpretation: Iteration time

- Execution time per epoch decreases significantly when using compile with torch inductor
- The difference in iteration time is noticeable in large and small LLMs but more pronounced in smaller models



Interpretation: Mean Flop Utilization

- Compilation with torch inductor yields improvements in mfu in both training of both small and medium LLM



Other findings

- Mean Flop Utilization is very stable when training a larger model
  - More volatile when training a smaller model
  - Could mean there are other inefficiencies in the data preparation pipeline for small datasets
- Use of cudagraphs did not result in improvements in performance

Do these results align with claims made by the PyTorch foundation?

- Claims by the PyTorch foundation:
  - 51% faster on average
- Our results:
  - 16.25% speedup training for a medium sized LLM
  - 27.2% speedup when training a small LLM
- But:
  - Assuming mixed precision (bfloat16)
  - Using NVIDIA A100
  - PyTorch foundation measurements are across a more diverse set of models!

Implications for a business use case?

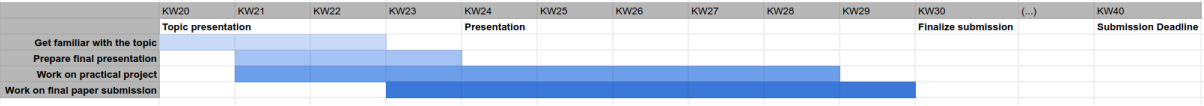
- Save double digit percentage of cost during model training
  - Futher efficiency gains by better *Mean Flop Utilization* on the GPU
- Given how easy it is to implement, there is hardly any reason not to use torch.compile(), especially when training at scale!
  - But: During development stick to eager execution as it creates more verbose error messages

Future testing? More architectures!

- Verify that ViTs yield similar results to LLMs
  - They share the same underlying architecture as LLMs
- More interestingly: Do testing on architectures which were not covered yet
  - Convolutional Neural Networks, LSTMs etc.

The timeline so far

- My presentation is early, so my personal deadline is set early in KW30
- Timeline formalized using Gantt chart



Questions and Feedback

References

Gokaslan, Aaron and Vanya Cohen. *OpenWebText Corpus*. http://Skylion007.github.io/OpenWebTextCorpus. 2019.

PyTorch 2.0 — pytorch.org. https://pytorch.org/get-started/pytorch-2.0/. [Accessed 14-05-2024].

Tillet, Philippe, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973. URL: https://doi.org/10.1145/3315508.3329973.