
Some Ideas on ML Systems Research
Jason Ansel, Horace He

Jason's thoughts on state of research

- Training is understudied in research compared to inference
  - Some much more interesting compiler problems in training
    - Less linear graphs with more choices
  - Training now taking >50% of flops at most companies and growing
  - AOTAutograd makes doing research on training much easier
- Benchmarking is often laughably bad
  - Way too few benchmarks in most papers
  - Results on BERT/ResNet often don't generalize
  - Often basic misconfigurations
    - Not enabling tf32/cudagraphs
  - Use our benchmarking infra!

What is Useful in Practice Not Coming From Systems Research

Triton:

- Probably the most successful ML systems project from academia.
- Its ideas have been extremely influential, with several imitation projects in both academia and industry.
- Unsuccessful in academia - rejected from ASPLOS/PLDI, only published in a workshop. Why?

What is Useful in Practice Not Coming From Systems Research

Triton:

- Probably the most successful ML systems project from academia.
- Its ideas have been extremely influential, with several imitation projects in both academia and industry.
- Unsuccessful in academia - rejected from ASPLOS/PLDI, only published in a workshop. Why?

FlashAttention

- One of the most impactful academic papers in ML systems
- Explores “how to speed up attention” from the computational perspective
- Coming from a ML researcher!

What are the main gaps?

There are two main goals when providing any abstraction (e.g. system)

1. Abstract away some details for the user.
    a. Virtual memory - abstracts location and management of memory.
    b. Array programming (numpy, PyTorch) - abstracts away the underlying kernels being called
2. Allow the user to program against your abstraction! (i.e. a programming model)
    a. Performance should be predictable!
    b. Your abstraction should not be leaky!

Often, academic systems focus too much on 1 and not enough on 2.

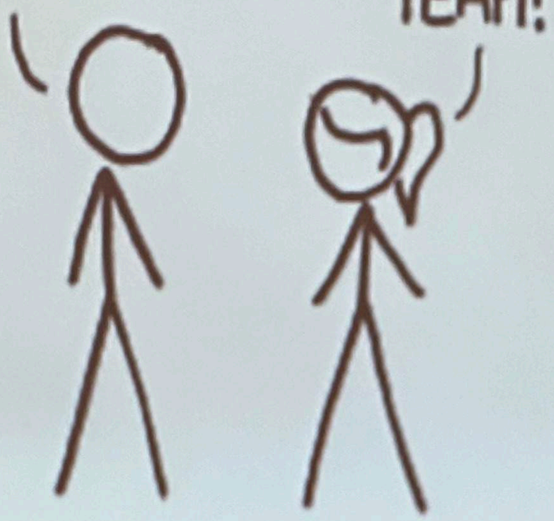
Abstractions

HOW ~~STANDARDS~~ PROLIFERATE:

(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)

SITUATION:
THERE ARE
14 things users need to understand

14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE automatic compiler
THAT COVERS EVERYONE'S
USE CASES.



SOON:

SITUATION:
THERE ARE
15 things users need to understand

Auto-vectorization is not a programming model

I think that the fatal flaw with the approach the compiler team was trying to make work was best diagnosed by T. Foley, who's full of great insights about this stuff: *auto-vectorization is not a programming model*.

The problem with an auto-vectorizer is that as long as vectorization can fail (and it will), then if you're a programmer who actually cares about what code the compiler generates for your program, you must come to deeply understand the auto-vectorizer. Then, when it fails to vectorize code you want to be vectorized, you can either poke it in the right ways or change your program in the right ways so that it works for you again. This is a horrible way to program; it's all alchemy and guesswork and you need to become deeply specialized about the nuances of a single compiler's implementation —something you wouldn't otherwise need to care about one bit.

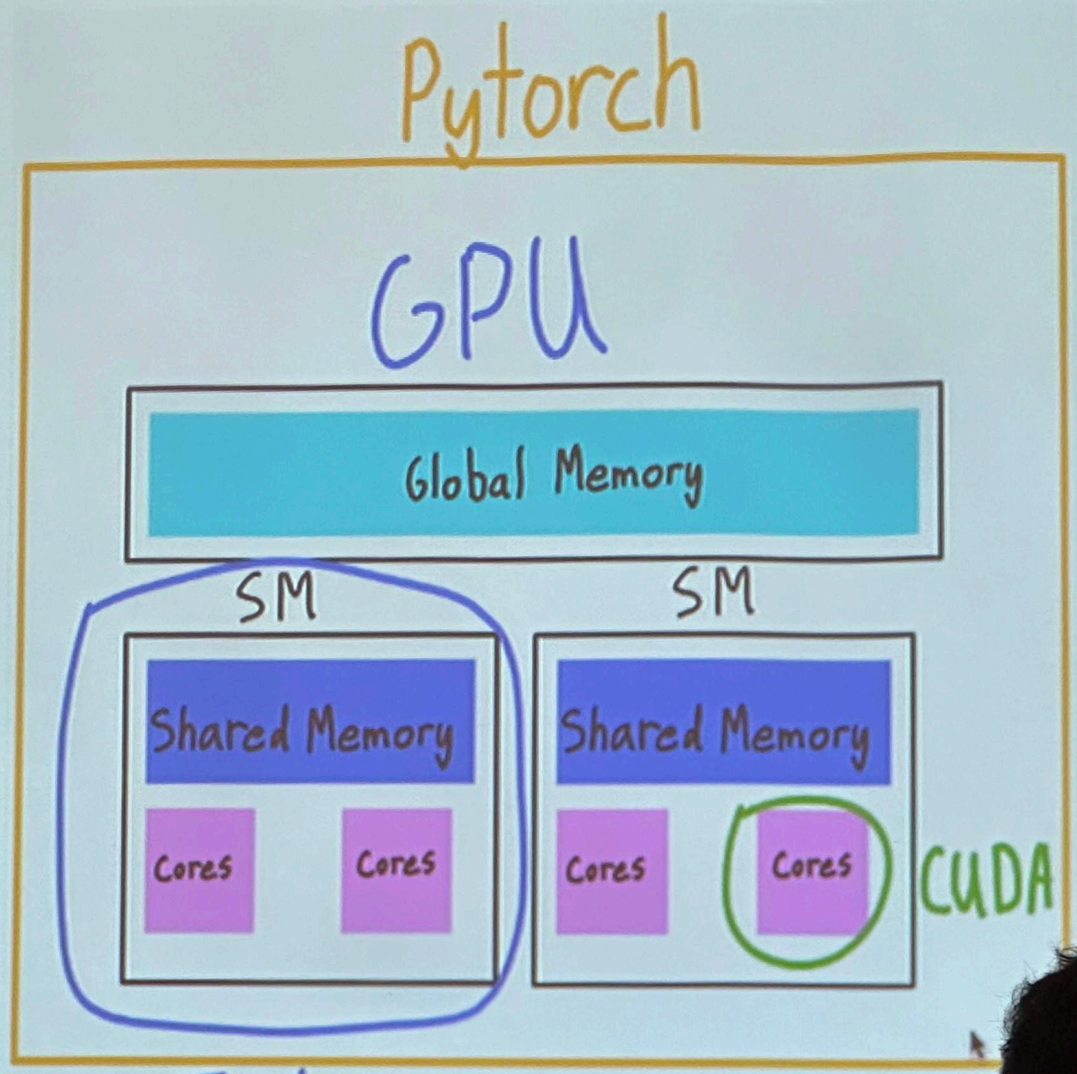
A “Sufficiently Smart Compiler” solves everything

```
perfectly_optimized_program = compile(program)
```

But, # of users >> # of compiler developers

Your prospective users are creative and smart!

Triton's Biggest Contribution is Not the System, but the Tile-Based Programming Model!



Triton handles the low-level details while providing enough flexibility!

It's easy-ish to take an existing Triton kernel and add a trivial change to it.

- Pointwise Epilogue
- Write to a signaling buffer when kernel is done
- Supported Jagged Tensor Dimensions
- Support Block-sparsity.

(WIP) FlexAttention

FlashAttention is an extremely important kernel, but folks often want to make modifications.

- PagedAttention
- Alibi Positional Bias
- Sliding Window Attention
- Jagged Sequence Lengths
- PrefixLM Models

Notably, although generating the flashattention kernel is hard, these modifications are easy!

Choices in the TorchInductor

Today: Local Autotuning

- Block sizes
  - "Coordinate descent tuner"
- Persistent/looped reductions
  - "Multi-kernel"
- Matmul/conv backend choice
  - triton vs cublas vs cutlass

Today: Heuristics

- Store vs recompute
- Fusions decisions
- Memory layouts
- Pattern matching
- Split reductions

Ideas:

- Global autotuning
  - Much larger search space
  - Interdependent choices
- ML cost models
- Reinforcement learning
- Apply LLMs to choices
- System to manage training data

Metrics:

- Performance
- Compile time

New backends

Integration points

- FX-graph
  - AOTAutograd for backwards
  - ~250 operators (tensor-level)
  - Python
- TorchInductor
  - ~50 operators (element/block-level)
  - Python
- Triton
  - LLVM/MLIR dialect
  - Today: AMD GPU/Intel GPU/MTIA/etc

Ideas

- New hardware
- New compilers/languages
- Apple Metal
- Pallas backend (TPUs)
- Halide backend
  - More directly compare Halide-style approach versus Triton
  - Current Halide-style backends struggle due to autotuning not scaling (TVM) or coverage issues (Hidet/nvFuser/NNC)

New compiler optimizations

- Approximation algorithms
- Grouped GEMMs
- Strassen-based GEMMs
- Improved Triton convolution/attention templates
- Model weight layout changes
- Direct codegen of matmuls
- Pattern matching replacements
- Your ideas!

Compile time becoming a bigger problem

- For larger training jobs
  - Large models can have huge graphs
  - Distributed
    - MTBF <1 hour
    - Time to restart a checkpoint becoming the bottleneck
    - Problem will become worse as scale up
- Some users of torch.compile are blocked on compile times

PyTorch distributed

- A huge open are of research, especially at larger scales
- Many people are building bespoke model-specific frameworks
- Cost of entry likely prohibitive for academics
- A theoretical model or simulator could be interesting research
  - Allows one to predict/explore different strategies without needing a massive cluster
- Empirically:
  - Small scales: compute bound (solved by DDP)
  - Medium scales: memory bound (solved by FSDP)
  - Large scales: communication bound (3D/pipeling/bespoke systems)
  - Extra-large scales: fault-tolerance bound

Profiling, debugging, and model understanding

- Fusions done by Inductor make profiles less understandable
- Understanding what the compiler did
- Why is this kernel slow?
- Where did the speedup come from?
- What causes this bug
  - Existing error minifier (similar to delta-debugging) could be improved
- Accuracy divergence debugging
- Better flow for fixing graph breaks
- Explain errors to users

A more expressive PyTorch

- Make it easier possible to generate new types of kernels
- Lambda reductions
- Kernel-level control flow
- Mixing triton/pytorch code better
- Custom dtypes

Profiling, debugging, and model understanding

- Fusions done by Inductor make profiles less understandable
- Understanding what the compiler did
- Why is this kernel slow?
- Where did the speedup come from?
- What causes this bug
  - Existing error minifier (similar to delta-debugging) could be improved
- Accuracy divergence debugging
- Better flow for fixing graph breaks
- Explain errors to users

Some examples of things where I'm not satisfied with current abstractions

- Parallelism, especially expressing things like pipeline parallelism.
- Fault Tolerance
- System for generating kernels at a level of abstraction between PyTorch and Triton
- Structured Sparsity in Kernels (JaggedTensors, etc.)
- Communication Collectives (i.e. communication across GPUs)
- Mixed Compute/Communication Kernels

Thanks for attending!

Can ask questions on pytorch.slack.com: #torchdynamo / #asplos24tutorial

Invite URL (valid for 7 days): https://tinyurl.com/pytorchslack