Computation Graph and Operator-level Optimization

Aaron Zhao, Imperial College London

Introduction

Computation graphs

We have looked a few algorithmic optimizations, and they are mainly ‘lossy’ optimizations. Most of them rely on the redundancy of NNs and also the recovering power of SGDs.

On the compiler stack, we also have the opportunity to issue a range of classic lossless optimizations.

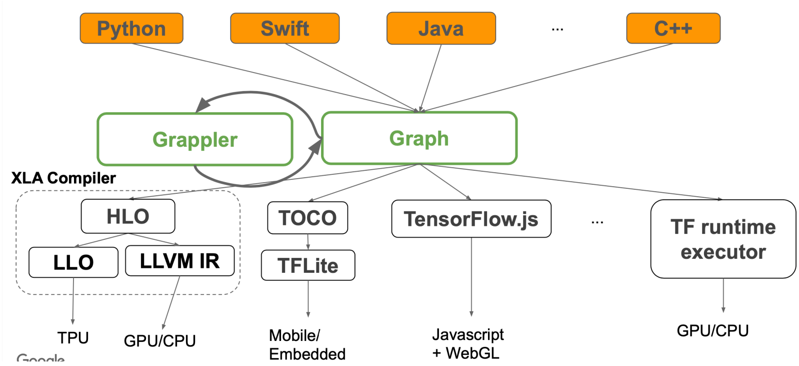
- Graph-level optimizations, we refer to a computation graph in this case, an example could be the MaseGraph.
- Operator-level optimizations, how do we perform lower-level optimization if given hardware information.

A high-level overview

Grappler (Tensorflow)

Grappler is designed for Tensorflow, to optimize the Graph.

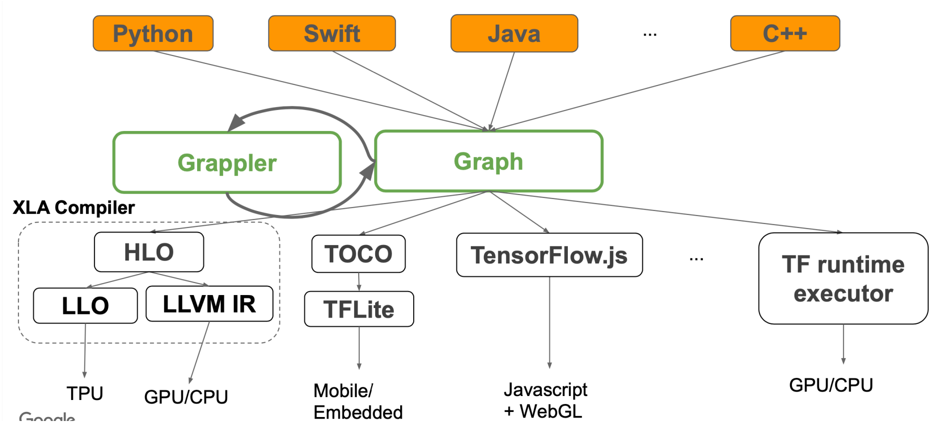
This is normally on the middleware level. Grappler takes various front-end languages (Python, Swift … C++). The middleware obtains a Graph and we have to deploy this to various hardware backends.



A high-level overview

These optimizations are normally at two levels:

- Graph-level
- Operator-level



Torch Compile

- TorchDynamo
  - Uses CPython Frame Evaluation to allow per-interpreter function pointer to handle the evaluation of frames.
  - An execution frame is how Python tracks the values of local variables during a function call.
  - This CPyton feature allows external C code to control frame evaluation.
  - This means a JIT can participate in the execution of Python code.
- TorchInductor
  - Triton-based, maps code to multiple backends (eg. AMD, Nvidia)
- AOT Autograd
  - Captures backward pass computation, ahead of time.

Optimizations

Constant folding

Rewrites certain operations in the graph, if we know the values ahead-of-time.

Constant propagation

$Add(c1, Add(x, c2)) => Add(x, c1 + c2)$

Operations with neutral & absorbing elements

$x^*Ones(s) => Identity(x), if\ shape(x) == output\_shape$

Graph Operation Fusion

Replaces commonly occurring subgraphs with optimized fused kernels

Examples of patterns fused:

- Conv2D + BiasAdd + Activation
- Conv2D + FusedBatchNorm + Activation
- Conv2D + Squeeze + BiasAdd
- MatMul + BiasAdd + Activation

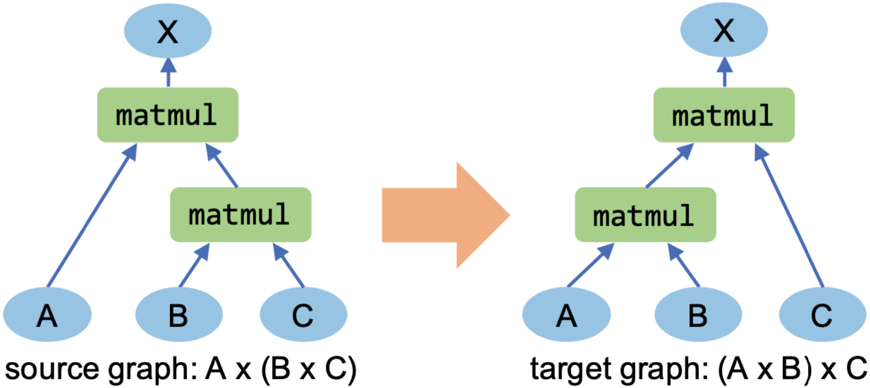
Graph Operation Fusion

Graph Operation Fusion Fusing ops together provides several performance advantages:

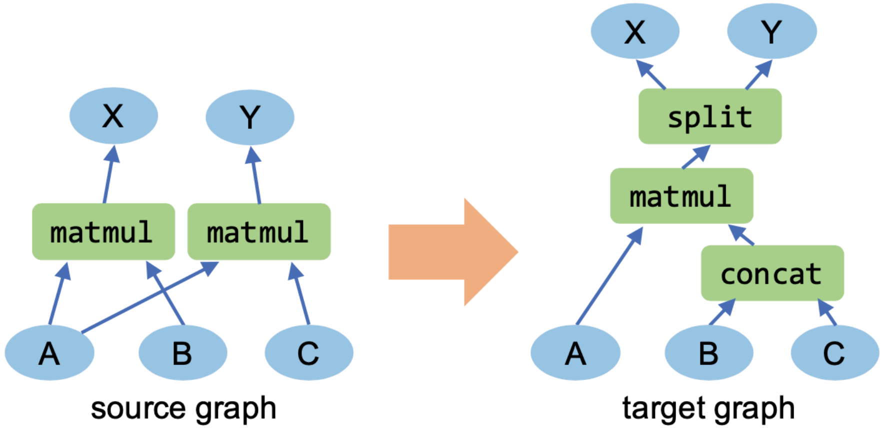
- Completely eliminates the scheduling overhead (great for cheap ops)
- Increases opportunities for ILP, vectorization etc.
- Improves temporal and spatial locality of data access. E.g. MatMul is computed block-wise and bias and activation function can be applied while data is still “hot” in cache.

Graph Operation Rewrites

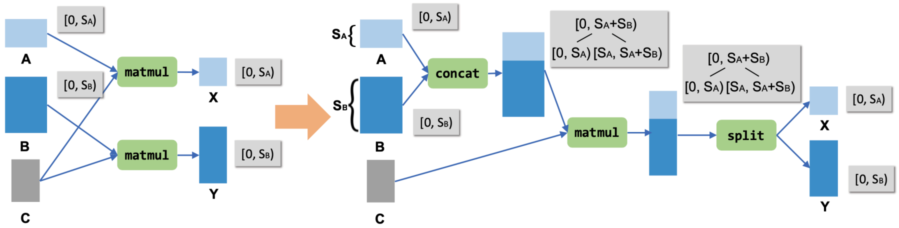
Rely on heuristics to repack operations on the graph level.



Graph Operation Rewrites (ii)



Graph Operation Rewrites (iii)



Graph level optimizations are normally at a coarse-grained level.

More importantly, most of them have to rely on certain heuristics, for instance, this could be the re-write rules for fusion strategies.

Internally, in Pytorch land, these re-write rules are implemented as `fx` passes!

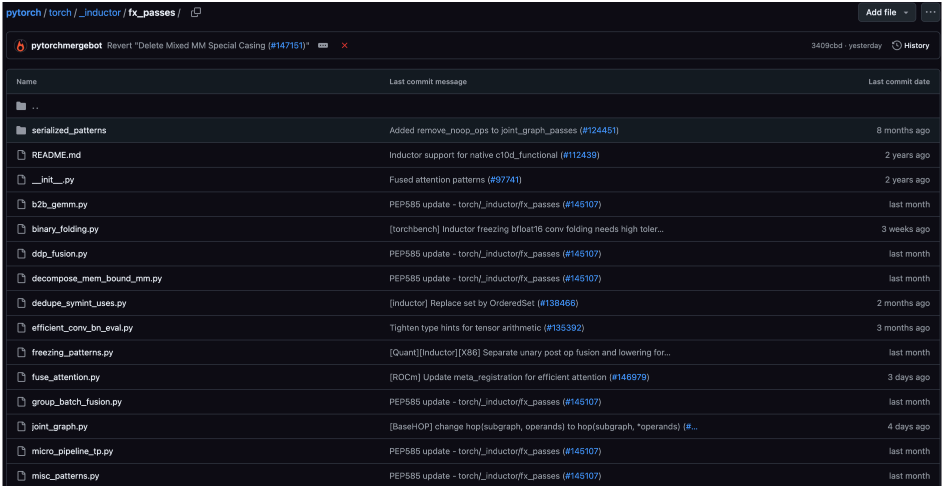
How graphs are handled in Pytorch

```python
import torch


# Simple module for demonstration
class MyModule(torch.nn.Module):
    def __init__(self) -> None:
        super().__init__()
        self.param = torch.nn.Parameter(torch.rand(3, 4))
        self.linear = torch.nn.Linear(4, 5)

    def forward(self, x):
        return self.linear(x + self.param).clamp(min=0.0, max=1.0)


module = MyModule()

from torch.fx import symbolic_trace
```

How graphs are handled in Pytorch (ii)

```python
# Symbolic tracing frontend - captures the semantics of the module
symbolic_traced: torch.fx.GraphModule = symbolic_trace(module)

# High-level intermediate representation (IR) - Graph
representation
print(symbolic_traced.graph)
"""
graph():
    %x : [num_users=1] = placeholder[target=x]
    %param : [num_users=1] = get_attr[target=param]
    %add : [num_users=1] = call_function[target=operator.add](args
= (%x, %param), kwargs = {})
    %linear : [num_users=1] = call_module[target=linear](args =
(%add,), kwargs = {})
    %clamp : [num_users=1] = call_method[target=clamp](args =
(%linear,), kwargs = {min: 0.0, max: 1.0})
    return clamp
"""
```

How graphs are handled in Pytorch (iii)



pytorch / torch / _inductor / fx_passes /

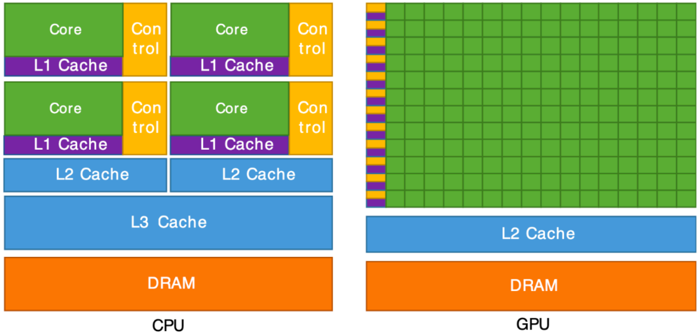
Add file

pytorchmergebot Revert "Delete Mixed MM Special Casing (#147151)" ✕ 3409cbd · yesterday History

Name	Last commit message	Last commit date
..		
serialized_patterns	Added remove_noop_ops to joint_graph_passes (#124451)	8 months ago
README.md	Inductor support for native c10d_functional (#112439)	2 years ago
__init__.py	Fused attention patterns (#97741)	2 years ago
b2b_gemm.py	PEP585 update - torch/_inductor/fx_passes (#145107)	last month
binary_folding.py	[torchbench] Inductor freezing bfloat16 conv folding needs high toler...	3 weeks ago
ddp_fusion.py	PEP585 update - torch/_inductor/fx_passes (#145107)	last month
decompose_mem_bound_mm.py	PEP585 update - torch/_inductor/fx_passes (#145107)	last month
dedupe_symint_uses.py	[inductor] Replace set by OrderedSet (#138466)	2 months ago
efficient_conv_bn_eval.py	Tighten type hints for tensor arithmetic (#135392)	3 months ago
freezing_patterns.py	[Quant][Inductor][X86] Separate unary post op fusion and lowering for...	last month
fuse_attention.py	[ROCm] Update meta_registration for efficient attention (#146979)	3 days ago
group_batch_fusion.py	PEP585 update - torch/_inductor/fx_passes (#145107)	last month
joint_graph.py	[BaseHOP] change hop(subgraph, operands) to hop(subgraph, *operands) (#...	4 days ago
micro_pipeline_tp.py	PEP585 update - torch/_inductor/fx_passes (#145107)	last month
misc_patterns.py	PEP585 update - torch/_inductor/fx_passes (#145107)	last month

The Operator-level Optimizations

Operator-level optimizations



On the graph-level, we care about large operations (more like layers).

On the operator level, this is now closer to the hardware. We typically prioritize the most compute-intensive tasks, thereby mostly focus on loop manipulations.

Loop tiling

The following is an example of matrix vector multiplication.

There are three arrays, each with 100 elements. The code does not partition the arrays into smaller sizes.

```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++):
  c[i] = 0;
  for (j = 0; j < n; j++):
    c[i] = c[i] + a[i][j] * b[j];
```

Loop tiling

After loop tiling is applied using $2 * 2$ blocks, the code looks like:

```
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2):
  c[i] = 0;
  c[i + 1] = 0;
  for (j = 0; j < n; j += 2):
    for (x = i; x < min(i + 2, n), x++):
      for (y = j; y < min(j + 2, n), y++):
        c[x] = c[x] + a[x][y] * b[y];
```

Loop tiling

- Tiling partitions a loop's iteration space into smaller chunks or blocks
- This ensures data used in a loop stays in the cache until it is reused.
- This leads to partitioning of a large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements.

Loop unrolling

```
int x;
for (x = 0; x < 100; x++):
  do_something(x);

for (x = 0; x < 100; x += 5):
  do_something(x);
  do_something(x + 1);
  do_something(x + 2);
  do_something(x + 3);
  do_something(x + 4);
```

The amount of unrolling is associated with the hardware parallelism.

This looks simple, but can become very complex if we have loop nests. It becomes more complex if we start to think about it in conjunction with tiling and other operations.

Loop permutation for data layout optimization

What is Data Layout?

Data layout optimization converts data into one that can use better internal data layouts for execution on the target hardware.

For instance, a DL accelerator might exploit 4×4 matrix operations, requiring data to be tiled into 4×4 chunks to optimize for access locality.

A good data layout:

- Improves locality
- Reduces the number of off-chip memory accesses
- May reduce Ops (eg. unnecessary transpose)

This is normally achieved by loop permutation.

More on loop optimizations

Loop fusion and fission: this breaks large loop to smaller ones or fuse small loops, normally for locality.

Loop Skewing (Polyhedral Optimization): Normally for deep nested loops, re-arrange loops to achieve a better access pattern.

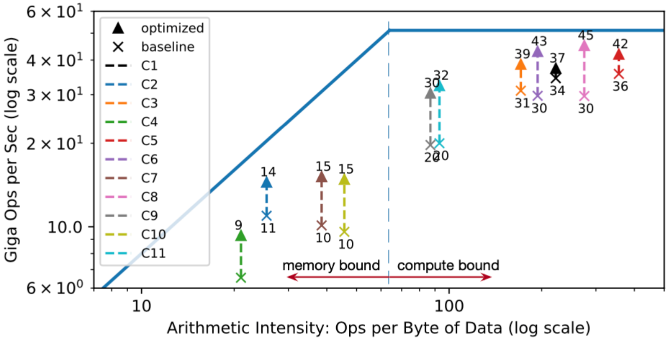
Loop Splitting: attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different portions of the index range.

and more…

Why we do this?

Loop-level manipulation is very complex, however, the core idea is to push the performance to the optimal point!

- Improve locality for memory-bound scenarios
- Improve parallelism for compute-bound scenarios



Arithmetic Operator Optimization

- **Arithmetic simplification**
  - Hoisting: $Add(x^*a, x^*bx^*c) = x^*Add(a, b, c)$
  - Node reduction: $x + x + x = 3x$, one op instead of two ops
- **Broadcast minimization** $(matrix1 + scalar1) + (matrix2 + scalar2) => (matrix1 + matrix2) + (scalar1 + scalar2)$
- **Better use of intrinsics** $Matmul(Transpose(x), y) => Matmul(x, y, transpose_x = True)$

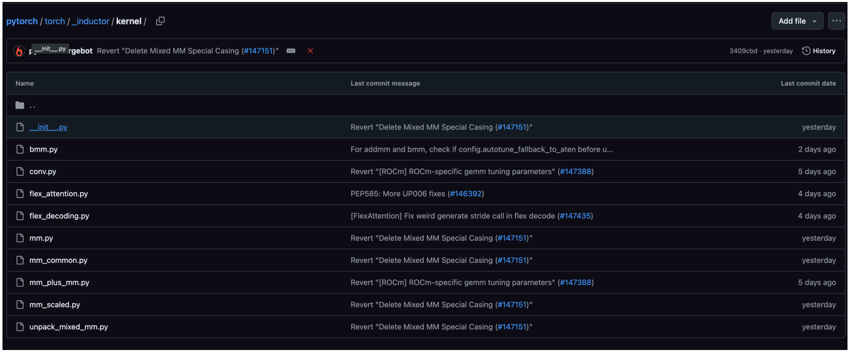
Memory Operator Optimization

- **Swap-in and Swap-out Optimizations**: actively estimate memory usage and swap in/out idle data to host memory
- **Recomputation optimization**: if moving in/out the data is slow, why not just re-compute that value.

Automated optimization?

Automated low-level optimization was very popular but now …

Hashed to a few popular kernels!



pytorch / torch / _inductor / kernel /

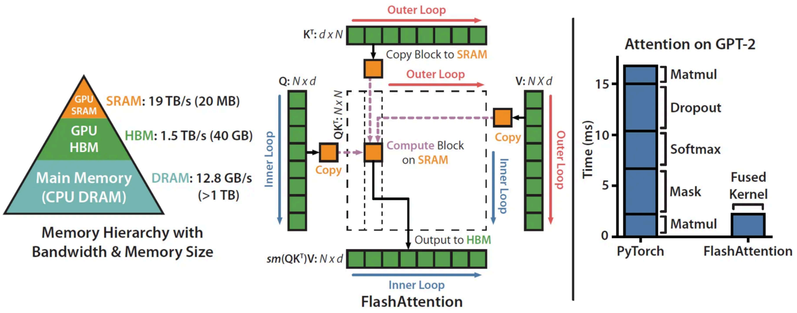
Add file

p __init__.py rgebot Revert "Delete Mixed MM Special Casing (#147151)"

3409cbd · yesterday History

Name	Last commit message	Last commit date
..		
__init__.py	Revert "Delete Mixed MM Special Casing (#147151)"	yesterday
bmm.py	For addmm and bmm, check if config.autotune_fallback_to_aten before u...	2 days ago
conv.py	Revert "[ROCm] ROCm-specific gemm tuning parameters" (#147388)	5 days ago
flex_attention.py	PEP585: More UP006 fixes (#146392)	4 days ago
flex_decoding.py	[FlexAttention] Fix weird generate stride call in flex decode (#147435)	4 days ago
mm.py	Revert "Delete Mixed MM Special Casing (#147151)"	yesterday
mm_common.py	Revert "Delete Mixed MM Special Casing (#147151)"	yesterday
mm_plus_mm.py	Revert "[ROCm] ROCm-specific gemm tuning parameters" (#147388)	5 days ago
mm_scaled.py	Revert "Delete Mixed MM Special Casing (#147151)"	yesterday
unpack_mixed_mm.py	Revert "Delete Mixed MM Special Casing (#147151)"	yesterday

Flash Attention



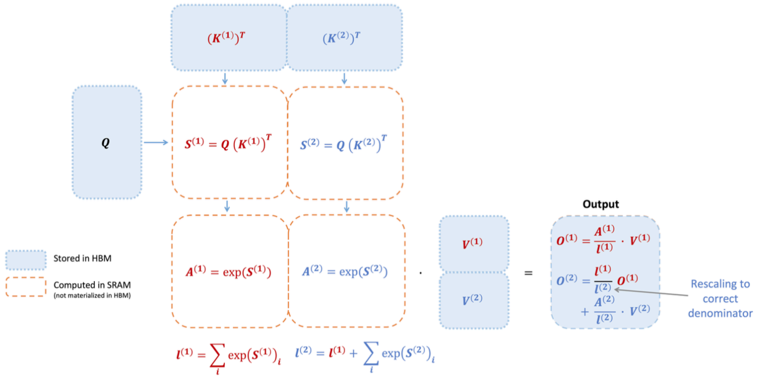
Attention is very memory-bound, if you implement them naively.

Standard implementation shows the utmost disrespect for the way HW operates. It’s basically treating HBM load/store ops as 0 cost (it’s not “IO-aware”).

FlashAttention mainly use two tricks:

Flash Attention (ii)

- Tiling
- Recomputation



$(K^{(1)})^T$
$(K^{(2)})^T$
Q
$S^{(1)} = Q\left(K^{(1)}\right)^T$
$S^{(2)} = Q\left(K^{(2)}\right)^T$
Stored in HBM
Computed in SRAM
(not materialized in HBM)
$A^{(1)} = \exp(S^{(1)})$
$A^{(2)} = \exp(S^{(2)})$
$V^{(1)}$
$V^{(2)}$
Output
$O^{(1)} = \frac{A^{(1)}}{l^{(1)}} \cdot V^{(1)}$
$O^{(2)} = \frac{l^{(1)}}{l^{(2)}} O^{(1)} + \frac{A^{(2)}}{l^{(2)}} \cdot V^{(2)}$
Rescaling to correct denominator
$l^{(1)} = \sum_i \exp(S^{(1)})_i$
$l^{(2)} = l^{(1)} + \sum_i \exp(S^{(2)})_i$

Summary

- The Computational Graph
  - Constant folding
  - Graph operation fusion
  - Graph operation rewrites
- The Operator-level Graph
  - Loop tiling
  - Loop unrolling
  - Loop permutation
  - Arithmetic operator optimization
  - Memory operator optimization