

How does TorchInductor work?

Peter Bell, Horace He

What does Deep Learning performance look like?

Matmuls + Moving Data
Around

Modern GPUs (or AI accelerators) are matmul ASICs

Tensor Core == “does a matmul”

15x more FLOPS for matmuls

Form Factor	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core	989 teraFLOPS ²
BFLOAT16 Tensor Core	1,979 teraFLOPS ²
FP16 Tensor Core	1,979 teraFLOPS ²
FP8 Tensor Core	3,958 teraFLOPS ²
INT8 Tensor Core	3,958 TOPS ²

Matmuls are already highly optimized (and fast moving)!

- Tensor Cores in 2017 with V100
- Async Loads in 2020 with A100
- Thread Block Clusters/TMA/Warp-specialization/persistent kernels in 2022 with H100
- Who knows what's introduced in 2024 with B100

Warp-Specialized Persistent Ping-Pong kernel design

The third kernel design is the [Warp-Specialized Persistent Ping-Pong](#) kernel. Like the Warp-Specialized Persistent Cooperative, kernel the concepts of warp groups, barrier synchronization between warp groups, and the shape of the grid launch remain the same in the persistent ping-pong design. The distinctive feature of the Warp-Specialized Persistent Ping-Pong kernel is the following :

- The two *consumer* warp groups are assigned a different output tile using the Tile Scheduler. This allows for *epilogue* of one *consumer* warp group to be overlapped with the math operations of the other *consumer* warp group - thus maximizing tensor core utilization.
- The *producer* warp group synchronizes using the [Ordered Sequence Barrier](#) to fill buffers of the two *consumer* warp groups one after the other in order.

How do we generate efficient matmuls?

A: We just use CuBLAS/CUTLASS/Triton!

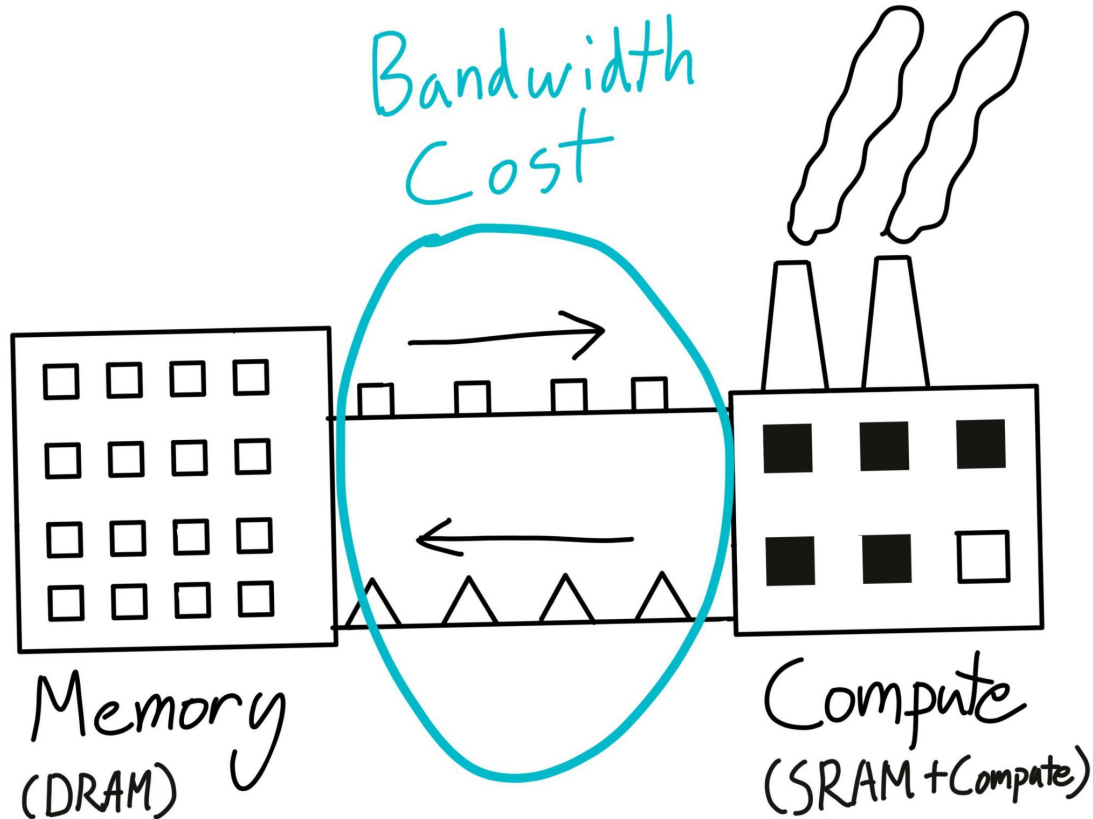
There are still interesting optimizations around matmuls, but the fundamental *codegen* is handled by (essentially) highly optimized handwritten libraries.

Matmuls + Moving Data Around

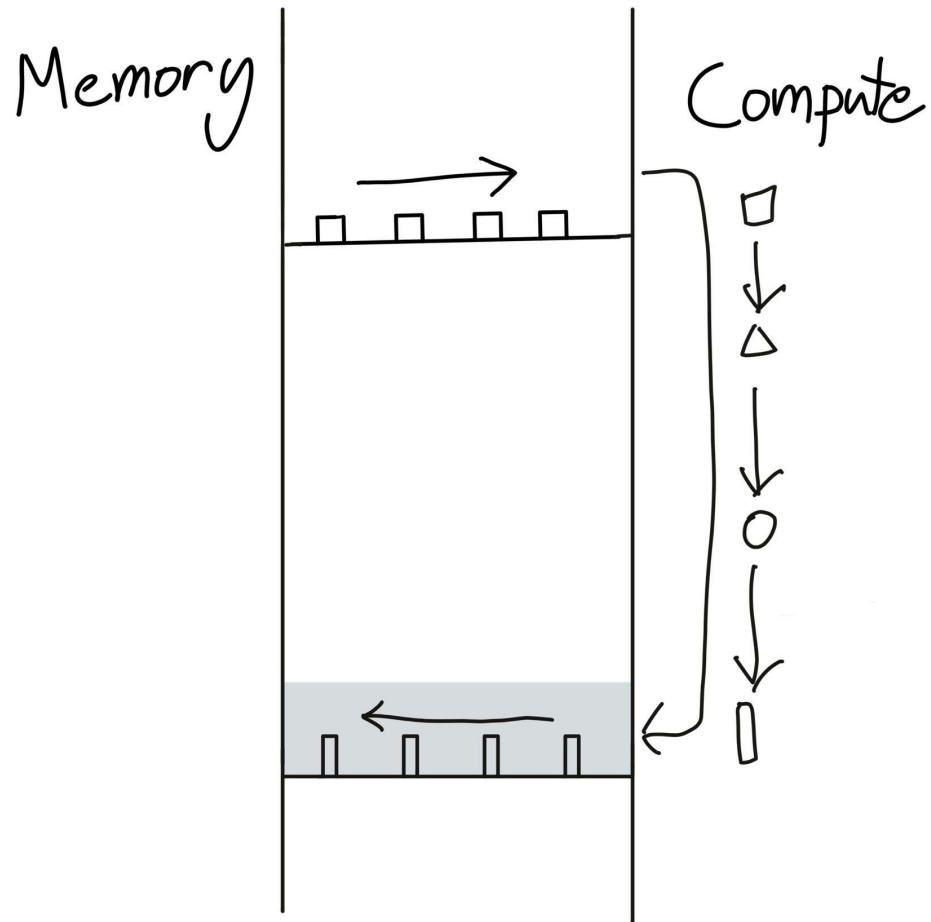
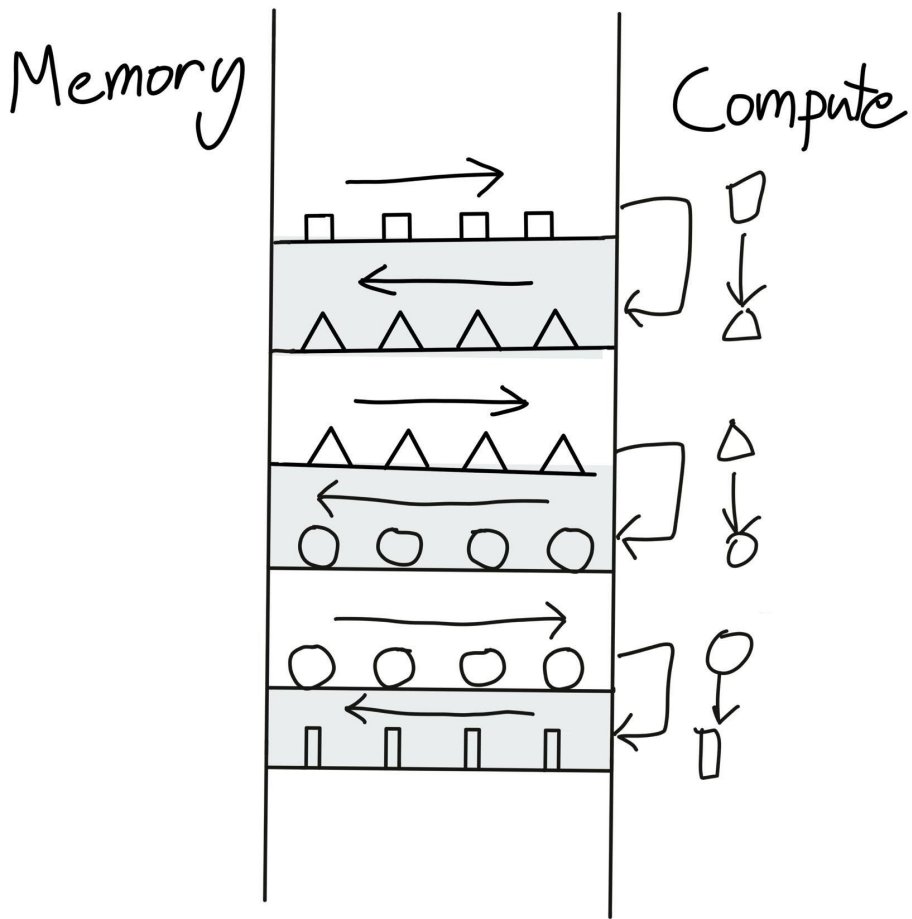
(Just call CuBLAS)

(Most of what TorchInductor does)

How do we optimize data movement?



Operator Fusion is the key!



Still Plenty of Difficult Problems to Solve...

1. Tiling
2. Loop Ordering
3. Rematerialization vs Recompute Decisions
4. Memory Planning
5. Layout Decisions
6. Block Size Choices
7. Split Reductions
8. Persistent vs. Non-Persistent Reductions
9. Fusion Segmentation

etc...

Enter TorchInductor

Representing training benchmarks across 172 models.

Represents:

46 Text Models (Transformers)

61 Vision Models (Timm)

65 TorchBench Models (misc.)

Geometric mean speedup (threshold = 0.95x) ?

Inductor config	Torchbench	Huggingface	TIMM models
cuda	2.01x	2.08x	1.82x
cuda_dyn	1.99x	2.07x	1.62x
default	1.27x	1.82x	1.72x

Peak memory footprint compression ratio (threshold = 0.9x) ?

Inductor config	Torchbench	Huggingface	TIMM models
cuda	0.79x	1.26x	1.11x
cuda_dyn	0.85x	1.08x	1.11x
default	0.79x	1.26x	1.11x

There exist only pointwise operators and reductions

Pointwise: $A * 2$, $A.\cos()$, $A.\text{relu}()$, $A + B$

Reduction: $\text{sum}(A)$, $\text{max}(A)$

Almost everything else can be written with these.

$\text{softmax}(x) = \exp(x - \text{max}(x)) / \text{sum}(\exp(x - \text{max}(x)))$

Technically, matmuls as well!

$\text{mm}(A: [M, K], B: [K, N]) = (a.\text{unsqueeze}(-1) * B).\text{sum}(\text{dim}=1)$

How does TorchInductor represent these operations?

Represent the abstract computation of each output element

$$\text{Out}[x, y] = A[x, y] + B[y, x] + C[x] + D[E[x]]$$
$$X = 4, y = 6$$
$$\text{Out}[4, 6] = A[4, 6] + B[6, 4] + C[4] + D[E[4]]$$


Reductions are the same, but accumulate into an output buffer with an extra dimension that is then reduced over.

$$\text{Tmp}[x, r] = A[x, r] + B[r, x] + C[x] + D[E[x]]$$
$$\text{Tmp.reduce}(r)$$

What's the point of this loop-level IR?


It disentangles what's being computed from the manner in which it's computed.

Loop Reordering



```
# Iterate x and then y
iter_order = (x, y)
out[x, y] = A[x, y]

# Iterate y and then x
iter_order = (y, x)
out[x, y] = A[x, y]
```



```
# Iterate x and then y
for x in range(1000):
    for y in range(1000):
        out[x, y] = A[x, y]

# Iterate y and then x
for x in range(1000):
    for y in range(1000):
        out[x, y] = A[x, y]
```

What's the point of this loop-level IR?

It disentangles what's being computed from the manner in which it's computed.

Split across GPU



```
# Iterate x and then y
iter_order = (x, y)
out[x, y] = A[x, y]

# 512 elements per threadblock
ELEM_PER_BLOCK = 512
iter_order = (x, y)
out[x, y] = A[x, y]
```



```
# Iterate x and then y
for x in range(1024):
    for y in range(1024):
        out[x, y] = A[x, y]

# 512 elements per threadblock
@parallelize_gpu
for x in range(1024):
    @parallelize_gpu
    for y_out in range(1024//512):
        for y_in in range(512):
            out[x, y] = A[x, y]
```

What's the point of this loop-level IR?

It disentangles what's being computed from the manner in which it's computed.

Tiling



```
# Iterate x and then y
iter_order = (x, y)
out[x, y] = A[x, y]
```

```
# 2d Tiling
```

```
ELEM_PER_BLOCK = (32, 32)
iter_order = (x, y)
out[x, y] = A[x, y]
```



```
# Iterate x and then y
for x in range(1024):
    for y in range(1024):
        out[x, y] = A[x, y]
```

```
# 512 elements per threadblock
```

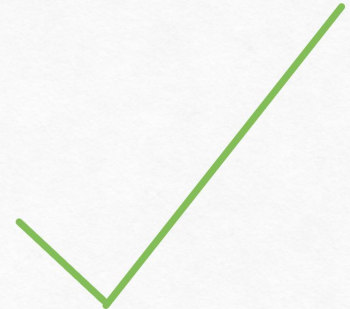
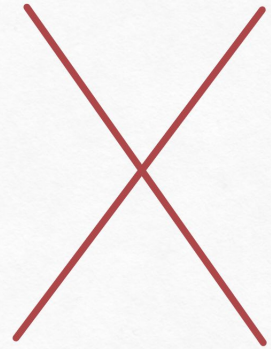
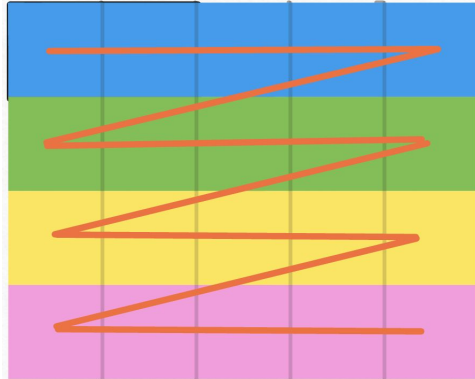
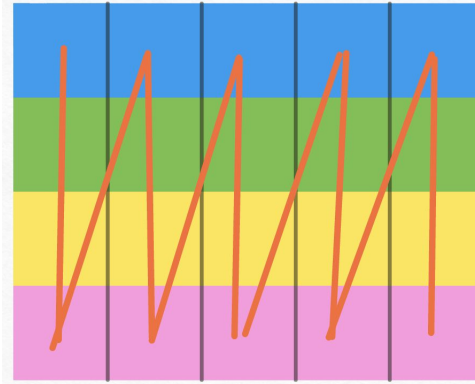
```
BLOCK_X = 32, BLOCK_Y=32
for x_out in range(1024//BLOCK_X):
    for y_out in range(1024//BLOCK_Y):
        for x_in in range(BLOCK_X):
            for y_in in range(BLOCK_Y):
                out[x, y] = A[x, y]
```

Why are Loop Ordering and Tiling Important?

Memory accesses are not truly “random” - we want to access consecutive elements.

For unary operators, this is easy.

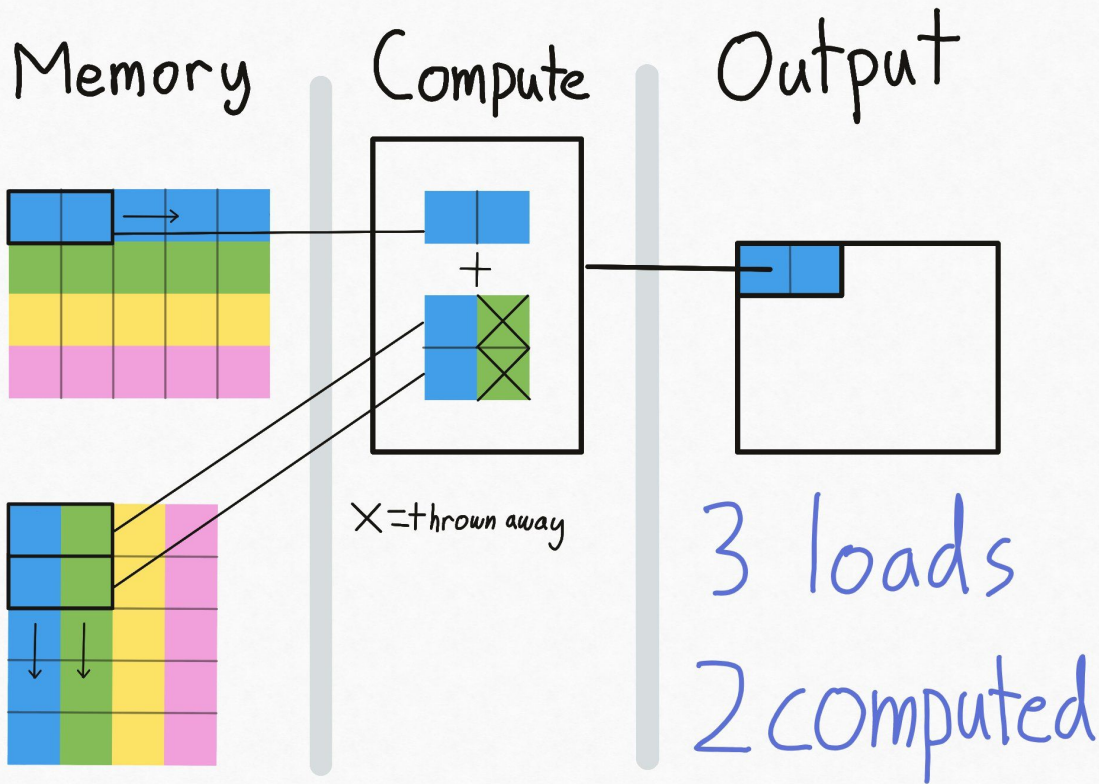
Modify your loop order to align with your memory layout.



$$A + B^T$$

What if we're doing an add with two inputs that don't have the same layouts?

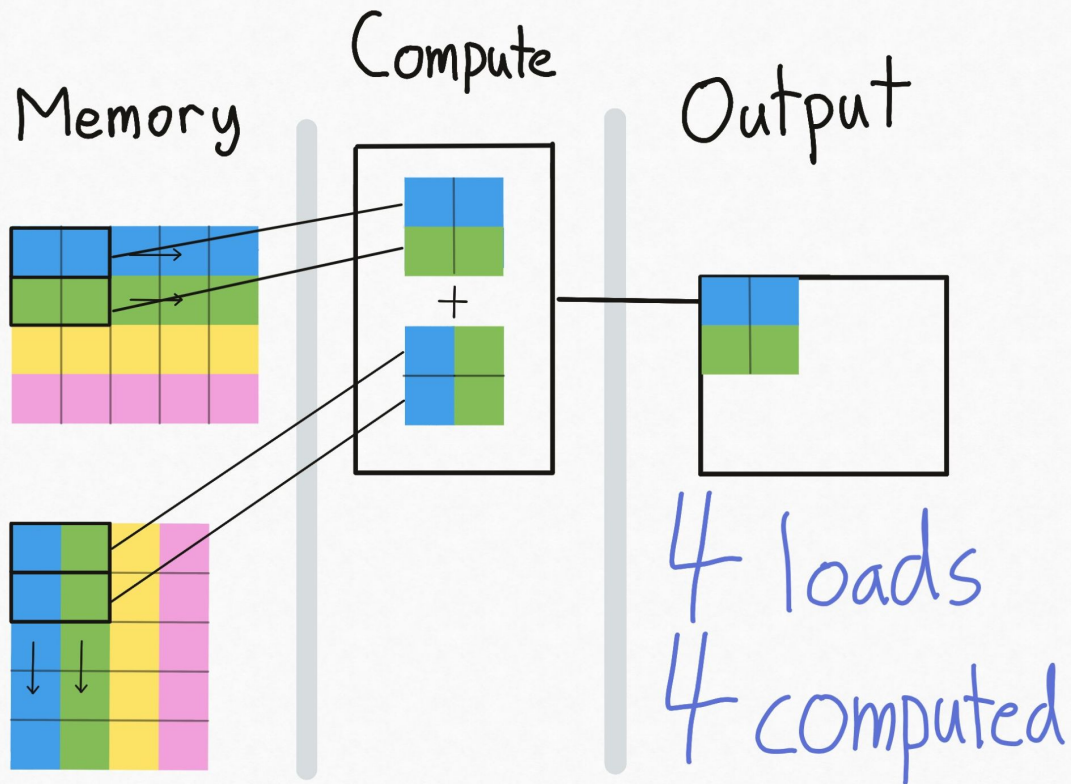
If we use the same iteration order for both inputs, it's not possible for both inputs to have contiguous reads.



Tiling to the rescue!

By relaxing the iteration order a little, tiling gives us enough leeway for both A and B to have contiguous reads.

Tiled $A + B^T$

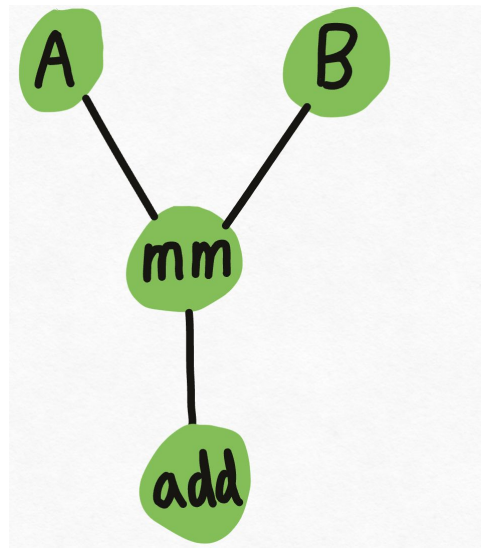


Loops are cool.... but we start with a graph.

Directed Acyclic Graph

Purely Functional

All autograd/transform/etc. “baked into” the graph



```
def forward(self, arg0_1: "f32[20, 20]", arg1_1: "f32[20, 20]"):
    # File: /data/users/chilli/pytorch/./t.py:12 in f, code: return torch.mm(a, b) + 3
    mm: "f32[20, 20]" = torch.ops.aten.mm.default(arg0_1, arg1_1); arg0_1 = arg1_1 = None
    add: "f32[20, 20]" = torch.ops.aten.add.Tensor(mm, 3); mm = None
    return (add,)
```

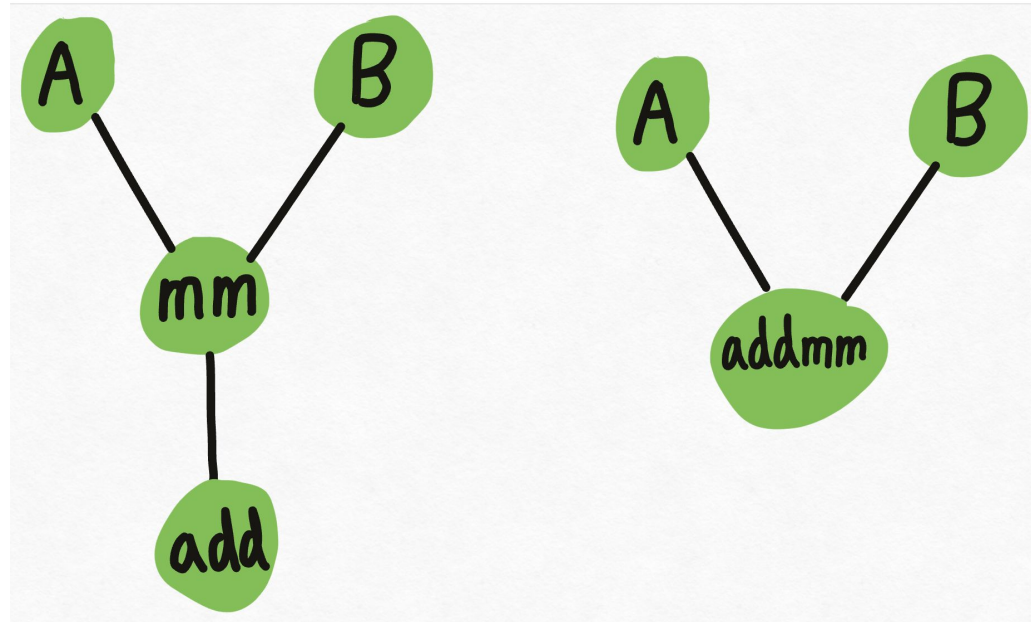
Why do we want a graph-level IR?

We lower the level of abstraction throughout compilation.

You start with user-code, and end up with machine code.

More structure
=> Easier to do optimizations

Less Structure
=> Optimizations are more general



Examples of graph-level Optimizations

1. Some amount of pattern-matching (addmm, attention, other misc stuff)
 - a. It's a very easy way to optimize a specific pattern.
 - b. Not very general, quite brittle.
 - c. Not a major component of Inductor's performance.
2. Reintroducing mutation for certain operators.
3. Matmul Shape Padding
 - a. Matmuls that have shapes that are multiples of 8 are much faster.



Andrej Karpathy ✓

@karpathy



The most dramatic optimization to nanoGPT so far (~25% speedup) is to simply increase vocab size from 50257 to 50304 (nearest multiple of 64). This calculates added useless dimensions but goes down a different kernel path with much higher occupancy. Careful with your Powers of 2.

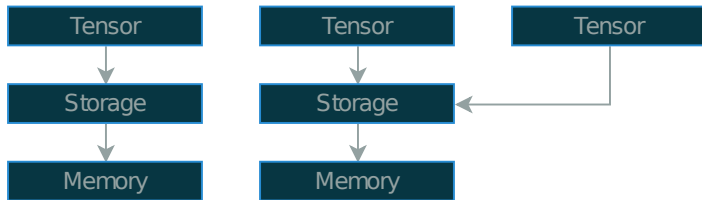
10:36 AM · Feb 3, 2023 · 1.2M Views

Overview of inductor codegen

1. Lower fx graph to loop level IR
2. Memory planning
3. Scheduling and loop fusion
4. Code generation

Background – Eager data model

- ▶ All intermediates are stored in device memory
- ▶ Device memory is owned uniquely by a `Storage` object
- ▶ Each `Tensor` views a storage with:
 - ▶ Shape
 - ▶ Strides
 - ▶ Storage offset
- ▶ Views create a new `Tensor` referencing the same `Storage`



Loop level IR – Lowering the graph

```
def forward(primals_1, primals_2):  
    add = aten.add.Tensor(  
        primals_1, primals_2)  
    div = aten.div.Tensor(add, 2)  
    return [div]
```

```
arg0 = TensorBox(StorageBox(InputBuffer(...))  
arg1 = TensorBox(StorageBox(InputBuffer(...))  
# These become TensorBox(StorageBox(Pointwise))  
add = lowering[aten.add.Tensor](arg0, arg1)  
div = lowering[aten.div.Tensor](add, 2)
```

- ▶ InputBuffers created from example inputs
- ▶ Execute fx graph and build IR on-the-fly

Loop level IR – Pointwise Loops

```
@register_lowering(
    aten.add, broadcast=True)
def add(a, b):
    a, b = promote_constants([a, b])
    a_loader = a.make_loader()
    b_loader = b.make_loader()

    def inner_fn(index):
        return ops.add(
            a_loader(index), b_loader(index))

    return ir.Pointwise.create(
        device=a.get_device(),
        dtype=a.get_dtype(),
        inner_fn=inner_fn,
        ranges=a.get_size())
```

- ▶ inner_fn builds the IR
- ▶ index is a tuple of `sympy.Expr`
- ▶ Loaders themselves just call ops
- ▶ `Pointwise.make_loader()` is inner_fn
- ▶ No concept of memory layout yet, only nd-indices

Loop level IR – Define by run

- ▶ ops calls recorded by running `inner_fn`
- ▶ Inputs and outputs become buffer loads and stores
- ▶ Symbolic indexing calculations get frozen in
- ▶ `i0` is a loop index, `s0` is a size

```
def inner_fn(index):  
    a = a_loader(index)  
    b = b_loader(index)  
    if alpha != 1:  
        b = ops.mul(b, alpha)  
    return ops.add(a, b)
```

```
tmp0 = ops.load("buf0", i1 * s0 + i0)  
tmp1 = ops.load("buf1", i1 * s0 + i0)  
tmp2 = ops.add(tmp0, tmp1)  
ops.store("buf2", i1 * s0 + i0, tmp2)
```

Loop level IR – Indirect indexing

- ▶ Data-dependent indexing, e.g. `aten.index`
- ▶ Indirect loads are still considered Pointwise loops
- ▶ Loop only needs to be pointwise on `inner_fn`
- ▶ Other weird Pointwise lowerings:
 - ▶ Pooling – Each input views a sliding window of input
 - ▶ Padding – Input is offset and masked relative to output

```
tmp0 = ops.load("buf0", i0)
index0 = ops.indirect_indexing(tmp0, s0)
tmp1 = ops.load("buf1", index0)
```

Loop level IR – Reduction

- ▶ `ir.Reduction` – associative reductions that reduce across loop iterations.
- ▶ Some loop indices are designated reduction indices
- ▶ `ops.store_reduction` to store only once per reduction loop

```
tmp0 = ops.load("buf0", i1 * s0 + i0)
tmp1 = ops.reduction(
    out_dtype, src_dtype, "sum", tmp0)
ops.store_reduction("buf1", i1, tmp1)
```

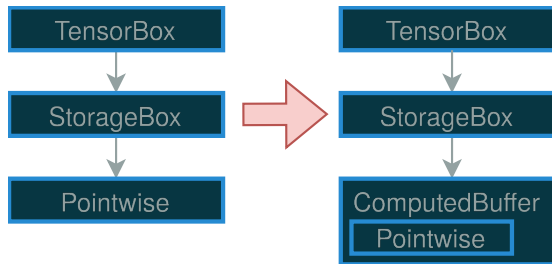
Loop level IR – Scans

- ▶ Associative scans are similar, but use a normal store instruction

```
tmp0 = ops.load("buf0", i1 * s0 + i0)
tmp1 = ops.scan(dtype, sum_combine_fn, (tmp0,))
ops.store("buf1", i1 * s0 + i0, tmp1)
```

Tensor realization

- ▶ Loops may be ‘realized’ into buffers
 - ▶ Stops inlining
 - ▶ Represents a tensor in memory
- ▶ This wraps the `ir.Loops` object in `ir.ComputedBuffer`
- ▶ `ComputedBuffer.make_loader()` loads from memory
- ▶ Strides are not fixed at this stage, buffer has `FlexibleLayout`



Tensor realization – Forced

Some cases require loops to be realized:

- ▶ Template kernels (e.g. `aten.addmm`)
- ▶ Fallback to ATen (non-codegen path, e.g. `aten.convolution`)
- ▶ Buffer mutation (e.g. `aten.index_put_`)
- ▶ Graph outputs
- ▶ Reduction and Scan outputs

Tensor realization – Heuristic-based

We also heuristically realize a `TensorBox` if:

- ▶ Has more than 8 reads OR
- ▶ Has more than 1 user¹ AND
 - ▶ Has more than 4 reads OR
 - ▶ `inner_fn` calls more than 30 ops (after CSE)

¹A single loop reading multiple indexes also counts as multiple ‘users’

Memory Planning – Layout constraints

FixedLayout

While lowering, we fix strides when

- ▶ We fallback to ATen and use strides from eager
- ▶ Some operator depends on particular strides, e.g. `aten.as_strided`
- ▶ Strides are visible in output tensors
- ▶ `aten.concat` may allocate buffer into a larger buffer



- ▶ Channels last format is preferred for convolutions

Otherwise, any remaining FlexibleLayouts are made contiguous.

Scheduling and Loop Fusion – Pointwise fusions

Original loops

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    buf0[i0] = tmp0
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

After horizontal fusion

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    buf0[i0] = tmp0
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

After vertical fusion

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

Benefits of fusion

- ▶ Reduce memory bandwidth requirements
- ▶ Enables common subexpression elimination

Scheduling and Loop Fusion – Pointwise fusions

Original loops

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    buf0[i0] = tmp0
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

After horizontal fusion

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    buf0[i0] = tmp0
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

After vertical fusion

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

Benefits of fusion

- ▶ Reduce memory bandwidth requirements
- ▶ Enables common subexpression elimination

Scheduling and Loop Fusion – Pointwise fusions

Original loops

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    buf0[i0] = tmp0
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

After horizontal fusion

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    buf0[i0] = tmp0
```

```
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

After vertical fusion

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    tmp1 = abs(tmp0)  
    buf1[i0] = tmp1  
    tmp2 = sqrt(tmp0)  
    buf2[i0] = tmp2
```

Benefits of fusion

- ▶ Reduce memory bandwidth requirements
- ▶ Enables common subexpression elimination

Scheduling and Loop Fusion – Pointwise-Reduction fusion

Before fusion

```
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    buf0[i0] = tmp0
```

```
acc0 = 0  
for i0 in range(s0):  
    tmp0 = buf0[i0]  
    acc0 = acc0 + tmp0  
buf1[0] = acc0
```

```
tmp0 = buf1[0]  
tmp1 = sqrt(tmp0)  
buf2[0] = tmp1
```

After fusion

```
acc0 = 0  
for i0 in range(s0):  
    tmp0 = inp[i0] + 1  
    acc0 = acc0 + tmp0  
tmp1 = sqrt(acc0)  
buf2[0] = tmp1
```

Inductor can also fuse

- ▶ Reductions broadcast back to their original size
- ▶ Horizontal reductions with pointwise ops

Scheduling and Loop Fusion

Why not fuse everything?

- ▶ Large kernels may use too many registers
- ▶ Different kernels benefit from different launch parameters
- ▶ Not all kernels can be fused (e.g. ATen Fallback)
- ▶ Templates only allow epilogue fusion

When can't we fuse?

- ▶ Kernels must be on the same device
- ▶ Loops must be compatible shapes
 - ▶ Same `shape` on CPU, or `numel` on CUDA
- ▶ Vertical fusions must access intermediate buffer at the same index
- ▶ Fusion must not create a dependency cycle

Scheduling and Loop Fusion – Algorithm

Algorithm

1. Record each loop's memory dependencies (buffer and index)
2. List pairs of nodes with a common buffer that can be fused
3. Rank possible fusions by common memory accesses
4. Greedily try to fuse all allowed pairs in ranked order
5. If any fusions happened, goto 2

Scheduling and Loop Fusion – Scoring

Scoring factors from most to least important

1. Prioritise fusing reductions with reductions, pointwise with pointwise.
 - ▶ Goal: Prefer doing work in pointwise kernels
 - ▶ Only applies if the kernels have common data accesses
2. Memory score: number of bytes of common data access
 - ▶ Goal: Reduce overall bandwidth requirement
3. Proximity score: maximum distance in a topological ordering
 - ▶ Goal: Avoid increasing intermediate lifetimes

Code Generation

- ▶ Fused loops now represent individual kernels to be generated
- ▶ GPU device code generated with triton

```
tmp0 = ops.load("buf0", i0)
tmp1 = ops.load("buf1", i0)
tmp2 = ops.add(tmp0, tmp1)
ops.store("buf2", i0, tmp2)
```

```
@triton.jit
def triton_(in_ptr0, in_ptr1, out_ptr0,
            xnumel, XBLOCK : tl.constexpr):
    xoffset = tl.program_id(0) * XBLOCK
    xindex = xoffset + tl.arange(0, XBLOCK)[:XBLOCK]
    xmask = xindex < xnumel
    x0 = xindex
    tmp0 = tl.load(in_ptr0 + (x0), xmask)
    tmp1 = tl.load(in_ptr1 + (x0), xmask)
    tmp2 = tmp0 + tmp1
    tl.store(out_ptr0 + (x0), tmp2, xmask)
```


Code Generation

- ▶ CPU kernels generated with C++ and OpenMP
- ▶ Multi-threaded and vectorized

```
tmp0 = ops.load("buf0", i0)
tmp1 = ops.load("buf1", i0)
tmp2 = ops.add(tmp0, tmp1)
ops.store("buf2", i0, tmp2)
```

```
#pragma omp parallel num_threads(32)
{
    #pragma omp for
    for(long x0 = 0L; x0 < 8L*(ks0 / 8L); x0 += 8L) {
        auto tmp0 = Vectorized<float>::loadu(
            in_ptr0 + x0, 8);
        auto tmp1 = Vectorized<float>::loadu(
            in_ptr1 + x0, 8);
        auto tmp2 = tmp0 + tmp1;
        tmp2.store(out_ptr0 + x0);
    }
    #pragma omp for
    for(long x0 = 8L*(ks0 / 8L); x0 < ks0; x0 += 1L) {
        auto tmp0 = in_ptr0[x0];
        auto tmp1 = in_ptr1[x0];
        auto tmp2 = decltype(tmp0)(tmp0 + tmp1);
        out_ptr0[x0] = tmp2;
    }
}
```

Code Generation – Wrapper Code

```
def call(args):
    arg0_1, arg1_1 = args
    args.clear()
    assert_size_stride(arg0_1, (1024, 16384), (16384, 1))
    assert_size_stride(arg1_1, (1024, 16384), (16384, 1))
    with torch.cuda._DeviceGuard(0):
        torch.cuda.set_device(0)
        buf0 = empty_strided_cuda((1024, 16384), (16384, 1), torch.float32)
        # Source Nodes: [add], Original ATen: [aten.add]
        stream0 = get_raw_stream(0)
        triton_poi_fused_add_0.run(
            arg0_1, arg1_1, buf0,
            16777216, grid=grid(16777216), stream=stream0)
    del arg0_1
    del arg1_1
    return (buf0, )
```

Triton 101

```
@triton.jit
def sum_kernel(
    in_ptr, out_ptr, xnumel, rnumel, RBLOCK: tl.constexpr):
    xindex = tl.program_id(axis=0)
    rindex = tl.arange(0, RBLOCK)
    offset = xindex * rnumel + rindex
    data = tl.load(in_ptr + offset, mask=rindex < rnumel)
    res = tl.sum(data, axis=0)
    tl.store(out_ptr + xindex, res)
```

- ▶ Tensor programming with NumPy/PyTorch like syntax
- ▶ Each 'program' runs on one CUDA block
- ▶ Not as high level as pytorch tensors
- ▶ Inductor has to decide memory layout, work distribution between blocks

Comparison with CUDA C++

```
template <int block_size>
__global__ void sum_kernel(
    const float *in_ptr, float *out_ptr,
    int64_t numel, int64_t rnumel) {
    auto xindex = blockIdx.x;
    auto rindex = threadIdx.x;
    auto offset = xindex * rnumel + rindex;
    auto data = rindex < rnumel ? in_ptr[offset] : 0.0f;
    auto res = blockSum<block_size>(data);
    if (rindex == 0) { out_ptr[xindex] = res; }
}
```

- ▶ Naive code misses vectorization
- ▶ The job of each thread is set in stone, tying compiler's hands
- ▶ No easy way to autotune

What optimizations matter?

	Inference	Training
All TorchInductor optimizations	$1.91\times$	$1.45\times$
Without loop/layout reordering	$1.91\times (-0.00)$	$1.28\times (-0.17)$
Without matmul templates	$1.85\times (-0.06)$	$1.41\times (-0.04)$
Without parameter freezing	$1.85\times (-0.06)$	$1.45\times (-0.00)$
Without pattern matching	$1.83\times (-0.08)$	$1.45\times (-0.00)$
Without cudagraphs	$1.81\times (-0.10)$	$1.37\times (-0.08)$
Without fusion	$1.68\times (-0.23)$	$1.27\times (-0.18)$
Without inlining	$1.58\times (-0.33)$	$1.31\times (-0.14)$
Without fusion and inlining	$0.80\times (-1.11)$	$0.59\times (-0.86)$

Fusing Pointwise Epilogues into Matmuls

```
@torch.compile(mode="max-autotune")
def f(x, y, bias):
    out = torch.mm(x, y)
    return out, out.cos(), out + bias
```



```
...
# rematerialize rm and rn to save registers
rm = pid_m * BLOCK_M + tl.arange(0, BLOCK_M)
rn = pid_n * BLOCK_N + tl.arange(0, BLOCK_N)
idx_m = rm[:, None]
idx_n = rn[None, :]
mask = (idx_m < M) & (idx_n < N)
```

Handwritten
template code!

```
# inductor generates a suffix
xindex = idx_n + (4096*idx_m)
tl.store(out_ptr0 + (tl.broadcast_to(xindex, mask.shape)), acc, mask)
tmp1 = tl.load(in_ptr2 + (tl.broadcast_to(idx_n, mask.shape)), mask, eviction_policy=
tmp0 = tl.cos(acc)
tmp2 = acc + tmp1
tl.store(out_ptr1 + (tl.broadcast_to(xindex, mask.shape)), tmp0, mask)
tl.store(out_ptr2 + (tl.broadcast_to(idx_n + (4096*idx_m), mask.shape)), tmp2, mask)
```

Automatically
generated suffix,
reading in an extra input
and writing two outputs

Generating For-Each Kernels

```
@torch.compile
def f(params):
    param1 = torch._foreach_add(params, 1)
    param2 = torch._foreach_mul(param1, params)
    return param2

f([torch.randn(i) for i in range(100, 150)])
```



```
def triton foreach(...):
    if xpid >= 0 and xpid < 1:
        xpid_offset = xpid - 0
        xnumel = 100
        xoffset = xpid_offset * XBLOCK
        xindex = xoffset + tl.arange(0, XBLOCK)[:xnumel]
        xmask = xindex < xnumel
        x0 = xindex
        tmp0 = tl.load(in_ptr0 + (x0), xmask)
        tmp1 = 1.0
        tmp2 = tmp0 + tmp1
        tmp3 = tmp2 * tmp0
        tl.store(out_ptr0 + (x0), tmp3, xmask)
    elif xpid >= 1 and xpid < 2:
        xpid_offset = xpid - 1
        xnumel = 101
        xoffset = xpid_offset * XBLOCK
        xindex = xoffset + tl.arange(0, XBLOCK)[:xnumel]
        xmask = xindex < xnumel
        x1 = xindex
        tmp4 = tl.load(in_ptr1 + (x1), xmask)
        tmp5 = 1.0
        tmp6 = tmp4 + tmp5
        tmp7 = tmp6 * tmp4
        tl.store(out_ptr1 + (x1), tmp7, xmask)
    elif xpid >= 2 and xpid < 3:
        ...
```

Taking advantage of block-level control flow!

Optimization Case Study: Concat Lowering Choices

`concat([1,2,3], [4, 5]) => [4, 5]`

Strategy 1: Memory Fusion



```
a = torch.mm(a)
b = torch.mm(b)
c = torch.cat([a, b])
```



```
output_buffer = torch.empty()
a = torch.mm(a, out=output_buffer[:1024])
b = torch.mm(b, out=output_buffer[1024:])
```

Best if all of the cat inputs are computed but unfusable.

Strategy 2: Masked Pointwise Operations



```
a = torch.cos(a)
b = torch.sin(b)
c = torch.cat([a, b])
c = torch.tanh(c)
```

Best if the user of cat is fusible
and inputs are all fusible too



```
masked_a: [2N] = torch.where(idx < N, torch.cos(a), nan)
masked_b: [2N] = torch.where(idx > N, torch.sin(b), nan)
c: [2N] = torch.where(idx < N, masked_a, masked_b)
c = torch.tanh(c)
```

Strategy 3: For-Each Kernel

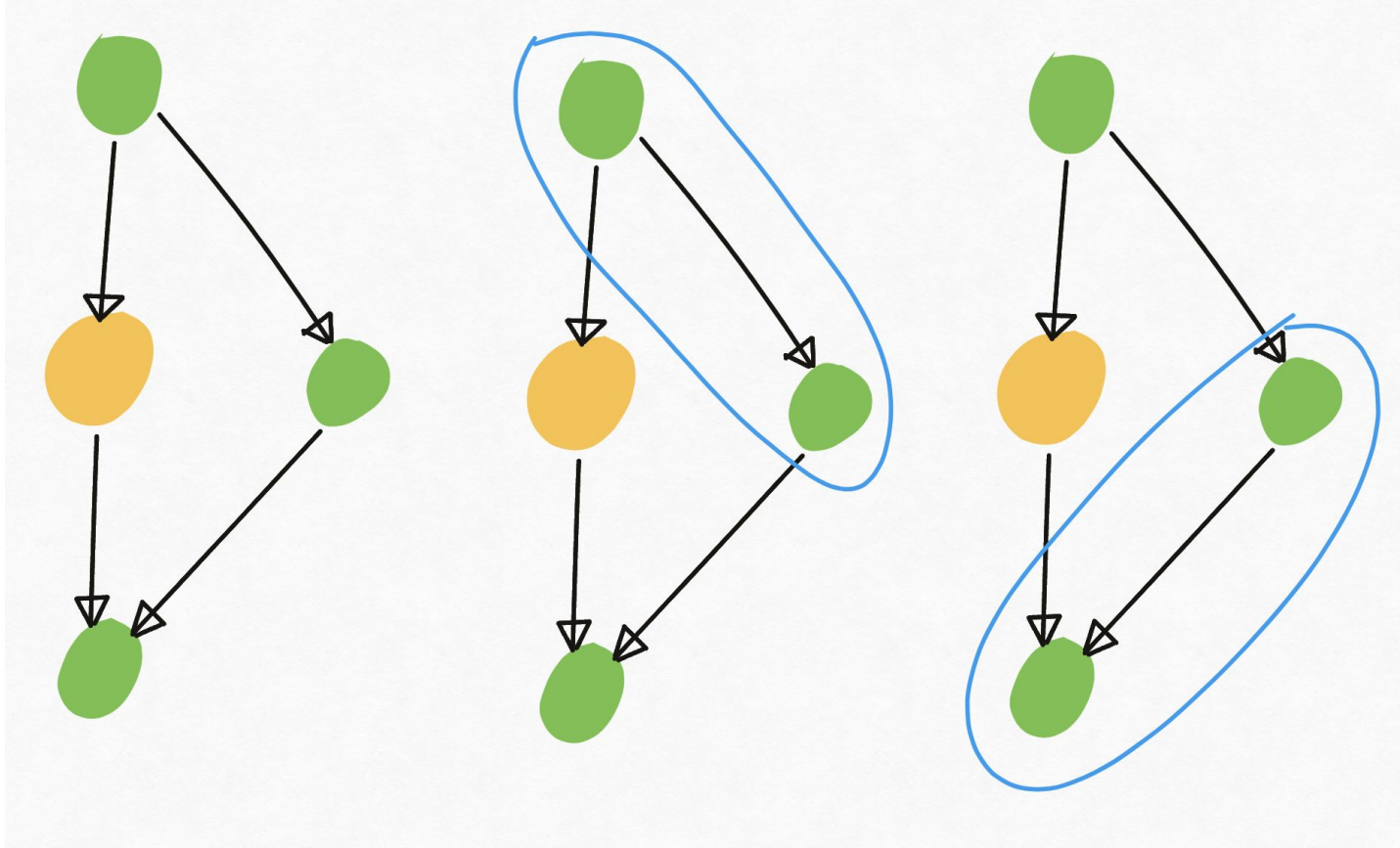
We want to launch one kernel, not 500.



```
c = torch.cat([x_0, x_1, x_2, ..., x_500])
```

Best with many inputs and as a fallback

Case Study: Fusion Segmentation Decisions



Case Study: Rematerialization vs. Reuse

Recomputing operations can allow us to improve our fusion segmentation.

