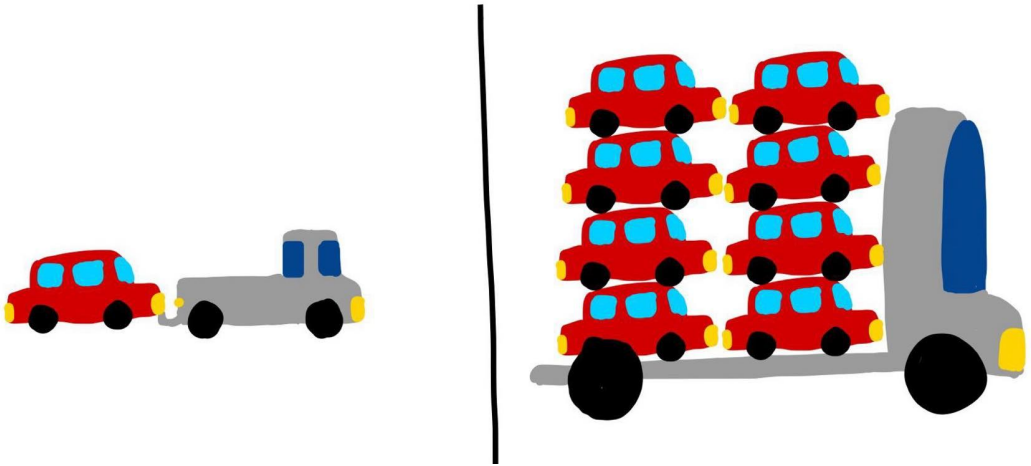
# Optimizing optimizers

in PyTorch

# Runtime vs memory

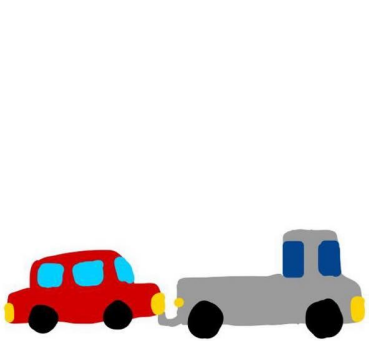
Runtime and memory usage are often at odds with each other

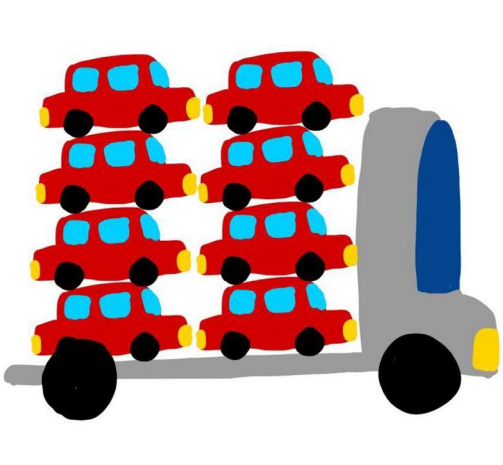



you're towing 512 cars from A to B. which truck do you take?

# Runtime vs memory

Runtime and memory usage are often at odds with each other



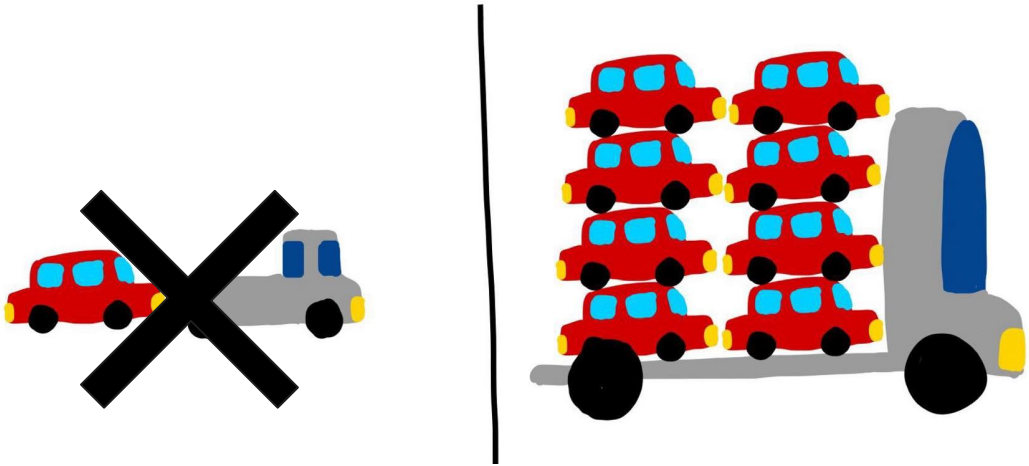




you're towing 512 cars from A to B. which truck do you take?
what if the only way to B had a low clearance bridge?

# Runtime vs memory

Runtime and memory usage are often at odds with each other





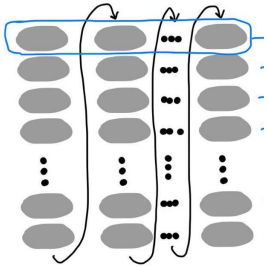
Today, we focus on speeeeeeeeed!

yes this does mean memory will take a hit, disclaimer

# Fusion, the high level idea

# your simplest optimizer

## for loop/single tensor

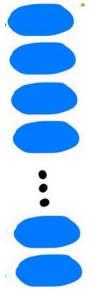


https://github.com/pytorch/pytorch/blob/b5ba80828f77c565bcda7558da97c792af32d517/torch/optim/adamw.py#L362

```
for i, param in enumerate(params):
    grad = grads[i] if not maximize else -grads[i]
    exp_avg = exp_avgs[i]
    exp_avg_sq = exp_avg_sqs[i]
    step_t = state_steps[i]

    # update step
    step_t += 1

    # Perform stepweight decay
    param.mul_(1 - lr * weight_decay)

    # Decay the first and second moment running average coefficient
    exp_avg.lerp_(grad, 1 - beta1)
    exp_avg_sq.mul_(beta2).addcmul_(grad, grad, value=1 - beta2)
    step = _get_value(step_t)

    bias_correction1 = 1 - beta1 ** step
    bias_correction2 = 1 - beta2 ** step

    step_size = lr / bias_correction1

    bias_correction2_sqrt = _dispatch_sqrt(bias_correction2)
    denom = (exp_avg_sq.sqrt() / bias_correction2_sqrt).add_(eps)

    param.addcdiv_(exp_avg, denom, value=-step_size)
```

# horizontally fused optimizer

foreach



https://github.com/pytorch/pytorch/blob/b5ba80828f77c565bcda7558da97c792af32d517/torch/optim/adamw.py#L480

```
torch._foreach_add_(device_state_steps, 1)

# Perform stepweight decay
if weight_decay != 0:
    torch._foreach_mul_(device_params, 1 - lr * weight_decay)

# Decay the first and second moment running average coefficient
torch._foreach_lerp_(device_exp_avgs, device_grads, 1 - beta1)

torch._foreach_mul_(device_exp_avg_sqs, beta2)
torch._foreach_addcmul_(device_exp_avg_sqs, device_grads,
device_grads, 1 - beta2)

. . .

torch._foreach_div_(exp_avg_sq_sqrt, bias_correction2_sqrt)
torch._foreach_add_(exp_avg_sq_sqrt, eps)
torch._foreach_addcdiv_(device_params, device_exp_avgs,
exp_avg_sq_sqrt, step_size)
```

# entirely fused optimizer

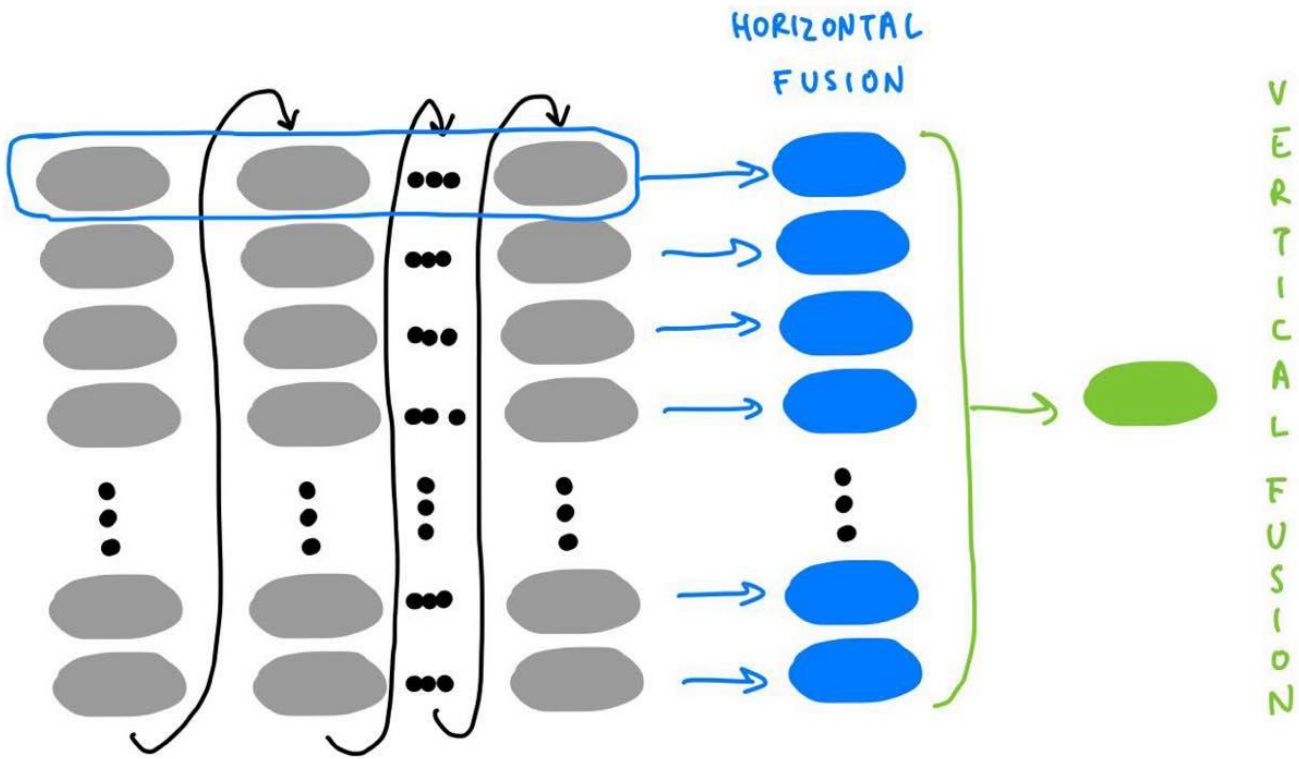
...fused

(thanks NVIDIA)

```
torch._fused_adamw_(
    device_params,
    device_grads,
    device_exp_avgs,
    device_exp_avg_sqs,
    device_max_exp_avg_sqs,
    device_state_steps,
    amsgrad=amsgrad,
    lr=lr,
    beta1=beta1,
    beta2=beta2,
    weight_decay=weight_decay,
    eps=eps,
    maximize=maximize,
    grad_scale=device_grad_scale,
    found_inf=device_found_inf,
)
```

https://github.com/pytorch/pytorch/blob/b5ba80828f77c565bcda7558da97c792af32d517/torch/optim/adamw.py#L616

The Gist: the fewer kernels you launch on CUDA, the faster.

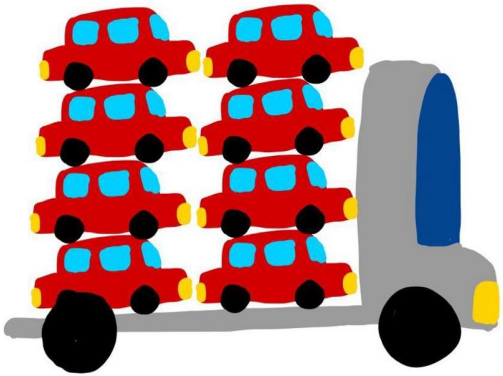


# Fusion, the nitty gritty

starting with multi_tensor_apply

you know how mitochondria is the powerhouse of the cell?

multi_tensor_apply is the powertruck of our speedy optimizers.

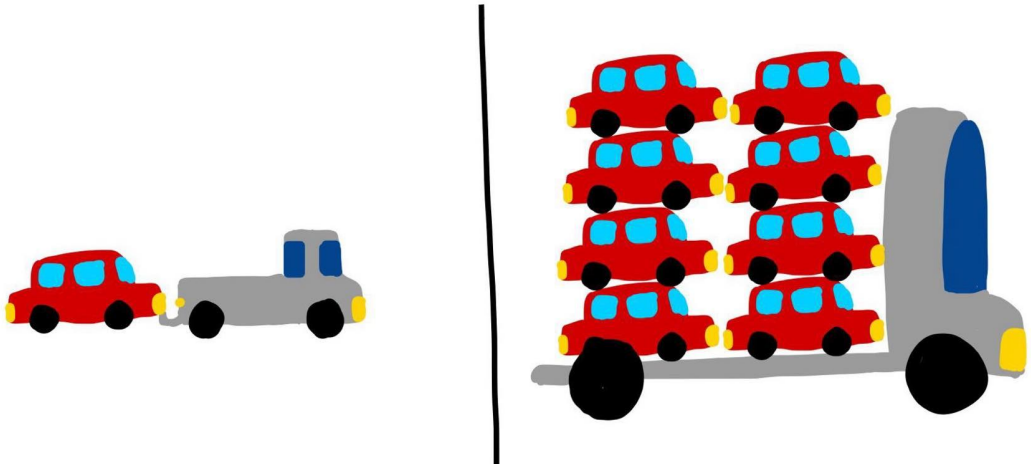



CPU

GPU

multi_tensor_apply allows us to operate over a list of Tensors vs a single tensor.

# Example with torch.add



```
add(
    Tensor self, Tensor other, *,
    Scalar alpha=1
) -> Tensor
```

```
_foreach_add(
    Tensor[] self, Tensor[] other, *,
    Scalar alpha=1
) -> Tensor[]
```

# Under the hood in CUDA

```
add(Tensor self, Tensor other, *,
Scalar alpha=1) -> Tensor
```

A simplified CUDA kernel signature, assuming we have float Tensors:

```
__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {
...
}
```

```
_foreach_add(Tensor[] self,
Tensor[] other, *, Scalar alpha=1)
-> Tensor[]
```

How would you write this one?

# Attempt 1: use std::vector

```
_foreach_add(Tensor[] self, Tensor[] other, *, Scalar alpha=1) -> Tensor[]

__device__ void _foreach_add_kernel(
        std::vector<float*> self,
        std::vector<float*> other,
        std::vector<float*> res,
        float alpha=1) {
...
}
```

Does this work?

```
add(Tensor self, Tensor
other, *, Scalar alpha=1) ->
Tensor

__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {
...
```

# Attempt 1: use std::vector

```
_foreach_add(Tensor[] self, Tensor[] other, *, Scalar alpha=1) -> Tensor[]

__device__ void _foreach_add_kernel(
        std::vector<float*> self,
        std::vector<float*> other,
        std::vector<float*> res,
        float alpha=1) {
...
}
```

**Does this work?**
**No because CUDA doesn't recognize std::vectors!**

```
add(Tensor self, Tensor
other, *, Scalar alpha=1) ->
Tensor

__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {
...
```

# Attempt 2: fine i'll use a C-style array then

```
_foreach_add(Tensor[] self, Tensor[] other, *, Scalar alpha=1) -> Tensor[]

__device__ void _foreach_add_kernel(
        float** self,
        float** other,
        float** res,
        float alpha=1) {
...
}
```

Does this work?

```
add(Tensor self, Tensor
other, *, Scalar alpha=1) ->
Tensor

__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {
...
```

# Attempt 2: fine i'll use a C-style array then

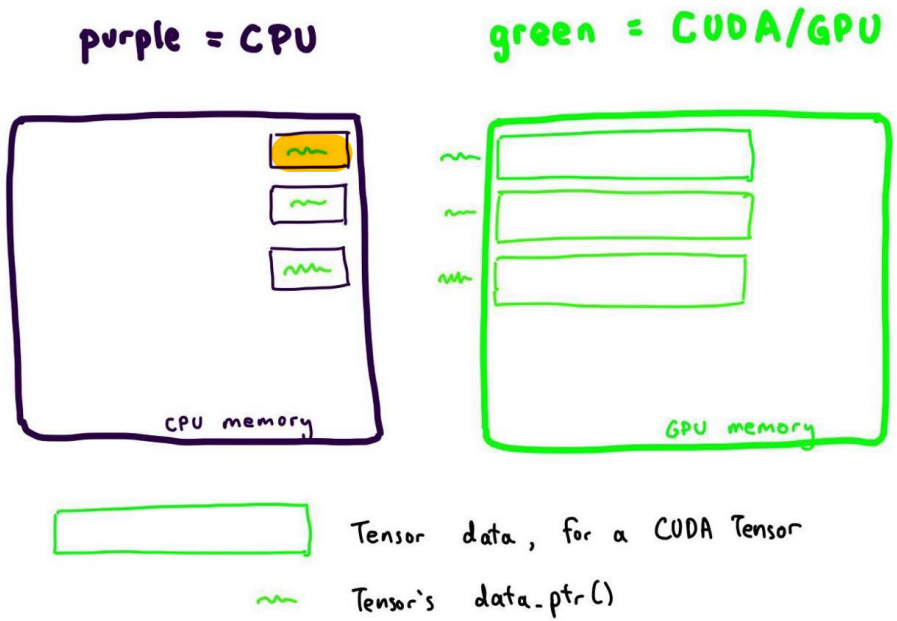
```
_foreach_add(Tensor[] self, Tensor[] other, *, Scalar alpha=1) -> Tensor[]

__device__ void _foreach_add_kernel(
        float** self,
        float** other,
        float** res,
        float alpha=1) {
...
}
```

Does this work?
Nope! This will cause an Illegal Memory Access (IMA) because the outer pointer * is a CPU address!

```
add(Tensor self, Tensor
other, *, Scalar alpha=1) ->
Tensor

__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {
...
```

# Attempt 2: fine i'll use a C-style array then

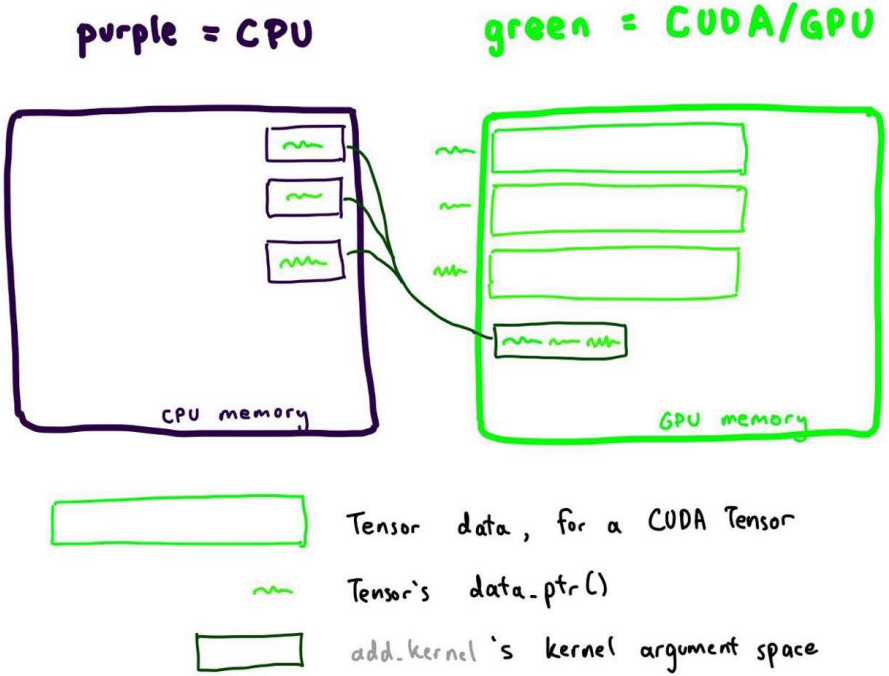
Look again at the add_kernel.




```
add(Tensor self, Tensor
other, *, Scalar alpha=1) ->
Tensor

__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {

...
```

# Attempt 2: fine i'll use a C-style array then

When we dereference in the CUDA kernel, it is OK! The address is in GPU.

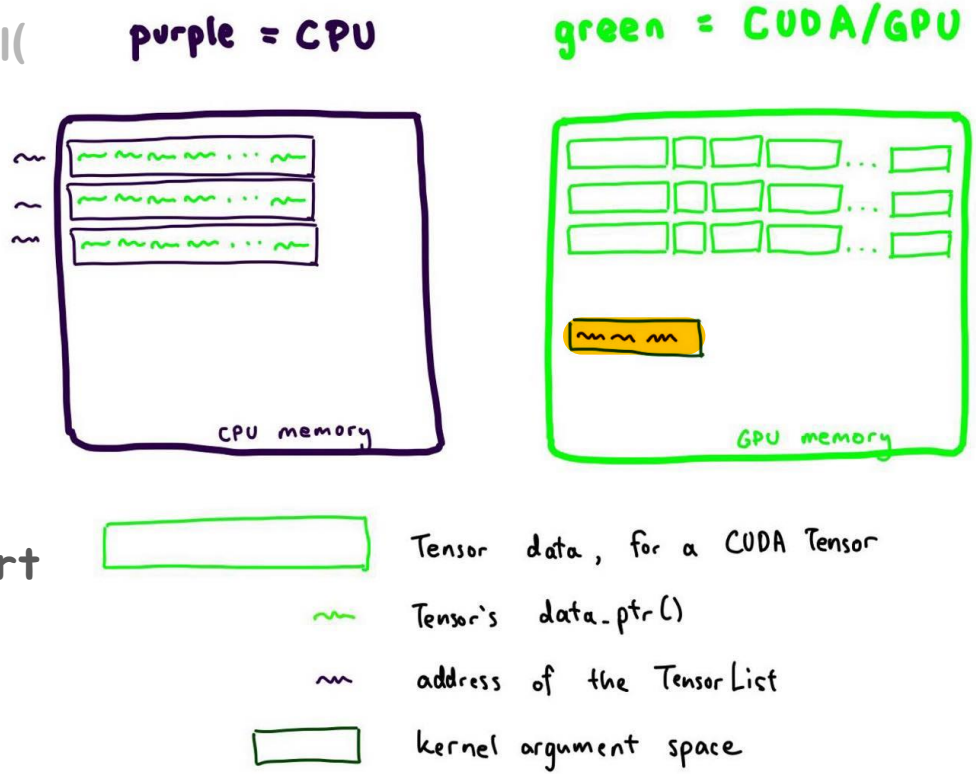



```
add(Tensor self, Tensor
other, *, Scalar alpha=1) ->
Tensor


__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {
...
```

you're going to dereference in here!

# Attempt 2: fine i'll use a C-style array then

```
__device__ void _foreach_add_kernel(
        float** self,
        float** other,
        float** res,
        float alpha=1) {
...
}
```

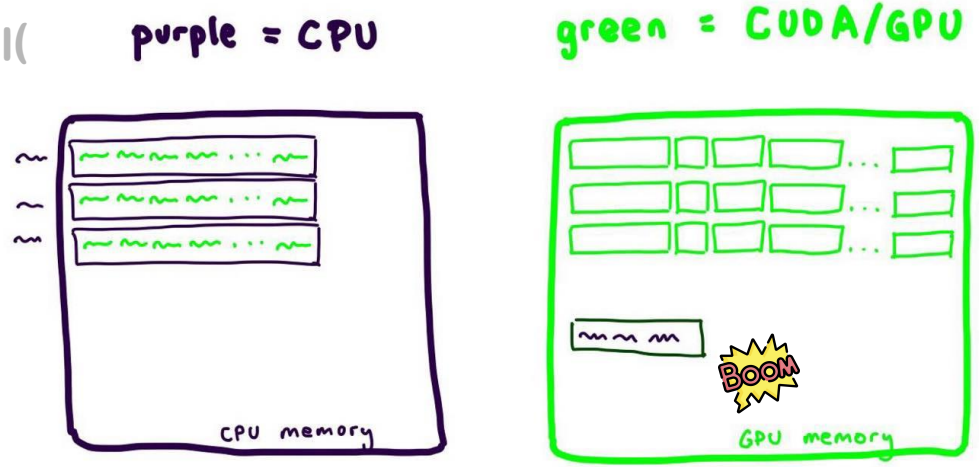



But not so our _foreach_add kernel. While Tensors live in GPU, the list part of TensorLists live in CPU.

# Attempt 2: fine i'll use a C-style array then

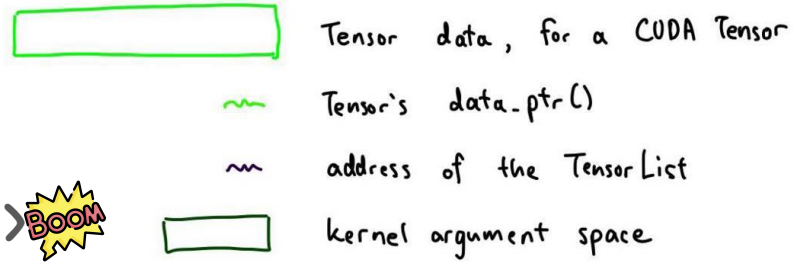
```
__device__ void _foreach_add_kernel(
        float** self,
        float** other,
        float** res,
        float alpha=1) {
... you're going to dereference in here!
}
```

BOOM




But not so our _foreach_add kernel. While Tensors live in GPU, the list part of TensorLists live in CPU.

Dereferencing a CPU address from GPU => BOOM



# Attempt 3: pass by chonky boi (not reference)

```
struct TensorListMetadata {
  const float* addresses[3][NUM_TENSORS];
};

<add all the addresses into the struct>

__device__ void _foreach_add_kernel(
        TensorListMetadata tlm,
        float alpha=1) {
...
}
```

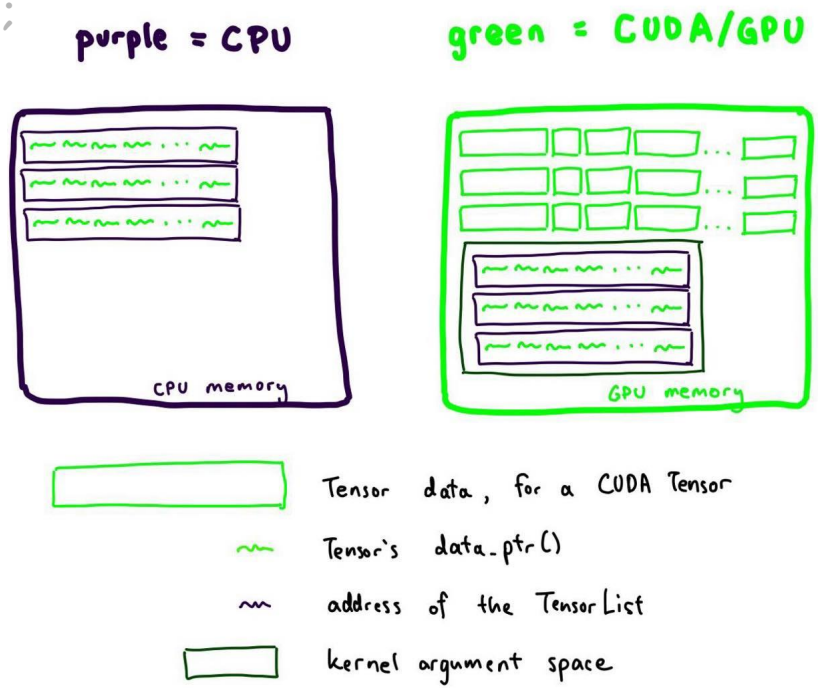
Does this work?

```
add(Tensor self, Tensor
other, *, Scalar alpha=1) ->
Tensor

__device__ void add_kernel(
        float* self,
        float* other,
        float* res,
        float alpha=1) {
...
```

# Attempt 3: pass by chonky boi (not reference)

```
struct TensorListMetadata {
  const float* addresses[3][NUM_TENSORS];
};

<add all the addresses into the struct>

__device__ void _foreach_add_kernel(
        TensorListMetadata tlm,
        float alpha=1) {
...
}
```

Does this work? It passes CI! Yay!



~ the end ~

~ the end ~

jk

# I actually did land a PR like this and it got reverted :(

Cuz some! how! an illegal memory access happened 

for some models (like timm_efficientdet).

I minified the repro to the following and played around with N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

Now would you join me on my (binary) search!

(yes, this example is for norm and not add, but the principle is the same!)

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 

N = 256 👌

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 

N = 256 👌
N = 400 👌

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 

N = 256 👌
N = 400 👌
N = 450 

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 

N = 256 👌
N = 400 👌
N = 450 

N = 425 

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 

N = 256 👌
N = 400 👌
N = 450 

N = 425 

N = 412 👌

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 💥
N = 256 👌
N = 400 👌
N = 450 💥
N = 425 💥
N = 412 👌
N = 420 👌

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 💥
N = 256 👌
N = 400 👌
N = 450 💥
N = 425 💥
N = 412 👌
N = 420 👌
N = 423 👌

# Let's binary search over N.

```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 💥
N = 256 👌
N = 400 👌
N = 450 💥
N = 425 💥
N = 412 👌
N = 420 👌
N = 423 👌
N = 424 💥

# Let's binary search over N.

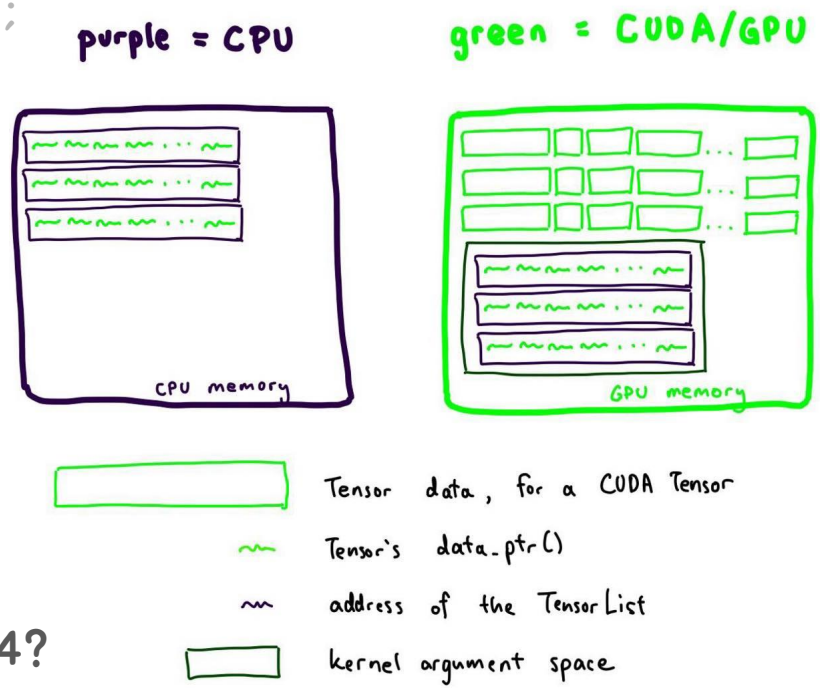
```
params = [torch.rand(2, 3, device="cuda") for _ in range(N)]
torch._foreach_norm(params, ord=1)
torch.cuda.synchronize()
```

N = 500 💥
N = 256 👌
N = 400 👌
N = 450 💥
N = 425 💥
N = 412 👌
N = 420 👌
N = 423 👌
N = 424 💥

what is so special about these numbers?
what could be going on here?

# Attempt 3: pass by chonky boi (not reference) cont.

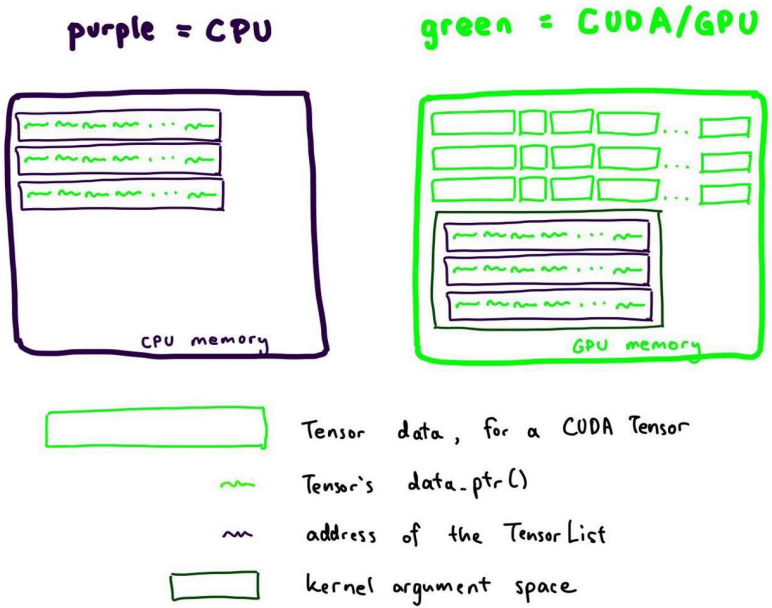
```
struct TensorListMetadata {
  const float* addresses[3][NUM_TENSORS];
};

<add all the addresses into the struct>

__device__ void _foreach_add_kernel(
        TensorListMetadata tlm,
        float alpha=1) {
...
}
```



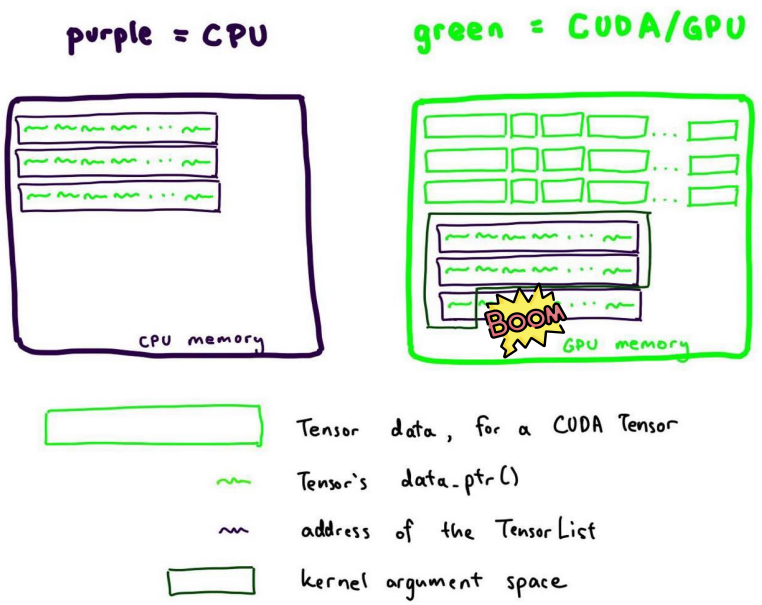

**Does this work? Only if NUM_TENSORS < 424?**

# Attempt 3: pass by chonky boi (not reference) cont.

Expectation:



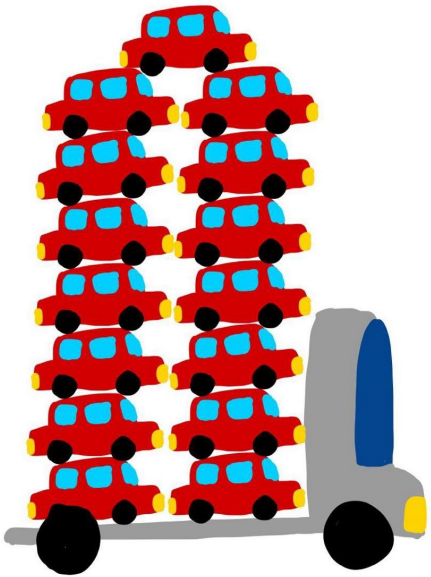

Reality:



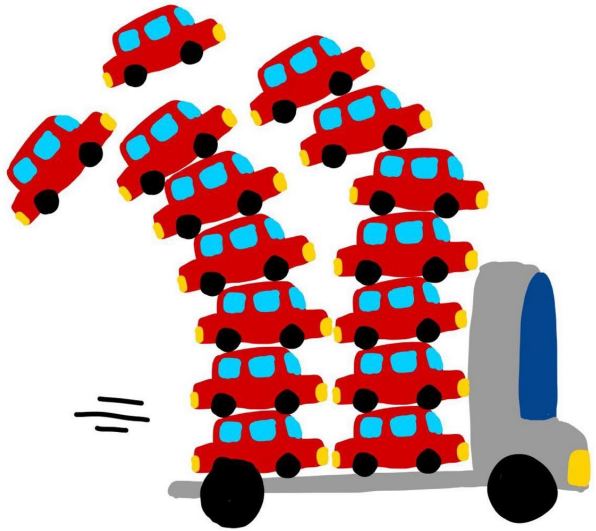

Fun fact: CUDA Kernel argument space has a max limit of 4KB 🥲

# Attempt 3: pass by chonky boi (not reference) cont.

Expectation:

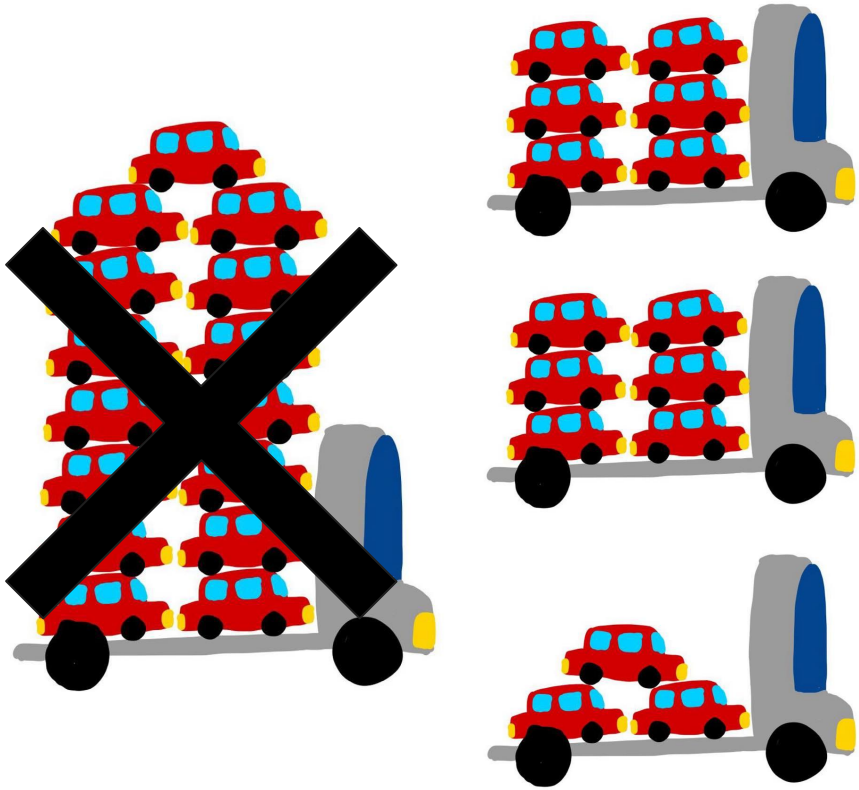


Reality:



Fun fact: CUDA Kernel argument space has a max limit of 4KB 🥲 so what now?

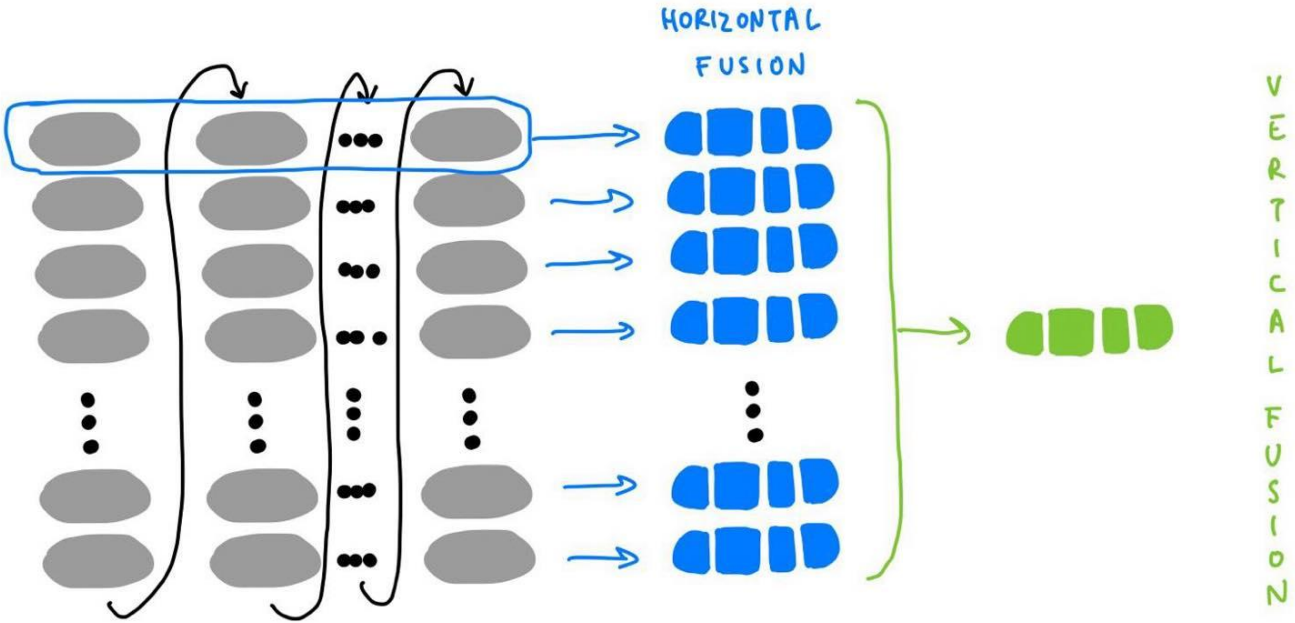
# Attempt 4: just launch more kernels; make more trips



```
struct TensorListMetadata {
  const float*
addresses[3][MAX_NUM_TENSORS];
};

<make multiple structs>
<add a chunk of the addresses to each>
<launch the kernel multiple times>

__device__ void _foreach_add_kernel(
        TensorListMetadata tlm,
        float alpha=1) {

...
}
```

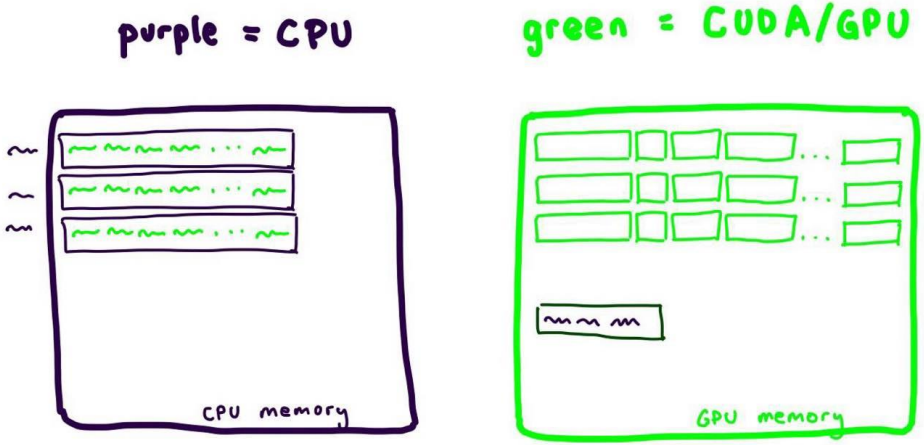
# Attempt 4 *is* what we do today. But we could do better.

While we claim to horizontally fuse into 1 kernel...we often end up with more:



# How: revisit attempt 2

```
__device__ void _foreach_add_kernel(
        float** self,
        float** other,
        float** res,
        float alpha=1) {
...
}
```

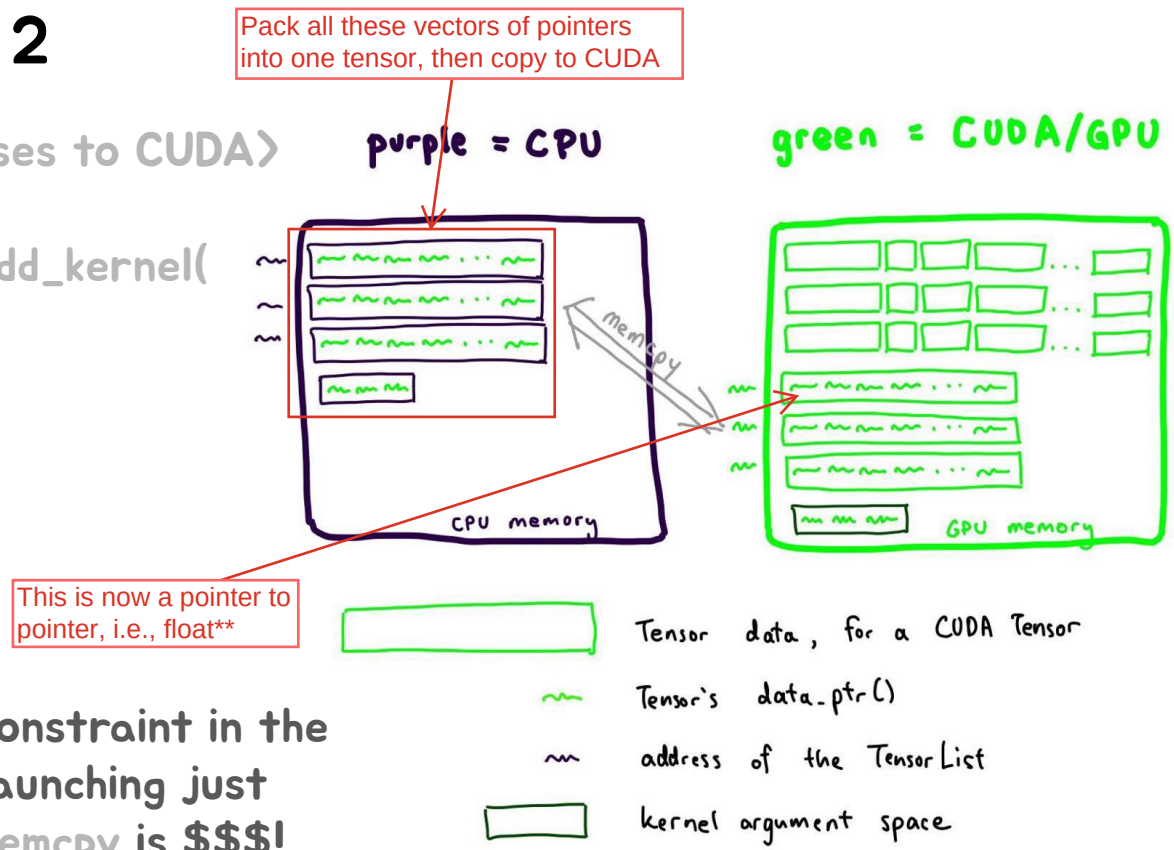



We wanna turn those purple * to green *.

How? Move them to CUDA beforehand!
(thanks Yifu Wang!)

# How: revisit attempt 2

```
<memcpy the lists of addresses to CUDA>

__device__ void _foreach_add_kernel(
        float** self,
        float** other,
        float** res,
        float alpha=1) {
...
}
```

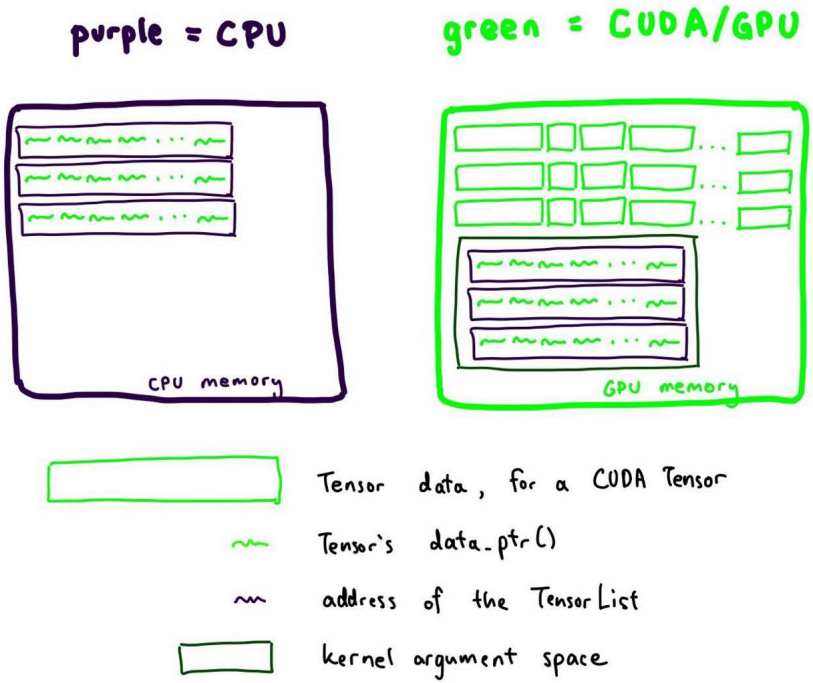



We thereby avoid the 4KB constraint in the kernel argument to enable launching just one kernel. But remember, memcpy is $$$!

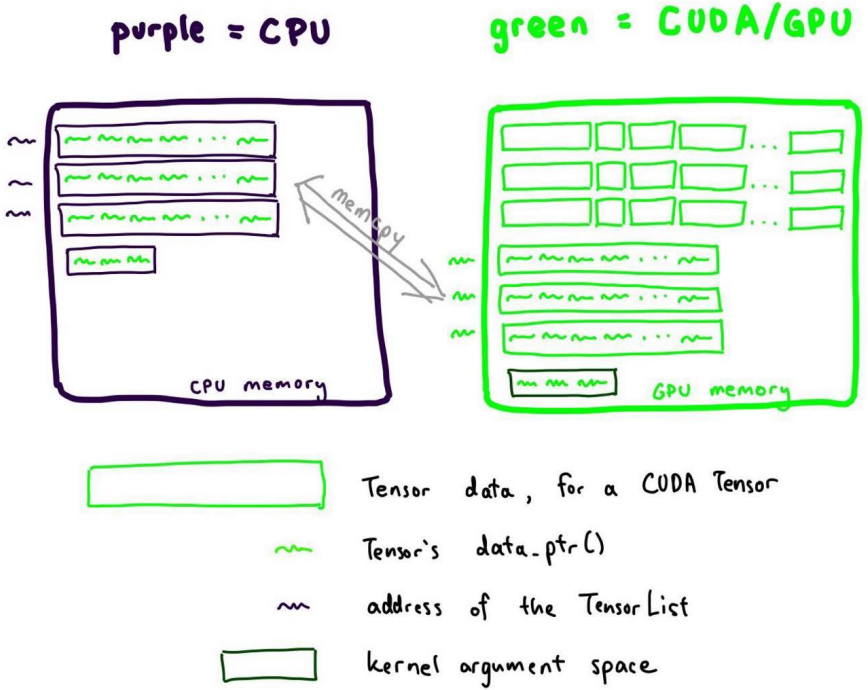
# Conclusion: we will be doing a mix of struct + memcpy

Can we use unified memory here? (currently not supported in PyTorch)

## if it fits, use the struct

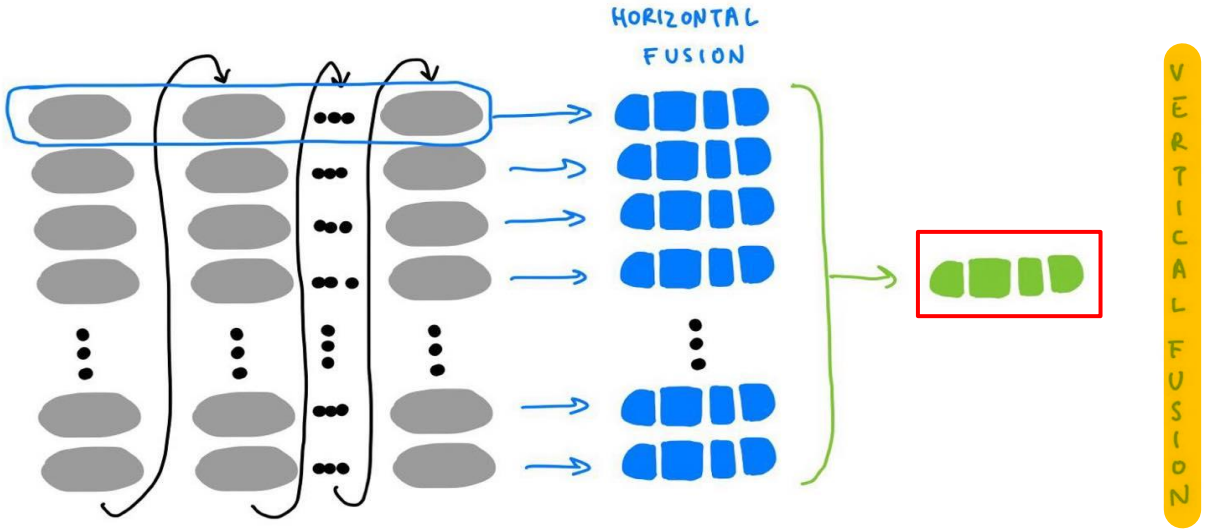



## otherwise, take the memcpy hit



# Did you notice I split up the fused implementation too?

This is cuz our fastest fused impls also rely on multi_tensor_apply!



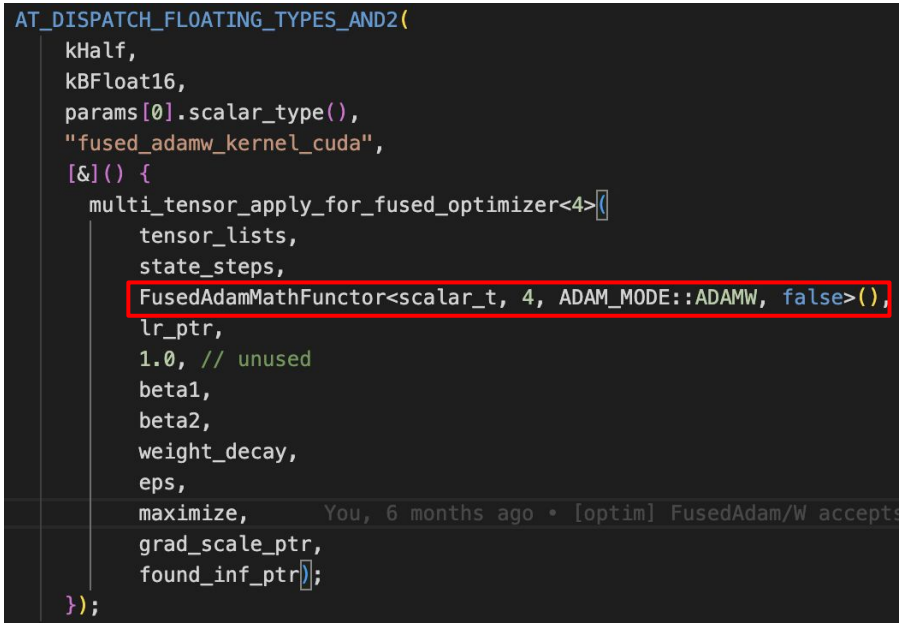
# Did you notice I split up the fused implementation too?

Whereas _foreach_add will call multi_tensor_apply with a Callable that does addition, _fused_adamw will call multi_tensor_apply with a bigger Callable.

```
multi_tensor_apply_kernel<<<
    loc_block_info,
    kBlockSize,
    0,
    at::cuda::getCurrentCUDAStream()>>>(tensorListMeta, callable, args...);
C10_CUDA_KERNEL_LAUNCH_CHECK();
```

# Did you notice I split up the fused implementation too?

Whereas _foreach_add will call multi_tensor_apply with a Callable that does addition, _fused_adamw will call multi_tensor_apply with a bigger Callable.



so let's peek at FusedAdamMathFunctor

```
struct FusedAdamMathFunctor {
  C10_DEVICE __forceinline__ void operator()(
    const auto tensor_loc = tl.block_to_tensor[blockIdx.x];
    const auto chunk_idx = tl.block_to_chunk[blockIdx.x];
    const double lr_double = lr_ptr ? *lr_ptr : lr;

    if (found_inf_ptr && *found_inf_ptr == 1) {
      return;
    }
    const auto [bias_correction1: <dependent type>, bias_correction2_sqrt: <dependent type>] =
        [&]() -> std::pair<double, double> {
      auto* step_count: const float * =
          reinterpret_cast<const float*>(tl.state_steps_addresses[tensor_loc]);      Masaki Kozuki, 2
      const auto bias_correction1: double const = 1 - at::native::pow_(base: beta1, exp: *step_count);
      const auto bias_correction2: double const = 1 - at::native::pow_(base: beta2, exp: *step_count);
      const auto bias_correction2_sqrt: const double = std::sqrt(x: bias_correction2);
      return {a: bias_correction1, b: bias_correction2_sqrt};
    }();

    scalar_type* args[depth];
    scalar_type r_args[depth][kILP];
    const auto n = tl.numel_for_tensor[tensor_loc] - chunk_idx * chunk_size;

    const bool all_aligned{
        init_args<depth>(args, tl, chunk_idx, chunk_size, tensor_loc)};
    if ((n % kILP == 0) && (chunk_size % kILP == 0) && all_aligned) {
      for (int64_t i_start = threadIdx.x;
           i_start * kILP < n && i_start * kILP < chunk_size;
           i_start += blockDim.x) {
#pragma unroll
        for (int i = 0; i < depth; i++) {
          load_store(dst: r_args[i], src: args[i], dst_offset: 0, src_offset: i_start);
        }
        adam_math<scalar_type, opmath_t, depth, adam_mode, amsgrad>(
            r_args,
            lr: lr_double,
```

Locate the thread

```
C10_DEVICE inline void adam_math(
#pragma unroll
  for (int ii = 0; ii < kILP; ii++) {
    // Load values.
    opmath_t param = static_cast<opmath_t>(r_args[kParamIdx][ii]);
    opmath_t grad = static_cast<opmath_t>(r_args[kGradIdx][ii]);
    if (grad_scale_ptr) {
      grad /= (static_cast<double>(*grad_scale_ptr));
    }
    const opmath_t grad_to_store = grad;
    if (maximize) {
      grad = -grad;
    }
    opmath_t exp_avg = static_cast<opmath_t>(r_args[kExpAvgIdx][ii]);
    opmath_t exp_avg_sq = static_cast<opmath_t>(r_args[kExpAvgSqIdx][ii]);
    opmath_t max_exp_avg_sq;
    if (amsgrad) {
      max_exp_avg_sq = static_cast<opmath_t>(r_args[kMaxExpAvgSqIdx][ii]);
    }
    // Update param, grad, 1st and 2nd order momentum.
    if (weight_decay != 0) {
      if constexpr (adam_mode == ADAM_MODE::ORIGINAL) {
        grad += param * weight_decay;
      } else if constexpr (adam_mode == ADAM_MODE::ADAMW) {
        param -= lr * weight_decay * param;
      }
    }
    // todo(crcrpar): use lerp
    // ref: https://developer.nvidia.com/blog/lerp-faster-cuda/
    exp_avg = beta1 * exp_avg + (1 - beta1) * grad;
    exp_avg_sq = beta2 * exp_avg_sq + (1 - beta2) * grad * grad;      Masak
    const opmath_t step_size = lr / bias_correction1;
    opmath_t denom;
    if (amsgrad) {
      max_exp_avg_sq = std::max(max_exp_avg_sq, exp_avg_sq);
      denom = (std::sqrt(max_exp_avg_sq) / bias_correction2_sqrt) + eps;
```

Math part of the optimizer

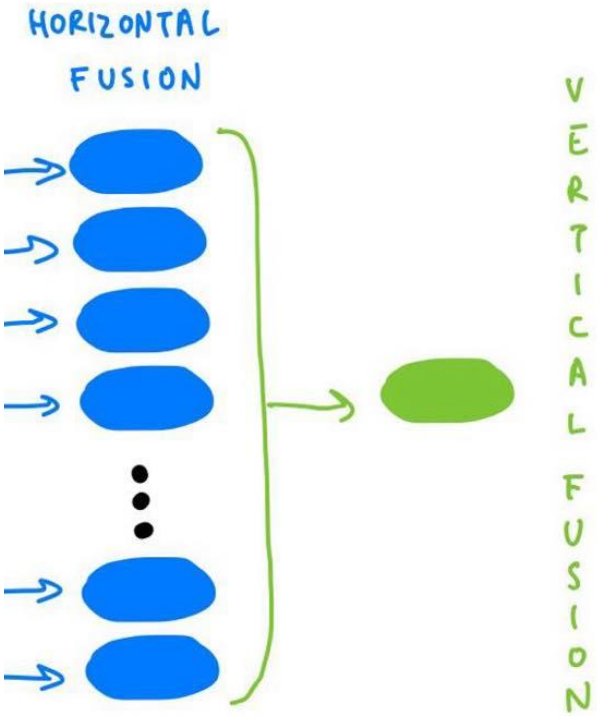
...that was very manual.

What if we could automate vertical fusion?

with just 1 line?

Enter `torch.compile()`

# torch.compile()'s strength is vertical fusion



# How do I use torch.compile() with optimizers?

```
optimizer = torch.optim.AdamW(params)

@torch.compile(fullgraph=False)
def compiled_step():
    optimizer.step()
```

Now call `compiled_step` instead of `optimizer.step()` in your training loop. That's it!

(okay I suppose that was 2 lines)

# Inductor will generate a large triton kernel

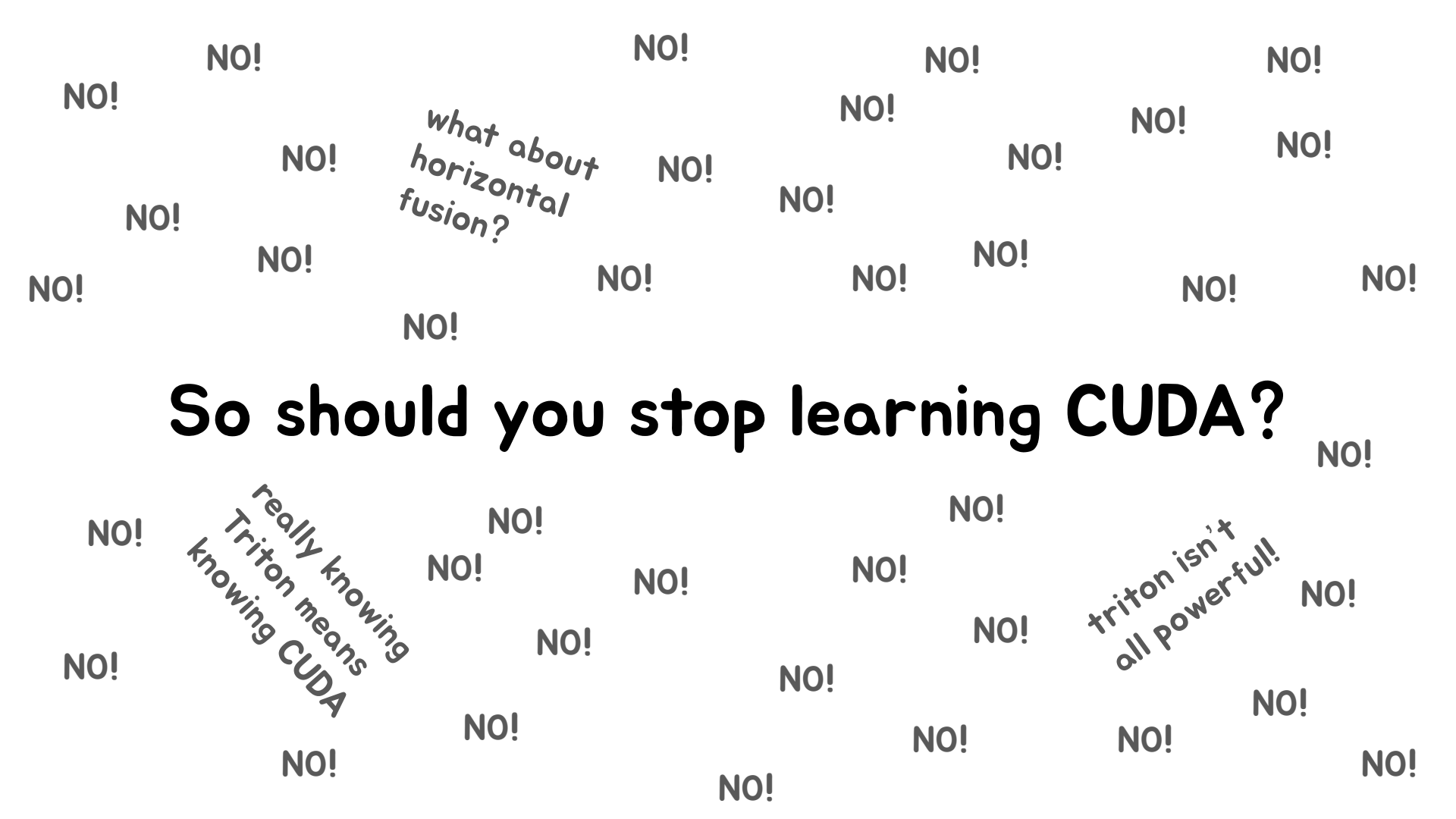
```
xpid_offset = xpid - 0
xnumel = 512
xoffset = xpid_offset * XBLOCK
xindex = xoffset + tl.arange(0, XBLOCK)[:]
xmask = xindex < xnumel
x0 = xindex
tmp0 = tl.load(in_ptr0 + (x0), xmask)
tmp1 = tl.load(in_ptr1 + (x0), xmask)
tmp6 = tl.load(in_ptr2 + (x0), xmask)
tmp13 = tl.load(in_ptr3 + (x0), xmask)
tmp17 = tl.load(in_ptr4 + (0))
tmp18 = tl.broadcast_to(tmp17, [XBLOCK])
tmp2 = tmp1 - tmp0
tmp3 = 0.09999999999999998
tmp4 = tmp2 * tmp3
tmp5 = tmp0 + tmp4
tmp7 = 0.999
tmp8 = tmp6 * tmp7
tmp9 = tmp1 * tmp1
tmp10 = 0.0010000000000000009
tmp11 = tmp9 * tmp10
tmp12 = tmp8 + tmp11
tmp14 = 0.99999
tmp15 = tmp13 * tmp14
tmp16 = tl.sqrt(tmp12)
tmp19 = tl.math.pow(tmp7, tmp18)
tmp20 = 1.0
tmp21 = tmp19 - tmp20
tmp22 = -tmp21
tmp23 = tl.sqrt(tmp22)
tmp24 = tmp16 / tmp23
tmp25 = 1e-08
tmp26 = tmp24 + tmp25
tmp27 = 0.9
tmp28 = tl.math.pow(tmp27, tmp18)
tmp29 = tmp28 - tmp20
tmp30 = 0.001
tmp31 = tmp29 / tmp30
tmp32 = 1 / tmp31
tmp33 = tmp26 / tmp32
tmp34 = tmp5 / tmp33
tmp35 = tmp15 + tmp34
tl.store(out_ptr0 + (x0), tmp5, xmask)
tl.store(out_ptr3 + (x0), tmp35, xmask)
tl.store(out_ptr4 + (x0), tmp12, xmask)
```

# When does it work? (or not work?)

- You must have CUDA capability 7.0+ for Triton
- All optimizers in pytorch/pytorch with a foreach implementation are now compilable
  - So everything except L-BFGS and SparseAdam
- Vertical fusion of *any* sequence of supported _foreach_* ops should work!
  - try out your experimental optimizers!
  - open an issue when this isn't true

Compiled optimizers is in beta! Try it out and complain lots here!

# So should you stop learning CUDA?


NO!
what about horizontal fusion?
So should you stop learning CUDA?
really knowing Triton means knowing CUDA
triton isn't all powerful!

thanks! questions?