Clad Meets PyTorch: Gradients Ahead of Time

CRCon Presentation



CRCon2026/Kacent Huang

A glimpse of the work

A LibTorch program $\xrightarrow{\text{Supply Tensor derivatives}}$ Clad-generated gradients in compile time

A glimpse of the work

A LibTorch program —Supply Tensor derivatives→ Clad-generated gradients in compile time

How to make customization simpler?

How to make it faster and use less memory?

The same setup, repeated for every function

Before: a reverse-forward helper

```
namespace clad::custom_derivatives {
ValueAndAdjoint<Tensor, Tensor>
make_tensor_reverse_forw(double x,
                         double /*d_x*/) {
  Tensor value = make_tensor(x);
  return {value, at::zeros_like(value)};
}
}
```

What the helper supplies

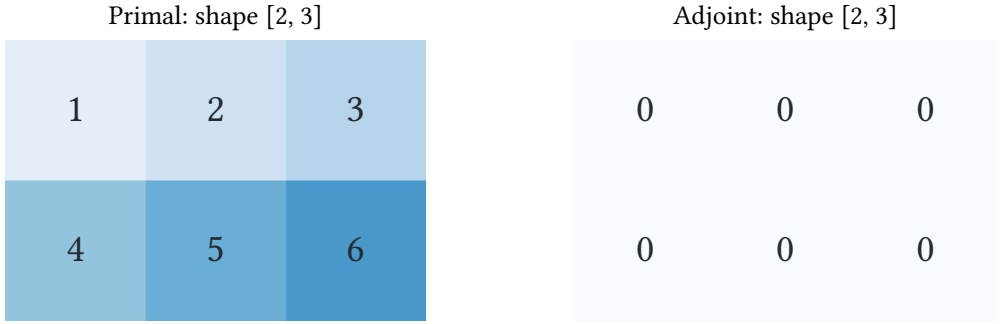
The original result, plus storage for its adjoint.

Another function returns the same Tensor type. Why repeat the same initialization rule?

Let the type define its zero

```
namespace clad {
at::Tensor zero_like(
    const at::Tensor& value) {
  at::NoGradGuard guard;
  return at::zeros_like(value);
}
}
```

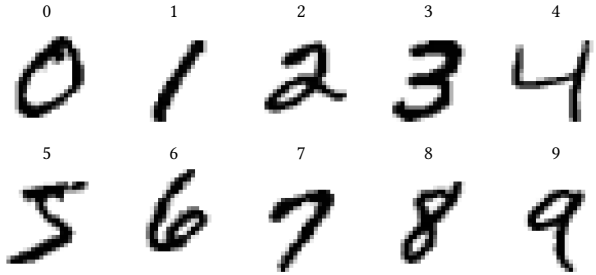
One rule for the Tensor type.



Same shape, dtype and device.
Independent zero storage.

Clad can construct a default result adjoint for a by-value return when no explicit reverse-forward helper is provided.

MNIST: a fixed training workload



784
Image pixels

→

64
Linear + ReLU

→

10
Class scores

$$H = \mathrm{ReLU}(XW_1 + b_1), \quad \hat{Y} = HW_2 + b_2$$

$$L = \frac{1}{2B} \sum_{b,k} \left(\hat{Y}_{bk} - Y_{bk}\right)^2$$

Data
60,000 training images · 10,000 test images.

Training
Batch 128 · FP32 · SGD 0.05 · 10 epochs.

Keep the model and training loop fixed. Improve the Clad backward one step at a time.

One CPU thread · -O3 · identical initialization and batch order.

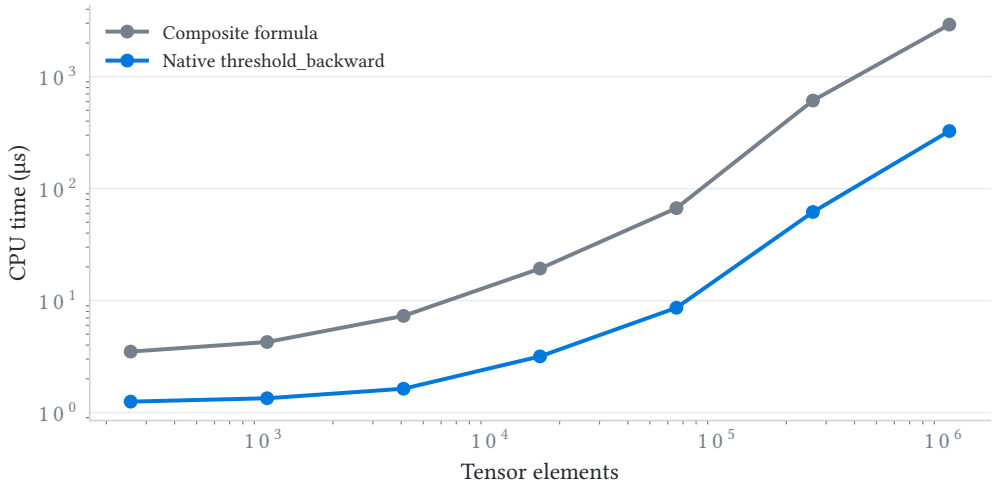
Before S0: use native backward kernels

Compose the formula

```
auto mask = x > 0;
auto d_x = g * mask;
```

Use the native backward kernel

```
auto d_x =
    at::threshold_backward(g, x, 0);
```



At 1,048,576 elements: 2919 µs ⟶ 327 µs, an 8.9× local improvement.

S0: keep forward values inside the loss

Model behind a helper

```
Tensor loss(...) {
  auto y = predict(x, w1, b1, w2, b2);
  return squared_error(y, target);
}
```

Forward	Pullback	Reverse
Run model	Replay model	Propagate

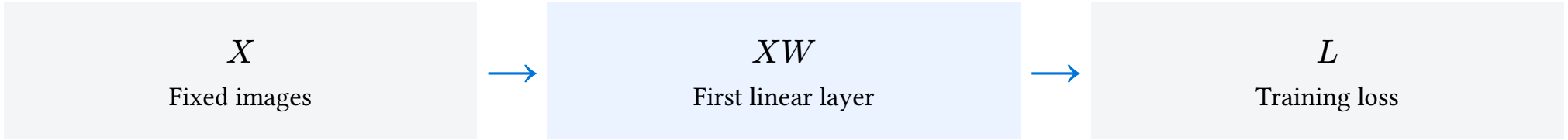
Forward inside the differentiated loss

```
Tensor loss(...) {
  auto h = (x.mm(w1) + b1).relu();
  auto y = h.mm(w2) + b2;
  return squared_error(y, target);
}
```

Forward	Reuse	Reverse
Keep values	Saved values	Propagate

S0 starts here: native ReLU backward, forward in the root loss, and ordinary zero_like.

S1: compute only the gradients we need



Requested weight contribution

$$d_W = X^T G$$

Needed to update the model.

Unneeded image contribution

$$d_X = G W^T$$

One matrix product and a 128 × 784 buffer.

For this fixed-data training loop, we can omit image and target adjoints: 397 KiB of FP32 storage per full batch.

S1: select a partial pullback with no_adjoint

The request

```
auto grad = clad::gradient<
  clad::opts::enable_va,
  clad::opts::assume_unselected_inactive>(
    loss, "w1,b1,w2,b2");
```

The user promises that the unselected root inputs carry no active state.

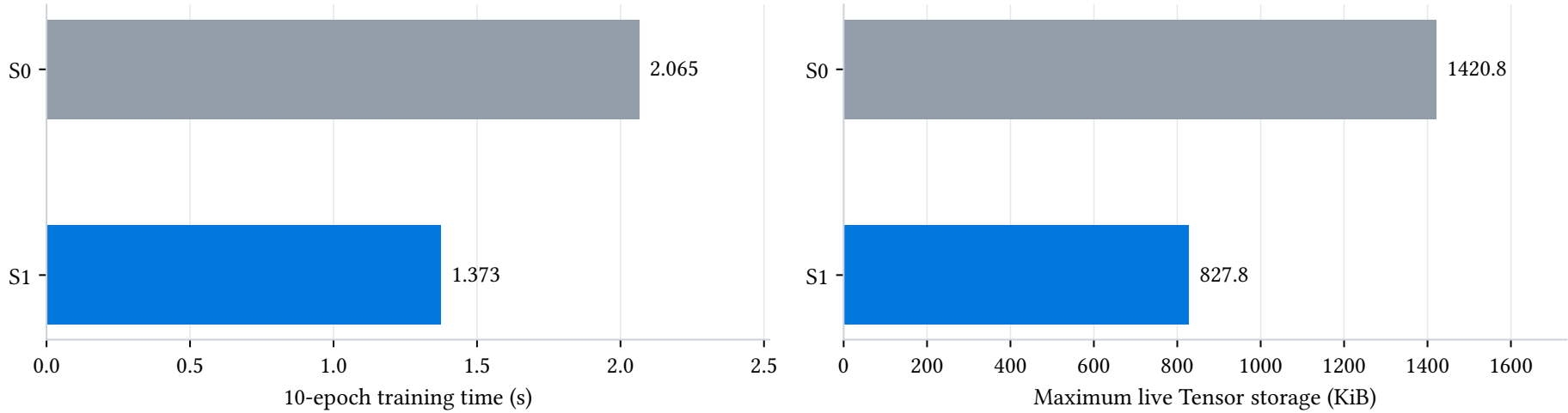
The selected custom overload

```
void mm_pullback(
    const Tensor& x, const Tensor& w,
    Tensor g, clad::no_adjoint,
    Tensor* d_w) {
  accumulate_mm(d_w, x.t(), g, w);
}
```

Image slot	Weight slot
`no_adjoint`	`Tensor* d_w`

S0 ⟶ S1: omit the fixed-data gradients

Same executable and backend; enable `assume_unselected_inactive`.



−33.5%
Training time / S0

−41.7%
Maximum live Tensor storage / S0

35 ⟶ 29
Allocations per batch

S2: use LibTorch's fused addmm_ kernel

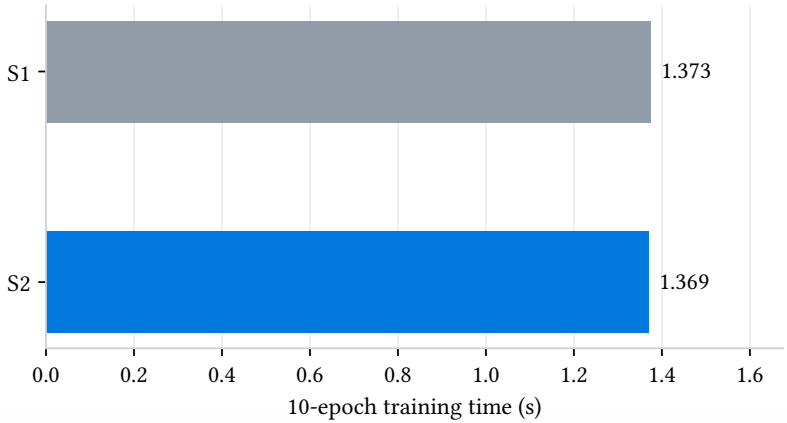
Separate product and addition

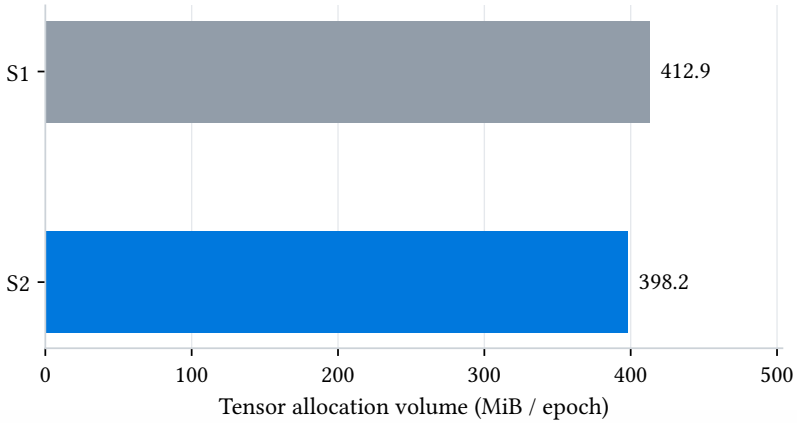
```
auto tmp = x.t().mm(g);
d_w.add_(tmp);
```

→

Native accumulating kernel

```
d_w.addmm_(x.t(), g);
// No separate product Tensor.
```

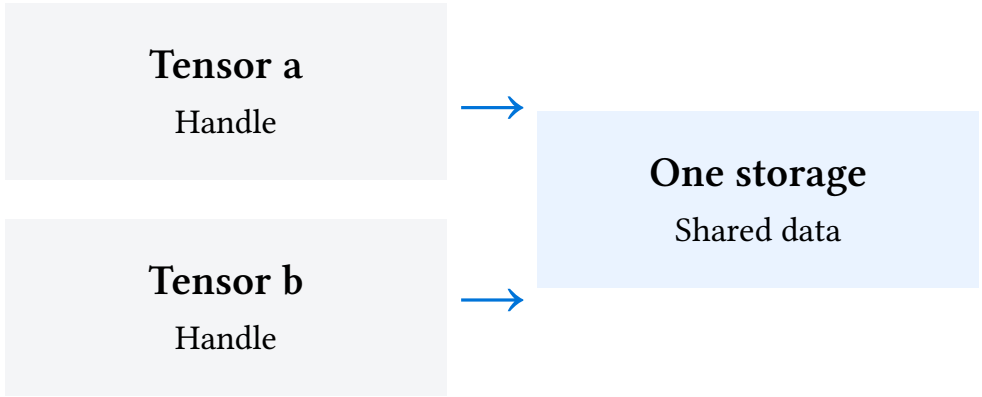




29 → 28 allocations per batch; maximum live storage stays at 827.8 KiB.

S3: reuse buffers and avoid extra copies

A Tensor copy is a handle copy



Mutating one alias can change the other. A deep clone creates independent data.

When installing a first contribution

```
// destination is still empty
if (owns_adjoint_storage(contribution))
  *destination = std::move(contribution);
else
  *destination = contribution.clone();
```

In MNIST, a fresh ReLU contribution can become the empty receiver adjoint without another clone.

28 ⟶ 27 allocations per batch; 398.2 MiB ⟶ 395.9 MiB allocated per epoch.

S4: remove an adjoint nobody reads

Generated code around a temporary receiver

```
Tensor t = a + b;
Tensor unused = clad::zero_like(t);
// The outer member call uses d_t instead.
Tensor d_t = {};
relu_pullback(&t, g, &d_t);
add_pullback(a, b, d_t, ...);
```

Keep the primal value. Skip the default adjoint that the receiver path discards.

Before

Two adjoint slots; one is unread

After

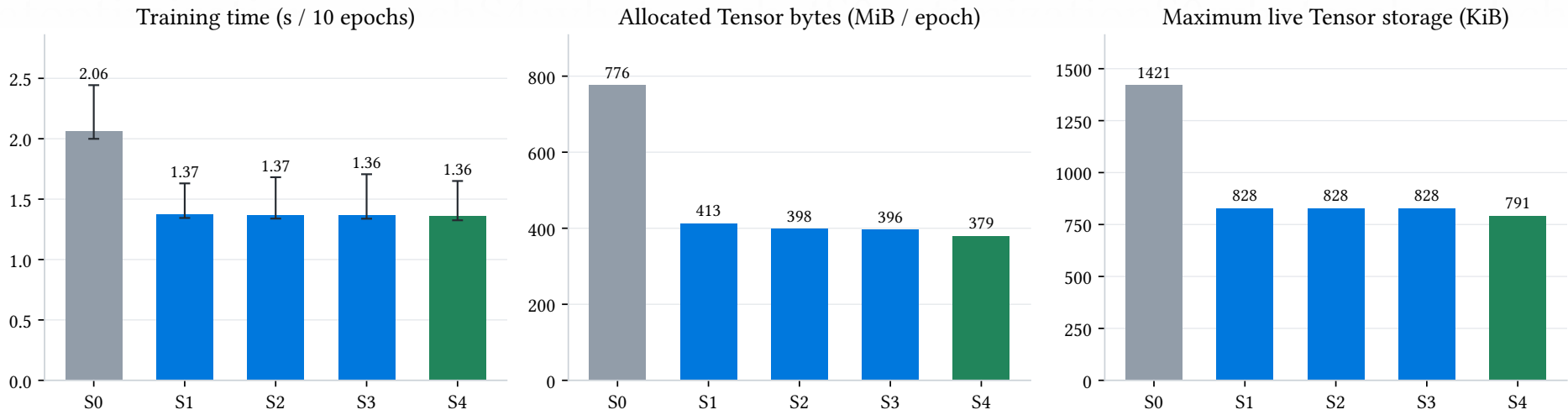
Only the slot used by backward

−3

Tensor allocations per batch

S3 ⟶ S4: 827.8 KiB ⟶ 790.8 KiB maximum live storage; 395.9 MiB ⟶ 379 MiB allocated per epoch.

S0 ⟶ S4: what each optimization changes

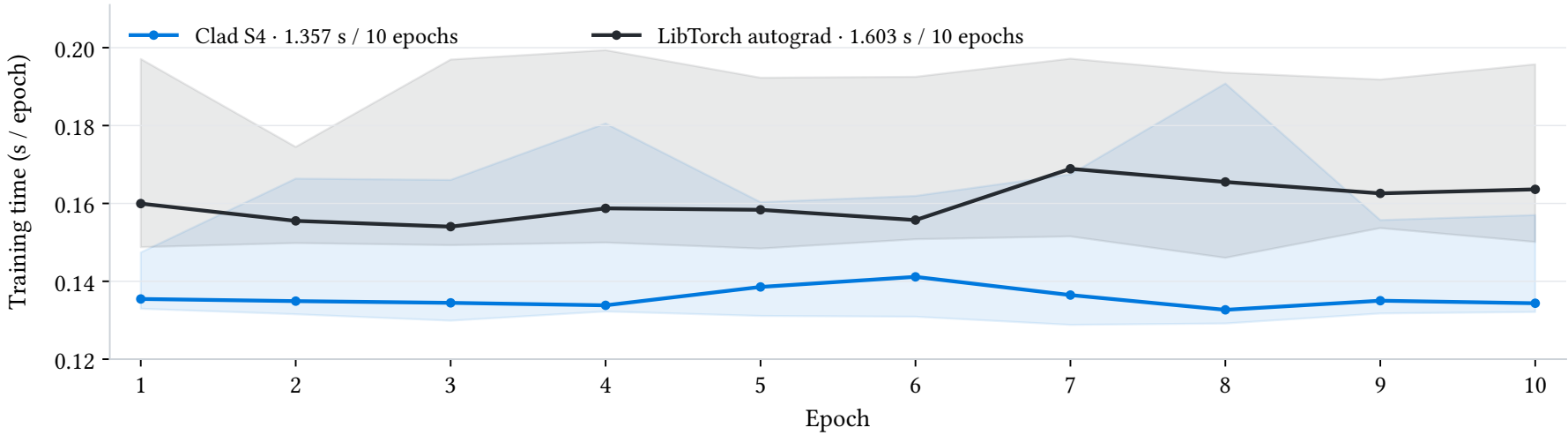


S0	**S1**	**S2**	**S3**	**S4**
Root loss	Partial pullbacks	Fused mm	Buffer reuse	Adjoint fix

Each stage retains the previous changes. All final parameters agree exactly (93.9% accuracy).

S1–S4 timing differences are within run-to-run variation. Whiskers show whole-run min–max.

S4 vs LibTorch: training time and memory

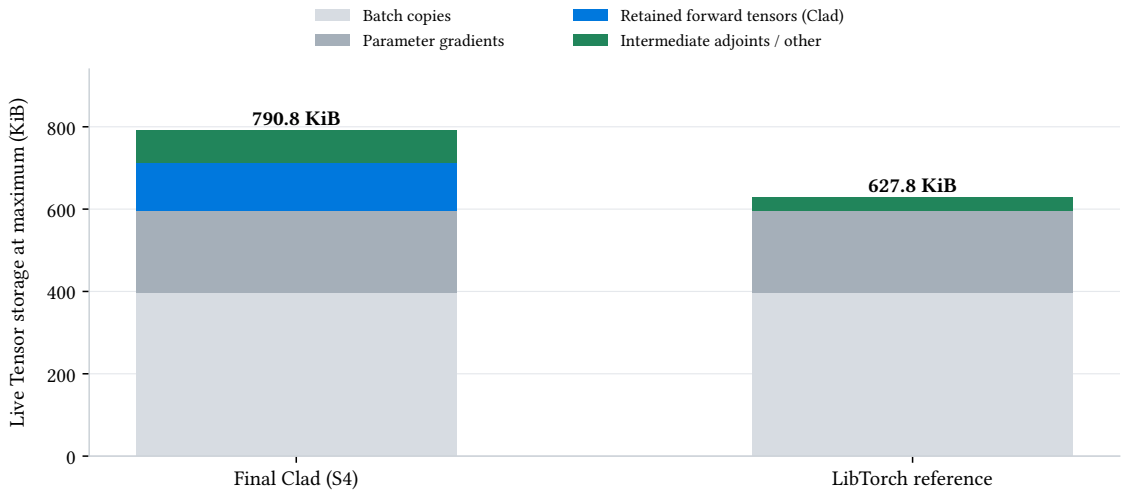


−15.4%
Training time in this experiment

+26%
Maximum live Tensor storage

93.9%
Accuracy in both; parameter difference 0

S4 vs LibTorch: where is the extra memory?



163 KiB

Final Clad / LibTorch gap

At its maximum, Clad still holds:
116 KiB of forward values;
79 KiB of intermediate adjoints.

Native autograd has already released the large forward values before its maximum.

Locate the retained storage, then release it

```
Tensor _r3 = clad::zero_like(_t2);
Tensor _d_hidden = _r3;
// ...
relu_pullback(
    &_t1,
    std::move_if_noexcept(_d_hidden),
    &_r0);
// Other reverse operations follow.
// _r3 and _t1 are still alive.
```

Handles keep storage alive

`_r3` keeps gradient storage alive after `_d_hidden` is moved. `_t1` keeps the ReLU input alive. Both outlive their last use.

Manual release after last use

Drop the extra handle. Release the ReLU input after its pullback. Apply this at every local Tensor's final use.

Locate the retained storage, then release it

```
Tensor _r3 = clad::zero_like(_t2);
Tensor _d_hidden = _r3;
_r3.reset();  // Drop the extra handle.
// ...
relu_pullback(
    &_t1,
    std::move_if_noexcept(_d_hidden),
    &_r0);
_t1.reset();  // Its final use is over.
_d_hidden.reset();
```

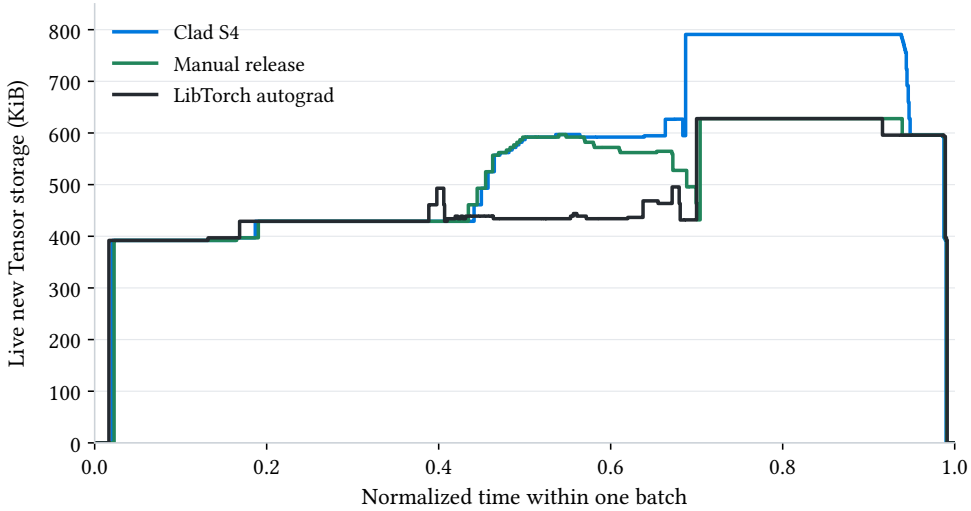
Handles keep storage alive

`_r3` keeps gradient storage alive after `_d_hidden` is moved. `_t1` keeps the ReLU input alive. Both outlive their last use.

Manual release after last use

Drop the extra handle. Release the ReLU input after its pullback. Apply this at every local Tensor's final use.

Manual release closes the remaining memory gap



−20.6%

Maximum live Tensor storage

Case	KiB
Clad S4	790.8
Manual release	627.8
LibTorch	627.8

Releasing handles after last use brings Clad to LibTorch's peak: 627.8 KiB. Final parameter difference: 0.

Future Works

Reuse forward values across function calls

Custom `reverse_forw` and `pullback` pairs can already share intermediate values through `clad::pullback_state`. Generate this state automatically for composed functions, so backward can reuse forward values instead of recomputing them.

Release values after their last use

Use last-use and alias analysis to release local Tensor handles when they are no longer needed.
Turn the releases validated by our manual experiment into an automatic compiler transformation.