Flashlight: Enabling Innovation in Tools for Machine Learning

Jacob Kahn[1], Vineel Pratap[1], Tatiana Likhomanenko[2], Qiantong Xu[3], Awni Hannun[4], Jeff Cai[5], Paden Tomasello[1], Ann Lee[1], Edouard Grave[1], Gilad Avidov[1], Benoit Steiner[1], Vitaliy Liptchinsky[6], Gabriel Synnaeve[1], Ronan Collobert[2]

1 — FAIR, Meta AI
2 — Apple
3 — Samba Nova Systems
4 — Zoom, Inc.
5 — Independent
6 — Paxos Trust Company

Presenter: Jacob Kahn <jacobkahn@fb.com>

ICML 2022 | Baltimore, Maryland

FAIR @ Meta AI

GOAL

Enable fundamental research in ML computation via frameworks with deeply-customizable internals.

What is Flashlight?

Tape-based Automatic Differentiation

```
Variable cos(const Variable& input) {
  auto result = fl::cos(input.tensor()); // get a Tensor from a Variable
  // Called with backward() to compute gradients for this op's inputs
  auto gradFunc = [](std::vector<Variable>& inputs,
                     const Variable& gradOutput) {
    inputs[0].addGrad( // Add a gradient to the input
        Variable(gradOutput * negate(sin(inputs[0].tensor())), false));
  };
  // Construct a Variable from a Tensor and a gradient-computing function
  return Variable(result, {input}, gradFunc);
}
```

Modules and Models

```
Sequential model;

model.add(View(fl::Shape({IM_DIM, IM_DIM, 1, -1})));
model.add(Conv2D(
    1 /* input channels */,
    32 /* output channels */,
    5 /* kernel width */,
    5 /* kernel height */,
    1 /* stride x */,
    1 /* stride y */,
    PaddingMode::SAME; /* padding mode */,
    PaddingMode::SAME; /* padding mode */));
model.add(ReLU());
```

Agenda

Why Flashlight?

Building small frameworks

Powerful internal APIs

Simplicity = performance

Past and present applications
and directions

01 Building Small Frameworks

Compact Tools and Research

Having flexible tools enables challenging foundational assumptions in ML. Tools shape our worldview. What affects tool flexibility?

- Framework complexity
- Internal API availability and size
- Compilation time
- Opinionated interfaces

Fast Compilation

As frameworks grow, compile times for incremental changes regress. This inhibits rapid prototyping and iteration on computational research done inside frameworks.

120+

PyTorch — mean incremental compile time, CPU minutes

350+

TensorFlow — mean incremental compile time, CPU minutes

1

Flashlight — mean incremental compile time, CPU minutes

Incremental compilation was benchmarked via recompilation after trivial modifications to 100 random files in each framework. Only relevant subsystems and common components were eligible. Stdev was under 5% of the mean.

Small Operator Sets

The proliferation of large operator sets makes foundational modifications to primitive operations intractable. Using existing codebases with new computational techniques can require significant, wide-ranging changes.

2100+
Operators in PyTorch

1400+
Operators in TensorFlow

50+
Operators in Flashlight

Source: framework-level operator schemas.

50+

PyTorch ops that implicitly perform Tensor addition

20+

TensorFlow ops that implicitly perform Tensor addition

1

Flashlight ops that implicitly perform Tensor addition

Source: framework-level operator schemas.

02 Powerful Internal APIs

A Small API Surface for Tensor Computation

Making changes to tensor internals facilitates developing compilers, new computation models, and general optimizations in parallel computation.

Tensor backends supporting new hardware or embedded systems can be easily added adapted with no changes to model code.

Define a TensorAdapter abstraction for tensor state:

```
class MyTensorImpl : public TensorAdapter {
  // State information goes here (e.g. buffers, shape)
 public:
  // Metadata
  const Shape& shape() override;
  dtype type() override;
  // Ops on Tensors
  Tensor flatten() const override;
  // …
};
```

A Small API Surface for Tensor Computation

Flashlight's Tensor abstraction isn't opinionated to any particular computation model — its single API accommodates eager, lazy and static setups.

Tensor implementations can store arbitrary state and can compose operations in implementation-defined patterns.

Define a TensorBackend abstraction for defining computations on tensors:

```
class MyTensorBackend : public TensorBackend {
  // State information goes here
  // (e.g. compute streams, compiler state)
 public:
  // Tensor operation primitives
  Tensor add(const Tensor& lhs, const Tensor& rhs) override;
  Tensor minimum(const Tensor& lhs, const Tensor& rhs) override;
  // ...
};
```

Full Control of Memory Management

Control how memory is managed on accelerators via Flashlight's ArrayFire tensor backend and the corresponding internal API for memory management.

This enables studying memory management in isolation, without having to implement a full tensor backend.

```
class CachingMemoryManager : public MemoryManagerAdapter {
  // Store state as needed
 public:
  void* alloc(bool userLock, unsigned ndim,
              dim_t* dims, unsigned elSize) override;
  // free memory
  void unlock(void* ptr, bool userLock) override;
  // ...
};
```

03 Simplicity = Performance

High-Performance Reference Implementations

Ensure that you’re bottlenecked by your new implementation, not by other framework components that preclude isolating and studying computation.

Framework Overhead Matters

Be bottlenecked by what you're building, not framework overhead.

Model	Num. Params (M)	Batch Size	1 GPU			8 GPUs		
			PT	TF	FL	PT	TF	FL
AlexNet	61	32	2.0	4.0	**1.4**	6.0	6.5	**2.1**
VGG16	138	32	14.8	**12.6**	13.2	16.3	17.9	**14.9**
ResNet-50	25	32	11.1	12.4	**10.3**	12.3	15.9	**11.9**
BERT-like	406	128	19.6	19.8	**17.5**	22.7	23.6	**19.2**
ASR Tr.	263	10	58.5	63.7	**53.6**	63.7	69.7	**57.5**
ViT	87	128	137.8	140.3	**129.3**	143.1	169.6	**141.0**

Average number of seconds to do 100 forward + backward iterations.

PT = PyTorch, TF = TensorFlow, FL = Flashlight

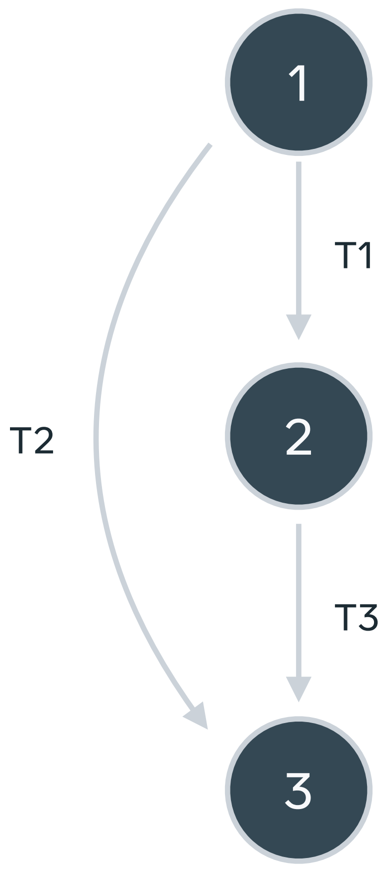
Random data is used for non-vision benchmarks to disambiguate data loading asymmetries.

Numbers gathered in NVIDIA 32GB V100 GPUs in DGX-1 systems with Intel E5-2698 CPUs with 512GB of RAM.

04 Past and present applications and directions

Case Study: Generalized memory management

With complete control over tensor memory management, current research on top of Flashlight optimizes tensor buffer placement by optimizing memory schedules of operator graphs.



Timesteps

	ts1	ts2	ts3	ts4
Ops	Run A	Run B		Run C
T1	Generate	Preserve		
T2	Generate		Fetch	Preserve
T3		Generate	Preserve	Preserve

Case Study: Swapping out element-wise addition

PyTorch

1. Search operator manifests for operators that might do addition, find stragglers, and change all call sites.

2. Hope that existing benchmarks don't use other specialized operators.

TensorFlow

1. Go through hundreds of operators that might do tensor addition, and change them.

2. Hope that existing benchmarks don't use other specialized operators.

Jax

1. Attempt to define an operator then use it via composition.

2. If your decomposition isn't usable with existing models, make deep modifications to XLA/ MLIR.

Flashlight

1. Modify the single addition operator in a tensor interface by overriding a class or changing code directly.

2. Profit!

github.com/flashlight/flashlight

Meta AI