Flashlight: Enabling Innovation in Tools for Machine Learning

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

Abstract

As the computational requirements for machine learning systems and the size and complexity of machine learning frameworks increases, essential framework innovation has become challenging. While computational needs have driven recent compiler, networking, and hardware advancements, utilization of those advancements by machine learning tools is occurring at a slower pace. This is in part due to the difficulties involved in prototyping new computational paradigms with existing frameworks. Large frameworks prioritize machine learning researchers and practitioners as end users and pay comparatively little attention to systems researchers who can push frameworks forward — we argue that both are equally important stakeholders. We introduce Flashlight, an open-source library built to spur innovation in machine learning tools and systems by prioritizing open, modular, customizable internals and state-of-the-art, research-ready models and training setups across a variety of domains. Flashlight allows systems researchers to rapidly prototype and experiment with novel ideas in machine learning computation and has low overhead, competing with and often outperforming other popular machine learning frameworks. We see Flashlight as a tool enabling research that can benefit widely used libraries downstream and bring machine learning and systems researchers closer together. Flashlight is available at this URL.

[1]Facebook AI Research, Menlo Park, CA, U.S.A. [2]Zoom AI, San Jose, CA, U.S.A. [3]Facebook AI Research, Paris, France [4]Facebook, Menlo Park, CA, U.S.A. [†]Currently at Apple, Cupertino, CA, U.S.A.. Correspondence to: Jacob Kahn <jacobkahn@fb.com>.

Proceedings of the 39^{th} International Conference on Machine Learning, Baltimore, Maryland, USA, PMLR 162, 2022.

1. Introduction

The recent rise of deep learning-based techniques has been accompanied and sustained by the wide availability of dedicated frameworks such as TensorFlow (Abadi et al., 2016) and PyTorch (Paszke et al., 2019). These frameworks have enabled the democratization of machine learning research by providing extensive collections of high level primitives to support common use cases. Lowering the barrier to entry for end users has boosted the popularity of both neural networks and the frameworks in which they are implemented. However, in order to support what are now vast ecosystems and a diverse user base, framework size and complexity have increased dramatically over time. As a result, deep, groundbreaking framework research has become onerous and time consuming, precluding rapid innovation. Given these barriers, major deep learning frameworks have become entrenched in their existing operating modes.

Innovation in this area remains as important as ever. Indeed, framework innovation accelerates machine learning (ML) and artificial intelligence (AI) research. Frameworks that are easier to use reduce the engineering burden on researchers, and frameworks that are higher-performance decrease the time required to iterate on experimental work and validate hypotheses. Even more critically, tooling plays a fundamental role in deciding which ideas succeed or fail. For example, LeCun et al. (1989) pioneered the use of convolutional neural networks (CNNs) (Fukushima & Miyake, 1982) trained using backpropagation for computer vision tasks in the late 1980s, which was subsequently applied to handwriting recognition. However, widespread success for CNNs was achieved two decades later when Krizhevsky et al. (2012a) leveraged the CUDA programming model to take advantage of graphics processing units (GPUs) to train a much deeper model (AlexNet).

While deep learning frameworks have been optimized to leverage existing hardware paradigms for common neural network architectures, they often fail to deliver similar efficiencies on designs that diverge from the mainstream. For example, Barham & Isard (2019) explain how the design of these frameworks results in poor hardware utilization for a novel type of neural network, known as a capsule net-

work (Hinton et al., 2018), that leverages new components such as squashing operations and routing by agreement. More generally, what are now unconventional approaches to modern problems in machine learning require highly-specialized additions to popular frameworks. As a result of narrowly-optimized systems, research beyond deep learning may be discounted due to purported computational infeasibility given modern frameworks' capabilities.

Furthermore, the waning of Moore's law (Theis & Wong, 2017) coupled with the ever-growing computational demands of deep learning are prompting several shifts in hardware. Massive-scale distributed computing is now required to train leading models — a process that established frameworks remain unable to handle truly automatically — that is, without some manual specification as to how to distribute work. In parallel, multiple specialized hardware products are now available to better support deep learning applications: Nvidia's TensorCores (Markidis et al., 2018), Google's TPUs (Jouppi et al., 2017), Graphcore's IPUs (Jia et al., 2019), Apple's Neural Engine[1], and others have been developed to improve total float-pointing operations (FLOPs), cost per-FLOP, or energy consumption. Additionally, numerous efforts are underway to move away from conventional von Neumann computing architectures in which memory and processing units are physically separated, either by storing data closer to compute units or by switching to in-memory computing altogether.

While tooling innovation is alive and well given these incentives for progress, working within large, well-established frameworks has become more and more challenging as framework size and scope grows. While some tools such as Halide (Adams et al., 2019; Steiner et al., 2021) and TVM (Chen et al., 2018; Zheng et al., 2020) are built from first-principles, many recent innovations have required the development of ad-hoc tools. For example, FlexFlow (Jia et al., 2018; 2020) underpins recent work aimed at improving the use of distributed computing to accelerate the training of large neural networks; PET (Wang et al., 2021) provides a framework that enables graph-level neural network optimizations; and DeepSpeed (Rasley et al., 2020) implements algorithms supporting custom distribution of computation. With ad-hoc approaches, researchers are required to start from scratch for new directions or adapt their ideas to fit into the scaffolding these frameworks provide — resulting in significant technical burdens.

To sustain framework innovation, we introduce Flashlight, an open source minimalist ML library designed to support research in machine learning frameworks, facilitate rapid iteration on ideas, reduce the engineering burden on researchers, and remove the need for new tools. Flashlight includes:

- A modular, component-based architecture that makes every aspect of the implementation fully **customizable** with simple internal APIs.
- A compact yet highly-performant **reference implementation** of each component.
- A comprehensive set of **benchmarks** representative of the state-of-the-art in machine learning on which to evaluate alternative implementations.

2. Related Work

Numerous frameworks have been implemented in recent years to support machine learning, including Lush (Bottou & LeCun, 2002), Theano (Bergstra et al., 2010), Torch (Collobert et al., 2011), Caffe (Jia et al., 2014), MXNet (Chen et al., 2015), deeplearning4j (Team, 2016), TensorFlow (Abadi et al., 2016), Flux (Innes, 2018), Jax (Bradbury et al., 2018), PyTorch (Paszke et al., 2019), Chainer (Tokui et al., 2019), and PaddlePaddle (Ma et al., 2019). These frameworks offer programming models designed around multidimensional arrays (TENSORS), modeled as first-class objects and supported by a comprehensive set of mathematical primitives (or operators) to manipulate them. To provide the computing power required by deep learning-based methods, most natively support hardware accelerators such as general-purpose GPUs or custom-designed ASICs such as TPUs.

Generally, framework implementations follow one of a few computational models:

- In the *deferred execution* model, the neural network to be trained is first encoded as a dataflow graph which can be optimized for a specific set of target hardware devices. The neural network is then executed in a distinct second phase. Since the dataflow graph represents the entire computation, both local and global optimizations can be applied, making the subsequent execution very efficient. However, only programs that can be represented as dataflow graphs can be processed with this approach, thus limiting flexibility. Frameworks such as Theano, Caffe, TensorFlow 2.0, or MXNet fall into this category.
- In the *eager* model, an interpreter (such as Python) is extended with the high level kernel-based operations needed to train a neural network. These operations are executed immediately when called, though this precludes many optimizations. By weaving neural network-related operations into a Turing complete programming language, this approach is extremely flexible. Furthermore, the imperative nature of the underlying programming language allows for fine-grained control over the execution order and memory utilization, which enables more specific user-driven optimiza-

[1] https://nr.apple.com/dE9q1p9M7t

tion. Frameworks such as Torch, TensorFlow 2.0 Eager, PyTorch, or Chainer exemplify this approach.

- In the *static* model, computation is explicitly specified ahead-of-time either via an implicit or explicit schedule. Operations are executed inside runtime sessions. Given that the entire graph of computation is fully-specified before execution, significant global optimizations can be applied here, such as explicit ahead-of-time (AOT) scheduling. Frameworks such as TensorFlow 1.0 fall into this category.
- The *hybrid* model simply combines multiple of the above computation models.

3. Principles

The aforementioned frameworks are designed and implemented to best-serve their user bases — namely, machine learning researchers and practitioners. They rely on large, internally complex codebases to provide comprehensive solutions, as is further discussed in Section 5.

In contrast, Flashlight targets an audience of researchers interested in experimenting with new designs and implementations of machine learning tools or broader computational or modeling paradigms. To foster this type of innovation, Flashlight balances simplicity and nimbleness with the need to provide enough functionality to support real use cases. Internal and external simplicity is the key design principle of Flashlight; the ability to dramatically modify software and drive it in new directions is inversely correlated with codebase size and complexity (Gill & Kemerer, 1990). More specifically:

- Flashlight is built on a shallow stack of **idiomatic, modular, and customizable** abstractions. Framework components interact through small, well-defined, stable APIs, which expose most internal aspects of its implementation. This ensures that every component of Flashlight can be modified or replaced with new custom implementations, even e.g. its memory manager and tensor implementation. To support the exploration of a wide array of alternative approaches, Flashlight interfaces are **flexible and unopiniated** by design. This is in contrast to other frameworks, which impose stricter implementation requirements based on tight design constraints for their computation models and support requirements across hardware, downstream frameworks, and other ecosystem members.
- Flashlight provides deliberately-**compact default implementations** of its APIs. This reduces out-of-the-gate engineering burden and the need for modifications, and enables fast compilation and rapid iteration when experimenting. Furthermore, to mitigate premature optimization, Flashlight deliberately **abstains from adding small efficiency improvements** if they conflict with the goals of keeping the codebase simple and APIs clean.
- Flashlight is a **research-first** framework, and is not intended for out of the box production use. To keep codebase size small, it forgoes features such as model servers for deployment and integration with cluster management tools.

Flashlight is a viable solution for **machine learning research**, shipping with a comprehensive set of benchmarks and research setups for state-of-the-art neural network architectures such as convolutional neural networks (CNNs) (Krizhevsky et al., 2012b) and Transformers (Vaswani et al., 2017), as well as task-specific models such as ViT (Dosovitskiy et al., 2020), DETR (Carion et al., 2020), or BERT (Devlin et al., 2018). The speech recognition system wav2letter (Pratap et al., 2019), is also built entirely on Flashlight.

Benchmarks built on these state-of-the-art models make Flashlight a **turn key solution for system researchers** who want to quickly evaluate their design and implementation choices without needing to build test benches from the ground-up. More importantly, Flashlight makes possible end-to-end benchmarking on real models rather than microbenchmarks or small-scale tests.

4. Design

Flashlight's design is centered around *internal* APIs for framework components which form the building blocks for domain-specific ML *packages* and *applications* — this structure is outlined in Figure 1. Flashlight is implemented as a C++ library and follows a Tensor-based programming methodology, with neural network building blocks that derive from a MODULE interface, communicate by exchanging Tensor data, and are composed functionally or imperatively to form complete neural network architectures. Tensor programming in Flashlight is fundamentally dynamic, but given that C++ is a compiled language, code describing models in Flashlight is compiled. This approach promotes type safety, foregoes the runtime overheads associated with interpreters, and, unlike eager-based approaches, enables global optimizations where possible.

4.1. Open Foundational Interfaces

Flashlight is built on top of three open *foundational* APIs, each addressing design and implementation challenges faced by machine and deep learning tools: a *Tensor* interface, a *memory management* subsystem, and a *distributed* computing interface. These APIs are backed by reference implementations that enable Flashlight to efficiently target CPUs, GPUs, and other accelerators. These include code generation and dedicated kernels for Intel, AMD, OpenCL,

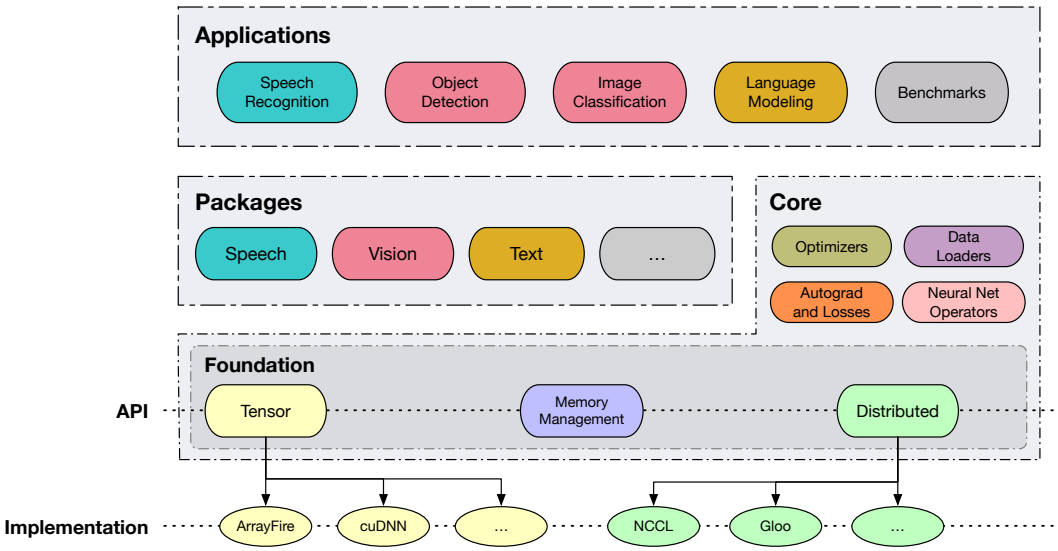

Figure 1. Components of the Flashlight library.

and CUDA devices, and leverage libraries such as cuDNN (Chetlur et al., 2014), MKL (Intel, 2020a), oneDNN (Intel, 2020b), ArrayFire (Yalamanchili et al., 2015), and MiOpen (Khan et al., 2019).

4.1.1. Tensor Interface

Modern deep learning frameworks feature tensor library internals which sit under deep layers of abstractions, requiring numerous framework modifications in order to iterate on tensor stack design. Flashlight's Tensor abstraction is defined in terms of existing tensor libraries via a simple, extensible interface and a high-level API that mirrors *numpy* (Harris et al., 2020) rather than using specific, opinionated intermediate representations (IRs) or large operator sets.

Flashlight Tensor backend implementations need not follow any particular computation mode as outlined in Section 2 and shown in Figure 2. Tensor values need only be materialized upon user request — typically when extracting the output values of a model or inspecting intermediary state. This provides a flexibility unique amongst deep learning frameworks to either defer or eagerly-compute intermediate values — or to experiment with new computation paradigms altogether.

Implementing a Tensor backend in Flashlight involves fulfilling a small set of implementation requirements. Users have full control of their implementations after subclassing two interfaces:

- the TensorAdapter interface (Listing 1) which allows a backend to attach custom stateful information and metadata to each tensor. This includes shape, type, and memory information — which may be implementation-dependent.
- the TensorBackend interface (Listing 2) which allows backends to store global state as needed (e.g device compute streams, dataflow graphs) and implement a small set of primitive tensor operations including unary and binary operations (e.g. arithmetic ops), reductions, matrix multiplication, and convolution.

Rather than require implementations of large, highly-specialized operator sets or interoperability with complex dispatch mechanisms or intermediate representations (IRs) as do other frameworks, Flashlight operators outside of the small TensorBackend API are derived by composition. For example: the ReLU activation is implemented by leveraging the max operator. Flashlight's reference Tensor implementation uses a *hybrid* approach, offloading computation to highly-optimized vendor libraries when advantageous and relying on deferred, on-the-fly code generation via ArrayFire for all other operations so as to increase kernel arithmetic intensity.

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

Listing 1. The *TensorAdapter* interface for implementing operations on tensor metadata and storing tensor state for individual tensor instances.

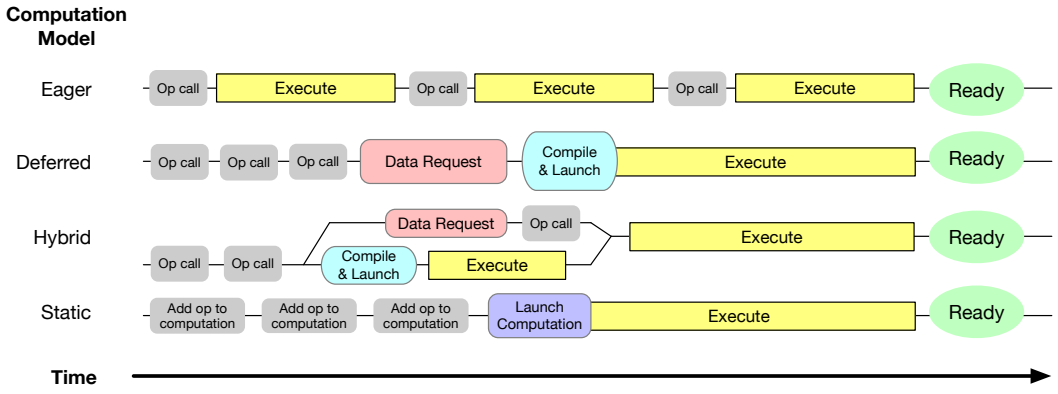

Figure 2. Flashlight's Tensor API supports backend implementations with any of the above computation modes (or entirely new modes that may result from further research).

```
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;
  // ...
};
```

Listing 2. The *TensorBackend* interface for implementing operations on tensors and storing global backend state.

4.1.2. Memory Management

Robust memory management is an important research area as model size increases. While individual Tensor backends in Flashlight can perform their own memory management as defined by implementers, Flashlight's default Tensor backend also provides a generic API for defining custom memory management schemes. By default, memory is only allocated when needed for just-in-time compilation. A sample of this API is shown in Listing 3. To support the lazy computation model as well as just in time code generation, memory allocations only occur when tensors need to be materialized per the compute graph. Buffers are used asynchronously after they are requested depending on the timing of kernel launches but are not freed until computation is complete.

4.1.3. Distributed Training

Flashlight provides a low-level API for distributed training primitives with performant default implementations in GPU and CPU settings using NCCL (2019) and Gloo (2019), respectively. Users can add new backends or custom methods of performing distributed computation, and can use primitives in other framework components as needed. Further details and samples can be found in Section A.4.1.

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

Listing 3. An implementation of a memory manager using the memory management API.

4.2. The Flashlight Core

The Tensor API, together with memory management and distributed computation abstractions provide a foundation on which to build other core machine learning algorithms and applications. These other core components are outlined below. Section A.4 provides code samples and linking documentation for the below components.

Neural Network Primitives To facilitate the implementation of neural networks, Flashlight ships with common neural building-blocks encompassing activation functions, normalization layers, regularizers, losses, and more. These derive from the Module abstraction, as discussed above, which provides a method of chaining and nesting operations. Section A.4.2 contains more details and sample implementations.

Automatic Differentiation Automatic differentiation (autograd) is implemented via a simple Variable abstraction. A Variable takes a Tensor argument when created, and its

underlying Tensor (or gradient Tensor) can be accessed at any time. Variables feature operators which call underlying Tensor operations and record those operations to a dynamic tape in a design similar to Paszke et al. (2017) while being lightweight enough to allow implementations of other autograd paradigms. In keeping with Flashlight's modularity, TENSOR and VARIABLE are separated to avoid performance and implementation overhead in non-gradient-based ML algorithms. Further, this gives implementers of a Flashlight Tensor backend an efficient, fully-featured autograd system with no additional effort. We provide an example of an autograd primitive implementation in Listing 4.

Optimizers Flashlight provides implementations of most common first-order stochastic optimization algorithms, as included in other frameworks. These are defined in terms of Variable and Tensor operations, allowing for open-ended experimentation (e.g. with distributed computation, in-place operations, etc).

Data Loaders Flashlight provides a simple DATASET class which abstracts away the notion of a *sample* in ML algorithms. A sample is viewed here as a TENSOR or vector of TENSORS. Datasets are trivially composable to create pipelines to transform, resample, or parallelize (via native C++ threads) the construction of such samples. While Flashlight core dataset abstractions are agnostic from the end-user task, data-specific datasets are provided in higher-level *packages*, to efficiently load from disk structured data (e.g. images, audio or text).

4.3. Packages and Applications

Flashlight contains additional domain-specific abstractions leveraging both core components as well as stand-alone implementations. These abstractions allow end-users or ML researchers to quickly get started on various ML applications. The *package* module provides building blocks for common ML tasks, domain-specific algorithms, and helpers. The *application* module leverages these building blocks to provide complete, ready to use solutions (e.g. models, training loops, evaluation pipelines). When not original to Flashlight, implementations reproduce the task performance those they reference. We leverage several of these applications to evaluate Flashlight's performance in Section 5.

Speech. Flashlight provides an implementation of classical featurization (spectogram, log-mel filterbanks, etc.) that can run on-the-fly with minimal overhead. It also provides a collection of data-augmentation techniques (including additive noise and reverberation), as well as implementations of speech-specific sequential criteria and model architectures. Flashlight contains a fast beam-search decoder (which can interface any language model) and beam rescorers (Collobert et al., 2016; Pratap et al., 2019). Research performed with the speech *application* have reached and are competitive with state-of-the-art results (Synnaeve et al., 2019; Likhomanenko et al., 2021).

Vision. Flashlight offers built-in data loaders for standard computer vision benchmarks (such as ImageNet (Deng et al., 2009) and COCO (Lin et al., 2014)) along with large set of efficient data-augmentations and transformations. It includes mainstream image classification models: convolutional (e.g. ResNet (He et al., 2016)) and transformer-based architectures (e.g. ViT (Dosovitskiy et al., 2020)), as well as a modern, transformer-based object detection model (DETR (Carion et al., 2020)) and helpers (e.g. Hungarian matching and object detection evaluation).

Text. Flashlight ships with support for text dataset manipulation and tokenization along with language modeling training pipelines for a variety of neural language models, including transformer (Vaswani et al., 2017) and CNN-based (Dauphin et al., 2017). Both autoregressive and masked, e.g. BERT, language modeling tasks are supported. These language models can be combined with other domain-specific packages such as speech.

5. Evaluation

In the sections that follow, we compare Flashlight to two widely-used deep learning frameworks — PyTorch and Tensorflow — with metrics relevant to framework research velocity. We also evaluate framework performance to demonstrate the potential of our approach and the quality of the default implementations of all our components. We outline the steps needed to reproduce all our results in the Appendix.

5.1. Code Complexity

Flashlight is built to minimize complexity and operating surface. As frameworks grow and are combined with other frameworks or take on new platform-specific requirements, internal modifiability decreases. Table 1 compares frameworks across well-established measures of complexity and portability — binary size, lines of code, and number of operators and operator implementations. Flashlight's small surface facilitates easily exploring new designs and prototyping on new hardware — having few sources of truth simplifies the process of replacing core components and ensures end-to-end tests don't opaquely fall back to existing implementations.

5.1.1. COMPILATION TIME

When modifying or adding significant new research code to framework internals, recompilation can be costly. Large frameworks depend on code generation for broad platform

```
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);
}
```

Listing 4. Defining a cosine autograd operator in Flashlight using TENSOR operations and VARIABLE.

Table 1. Complexity of various frameworks based on high-level metrics. We provide additional analysis disambiguating tensor library components of each framework in Section A.2.2.

METRIC	PYTORCH	TENSORFLOW	(OURS) FLASHLIGHT
BINARY SIZE (MB)	527	768	10
LINES OF CODE	1,798,292	1,306,159	27,173
NUMBER OF OPERATORS	2,166	1,423	60
APPROX NUM. OPS. THAT PERFORM:			
ADD	55	20	1
CONV	85	30	2
SUM	25	10	1

support[2], increasing compilation time. Further, expensive incremental rebuilds can slow iteration speed.

Flashlight is sufficiently-lightweight and modular so as to enable from-source build times that are orders of magnitude faster than other frameworks, as shown in Table 2. Times were measured for both from-scratch and incremental builds with Intel Xeon Gold 6138 CPUs with 80 cores and 750 GB of memory. To estimate incremental build performance, we randomly sample 100 source files without replacement, make trivial modifications that force recompilation, and time the resulting rebuild. While we constrain the set of files that can be modified for incremental compilation to those that are part of core systems (tensor library, autograd, modules, distributed computation), framework subsystems differ and constraints were specialised per-framework. The standard deviation for all incremental compilation benchmarks was under 5% of the overall compilation time. The Appendix contains information and resources to reproduce these results.

5.1.2. PERFORMANCE

When improving framework components or modifying internals, framework overhead makes it difficult to disambiguate performance changes due to in-flight modifications from existing bottlenecks or overhead due to other framework components as discussed in Section 3. Table 3 compares the performance of Flashlight 0.3.1, PyTorch 1.8, TensorFlow 2.4 on six common large-scale deep neural networks. For each configuration, we benchmark 100 iterations of data loading[3], preprocessing, and forward/backward passes, with data-parallel gradient synchronization in distributed settings. Benchmarks are performed on Intel E5-2698 CPUs with 512GB of RAM, and NVIDIA V100-32GB GPUs in a DGX-1 server. Inter-GPU interconnects in the 8 GPUs (1 node) setting are Nvidia NVLink-based. All models were warmed up with 100 iterations of forward and backward passes. For consistency and reproducibility, no third-party libraries are

Table 2. Compile times in CPU minutes across frameworks, with incremental compilation times averaged over 100 samples. Flashlight compilation time includes compiling with the default ArrayFire CUDA backend.

PLATFORM	FROM SCRATCH	INCREMENTAL
PYTORCH	754	132
TENSORFLOW	2061	371
(OURS) FLASHLIGHT	34	0.6

[2]Examples — PyTorch: https://git.io/Jzel9, TensorFlow: https://git.io/JzeRw

[3]To ensure fairness, due to Flashlight's significantly better dataloading performance as to compared to other frameworks, BERT-like models use random data in-memory; ViT models exclude data augmentation.

Table 3. Performance on common state-of-the-art models across frameworks. Values are the number of seconds needed to perform 100 iterations of the forward and backwards passes, with data loading (unless indicated). Number of parameters are in millions. Framework labels: PT = PyTorch, TF = TensorFlow, and FL = Flashlight.

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**

used to enhance performance beyond optimization tools already contained in frameworks (e.g. `@tf.function` in TensorFlow). Flashlight is benchmarked as is with no optimizations. While orthogonal to the paper, Flashlight's default backend has empirically outperformed other frameworks due to the quality of the ArrayFire JIT, dataloading performance, and low framework overhead.

Flashlight is competitive and can exceed the performance of other frameworks, especially on architectures which are of lower arithmetic intensity and spend less compute time in vendor-optimized libraries, such as AlexNet. Given strong performance with simple reference implementations that have undergone far less optimization than have large frameworks, we see exciting potential for improvement with future research done in Flashlight.

5.2. Case Studies

Ongoing research efforts enabled by Flashlight include work in code generation, compilers and IRs, memory management, and distributed computing. Below, we give examples of recent research made possible with Flashlight.

5.2.1. Optimizations on Large, Specialized Autograd Graphs

The ability to change the lightweight implementation of Flashlight's tensor and automatic differentiation (autograd) components via extensible APIs facilitated research in building a fully differentiable beam search decoder (Collobert et al., 2019), which required operating on unconventional computation graphs not supported by other frameworks' autograd systems. Other frameworks were unable to handle these autograd graphs for several reasons:

- Graphs contained millions of nodes/operations that created significant memory pressure;
- There existed only small operator overhead per autograd graph node (many addition and log operations);
- Graph operations had few opportunities for vectorization;
- Only sparse components of the graph were required.

Authors modified Flashlight's autograd to support:

- On-the-fly graph pruning to take advantage of sparsity and reduce memory footprint;
- Dynamic, pre-fused gradient computation for common sequences of gradient computation operations;
- Custom autograd node lifetime for avoiding reference-counting overhead for graph mutations.

To our knowledge, these capabilities only exist in autograd implementations like Flashlight's that feature public APIs built for customization. Indeed, these ideas may be broadly applicable to other modeling settings, but implementing and researching such logic in other frameworks' autograd systems is not realistically possible.

5.2.2. Fragmentation Reduction in Accelerator Memory Management

While those researching memory management techniques in machine learning computation can make ad-hoc modifications to other frameworks' memory managers, build time, internal complexity, and lack of a unified interface makes this challenging. Using Flashlight's open memory management interface, researchers studied and developed new techniques for fragmentation reduction for memory management on the GPU. This was made possible by the ease of extending a lightweight memory manager interface in Flashlight along with clear implementation requirements and tests that made rapid prototyping possible.

Various caching memory allocators are used across deep learning frameworks to reduce the cost of native device memory allocations and reuse already-allocated memory. These caching memory managers are subject to fragmentation as they bucket allocations based on rounded size. Reducing this fragmentation allows for training larger models and significantly improves performance as it removes sometimes-expensive allocation overhead.

Given these challenges, researchers aiming to reduce external fragmentation implemented a custom caching memory manager in Flashlight to study memory behavior and built highly-specialized telemetry that tied individual tensor operations to specific allocations. Researchers detailed a myriad of prototypes to reduce fragmentation and described rapid rebuilds and custom telemetry as critical to their experimentation. Ultimately, a memory manager that restricted splitting large cache blocks (or blocks beyond a certain tunable size) showed promise and reduced internal fragmentation for most models by over 20%.

These techniques were tested on a variety of models in Flashlight before researchers shared their findings with maintainers of other large deep learning frameworks, where knowledge were contributed upstream[4].

5.2.3. Automatic Optimization of Memory and Distributed Compute

Flashlight enables developing generalized approaches to both memory management and distributed computation. Approaches such as ZeRO (Rajbhandari et al., 2020) optimizer state sharding or GPUSwap (Kehne et al., 2015) memory oversubscription, for example, are specialized to particular components of training pipelines. Generalizations of these approaches which involve shuffling around buffers to a variety of devices or performing pieces of computation on certain hardware should discover new techniques for efficient large-scale training.

Given that Flashlight offers complete control over memory and distributed computation models, tensors can follow any preordained allocation schedule or rules. They can be sharded or computations dispatched to arbitrary devices. Operating under such general assumptions in other frameworks that are without internal APIs for memory management or feature strict requirements around computation model makes specifying such a general system difficult.

5.2.4. Researching Efficient Primitives in Tensor Computation

New ideas in basic tensor computation are notoriously difficult to test at scale in an end-to-end fashion. For demonstrative purposes, consider a research artifact which includes new, more-efficient ways to perform element-wise arithmetic operations on tensors (e.g. addition, multiplication). While such operations can be implemented as one-off operators in most machine learning frameworks, swapping out the source of truth for the tensor `add` function, for instance, such that existing operators and framework components (including baselines, benchmarks, etc) uses the custom implementation may be difficult or untenable. Across various frameworks, this process might involve:

- **PyTorch** — per Figure 1, 55 PyTorch operators explicitly perform addition as part of their computation (and perhaps far more implicitly); to benchmark end-to-end with existing PyTorch code, all of these callsites must be changed to call the new implementation.
- **TensorFlow** — similarly to PyTorch, many operators perform addition, and while defining custom operators is straightforwards, changing a wealth of existing library code is error-prone and labor-intensive.
- **Jax** — defining a custom operator in Jax and using it via composition with other operators is relatively simple. However, swapping out default behavior of addition for most all Jax operators would require changes to XLA via MLIR, a large, complex codebase.
- **Flashlight** — given the default backend, an implementer can simply subclass or swap out the existing implementation of the `add` function with their custom logic. All `add` operations in Flashlight dispatch to that operator, so existing baselines and operations will run with the new implementation without any additional code changes.

6. Conclusion

We presented Flashlight, a modular machine learning library supporting modern, state-of-the-art baselines that features orders of magnitude less code and binary size as compared to frameworks such as PyTorch and TensorFlow. These large frameworks come fully-featured; Flashlight aims to complement them in providing a first-of-its-kind tool with which to do machine learning *framework* and *computational* research. To this end, Flashlight features a lightweight, modular design, as well as full implementations of mainstream models across a variety of domains, making it easy for researchers to implement new internal tensor, memory management, or distributed computation backends. Flashlight includes a variety of reference implementations for its APIs that compete with and often outperform popular machine learning frameworks, thus demonstrating the viability of our approach.

Acknowledgements

We would like to thank Chaitanya Talnikar and Mohammed Motamedi for valuable contributions to the Flashlight codebase, Shubho Sengupta for valuable support and input, as well as Horace He and Edward Yang for their work on operator size and complexity in deep learning frameworks.

[4] Caching allocator improvements in PyTorch.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pp. 265–283, 2016.

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., Steiner, B., Johnson, S., Fatahalian, K., Durand, F., and Ragan-Kelley, J. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.*, 38(4), July 2019. ISSN 0730-0301. doi: 10.1145/3306346.3322967. URL https://doi.org/10.1145/3306346.3322967.

Barham, P. and Isard, M. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, pp. 177–183, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367271. doi: 10.1145/3317550.3321441. URL https://doi.org/10.1145/3317550.3321441.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4, pp. 1–7. Austin, TX, 2010.

Bottou, L. and LeCun, Y. Lush. 2002. URL http://lush.sourceforge.net.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., and Zagoruyko, S. End-to-end object detection with transformers. In *European Conference on Computer Vision*, pp. 213–229. Springer, 2020.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, T., Zheng, L., Yan, E., Jiang, Z., Moreau, T., Ceze, L., Guestrin, C., and Krishnamurthy, A. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, pp. 3393–3404, Red Hook, NY, USA, 2018. Curran Associates Inc.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

Collobert, R., Kavukcuoglu, K., and Farabet, C. Torch7: A matlab-like environment for machine learning. 01 2011.

Collobert, R., Puhrsch, C., and Synnaeve, G. Wav2letter: an end-to-end convnet-based speech recognition system. *arXiv preprint arXiv:1609.03193*, 2016.

Collobert, R., Hannun, A., and Synnaeve, G. A fully differentiable beam search decoder. In *International Conference on Machine Learning*, pp. 1341–1350. PMLR, 2019.

Dauphin, Y. N., Fan, A., Auli, M., and Grangier, D. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, pp. 933–941. JMLR.org, 2017.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Fukushima, K. and Miyake, S. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern Recognition*, 15(6):455–469, 1982. ISSN 0031-3203. doi: https://doi.org/10.1016/0031-3203(82)90024-3. URL https://www.sciencedirect.com/science/article/pii/0031320382900243.

Gill, G. and Kemerer, C. Productivity impacts of software complexity and developer experience. *MIT Sloan*, 1990.

Gloo. Gloo: a collective communications library. https://github.com/facebookincubator/gloo, 2019.

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F.,

Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL https://doi.org/10.1038/s41586-020-2649-2.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hinton, G., Sabour, S., and Frosst, N. Matrix capsules with em routing. 2018. URL https://openreview.net/pdf?id=HJWLfGWRb.

Innes, M. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 3(25):602, 2018.

Intel. Mkl developer reference. https://software.intel.com/content/www/us/en/develop/documentation/mkl-developer-reference-c/top.html, 2020a.

Intel. Onednn. https://github.com/oneapi-src/oneDNN, 2020b.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

Jia, Z., Zaharia, M., and Aiken, A. Beyond data and model parallelism for deep neural networks. *CoRR*, abs/1807.05358, 2018. URL http://arxiv.org/abs/1807.05358.

Jia, Z., Tillman, B., Maggioni, M., and Scarpazza, D. Dissecting the graphcore ipu architecture via microbenchmarking. *ArXiv*, abs/1912.03413, 2019.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 187–198, 2020. URL https://proceedings.mlsys.org/paper/2020/file/fe9fc289c3ff0af142b6d3bead98a923-Paper.pdf.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017. ISSN 0163-5964. doi: 10.1145/3140659.3080246. URL https://doi.org/10.1145/3140659.3080246.

Kehne, J., Metter, J., and Bellosa, F. Gpuswap: Enabling oversubscription of gpu memory through transparent swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 65–77, 2015.

Khan, J., Fultz, P., Tamazov, A., Lowell, D., Liu, C., Melesse, M., Nandhimandalam, M., Nasyrov, K., Perminov, I., Shah, T., et al. Miopen: An open source library for deep learning primitives. *arXiv preprint arXiv:1910.00078*, 2019.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pp. 1097–1105, Red Hook, NY, USA, 2012a. Curran Associates Inc.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012b.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, 1989. doi: 10.1162/neco.1989.1.4.541.

Li, S., Zhao, Y., Varma, R., Salpekar, O., Noordhuis, P., Li, T., Paszke, A., Smith, J., Vaughan, B., Damania, P., et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

Likhomanenko, T., Xu, Q., Pratap, V., Tomasello, P., Kahn, J., Avidov, G., Collobert, R., and Synnaeve, G. Rethinking Evaluation in ASR: Are Our Models Robust

Enough? In *Proc. Interspeech 2021*, pp. 311–315, 2021. doi: 10.21437/Interspeech.2021-1758.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft coco: Common objects in context. In *European conference on computer vision*, pp. 740–755. Springer, 2014.

Ma, Y., Yu, D., Wu, T., and Wang, H. Paddlepaddle: An open-source deep learning platform from industrial practice. *Frontiers of Data and Computing*, 1(1):105–115, 2019.

Markidis, S., Chien, S. W., Laure, E., Peng, I., and Vetter, J. Nvidia tensor core programmability, performance & precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 522–531, 2018.

NCCL. Nvidia collective communications library (nccl). https://github.com/NVIDIA/nccl, 2019.

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in pytorch. 2017.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

Pratap, V., Hannun, A., Xu, Q., Cai, J., Kahn, J., Synnaeve, G., Liptchinsky, V., and Collobert, R. Wav2letter++: A fast open-source speech recognition system. In *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6460–6464, 2019. doi: 10.1109/ICASSP.2019.8683535.

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3505–3506, 2020.

Steiner, B., Cummins, C., He, H., and Leather, H. Value learning for throughput optimization of deep learning workloads. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 323–334, 2021. URL https://proceedings.mlsys.org/paper/2021/file/73278a4a86960eeb576a8fd4c9ec6997-Paper.pdf.

Synnaeve, G., Xu, Q., Kahn, J., Likhomanenko, T., Grave, E., Pratap, V., Sriram, A., Liptchinsky, V., and Collobert, R. End-to-end asr: from supervised to semi-supervised learning with modern architectures. *arXiv preprint arXiv:1911.08460*, 2019.

Team, E. D. D. Deeplearning4j: Open-source distributed deep learning for the jvm. 2016. URL https://github.com/eclipse/deeplearning4j.

Theis, T. N. and Wong, H.-S. P. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017. doi: 10.1109/MCSE.2017.29.

Tokui, S., Okuta, R., Akiba, T., Niitani, Y., Ogawa, T., Saito, S., Suzuki, S., Uenishi, K., Vogel, B., and Yamazaki Vincent, H. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2002–2011, 2019.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 6000–6010, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Wang, H., Zhai, J., Gao, M., Ma, Z., Tang, S., Zheng, L., Li, Y., Rong, K., Chen, Y., and Jia, Z. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pp. 37–54. USENIX Association, July 2021. ISBN 978-1-939133-22-9. URL https://www.usenix.org/conference/osdi21/presentation/wang.

Yalamanchili, P., Arshad, U., Mohammed, Z., Garigipati, P., Entschev, P., Kloppenborg, B., Malcolm, J., and Melonakos, J. ArrayFire - A high performance software library for parallel computing with an easy-to-use API, 2015. URL https://github.com/arrayfire/arrayfire.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879. USENIX Association, November 2020. ISBN 978-1-939133-19-9.

URL https://www.usenix.org/conference/osdi20/presentation/zheng.

A. Appendix

A.1. Reproducibility

All tools and code used in our evaluation are included as supplementary material — this includes scripts to reproduce benchmarks and other quantitative codebase metrics. Flashlight code can be found on Github at `https://github.com/flashlight/flashlight`.

As discussed in Section 5, Flashlight v0.3.1 is used to reproduce results using ArrayFire 3.8 as the underlying tensor backend. No other specialized configuration is used for either Flashlight or PyTorch or TensorFlow.

A.1.1. From-Source Compilation

Flashlight is built in CMake release mode, which is also the default for both PyTorch and TensorFlow builds. Build settings are kept default for all frameworks; Flashlight uses Ninja[5] with CMake in accordance with PyTorch's build, while Tensorflow employs Bazel.

Exact build step reproduction can be found in the aforementioned repository.

A.1.2. Incremental Compilation

Incremental compilation benchmarks are performed using similar build setups per Section A.1.1. To test incremental rebuilds, source files are randomly selected without replacement from a distribution constructing by weighting each file by the number of lines in the file, and using that number to determine the probability of selecting it for modification.

Scripts to perform and time this incremental compilation can be found in the aforementioned public repository.

A.2. Code Complexity

A.2.1. Operator Counting

To count the number of operators for each framework, we utilize operator schemas for PyTorch and Tensorflow (which generate code from those schemas, accordingly) written in YAML and Protobuf, respectively. For Flashlight, we count the number of functions in the Flashlight Tensor interface and autograd interfaces, as these form the full implementation requirements for a full tensor backend that functions on all platforms. The scripts released on Github detail the files and filtering techniques used to reproduce the number of results.

To count the number of operators for each implementation, we use the above operator lists, then count the number of operators that perform the specified function, even if those operators perform other functions. For example: an operator called addmm, which performs an addition operation followed by a matrix-matrix multiplication, performs an addition operation, and would thus be counted when tallying the number of add operators.

A.2.2. Codebase Complexity Disambiguating Tensor Library Size

Given that Flashlight can be compiled with arbitrary tensor libraries that are adapted to the Flashlight Tensor framework, we provide brief further evaluation here exhibiting the contribution of tensor libraries to overall framework size. In an effort to remote auxiliary and irrelevant framework components, both this analysis and the analysis from Section 5 counts only C, C++, Python, YAML, CUDA, and CMake files from a relevant subset of core framework components when assessing overall lines of code.

Table 4 details the relative sizes and number of lines per framework with and without tensor libraries. Note that PyTorch cannot be compiled without its tensor components, so one cannot directly assess binary size.

A.3. Performance

The aforementioned public repository provides scripts required to fully-reproduce all performance measures.

[5] `https://ninja-build.org/`

Table 4. Complexity of various framework components including and excluding building with tensor library components.

METRIC	PYTORCH	TENSORFLOW	(OURS) FLASHLIGHT
BINARY SIZE (MB) (NO TENSOR LIB)	N/A	423	7
LINES OF CODE (NO TENSOR LIB)	924K	602K	27K
BINARY SIZE (MB) (WITH TENSOR LIB)	527	768	10
LINES OF CODE (WITH TENSOR LIB)	1798K	1306K	17K
NUMBER OF OPERATORS	2,166	1,423	60

A.4. Design Details and Code Samples

In the following sections, we show brief code samples expounding on those in Section 4.

A.4.1. DISTRIBUTED TRAINING

Flashlight's distributed training API is of a similar flavor to its Tensor library, in that it invites custom implementations of distributed computation primitives with an explicit API. The API is unopinionated and supports both synchronous and asynchronous communication, unlike other frameworks. Alsi included is an internal API for implementing specialized rendezvous schemes for new distributed computation environments. Listing 5 shows part of this API, which is structured in a similar manner to (Li et al., 2020). Implementers need simply derive from this interface, and their distributed computation primitives will interoperate with other Flashlight computation streams.

```
class DistributedInterface {
  // store arbitrary state here
 public:
  // metadata about the distributed computation environment
  virtual int getWorldRank() = 0;
  virtual int getWorldSize() = 0;
  // ...

  // distributed computation primitives
  virtual void allReduce(Tensor& var, double scale = 1.0, bool async = false) = 0;
  virtual void allReduceMultiple(
    std::vector<Tensor> vars,
    double scale = 1.0,
    bool async = false) = 0;
  // ...

  // synchronization primitives
  virtual void syncDistributed() = 0;
  virtual void barrier() = 0;
  // ...
};
```

Listing 5. Part of Flashlight's distributed computation API.

A.4.2. MODULES

Flashlight's MODULE abstraction is similar to that of frameworks such as Torch and PyTorch. It can recursively store other modules and interoperate with more sophisticated abstractions including CONTAINER, which wraps multiple modules, SEQUENTIAL, which stores sequences of modules and forwards data through them sequentially, and user-defined abstractions. Listing 8 in Section A.4.3 shows an example of Sequential usage.

Listing 6 shows a small Dropout module implementation that calls into the dropout autograd primitive, stores and serializes a small amount of state, and defines a simple forward function on a Variable.

```
class Dropout : public Module {
 private:
  double ratio_;
  FL_SAVE_LOAD_WITH_BASE(Module, ratio_) // serialization
```

```
 public:
  Dropout(double drop_ratio = 0.5);
  Variable forward(const Variable& input) override {
    if (train_) {
      return dropout(input, ratio_); // autograd primitive
    } else {
      return input;
    }
  }
  // ...
};
```

Listing 6. A Dropout layer implemented as a Flashlight module.

The `FL_SAVE_LOAD_WITH_BASE` macro defines serialization of the Dropout class as a module, including any fields to be serialized (in this case, only the dropout ratio).

A.4.3. An End-to-End Example: MNIST

Below, we detail a simple end-to-end training setup following Flashlight's open-source documentation[6].

First, data is loaded using the BatchDataset abstraction in Listing 7:

```
const int kTrainSize = 60000;
const int kValSize = 5000;

auto& [train_x, train_y] = load_dataset(data_dir);

// Hold out a dev set
auto val_x = train_x(span, span, range(0, kValSize));
train_x = train_x(span, span, range(kValSize, kTrainSize));
auto val_y = train_y(range(0, kValSize));
train_y = train_y(range(kValSize, kTrainSize));

// Make the training batch dataset
BatchDataset trainset(
    std::make_shared<TensorDataset>(std::vector<Tensor>{train_x, train_y}),
    batch_size);

// Make the validation batch dataset
BatchDataset valset(
    std::make_shared<TensorDataset>(std::vector<Tensor>{val_x, val_y}),
    batch_size);
```

Listing 7. Loading MNIST data into a train and evaluation set.

A full description of the `load_dataset` function can be found in the MNIST training example on Github[7].

We can construct the model using a simple Sequential in Listing 8:

```
const int kImageDim = 28;
auto pad = PaddingMode::SAME;

Sequential model;
model.add(View({kImageDim, kImageDim, 1, -1})); // WHCN (col major)
model.add(Conv2D(
    1 /* input channels */,
    32 /* output channels */,
    5 /* kernel width */,
    5 /* kernel height */,
    1 /* stride x */,
```

[6] https://fl.readthedocs.io/en/latest/mnist.html
[7] https://git.io/JVI6O

```
    1 /* stride y */,
    pad /* padding mode */,
    pad /* padding mode */));
model.add(ReLU());
model.add(Pool2D(
    2 /* kernel width */,
    2 /* kernel height */,
    2 /* stride x */,
    2 /* stride y */));

model.add(Conv2D(32, 64, 5, 5, 1, 1, pad, pad));
model.add(ReLU());
model.add(Pool2D(2, 2, 2, 2));
model.add(View({7 * 7 * 64, -1}));
model.add(Linear(7 * 7 * 64, 1024));
model.add(ReLU());
model.add(Dropout(0.5));
model.add(Linear(1024, 10));
model.add(LogSoftmax());
```

Listing 8. Constructing a CNN for MNIST training.

In Listing 9, we create a simple custom training loop. This uses optimizer, loss function, and meter abstractions as provided by default by Flashlight. We perform the forward and backward pass, step the optimizer to update parameters, and zero out gradients before moving to the next batch. We evaluate the model using the function defined in Listing 10, pulls out the max prediction and comparing it against the ground truth, updating the loss meter as we go, then returning the final loss values.

```
// Make the optimizer
SGDOptimizer opt(model.params(), learning_rate);

// The main training loop
for (int e = 0; e < epochs; e++) {
  AverageValueMeter train_loss_meter;

  // Get an iterator over the data
  for (auto& example : dataset) {
    auto inputs = noGrad(example[INPUT_IDX]);
    auto output = model(inputs);

    auto target = noGrad(example[TARGET_IDX]);
    // Compute and record the loss.
    auto loss = categoricalCrossEntropy(output, target);
    train_loss_meter.add(loss.tensor().scalar<float>());
    // Backprop, update the weights and then zero the gradients.
    loss.backward();
    opt.step();
    opt.zeroGrad();
  }

  double train_loss = train_loss_meter.value()(0).scalar<double>();

  // Evaluate on the dev set.
  auto [val_loss, val_error] = eval_loop(model, valset);

  std::cout << "Epoch " << e << std::setprecision(3)
            << ": Avg Train Loss: " << train_loss
            << " Validation Loss: " << val_loss
            << " Validation Error (%): " << val_error << std::endl;
}
```

Listing 9. A simple training loop.

```
std::pair<double, double> eval_loop(Sequential& model, BatchDataset& dataset) {
  AverageValueMeter loss_meter;
```

```
  FrameErrorMeter error_meter;

  // Place the model in eval mode.
  model.eval();
  for (auto& example : dataset) {
    auto inputs = noGrad(example[INPUT_IDX]);
    auto output = model(inputs);
    // Get the predictions in max_ids
    Tensor max_vals, max_ids;
    max(max_vals, max_ids, output.tensor(), 0);

    auto target = noGrad(example[TARGET_IDX]);
    // Compute and record the prediction error.
    error_meter.add(reorder(max_ids, 1, 0), target.tensor());
    // Compute and record the loss.
    auto loss = categoricalCrossEntropy(output, target);
    loss_meter.add(loss.tensor().scalar<float>());
  }
  // Place the model back into train mode.
  model.train();

  double error = error_meter.value().scalar<double>();
  double loss = loss_meter.value().scalar<double>();
  return std::make_pair(loss, error);
}
```

Listing 10. Evaluating a training model on MNIST.

Finally, in Listing 11, we evaluate the trained model on the test set by creating a test dataset and using the previously-defined evaluation function.

```
std::pair<double, double> eval_loop(Sequential& model, BatchDataset& dataset) {
  AverageValueMeter loss_meter;
  FrameErrorMeter error_meter;

  // Place the model in eval mode.
  model.eval();
  for (auto& example : dataset) {
    auto inputs = noGrad(example[INPUT_IDX]);
    auto output = model(inputs);
    // Get the predictions in max_ids
    Tensor max_vals, max_ids;
    max(max_vals, max_ids, output.tensor(), 0);

    auto target = noGrad(example[TARGET_IDX]);
    // Compute and record the prediction error.
    error_meter.add(reorder(max_ids, 1, 0), target.tensor());
    // Compute and record the loss.
    auto loss = categoricalCrossEntropy(output, target);
    loss_meter.add(loss.tensor().scalar<float>());
  }
  // Place the model back into train mode.
  model.train();

  double error = error_meter.value().scalar<double>();
  double loss = loss_meter.value().scalar<double>();
  return std::make_pair(loss, error);
}
```

Listing 11. Evaluating a training model on MNIST.