XLA Overview

Note: XLA is experimental and considered alpha. Most use cases will not see improvements in performance (speed or decreased memory usage). We have released XLA early so the Open Source Community can contribute to its development, as well as create a path for integration with hardware accelerators.

XLA (Accelerated Linear Algebra) is a domain-specific compiler for linear algebra that optimizes TensorFlow computations. The results are improvements in speed, memory usage, and portability on server and mobile platforms. Initially, most users will not see large benefits from XLA, but are welcome to experiment by using XLA via just-in-time (JIT) compilation (https://www.tensorflow.org/performance/xla/jit) or ahead-of-time (AOT) compilation (https://www.tensorflow.org/performance/xla/tfcompile). Developers targeting new hardware accelerators are especially encouraged to try out XLA.

The XLA framework is experimental and in active development. In particular, while it is unlikely that the semantics of existing operations will change, it is expected that more operations will be added to cover important use cases. The team welcomes feedback from the community about missing functionality and community contributions via GitHub.

Why did we build XLA?

We had several objectives for XLA to work with TensorFlow:

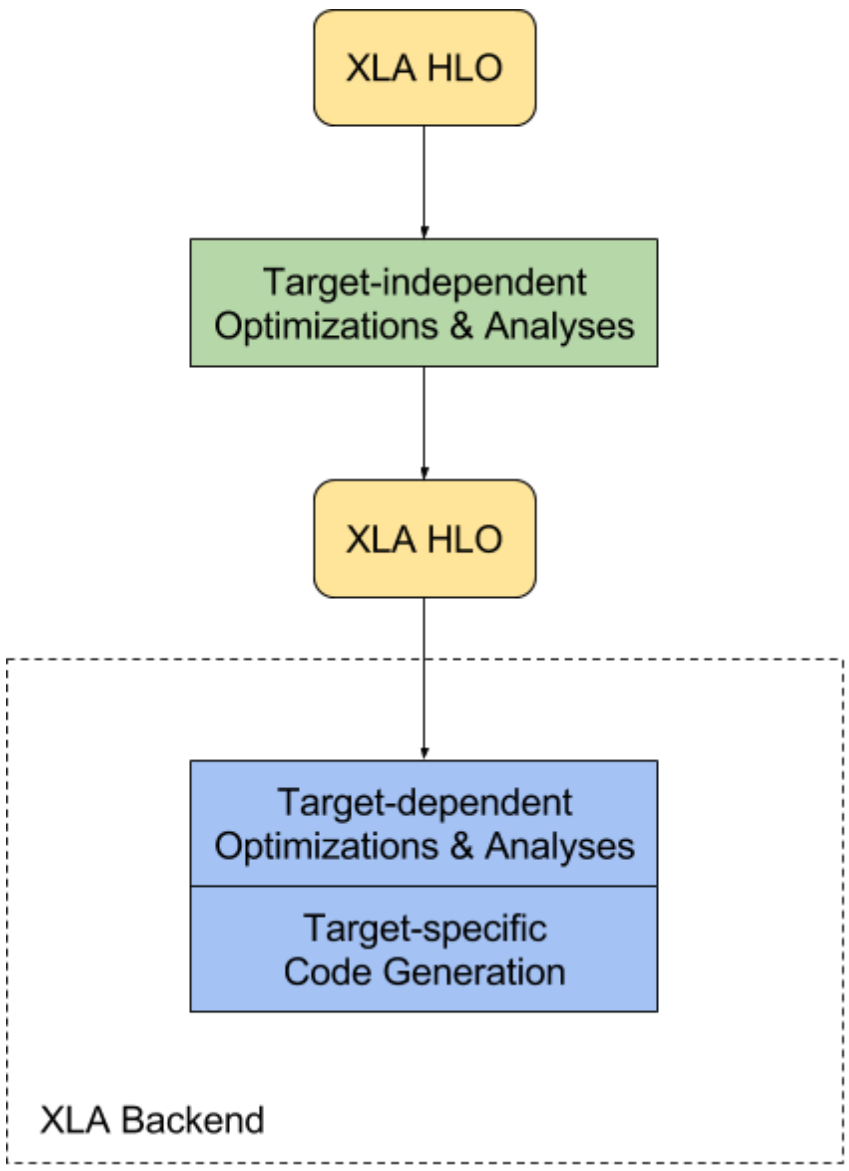
- *Improve execution speed.* Compile subgraphs to reduce the execution time of short-lived Ops to eliminate overhead from the TensorFlow runtime, fuse pipelined operations to reduce memory overhead, and specialize to known tensor shapes to allow for more aggressive constant propagation.
- *Improve memory usage.* Analyze and schedule memory usage, in principle eliminating many intermediate storage buffers.
- *Reduce reliance on custom Ops.* Remove the need for many custom Ops by improving the performance of automatically fused low-level Ops to match the performance of custom Ops that were fused by hand.
- *Reduce mobile footprint.* Eliminate the TensorFlow runtime by ahead-of-time compiling the subgraph and emitting an object/header file pair that can be linked directly into another application. The results can reduce the footprint for mobile inference by several orders of magnitude.
- *Improve portability.* Make it relatively easy to write a new backend for novel hardware, at which point a large fraction of TensorFlow programs will run unmodified on that hardware. This is in contrast with the approach of specializing individual monolithic Ops for new hardware, which requires TensorFlow programs to be rewritten to make use of those Ops.

How does XLA work?

The input language to XLA is called "HLO IR", or just HLO (High Level Optimizer). The semantics of HLO are described on the Operation Semantics (https://www.tensorflow.org/performance/xla/operation_semantics) page. It is most convenient to think of HLO as a compiler IR (https://en.wikipedia.org/wiki/Intermediate_representation).

XLA takes graphs ("computations") defined in HLO and compiles them into machine instructions for various architectures. XLA is modular in the sense that it is easy to slot in an alternative backend to target some novel HW architecture (https://www.tensorflow.org/performance/xla/developing_new_backend). The CPU backend for x64 and ARM64 as well as the NVIDIA GPU backend are in the TensorFlow source tree.

The following diagram shows the compilation process in XLA:



XLA comes with several optimizations and analyzes that are target-independent, such as CSE (https://en.wikipedia.org/wiki/Common_subexpression_elimination), target-independent operation fusion, and buffer analysis for allocating runtime memory for the computation.

After the target-independent step, XLA sends the HLO computation to a backend. The backend can perform further HLO-level analyzes and optimizations, this time with target specific information and needs in mind. For example, the XLA GPU backend may perform operation fusion beneficial specifically for the GPU programming model and determine how to partition the computation into streams. At this stage, backends may also pattern-match certain operations or combinations thereof to optimized library calls.

The next step is target-specific code generation. The CPU and GPU backends included with XLA use LLVM (http://llvm.org) for low-level IR, optimization, and code-generation. These backends emit the LLVM IR necessary to represent the XLA HLO computation in an efficient manner, and then invoke LLVM to emit native code from this LLVM IR.

The GPU backend currently supports NVIDIA GPUs via the LLVM NVPTX backend; the CPU backend supports multiple CPU ISAs.

Supported Platforms

XLA currently supports JIT compilation (https://www.tensorflow.org/performance/xla/jit) on x86-64 and NVIDIA GPUs; and AOT compilation (https://www.tensorflow.org/performance/xla/tfcompile) for x86-64 and ARM.

Last updated August 17, 2017.