Performance Analysis of Accelerated Linear Algebra Compiler for TensorFlow

Parth Chadha, Tejus Siddagangaiah
pchadha@andrew.cmu.edu, tsiddaga@andrew.cmu.edu

Machine learning algorithms are being widely used in systems for autonomous driving, image recognition, language translation. TensorFlow is an environment used by researchers to implement and test such systems. C++/Python is used as the front-end language to implement systems using TensorFlow. Until recent times, TensorFlow was executed on a system by run-time interpretations due to which no compiler level optimizations were performed. Google released an Accelerated Linear Algebra Just-in-time compiler for TensorFlow to optimize applications to improve speed, memory usage, portability and improved mobile footprint. In our work, we propose to analyze the performance of XLA compilation tool on machine learning algorithms like Convolutional Neural Networks, Long Short Term Memory and custom control flow graphs. If time permits, we aim to identify the issues and bottlenecks of the compiler and optimize both the compilation process or design control flow graphs taking these constraints into account.

I. Introduction

Researchers have used Machine Learning algorithms in the last decade to solve complex problems of Computer Vision, Natural Language Processing, robotics, information retrieval and medical research to name a few. Few of the tools used in machine learning research include TensorFlow, Torch and Caffe. TensorFlow [1] is an interface for expressing such algorithms and implementation for executing these algorithms. An algorithm expressed using TensorFlow can be ported across platforms with little to no change from general purpose CPUs, GPUs and distributed computing platforms. The tool is quite flexible and is extensively used to train and infer large Convolutional Neural Networks and Long Short Term Memory(LSTM) based algorithms. In TensorFlow, computations are described using data-flow like model and the computations are mapped on to a different hardware right from Android based mobile platforms, CPUs and GPUs.
A TensorFlow computation is described using a data-flow model described by a graph composed of a set of nodes. Programmers typically construct a computational graph using one of the front-end languages - Python or C++. The nodes are used to maintain the state of the program and the dataflow (loops and branches) of the program is maintained using edges connecting these nodes. Each node has a zero or more inputs and zero or more outputs. During the graph construction process, the operations that have to be performed are represented in the nodes and each operation can have several attributes which are provided or inferred from the graph. Each node is represented by an operation such as MatMul, Add, ReLU with one or more inputs. A kernel is a particular implementation of these operations and are designed according to the platform. These kernels are invoked at runtime according to traversal in computation graph, and the inputs, attributes for the kernels are managed at runtime.

These kernels are optimized implementations of operations and some of the libraries used for GPU kernels are cuBLAS, cuDNN, cuda-convnet. The nodes of the graph are executed in the control-flow order of the graph to maintain dependencies across nodes.

Google recently released an Accelerated Linear Algebra(XLA) compiler [3] [2] for TensorFlow to improve the execution speed, memory usage mobile footprint and improved portability of machine learning applications. XLA compiler provides the support for Just-In-Time compilation, which gives the following advantage:

- Fused pipeline operations to reduce memory overhead
- Memory usage analysis to eliminate intermediate buffer usage
- Fusing of operations/kernels to form a low-level op to match the performance of custom tuned low level operations

In the next section we describe XLA in detail.

II. Accelerated Linear Algebra Compiler(XLA)

XLA is a domain specific linear algebra compiler that optimizes the vector operations in machine learning algorithms implemented in TensorFlow. The XLA optimizations can be performed in two ways: Just-in-Time(JIT) or Ahead-of-Time(AOT). JIT compilation is performed to optimize the computational graph at runtime and perform fusing of operations. AOT compilation can be used to generate binaries for a specific architecture(mobile platform) so that runtime inference and Just-In-Time compilation is not required.

A. Just-In-Time Compilation

Just-In-Time compilation or dynamic translation is compiling the program during run-time. The compilation tool chain translates the front end program to machine code during run-time. The code is compiled when a particular section, function or file is about to be executed. A popular example of Just-In-Time compilation is how a Java virtual machine compiles the byte code into machine code at run-time. One of the key advantages of JIT is that the code is portable. Architecture specific optimizations can be performed at run-time.

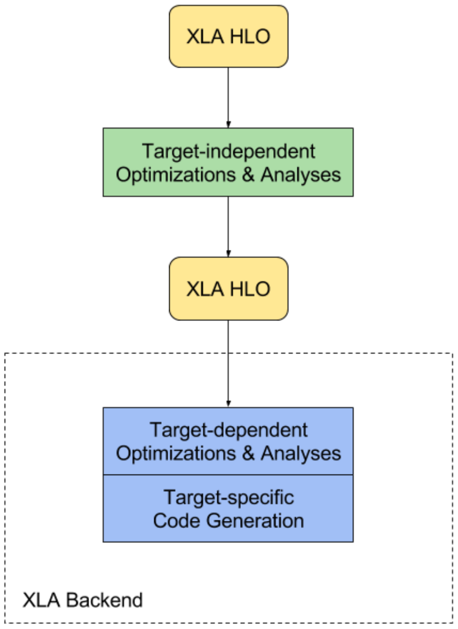

XLA HLO
Target-independent Optimizations & Analyses
XLA HLO
Target-dependent Optimizations & Analyses
Target-specific Code Generation
XLA Backend

Fig. 1. XLA compilation process

B. Ahead-Of-Time Compilation

XLA compiler also supports AOT compilation, where the code is compiled into a machine dependent byte code. XLA AOT compilation aims at generating code for mobile computing platforms where additional environments cannot be used for just in time compilation.

Figure 1 shows the compilation process of XLA. The input to the XLA compiler is the High Level Optimizer Intermediate Representation. A target independent optimization is performed on this intermediate representation. Using the optimized IR, a target-dependent analysis is done and optimzations are performed. Finally, an LLVM based compiler backend is used to generate the target specific code. The XLA currently supports backend for x86 processors and Nvidia GPUs.

C. Cluster Formation

The XLA compiler performs cluster identification and transformation as shown in Figure 2. A cluster of TensorFlow nodes are identified by the compiler and it performs optimization and JIT compilation for these sets of clustered nodes. Compiling subgraphs helps in the following ways:

- Reducing the execution time of short-lived Ops and eliminates the overheads from the TensorFlow runtime
- Fuse pipelined operations help in reducing memory overhead
- Specialization to known tensor shapes in XLA compilation helps to allow for more aggresive constant propogation.

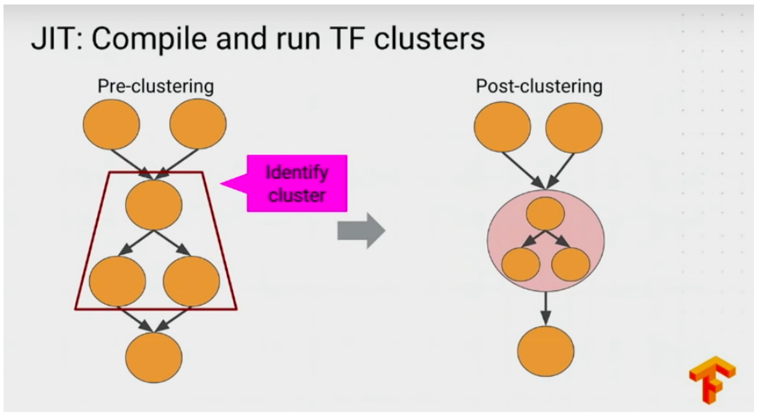


Fig. 2. Cluster formation in XLA

III. Experimental Setup

We started with setting up the XLA compilation tool with TensorFlow. As per this date, the XLA compilation tool-chain is not included in default TensorFlow installation and requires compiling from source with added support of XLA. During this phase, it is also required to compile if CUDA support is required or not. Since we do not currently have a GPU available, all our experimentation is based on CPU (Macbook Air, Intel Core i5,1.6Ghz, Dual-core with 2 way hyperthreading).

XLA performs multiple optimizations on the HLO-IR as listed below:

- Dead Code Elimination
- Common Subexpression elimination
- Matrix Transpose
- Algebraic Simplifier
- Transpose Folding
- Cpu Layout Assignment

The effect of some of these optimizations were evaluated and the most significant optimization is discussed in the next section.

XLA uses LLVM as the back-end to generate architecture specific binary. LLVM's optimization level (-O0 to -O3) can be controlled from within the XLA source code. Our experiments were performed with various optimization levels and the results are discussed in the next section.

- *Two layer Convolutional Neural Network:* The Convolutional Neural Network trained for MNIST dataset for 200,000 iterations.
- *SAXPY:* A simple SAXPY operation of $Z = AX + Y$. This test was performed to evaluate fusing multiply and add operation to utilize cache and optimize memory bandwidth.
- *Matrix Multiplication:* Matrix Multiplication was performed with XLA compilation enabled. This test case checks for Matrixmultiply fusion to create a common kernel for $a \times b \times c$.
- *Softmax:* Softmax is a common operation performed in Machine Learning. Softmax operation is defined as : $e_i^x / \sum e^{x_j}$

This test case evaluates fusing of operations performed by XLA.

- *Long Short Term Memory:* A simple LSTM test case is written to evaluate the performance gains of XLA compilation as the input size and the LSTM parameters are varied.

We performed experimentation with and without XLA compilation for both of the above listed networks. Table I shows the runtime observed for all listed configurations.

IV. Evaluation

A. Analysis of results

We observed that the optimization of Just-In-Time compilation is highly dependent on:

- Computational Graph Structure
- Extend to which the given computational graph can be clustered based on whether particular operations/kernels have XLA support or not

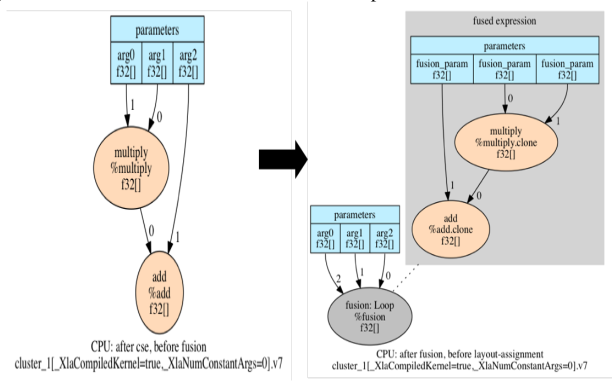
In this section we provide the analysis of computation graph for various operations described above. We have provided empirical evidence for complex computation graphs only as they provide meaningful data for analysis. Whereas the small computation graphs take negligible time to execute and can't be used for rigourous analysis.

The analysis of computational graphs we experimented with are presented below:

B. SAXPY

SAXPY operation involves 1 matrix multiplication and 1 addition and its arithmetic intensity is very low due to high number of memory access. The computation graph of SAXPY is as shown in Figure 3. The graph on left shows computation graph without XLA compilation and the one on right is with XLA compilation. XLA compiled graph shows the multiply and add unit to be fused together and this performs operation fusion. With the use of operation fusion, extra cycles required for memory read in add operation are avoided.

Fig. 3. SAXPY: Before and After XLA compilation


parameters
arg0 f32[]
arg1 f32[]
arg2 f32[]
multiply %multiply f32[]
add %add f32[]
CPU: after cse, before fusion
cluster_1[_XlaCompiledKernel=true,_XlaNumConstantArgs=0].v7
fused expression
fusion_param f32[]
multiply %multiply.clone f32[]
add %add.clone f32[]
fusion: Loop %fusion f32[]
CPU: after fusion, before layout-assignment
cluster_1[_XlaCompiledKernel=true,_XlaNumConstantArgs=0].v7

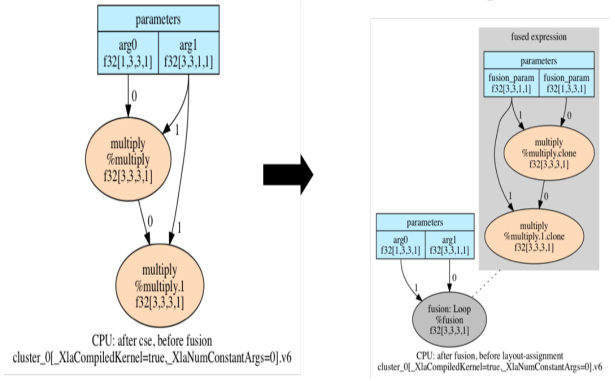
C. Matrix Multiplication

Matrix multiplication computation graph was defined as follows:

- Step 1: $temp = a \times b$
- Step 2: $res = temp \times c$

Figure 4 shows the computation graph obtained with and without use of XLA.

Fig. 4. Matrix Multiplication: Before and After XLA compilation


parameters
arg0 f32[1,3,3,1]
arg1 f32[3,3,1,1]
multiply %multiply f32[3,3,3,1]
multiply %multiply.1 f32[3,3,3,1]
CPU: after cse, before fusion
cluster_0[_XlaCompiledKernel=true,_XlaNumConstantArgs=0].v6
fused expression
fusion_param f32[3,3,1,1]
fusion_param f32[1,3,3,1]
multiply %multiply.clone f32[3,3,3,1]
multiply %multiply.1.clone f32[3,3,3,1]
fusion: Loop %fusion f32[3,3,3,1]
CPU: after fusion, before layout-assignment
cluster_0[_XlaCompiledKernel=true,_XlaNumConstantArgs=0].v6

As seen from the computation graph for matrix multiplication with XLA, the matrix multiplication operations are fused together to form a common kernel.

D. SoftMax

Softmax operation $e_i^x / \sum e^{x_j}$ involves point-wise exponential over the input vector, reduction of the computed exponential vector and division operation. Since this operation can lead to numerical instability, we often apply a trick of subtracting from the maximum element in vector and then applying the above operations.
The computation graph of the Softmax is as shown in Figure 5. XLA compiled graph on the right shows that XLA fuses the point-wises subtraction from maximum and exponential operations to form a combined kernel. This helps in efficient use of memory operations because the same memory element can be used for subtraction and exponential without reading/writing from memory.
Also, we observed that the XLA compilation does not fuse reduction operation in computational graphs and these operations are performed without JIT compilation.

E. CNN for MNIST dataset

All the test cases described before this involved simple operations. We have chosen convolutional neural network [5] as a test case because it performs all of the above operations multiple times and results in meaningful emperical data for our analysis. The network is trained using the MNIST dataset for 200,000 iterations.

The Neural Network was trained with multiple XLA configurations and without XLA JIT compilation. We observed a significant difference in the runtimes and the results are shown in table I. As a baseline implementation, we executed

Fig. 5. SoftMax: Before and After XLA compilation

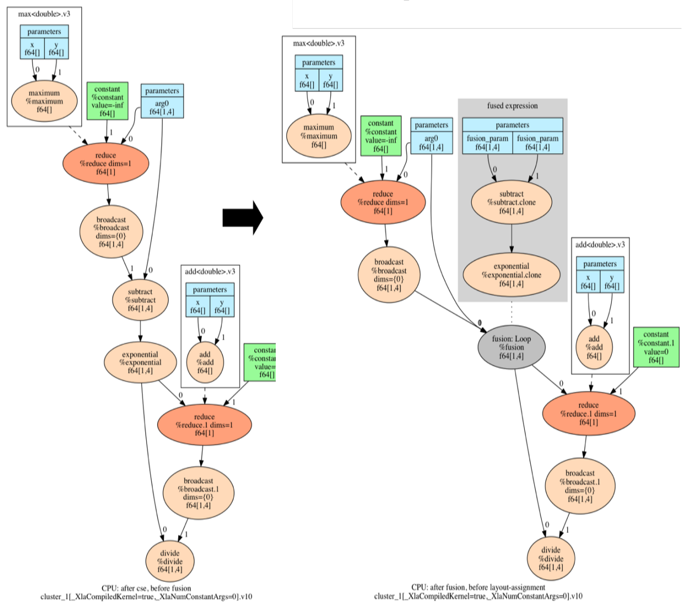


TABLE I
EXPERIMENT RESULTS OF MNIST NETWORK

Optimizations	Runtime
Baseline	605.03s
XLA - LLVM - O0	1041.77s
XLA - LLVM - O3	448.77s
XLA - HLO Optmization (No Matrix Transpose)	524.11s

our neural network code without XLA compilation and the runtime is 605.03s. The same experiment was executed with XLA enabled and we achieve a speedup of 1.34x. We achieve this speedup due to a few key optimizations performed by XLA:

- *Optimized Libraries for Matrix Multiplication:* XLA uses optimized libraries like Eigen to perform matrix multiplication and convolution operations. Usage of highly optimized libraries to generate the in-memory binary has a significant impact on the performance.
- *Matrix Transpose:* At the HLO-IR, XLA performs an optimization called *Transpose Folding:*, which performs matrix transpose to improve cache locality. Disabling this optimization decreased the performance and the speedup reduced to 1.15x.
- *Multi-threaded operations:* XLA utilized multi-core CPU architectures by dividing the workload of Convolution and Matrix Multiplication across different cores. Usage of Eigen library with multi-threaded computation improves the performance further.

The optimization level of LLVM backend can be controlled through the XLA source code. The optimization level was changed to -O0 and the performance degraded significantly to 0.58x.

To analyze if there was further scope of optimization within the LLVM backend, we dumped the LLVM IR generated by XLA to *.ll* files. As linking a custom LLVM build was out of the project scope, we used the XLA generated LLVM IR and ran optimization passes on these. We observed that LLVM-3.7 optimized the LLVM IR (Loop Unrolling) further with the standard -O3 flag. However, we were not able to compare the runtimes as the generated LLVM IRs are modules without a main function.

TABLE II
EXPERIMENT RESULTS

LSTM Matrix Size	Runtime w/ XLA(s)	Runtime w/o XLA(s)
10	0.56	0.063
20	1.176	0.063
50	0.388	0.082
100	0.466	0.250
250	1.006	3.22
500	4.469	24.14
600	8.33	51.07
750	57.95	513.76
1024	1057.1	Not Available

F. LSTM

A LSTM [4] cell operation includes 8 matrix-matrix multiplication and a number of point-wise addition operations as shown in equations 1-6. XLA improves the performance of LSTM cells significantly. Matrix size of the LSTM cell was varied from 10 to 1024, the runtimes for these configurations are shown in table II. When the matrix size is small, the XLA JIT compilation is becomes a overhead and the performance decreases as expected. However, as the matrix size increases, XLA improves the performance significantly. The change in speedup as the matrix size varies is shown in figure 6. There is an exponential increase in performance with XLA as the matrix size increases, this is due to the kind of optimizations XLA performs as explained in section IV-E.

$$i_t = sigm(\theta_{xi} \times X_t + \theta_{hi} \times h_{t-1} + b_i) \quad (1)$$

$$f_t = sigm(\theta_{xf} \times X_t + \theta_{hf} \times h_{t-1} + b_f) \quad (2)$$

$$o_t = sigm(\theta_{xo} \times X_t + \theta_{ho} \times h_{t-1} + b_o) \quad (3)$$

$$g_t = sigm(\theta_{xg} \times X_t + \theta_{hg} \times h_{t-1} + b_g) \quad (4)$$

$$c_t = f_t \cdot c_{t-1} + i_t \cdot g_t \quad (5)$$

$$h_t = o_t \cdot tanh(c_{t-1}) \quad (6)$$

Another key observation with respect to XLA optimization was regarding the memory consumption of Tensorflow code. As the matrix size increases, the memory consumed by Tensorflow increases significantly. For our LSTM test case, with matrix size 1024, the memory utilized without XLA is approximately 25.46GB. There is a significant reduction in the memory usage with XLA and the memory utilized reduces to approximately 16.25GB. During the execution, we observed that maximum time was spent on fetching the data from memory than on computation. The CPU usage during the execution is shown in Figures 7 and 8. The CPU utilization goes as low as 7.16% while the kernel spends majority of the time to fetch the data from Swap Memory (36.65%). During LSTM computation, the system utilizes

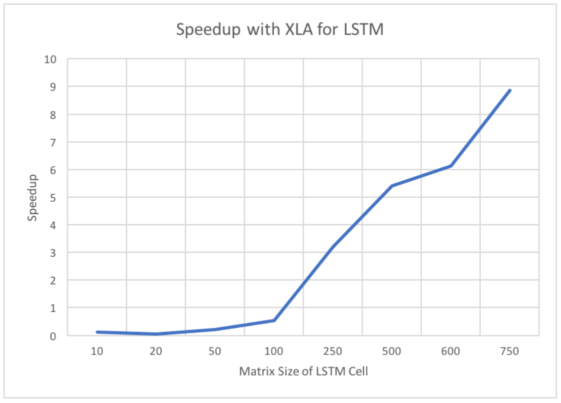
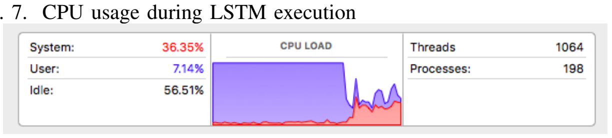


Fig. 6. Speedup with XLA for LSTM Cells

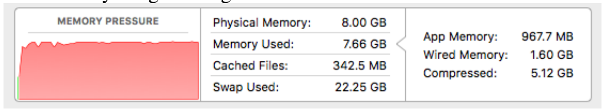
maximum RAM available to the CPU and uses the SWAP memory extensively.This improvement in performance is mainly due to the techniques used by XLA for optimum buffer assignment.

Fig. 7. CPU usage during LSTM execution



System:	36.35%	Threads 1064
User:	7.14%	Processes: 198
Idle:	56.51%	

Fig. 8. Memory usage during LSTM execution



Physical Memory:	8.00 GB	App Memory:	967.7 MB
Memory Used:	7.66 GB	Wired Memory:	1.60 GB
Cached Files:	342.5 MB	Compressed:	5.12 GB
Swap Used:	22.25 GB		

V. Challenges Faced

We invested a lot of time into reading the documentation/code of XLA and setting up the compilation environment. Unfortunately since the XLA project is still new, there are a lot of gaps in documentation and tutorials available. We have listed below few of those concerns:

- No documentation for linking custom LLVM build with XLA compilation environment. As a stretch goal, we had planned to experiment with custom LLVM passes over the LLVM IR generated by XLA for optimization. We have shown that is there is scope for improvement in LLVM optimizations for XLA generated IR via experimenting on the IR level, we were not able to execute these .ll files due to a lot of dependencies.
- Lack of documentation for observing the spilled out assembly code on CPU/GPU. The spilled out CPU/GPU code is demonstrated in Google's XLA presentation, but it seems it is just an internal tool used by Google and not open-sourced.
- Not enough documentation for understanding of the computational graph, since it produces 100's of graph files during optimization process

VI. Conclusion

In this project, we performed extensive analysis of Accelerated Linear Algebra Compiler (XLA) for TensorFlow compuation graphs. The algorithms chosen for the evaluation are popular machine learning algorithms. The test cases written result in complex data flow graphs, which enables XLA to exploit maximum cluster formation and perform aggressive optimizations.

The goals of the project during the proposal stage are listed below:

- **75% Goal:** In detail analysis of two experiments mentioned above.
- **100% Goal:** In detail analysis of all the experiments with suggestions to improve performance for the identified bottlenecks or issues.
- **125% Goal:** Implement or change the compiler implementation to improve performance for the identified bottlenecks or issues.

We have successfully achieved our 100% goal. However, our stretch goal involved linking XLA with a custom LLVM build. Due to the lack of documentation, we were not able to build an end-to-end working Tensorflow binary with custom LLVM build. We dumped the XLA generated LLVM IRs and performed experimented with different optimization passes on them with LLVM-3.7. After analyzing the two IRs, we observed that there is scope for futher optimizations at both HLO-IR and LLVM-IR level.

VII. Distribution of Total Credit

Both Parth and Tejus worked on this project together and we have each contributed 50%.

References

[1] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Jeff Dean. *XLA Jeff Dean presentation*. URL: https://autodiff-workshop.github.io/slides/JeffDean.pdf.

[3] Google. *XLA Compilation:Google*. URL: https://www.tensorflow.org/performance/xla/.

[4] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.