Operator Fusion in XLA: Analysis and Evaluation

Daniel Snider   Ruofan Liang

University of Toronto

{firstname.lastname}@mail.utoronto.ca

Abstract—**Machine learning (ML) compilers are an active area of research because they offer the potential to automatically speedup tensor programs. Kernel fusion is often cited as an important optimization performed by ML compilers. However, there exists a knowledge gap about how XLA, the most common ML compiler, applies this nuanced optimization, what kind of speedup it can afford, and what low-level effects it has on hardware. Our paper aims to bridge this knowledge gap by studying key compiler passes of XLA's source code. Our evaluation on a reinforcement learning environment Cartpole shows how different fusion decisions in XLA are made in practice. Furthermore, we implement several XLA kernel fusion strategies that can achieve up to 10.56x speedup compared to our baseline implementation.**

I. INTRODUCTION

Machine learning (ML) becomes more and more important in various computer tasks including computer vision, natural language processing, robotic control, etc. The computation efficiency of ML also arises as a popular topic for computer system and architecture research. Today's machine learning (ML) applications rely on specialized hardware accelerators like GPUs in order to be performant. However, because hardware accelerators are designed to be simple and fast, they lack features that CPUs have to automatically speed up user code. As a result of this and with the addition of the complexity of designing efficient parallel programs, modern ML programs require complex optimizations. Furthermore, these optimizations are often unique to each combination of algorithm, data shape, and hardware. To address this problem, ML compilers can make these optimizations automatically, but they are still limited and an emerging field of research. The ML compilers take the model definitions described in the ML frameworks as inputs, and generate efficient code implementations on various ML hardware as outputs. The transformation between model definition and specific code implementation are highly optimized targeting the model specification and hardware architecture [11].

The adoption of ML compilers for GPUs is still nascent. The existing works like TVM AutoScheduler [5], Ansor [22], Rammer [12], and TorchScript compiler [1] are limited to supporting ML inference. On the other hand, XLA compiler [16] is more mature and flexible. XLA is the most widely used ML compiler because it is used in TensorFlow [3] and JAX frameworks [7]. A common optimization step

This is a CSC2224 course project report.

for XLA and other ML compilers is operator fusion. ML compilers can automatically fuse multiple operations into one computation kernel to reduce the memory transfer and kernel launch overhead. XLA uses rule-based fusion strategies to fuse operations that meet certain fusion patterns and requirements set by experienced developers. Recent research often compares XLA's performance to hand-optimized GPU kernels like cuDNN and super-optimizations like Ivanov et al. [8]. These papers find performance gaps between their optimized kernels and XLA's generated kernels. We are motivated to look closely at this performance gap by investigating what optimizations and sacrifices XLA makes in its approach to automatic optimization.

By analyzing XLA's source code and evaluating a microbenchmark of XLA, we are now able to give an in-depth explanation on XLA's operation fusion behavior. XLA's fusion heavily depends on the initial computation graph converted from the Python frontend. A low-quality python implementation can cause a major performance drop. Additionally, conservative fusion criteria in XLA also limits the opportunities for optimization. Our evaluation on a reinforcement learning environment Cartpole [17] shows how different fusion decisions in XLA are made in practice. Furthermore, we also explore some potential optimizations that can achieve up to 10.56x speedup compared to our baseline JAX-XLA implementation.

To summarize, we make the following contributions in this project:

- We give an in-depth introduction to the fusion mechanism of XLA, which has not yet been well described in public papers or documents.
- We evaluate XLA's performance with one JAX-based RL task and explore ways to further improve the performance of XLA's generated code.
- We make a low-level profiling of XLA programs to identify the existing limitations and potential optimizations for the XLA compiler.

II. RELATED WORK

A. Optimized ML Frameworks

The conventional ML frameworks such as PyTorch [14], TensorFlow (without XLA) [3], and MXNet [4] commonly

map DL operations to cuDNN/cuBLAS primitives or pre-implemented CUDA kernels for the full flexibility of ML algorithm prototyping, but this design cannot provide full optimizations for ML programs. The optimized deep learning frameworks are later proposed to generate or use workload-specific kernels for better performance. The typical DL optimization frameworks include XLA [16], TensorRT [2], TVM [5], Tensor Comprehensions [18], etc. XLA and TensorRT use some manually defined rules to fuse simple operations, while for complicated operators such as convolution, matrix multiplication, these frameworks still rely on the cuDNN/cuBLAS primitives. One advantage of this type of ML compilers is that they require relatively shorter compile time to finish optimization. On the other hand, frameworks like TVM and Tensor Comprehension have more flexible code generation, instead of using primitive kernels, these frameworks can automatically tune the fused kernels by using some learning algorithm such as GBM, simulated annealing, and genetic algorithm, which can result in relatively longer compilation time and limited sub-graph level searching and tuning.

Compared to other ML compilers, XLA is one of the most widely-used ML optimization compilers because of its core position in Google's AI computation environment from the frontend (TensorFlow/JAX) to the backend (TPU). The developers can directly benefit from the speedup provided by XLA from their TensorFlow or JAX code without additional effort on the code conversion. This is also one the main reasons we choose XLA as our target ML optimization framework.

B. Operation Fusion in DL

The operation fusion is also an active topic for ML compiler research. In addition to aforementioned ML optimization frameworks, recent works explore more complicated fusion strategies for better hardware resource utilization.

Ivanov et al. [8] do an exhaustive search of all possible data layout and operator fusion in the Transformer model [19]. Their results show that an optimal fusion strategy can provide up to 22.91% data movement reduction and overall achieve a 1.30x performance improvement over the state-of-the-art implementations. PET [20] allows partially equivalent transformations such as joining two tensors and performing a single convolution instead of two. Such equivalent transformation can increase the optimization search space, and PET uses the beam search to optimize the entire DNN. DeepCuts [9] applies a greedy exploration guided by an analytical cost model to tune graph-level fusion decisions along with some CUDA kernel parameters. However, DeepCuts only considers limited vertical operation fusions in the computation graph, and it has relatively high implementation cost for the new operations because of its analytical cost model. Unlike the XLA compiler, [8], [20], and [9] all employ some types of design space search algorithms to find some better fusion/optimization combinations. These methods require much longer compilation time (it can be multiple hours) to find a good enough solution, thus algorithm developers cannot instantly benefit from these optimizations during the fast algorithm prototyping stage. The equivalent computation transformation mentioned in [20] and [13] could be a promising future optimization direction, which will also be discussed in our XLA analysis.

There are other non-trivial recent ML fusion optimization works. For example, XTAT [15] uses the auto-tuning method directly on the XLA's multi-pass optimization to generate better kernels for TPU. XTAT can perform joint optimization for different XLA passes at both graph-level and subgraph-level. However, this autotuner is relatively compute-heavy and time-consuming, which cannot be finished by the JIT compilation of TensorFlow/JAX. Besides, XTAT autotuner only targets TPU accelerators, and is mainly used for improving the existing heuristic algorithm in Google's internal TPU-XLA.

HFTA [21] is another fusion technique that vectorizes the training of multiple models together. This reduces overhead (number of CUDA API calls, DL framework GPU memory footprint, etc.) for GPU computation which in turn increases throughput. Though XLA also has some horizontal fusion mechanisms, HFTA supports end-to-end training and is an ideal way to accelerate hyperparameter tuning.

III. XLA's Multi-pass Optimization

A. XLA Computation Graph Optimization

Once the traced computational graph from TensorFlow or JAX is sent to the XLA compiler, a series of fine-grained optimization passes are executed to gradually optimize the initial computation graph (represented as an XLA HLO IR). Some optimization passes are performed more than once, with the most common being Dead Code Elimination (DCE) and Common Subexpression Elimination (CSE). Optimization passes are logically grouped into what are called "`Pass Pipelines`". XLA's graph-level optimization pipeline passes are listed below. Our experimentation has found that kernel fusion is one of the last optimization pipelines to run.

- **SPMD partitioner**. This step partitions tensors to be operated on in parallel across devices. (SPMD, Single Program Multiple Data) [10].
- **Optimization**. Includes passes for canonicalization, expansion, and simplification. Mostly simple rule-based operator conversions for later use, e.g., BatchNorm Expander and Logistic Expander converts convert a complicated computation into a sequence of simple operations).
- **Simplification**. Performs simplification to specific operations, e.g., inlining and constant propagation, and WhileLoopSimplifier removes dead tuple elements.
- **Collective optimizations**. Optimizes collective operations (e.g., reduce, gather) generated by SPMD partitioning for multiple devices.

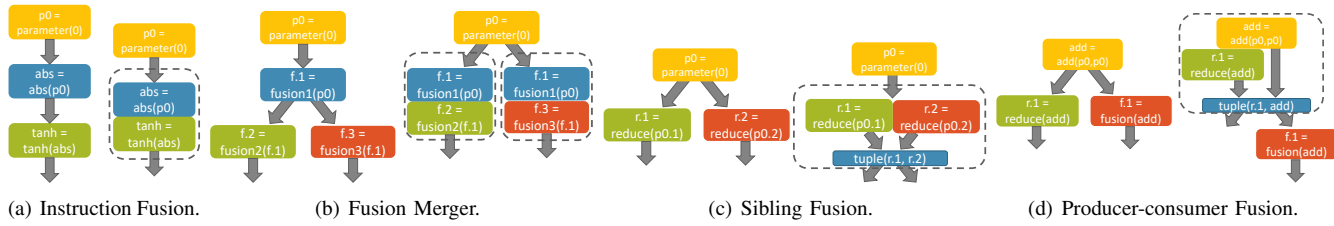(a) Instruction Fusion.   (b) Fusion Merger.   (c) Sibling Fusion.   (d) Producer-consumer Fusion.

Fig. 1. Fusion strategies in XLA.

- **Conv canonicalization**. Canonicalize convolution operations, i.e., reshaping the input and filter to NHWC and HWIO order, respectively.
- **Layout assignment**. Pre-assigns layouts of some operands to satisfy layout constraints and results of library calls (e.g., cuDNN, cuBLAS).
- **Post layout assignment** Performs target-specific HLO optimization passes after layout assignment, e.g., Optimize padding for cuBLAS and pick GEMM or Conv algorithm.
- **Fusion** Performs different types of vertical operation fusion, e.g., simple instruction fusion, fusion merger, and multi-output fusion.
- **Horizontal fusion**. Performs the horizontal operation fusion. It includes horizontal loop fusion and horizontal input fusion.
- **Post fusion optimization**. Combines small non-dependent collective operations into larger combined operations.
- **GPU IR emit prepare**. Sanitizes the given HLO module so that it will be accepted by IR Emitter.

In this project, we mainly focus on the fusion optimization passes that help XLA gain speedups. Fusion strategies are discussed in detail in the next subsection.

B. Operation Fusion

Because XLA's fusion strategies are not described in any official documentation or paper we have reviewed the XLA source code [1] in-depth. We have extracted all the fusion strategies used by XLA. Fig. 1 shows four typical fusion strategies commonly used by XLA. More details about different types of kernel fusions are listed as follows.

Instruction Fusion. This is a simple vertical operation fusion step, in which producing instructions are fused into their consumers with the intent that the sequential operations will be fused in code generation (see Fig. 1(a)). XLA does a reverse post-order traversal in this step to determine whether two dependent operations should be fused or not via a `ShouldFuse` function. XLA defines several rules to check whether the operation is fusible. For example, XLA explicitly

maintains a list of "expensive" operations[2] (e.g., convolution, sort, all reduce, etc.) that should not be fused. XLA also checks whether the fused kernel will be too large for the GPU, whether the fused kernel will cause a nested loop, etc. XLA will make sure not to exceed several GPU hardware limits including threads per block, shared memory per block, and threads per SM.

Fusion Merger. This fusion pass attempts to merge fusion instructions to reduce memory bandwidth requirements and kernel launch overhead (Fig. 1(b)). Fusion instructions are merged into their users if some conditions are met. For example, the result of merging the fusion instruction into its users would not increase bytes transferred; and if producer operations are fusible with all consumers (if they are not fusible with at least one consumer, they won't be fused at all).

Multi-Output Fusion. Multi-output fusion of sibling and producer-consumer instructions for the GPU backend is also intended to reduce memory bandwidth requirements. Typically, there are two types of multi-output fusion this pass performs–sibling multi-output fusion (Fig. 1(c)) and producer-consumer multi-output fusion (Fig. 1(d)). Fusion of sibling operations can reduce memory bandwidth requirements, because common input parameters have to be read only once. Fusion of producer-consumer operations reduces memory bandwidth requirements by eliminating one read from memory. Sibling fusion and producer-consumer fusion can usually meet the fusion constraints at the same time. XLA will select the one that can give more fusion opportunities for later fusion optimizations, and sibling has a higher priority over producer-consumer by default. These two types of multi-output fusion can also be combined in this fusion pass.

Horizontal Fusion. This fusion pass horizontally fuses computations to reduce kernel launch overhead while increasing kernel launch dimensions on the GPU. The initial motivation of the horizontal fusion is due to the observation that the training optimizer phase (e.g., Adam optimizer and L2Loss, etc.) typically has many small kernels as a result of applying the same formula on many training parameters (or variables in Tensorflow). Fusing these small kernels, hence, provides performance gain. This fusion style provides an important

---

[1] https://github.com/tensorflow/tensorflow/tree/master/tensorflow/compiler/xla

[2] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/service/instruction_fusion.cc#L62

advantage that <mark>kernels of different shapes can be horizontally fused</mark>[3]. For example, consider <mark>a multiply operation and an add operation with separate input shapes, but their output is consumed by a common operation</mark>, here horizontal fusion is triggered.

C. Other Findings

Kernel scheduling and CUDA streams. At compile time, XLA's `IrEmitter` also generates `KernelThunks` which contain necessary arguments for launching kernels. At run-time, <mark>`GpuExecutable` launches the kernel using the `KernelThunk` which specifies the buffer addresses of the data needed for the kernel launch</mark>. An initial finding is that the function `BFSLaunchOrder()` <mark>computes a topological launch order</mark> that is close to a breadth-first order. <mark>This enables the possibility of launching kernels concurrently in different CUDA streams</mark>. The function `CanRunConcurrently()` returns whether the two HLOs can run concurrently, however <mark>in practice we have not seen multiple streams utilized by XLA</mark>.

JAX Compile-time speed. XLA's compile time can be slow, especially when the input JAX source code includes native python loops. <mark>Each python loop iteration is traced and adds a repetition of body into the XLA intermediate representation</mark>. To avoid this a user must use JAX's built-in loop constructs. Through detailed investigation of Nsight System execution timelines we have identified two ways to speed up JAX's compile time that are not reported anywhere that we have seen: (1) <mark>disabling JAX tree flatten conversion and operate only on arrays</mark> (not objects), or (2) <mark>disable GEMM autotuning</mark> because it often doesn't help.

Rules-based optimization. XLA compiler does not search for any optimization, and the number of compiler passes is fixed. The trade-off between time spent waiting for compilation and compiler-based speedups has been chosen to work well for general purpose, everyday development. This trade off can be controlled, but the interface is not easy to control for normal uses. You can disable compiler passes using an environment variable, but this requires you to know the low-level names of XLA's compiler passes.

IV. CASE STUDY ON CART-POLE

We use a classic reinforcement learning environment Cart-pole [17] as the benchmark task to study the XLA graph optimizations (this is inspired by the recent success of JAX-based RL simulation framework [6]). Cart-pole is a dynamic control system with an inverted pendulum on a cart. We implement the Cart-pole environment in JAX, Fig. 2 shows our baseline implementation of the environment update function, the core computation of our benchmark task. We will show

---

[3]See here for more details https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/service/gpu/horizontal_loop_fusion.h#L63

```python
def dynamics(self, state, action):
    x, x_dot, theta, theta_dot = state
    force = self.force_mag if action == 1 else -self.force_mag
    costheta = np.cos(theta)
    sintheta = np.sin(theta)
    temp = (force + self.polemass_length * theta_dot**2 * sintheta)\
        / self.total_mass
    thetaacc = (self.gravity * sintheta - costheta * temp) / (
        (4.0/3.0 - self.masspole * costheta**2 / self.total_mass)
        * self.length)
    xacc = temp - self.polemass_length * thetaacc * costheta\
        / self.total_mass
    x = x + self.tau * x_dot
    x_dot = x_dot + self.tau * xacc
    theta = theta + self.tau * theta_dot
    theta_dot = theta_dot + self.tau * thetaacc
    return np.array([x, x_dot, theta, theta_dot])

def step(self, action):
    self.state = self.dynamics(self.state, action)
    [x, x_dot, theta, theta_dot] = self.state.transpose()
    done = np.where((np.abs(x) > self.x_threshold) |
        (np.abs(theta) > self.theta_threshold_radians), 1, 0)
    self.state = self.reset_some(done)
    reward = np.ones(done.shape)
    return self.state, reward, done, {}
```

Fig. 2. The JAX code for the Cart-pole environment update step.

how XLA converts this code into fused kernels as well as further analysis and optimization of this code in the following sections.

A. XLA Compilation

When running our JAX-Cartpole simulation environment, the JAX JIT compilation first traces and converts the Python code into the HLO computation graph. This HLO IR then goes through multiple optimization passes, as described in Sec. III-A. Fig. 3(a) shows the initial HLO graph converted by JAX, and Fig. 3(b) shows the HLO graph after several optimization passes (e.g., simplification, layout assignment optimization, etc.) and right before the fusion optimization pass. By comparing the two computation graphs, we can see that XLA removes or replaces the duplicate or redundant operations from the initial HLO graph in a multi-pass manner. Because our focus is on the operation fusion optimization, we skip the detailed discussion of XLA optimization passes before the fusion in this report. Next, we will explain XLA's fusion decisions performed on the HLO graph 3(b).

Fig. 3(c) shows the HLO graph after all the fusion passes. We see from the graph that operations in graph 3(b) are fused into 6 fused kernels. To understand how XLA makes the current fusion decisions, we investigate the XLA's source code and runtime log file. We find that most of operations in the graph are fused in the instruction fusion pass, and there three interesting fusion boundaries are shown in the graph (the boxes with dashed lines in 3(c)) which form final set of fused kernels. Next we explain these three XLA fusion boundaries.

❶ The first one is at the bottom of the computation graph. The fused kernel shown in box 1 is used only for XLA's implementation of while loop control. By looking at

(a) Before any optimization.   (b) Before fusion optimization   (c) After the fusion optimizations; right: 3 fusion boundaries for the fused kernels.
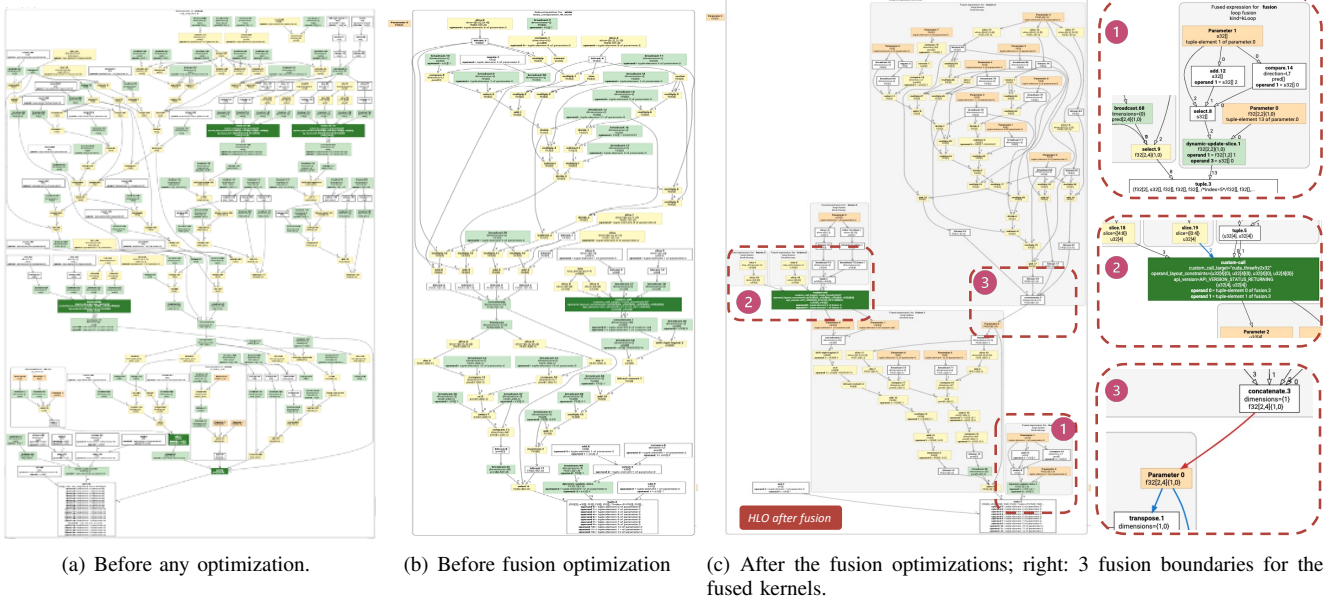
Fig. 3. HLO computation graph for Cartpole update step.

XLA's fusion code[4] we find that XLA does not fuse a tuple into its producer because it does not provide performance gain. This is because a tuple is not a kernel operation, it is a location in global memory. This tuple is used as the output of each loop step and the final output when the loop terminates.

2 The second one is in the middle left of the computation graph. This fusion boundary involves a custom-call operation "cuda_threefry2x32", which is a pre-defined kernel for the random number generation. XLA does not have the ability to fuse such custom operations into its consumer or producer. Other common custom-calls such as cuDNN/cuBLAS primitives can also halt the expansion of fused kernels at these operations.

3 The third one is in the middle right of the computation graph for the concatenation operation. Fusing `concatenate` in this case violates the XLA's predefined fusion rules that a `concatenate` operation with more than one user (blue arrows in the graph) cannot be fused, because XLA developers think such fusion may cause potentially high code duplication in general cases. However, we do not think the `concatenate` operation in our HLO graph could cause relatively high code duplication. We will show our attempt to bypass this limitation by modifying XLA's fusion decision functions.

V. EVALUATIONS

We performed 6 experiments to gather insights about XLA's fusion behaviour and performance. In each experiment we measured 2048 parallel simulation environments for 10,000

---

[4]https://github.com/tensorflow/tensorflow/blob/master/tensorflow/compiler/xla/service/gpu/gpu_fusible.cc#L215

steps. We chose 2048 because it is the largest number of environments used in the Brax paper to make effective use of the GPU, while not being so large to cause RL learning to diverge. We choose 10,000 steps because it is enough to amortize the one-time overhead caused by the JAX framework which includes serializing objects and moving data to the GPU to begin execution. We tested on the Eco-13 server which contains an RTX 2080Ti GPU with CUDA driver 11.2.

A. Remove cuRAND Kernels (Baseline)

The first thing we did was to remove the unfusable "cuda_threefry" cuRAND kernel which is responsible for randomness. We did this by precomputing a pool of random values to be used as random actions in the simulator and random start states for environment resets. The result was a reduction of the cuRAND kernel and it's 3 parent kernels as seen in figure 4. This yielded a 1.87x speedup. What remains are four kernels. Two kernels for the simulation and
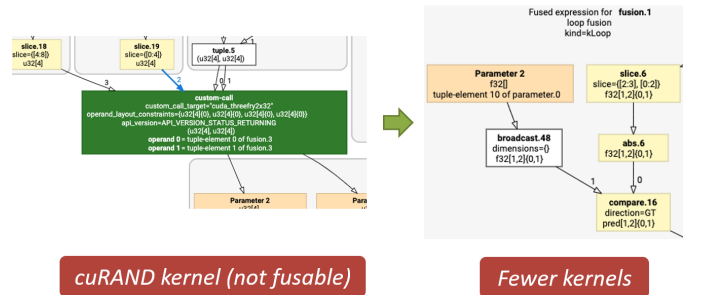

Fig. 4. We replaced the cuRAND kernel (in green) with precomputed random values to bring our cartpole implementation closer to a single fully fused kernel.
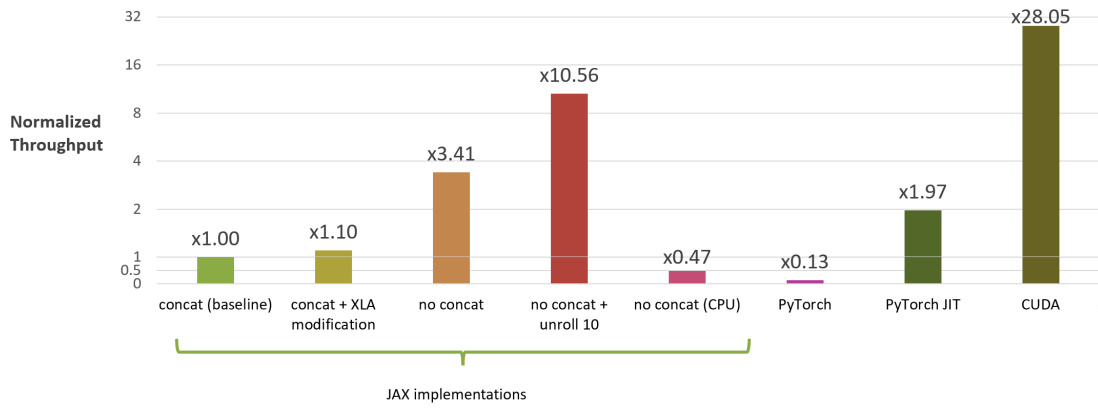
Fig. 5. Normalized throughput of different implementations of cartpole simulation.

two kernels involved in JAX's implementation of the "scan" loop. In our throughput plot, Figure 5, this implementation corresponds to "concat (baseline)".

B. Fusion via XLA Modification

With two kernels left to run the simulator, we investigated why XLA didn't fuse them. We found that the situation resembled the Fusion Merger type seen in Figure 1. XLA's prevented this fusion because of a function called function CodeDuplicationTooHigh(). Fusion merger requires the parent operation to be fused with the child, however, there are two consumer operations, each needing the parent to be duplicated, and it appears that XLA's code doesn't want to support this. We modified XLA's source code to allow up to three consumers of the parent operation which led to fusion of the two simulation kernels (see Figure 6).
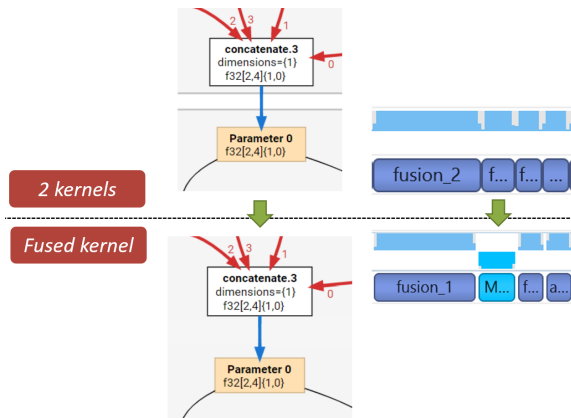

Fig. 6. We modified XLA so that it would fuse this concatenation operation into its child operation.

However, we only saw a marginal 10% speedup. It is important to note that the operation that was separating the two kernels was a "concatenate" operation which writes a new array that is too large to fit into registers. We hypothesize that memory movement is a larger bottleneck than launch

overhead, and because we haven't changed the amount of memory operations, the speedup is negligible. Using Nsight Systems we confirmed that one kernel was eliminated by in its place additional Device-to-Device memory transfer remained.

C. Fusion via Memory Movement Optimization (no concat)

Next we approached kernel fusion by addressing the core problem of our simulator's design. The concatenation operation was an unnecessary convenience to communicate state between our functions, so we instead passed the four state values individually as seen in Figure 7.
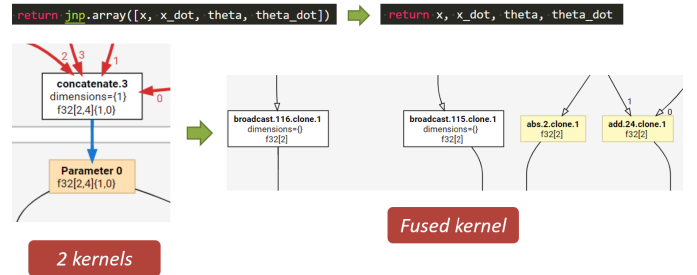

Fig. 7. We improved our code to remove the concatenation operation which allowed XLA to fuse together two cartpole kernels into one.

This memory movement optimization allowed XLA to fully fuse the simulation and the variables could remain local in registers without needing to be combined at a higher level of memory. This resulted in a 3.41x speedup. Using Nsight Compute we confirmed that register usage went up by 40%, the total executed instructions went down by 50% (most of them memory requests), and the number of stalled cycles went down by 33%.

D. Fusion via Unroll

At this point our simulation kernel was still characterized as a tiny kernel full of elementwise operations which made it bound by kernel launch overhead. Our next strategy to kernel

fusion was to use the built-in support to unroll Jax's scan loop. We unrolled the loop 10 times which means that one kernel contains the duplicated instructions for 10 loop iterations.
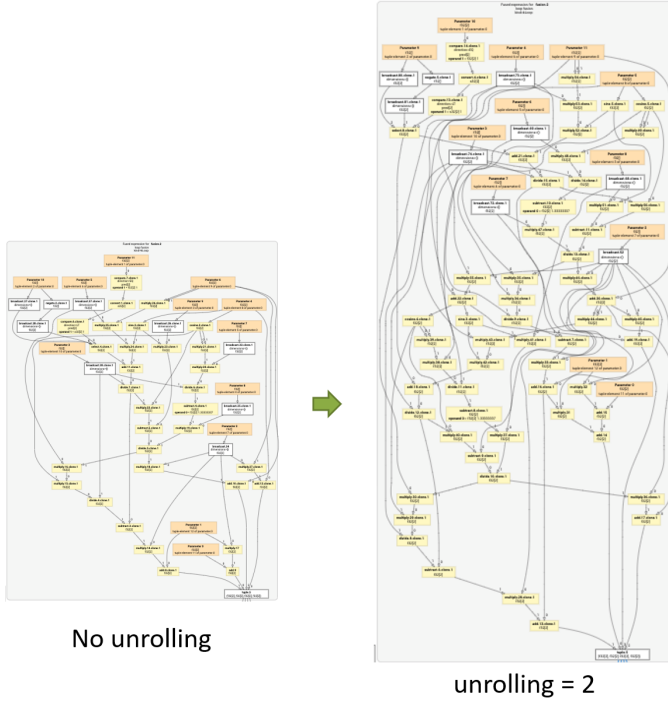

Fig. 8. Illustration of XLA's HLO IR computational graph before and after unrolling. Operations in the loop body become duplicated.

This reduced the number of launched kernels by 10x and we saw a 3.5x speedup over the previously optimized but not unrolled implementation. While the downside of unrolling is increased program size (as seen in Figure 8) and increased compile time (from roughly 300ms to 1400ms), the upside is that memory locations can be precomputed by the compiler, jumps to the start of the loop are eliminated, and intermediate between values stay in thread-local registers. We used Nsight Compute to confirm that the arithmetic intensity increased by 10x (because values are loaded once and operated on 10 times), memory unit stalls dropped by 5x, and achieved FLOP/s increased by 3.5x. We also saw math unit stalls increase by 2x, but this is because we are doing a better job keeping the Float32 compute units busy.

E. Comparison with Jax CPU implementation

We also compared the throughput speed of our fastest implementation (unroll 10) using the XLA CPU backend on an AMD Ryzen 7 5800X 8-core CPU. We found that the CPU is faster than the GPU when there are 70 or less simulators running in parallel. However, because the GPU is capable of higher thread parallelism, it achieves higher throughput when running more than 70 parallel simulators.

F. Comparison with PyTorch and TorchScript implementations

Our PyTorch implementation of cartpole eagerly executes tiny kernels for individual operations, leading to dozens more kernels launched than our XLA baseline. As a result we measured a slowdown to 0.13x compared to our baseline. However, TorchScript includes a compiler that performs instruction fusion. So we implemented cartpole in TorchScript and verified that it emits a single CUDA kernel that is fully fused. We found a 1.97x speedup and believe this is on a similar order of magnitude of throughput as the fully fused Jax-XLA kernel. However, Jax is much more flexible because it can differentiate functions and can use more of python's native features like loops, without breaking compilation.

G. Comparison with CUDA implementation

An implementation of cartpole written in CUDA was contributed by James Gleeson. While we found that his implementation differs in the datatype that it used (his CUDA uses F16, F32, F64 and our Jax uses F32 datatype only), it is otherwise functionally equivalent to our Jax baseline. We find that the CUDA implementation is 2.7x faster than our best XLA implementation, and 28x faster than our naïve XLA implementation. We find that XLA introduces framework overhead and we used Nsight System to verify the speed differential is fully accounted for. Specifically, XLA launches extra 2 kernels (see Figure 9) per simulation loop iteration due to the way it implements lops. And these two kernels are launch overhead bound.
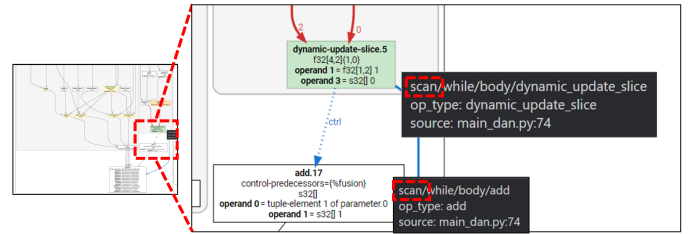

Fig. 9. The source of XLA's main slowdown as compared to CUDA are these two extraneous kernels which are involved in XLA's implementation of loops.

To provide more details, we found the runtime of the CUDA cartpole implementation to be 23% slower for a single CUDA kernel to run 5 simulation steps, as compared to the Jax with 5 loop unrolls. However, it's a very different story for 10,000 steps. In that case, the CUDA implementation still runs only 1 kernel, while our best XLA implementation runs 3 kernels for each loop iteration, and requires 1000 iterations when we unroll by 10 to reach 10,000 steps, which totals 3,000 kernel launches for our best XLA implementation. These additional kernels are small but together add up to 2.7x longer runtime than the CUDA implementation.

VI. Discussions

In this section, we discuss the existing limitations of XLA compiler and potential directions for the future research.

A. Limitations

Though XLA provides a simplest way for ML developers and researchers to optimize their codes, XLA's generated code is still sub-optimal compared to other heavily tuned ML compilers. Combining with our investigations shown in the previous sections, we give the following limitations/challenges for XLA.

1) The performance of the generated kernel heavily relies on the quality of frontend Python code (e.g., concatenate operation in our Cartpole environment). This is not a unique problem for XLA, other ML compilers also have similar problems. The fusion mechanism itself cannot address such frontend code quality issues, instead, it requires additional optimization passes that can diagnose inefficient frontend code and perform proper equivalent code transformation for better performance.

2) The custom CUDA kernel calls limit the further fusion opportunities for XLA-GPU. Compared to XLA-TPU, XLA-GPU still uses third-party cuDNN/cuBLAS primitives for compute-intensive DL operations, which results in separate fused kernels and additional layout conversion overhead. Further optimization is obtainable if XLA has its own efficient implementations of tunable DL operations.

3) Rule-based fusion in XLA is inflexible and conservative. XLA's fusion rule needs to consider various cases that could appear in the HLO graph, thus XLA has relatively conservative fusion strategies that are guaranteed to not hurt the performance of the code. For example, the inability of XLA's fusion at the concatenate operation in our initial Cartpole implementation can be treated as overkill. The task-specific autotuning methods [5], [9], [15] are able to address this problem and generate better fused kernels, but this can also result in much higher compilation overhead.

4) XLA relies on frontend code conversion and JIT compilation, which can introduce non-trivial runtime overhead for lightweight arithmetic computation with varied input shape. This is an inevitable tradeoff between flexibility and efficiency. For lightweight or varied programs, PyTorch-like eager execution may give better performance as compared to heavy JIT.

B. Future Work

In addition to potential optimizations mentioned above, there are other future works we can explore:

1) Fusing simulation with neural network inference and extending our characterization of kernel fusion. DL models will make the study of fusion much more complex, and it would be interesting to see if there could be new optimization opportunities for ML compilers, for example, multi-stream

parallelism for simulation and inference.

2) Auto-tuning the loop unrolling. We see a great performance gain from the loop unrolling. Due to the nature of iterative execution in ML tasks, automatic loop unrolling can be a good way to avoid frequent kernel stalls for repetitively executed kernels.

3) Fusion in DL training. Training support is one of the important advantages of XLA over other ML compilers. DL training requires the intermediate features for the backward gradient-based optimization, which limits the benefits of the operation fusion, and introduces other optimization options e.g., rematerialization. We have not done such characterization in this project, we will leave it as our future exploration.

VII. Conclusions

Throughout this paper we shed light on the concepts and inner workings of XLA's a poorly documented, but important optimization. Automatic kernel fusion is far from a solved problem. There are many fusion techniques (ie. instruction fusion, fusion merger, sibling fusion, and producer-consumer fusion), who's automatic application is hindered by practical considerations like code duplication and hardware limits (eg. register size), and engineering limitations such as arbitrary expensive ops (eg. log, power) and in-fusable custom kernels (eg. cuRAND, cuDNN). We fused kernels in three different ways (modifying XLA, optimizing our code, and loop unrolling). Our key takeaway is that we were able to utilize kernel fusion in XLA for a speedup up to 10.5x, but XLA's has framework overhead that makes a handwritten CUDA implementation even faster.

We hope that our insights into how XLA operates helps ML developers better understand the systems they use, and leads improvements in ML compilers, an open area of systems research.

References

[1] Introduction to torchscript. https://pytorch.org/tutorials/beginner/Intro_to_TorchScript_tutorial.html. Accessed: 2021-11-01.

[2] Nvidia tensorrt. https://developer.nvidia.com/tensorrt. Accessed: 2021-11-01.

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 578–594, 2018.

[6] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021.

[7] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[8] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems*, 3, 2021.

[9] Wookeun Jung, Thanh Tuan Dao, and Jaejin Lee. Deepcuts: a deep learning optimization framework for versatile gpu workloads. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 190–205, 2021.

[10] Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. Gshard: Scaling giant models with conditional computation and automatic sharding. *arXiv preprint arXiv:2006.16668*, 2020.

[11] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. *IEEE Transactions on Parallel and Distributed Systems*, 32(3):708–727, 2020.

[12] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 881–897, 2020.

[13] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[15] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, et al. A flexible approach to autotuning multi-pass machine learning compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–16. IEEE, 2021.

[16] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[17] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[20] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. {PET}: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 37–54, 2021.

[21] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models. *Proceedings of Machine Learning and Systems*, 3, 2021.

[22] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.