Effortless Machine Learning on TPUs with Julia

Keno Fischer
Julia Computing, Inc.
Cambridge, MA 02139
keno@juliacomputing.com

Elliot Saba
Julia Computing, Inc.
Seattle, WA
elliot.saba@juliacomputing.com

Abstract

Google's Cloud TPUs are a promising new hardware architecture for machine learning workloads. They have powered many of Google's milestone machine learning achievements in recent years. Google has now made TPUs available for general use on their cloud platform and as of very recently has opened them up further to allow use by non-TensorFlow frontends. We describe a method and implementation for offloading suitable sections of Julia programs to TPUs via this new API and the Google XLA compiler. Our method is able to completely fuse the forward pass of a VGG19 model expressed as a Julia program and into a single TPU exectuable to be offloaded to the device. Our method composes well with existing compiler-based automatic differentiation techniques on Julia code, and we are thus able to also automatically obtain the VGG19 backwards pass and similarly offload it to the TPU. Targeting TPUs using our compiler, we are able to evaluate the VGG19 forward pass on a batch of 100 images in 0.23s which compares favorably to the 52.4s required for the original model on the CPU. Our implementation is less than 1000 lines of Julia, with no TPU specfic changes made to the core Julia compiler or any other Julia packages.

1 Introduction

One of the fundamental changes that have enabled the steady progress of machine learning techniques over the past several years has been the availability of vast amounts of compute power train and optimize machine learning models. Many of the fundamental techniques are decades old, but only in recent years have advances in hardware manufacturing yielded the compute power to be interesting for real world problems. A significant chunk of this compute power has been available on Graphics Processing Units (GPUs) whose vector compute capability, while originally intended for graphics, have shown to deliver very good performance on the kind of matrix-heavy operations generally performed in machine learning models.

The real world success of these approaches and of GPUs in this space in particular has kicked off a flurry of activity among hardware designers with multiple independent efforts to create novel accelerators for machine learning workloads. However, while GPUs have a relatively long history of support in software, that support generally does not extend to new, non-GPU accelerators, and so finding the correct programming model remains a challenge.

In 2017, Google announced that they would make their proprietary Tensor Processing Unit (TPU) machine learning accelerator available to the public via their cloud offering. Originally, the use of TPUs was restricted to applications written using Google's TensorFlow machine learning framework. Fortunately, in September 2018, Google opened up access to TPUs via the IR of the lower level *XLA* ("Accelerated Linear Algebra") compiler. This IR is a general purpose IR and optimizing compiler for expressing arbitrary computations of linear algebra primitives and thus provides a good foundation for targeting TPUs by non-Tensorflow users as well as for non-machine learning workloads.

Preprint. Work in progress.

In this paper, we present initial work to compile general Julia code to TPU using this interface. This approach is in contrast to the approach taken by TensorFlow (1), which does not compile Python code proper, but rather uses Python to build a computational graph, which is then compiled. It is aesthetically similar to JAX (2), which does aim to offload computations written in python proper by tracing and offloading high-level array operations. Crucially, however, we do not rely on tracing, instead leveraging Julia's static analysis and compilation capabilities to compile the full program, including any control flow to the device. In particular, our approach allows users to take advantage of the full expressiveness of the Julia programming language in writing their models. This includes higher-level features such as multiple dispatch, higher order functions and existing libraries such as those for differential equation solvers (8) and generic linear algebra routines. Since it operates on pure Julia code, it is also compatible with the Zygote.jl (3) automatic differentiation tool, which performs automatic differentiation as a high-level compiler pass. Putting these together, we are able to compile full machine learning models written using the Flux machine learning framework, fusing the forward and backwards model passes as well as the training loop into a single executable that is offloaded to the TPU.

2 Background

2.1 The TPU Hardware

Google has developed three generations of TPU hardware. The second and third generations of TPUs (which are commerically available) have the ability to operate on IEEE 754 32-bit floating point numbers (*float32*), as well a custom non-IEEE 16-bit floating point format (*bfloat16*), that matches the bit width of IEEE 32-bit in the exponent, but trades that off for significantly reduced mantissa space. As with the previous generation, TPUv2 and v3 feature a systolic array matrix multiply unit, though in this case operating using bfloat16 multiplies and float32 accumulates. At full speed, each TPU core is capable of 22.5 TFLOP/s, as well as having 300GB/s bandwidth to an attached 8GB RAM of high bandwidth memory. Each TPU chip features two such cores for a total of operation speed of 45 TFLOP/s and total memory bandwidth of 600GB/s. Additionally, TPU chips are designed to be connected in a high-performance mesh network allowing scalability to larger models and data sets.

Unlike most accelerator hardware, Cloud TPUs are not made available to the user directly via PCIe, rather a webservice is exposed that accepts serialized XLA IR as well as exposing lower level memory management capabilities. Thus API, dubbed XRT, enables non-TensorFlow clients to generate XLA IR. XRT went live with the deployment of TensorFlow 1.11 to Cloud TPUs on September 27th 2018.

2.2 XLA

XLA ("Accelerated Linear Algebra") is a partially open source compiler project by Google. It features a rich input IR for specifying multilinear algebra computations and provides backend code generation capabilities for CPUs, GPUs and TPUs. XLA's Input IR (dubbed the HLO *High-Level Optimization* IR) operates on arbitrary dimensional arrays of basic data types (integers and floats of various bit widths, bfloat16s and complex numbers) or tuples thereof (but no arrays of tuples). HLO operations include basic arithmetic operations, special functions, generalized linear algebra operations, high level array operations, as well as primitives for distributed computation. XLA can perform semantic simplifications of input programs, as well as performing whole-program memory scheduling for efficient use and re-use of available memory (a very important consideration for large machine learning models).

2.3 The Julia Compiler

Julia is semantically a very dynamic language. However, in standard configuration, Julia's ultimate backend compiler is LLVM (6) which is a static compiler backend. The Julia compiler needs to bridge the semantic gap between the dynamic semantics of the language to the static semantics of the LLVM representation. The core idea of our work is to re-use most of Julia's compiler infrastructure but replace LLVM by XLA in the backend through a combination of careful language design and a sophisticated interprocedural type inference scheme. The Julia compiler is able to extract large static subregions from otherwise dynamic programs, allowing us to lower these static subregions to XLA,

run them, and then break back into the Julia compiler. In standard execution, these static subregions are chained together by the runtime to form the whole program.

3 Mapping XLA into Julia

In order to be able to replace LLVM by XLA, we need to represent XLA's operations as intrinsics to the Julia compiler. Suppose we have an example XLA operation 'Foo' taking one static operand (e.g. a single integer) and two dynamic operands. We enable translation to XLA by defining intrinsics within Julia that have equivalent semantics to any given HLO operation. Given this representation, it is easy to add appropriate methods to make Julia take advantage of them:

```
# Matrix-Matrix and Matrix-Vector product
function Base.:*(A::XRTMatrix, B::Union{XRTMatrix, XRTArray})
  # DimNums configures the contraction order of the HLO operation.
  # It must be a compile time constant in the XLA representation, but
  # we allow it to be dynamic as long as Julia can prove that it will
  # be constant before generating XLA IR.
  ddots = DimNums((1,), (0,), (), ())
  HloDot(ddots)(A, B)
end
Base.transpose(A::XRTArray) = HloTranspose((1,0))(A)
# Scalar addition
function Base.:+(A::XRTArray{T, (), 0}, B::XRTArray{T, (), 0}) where T
    # Many simple HLO operations can be represented by a single
    # GenericHloOp intrinsic that is parameterized on the GLO opcode
    # it generates and the resulting type/shape
    GenericHloOp{:add}(T, ())(A, B)
end
```

It is similarly convenient to implement Julia's higher level abstractions such as *mapreduce* or *broadcast*, allowing large amounts of Julia code to execute on the TPU without significant modification.

4 Results

The method described in this paper heavily relies on the Julia middle end compiler to determine sufficiently precise information (in particular the values of static operands and the shapes of dynamic operands), in sufficiently large subregions of the program to amortize any launch overhead. In this section, we demonstrate that the Julia compiler is indeed precise enough to make this method applicable to program of practical interest.

4.1 One simple example

Before moving on to more complicated examples, let us consider one simple examples that is a subproblem of the full VGG19 example:

```
softmax(xs) = exp.(xs) ./ sum(exp.(xs))
```

In figure 1, we show the final XLA IR generated by our compiler. There are several interesting aspects to this listing that are worthy of mention. Let us first consider how the `sum` invocation is represented in the XLA IR. The Julia standard library contains the following definition of sum:

```
add_sum(a::T, b::T) where {T} = T
sum(f, a) = mapreduce(f, add_sum, a)
sum(a) = sum(identity, a)
```

The XLA backend has one generic implementation of mapreduce (lowering it to the *Map* and *Reduce* HLO operations) and we can see this represented in the XLA output. The *c0m2* operation corresponds

```
@code_xla opt=true dense(x)
 c1 {
   c1p0 = f32[] parameter(0)
   ROOT c1e1 = f32[] exponential(c1p0)
 }
 c2 {
   ROOT c2p0 = f32[] parameter(0)
 }
 c3 {
   c3p0 = f32[] parameter(0)
   c3p1 = f32[] parameter(1)
   ROOT c3a2 = f32[] add(c3p0, c3p1)
 }
 c4 {
   c4p0 = f32[] parameter(0)
   c4p1 = f32[] parameter(1)
   c4t2 = (f32[], f32[]) tuple(c4p0, c4p1)
   c4gte3 = f32[] get-tuple-element(c4t2), index=0
   c4e5 = f32[] exponential(c4gte3)
   c4gte4 = f32[] get-tuple-element(c4t2), index=1
   ROOT c4d6 = f32[] divide(c4e5, c4gte4)
 }
 ENTRY softmax {
   c0p0 = f32[10]{0} parameter(0)
   c0m1 = f32[10]{0} map(c0p0), dimensions={0}, to_apply=c1
   c0m2 = f32[10]{0} map(c0p0), dimensions={0}, to_apply=c2
   c0c3 = f32[] constant(0)
   c0r4 = f32[] reduce(c0m2, c0c3), dimensions={0}, to_apply=c3
   c0b5 = f32[10]{0} broadcast(c0r4), dimensions={}
   ROOT c0m6 = f32[10]{0} map(c0p0, c0b5), dimensions={0}, to_apply=c4
 }
```

Listing 1: The final XLA IR for the softmax example. See section 4.1.

to the map of identity *identity* (represented as computation *c2*) over the array, while the reduction *c0r4* corresponds to the reduction using '+' (represented as computation *c3*). An additional interesting feature of the softmax IR is computation *c4*. It is generated from *syntactic broadcast fusion* (5), a Julia feature for more efficient broadcasts in the absence of an optimizing array compiler (and is thus likely not beneficial for the XLA backend, but not hurtful either).

4.2 VGG19 forward pass

Our first, more complex example is the full VGG19 forward pass. We use the implementation of VGG19 as found in the Metalhead package (7), which leverages the Flux (4) framework to translate the familiar machine learning layers (convolutional layer, dense layer) into linear algebra operations. However, importantly each layer in the Flux framework is just a regular function that in turn calls regular linear algebra operations. As such, machine learning models expressed in Flux, including VGG19, are just simply regular Julia functions and thus amenable to the methods described in this paper. Our compiler is able to fully infer, offload and fuse the entire forward pass of VGG19. After Julia-level optimizations, the final IR for the top level function contains 164 instructions (each an HloOp with properly inferred constant static parameters and properly shape inferred dynamic paramters. The total number HLO operands in the entry level computation is 166 (two extra for the parameter instructions which are implicit in the embedding) and 344 total over 29 computations[1]. The breakdown of instructions by instruction type is shown in table 1. Since we are able to offload the entire forward pass computation, the Julia is not involved at any step of the evaluation and can thus simultaneously perform other tasks (e.g. data preparation for the next batch). Additionally, the performance of the generated code is limited only by the quality of the code generated by

[1]However, many of these computation results are the same because of the regular structure of VGG19. For simplicity our compiler currently does not cache and re-use generated XLA IR, though that is a planned enhancement.

VGG19		Entry	Total
Forward	Unopt	183	361
	Opt	130	242
Backward	Unopt	577	2775
	Opt	362	925

Figure 1: Summary of XLA instruction generated by the Metalhead.jl VGG19 forward pass and backwards pass after compilation to XLA. Both unoptimized (after the Julia frontend) and optimized counts (after an XLA optimization pipeline similar to that used by the CPU backend, but without HLO fusion) are shown. For each, the count is further broken down into instructions in the entry (top-level) computation and instruction counts in all computations.

XLA, not by frontend considerations (we perform a performance evaluation in section 4.4). We validated correctness of the generated XLA code by evaluating the VGG19 model on images from the ImageNet validation set and validating that the obtained results match the results obtained from vanilla Metalhead (up to minor floating point rounding differences that generally don't affect the prediction).

4.3 VGG19 backward pass

To obtain the backwards pass, we make use of the Zygote.jl compiler-based AD framework (3). Zygote operates on Julia code and its output is again a Julia function (suitable for reintroduction into Zygote to obtain higher order derivatives, but also suitable for compilation to TPUs).

In particular, the example we are looking at is:

```
using Zygote
backwards(m::VGG19, x) = derivative(m -> sum(m(x)), m)
```

i.e. the derivative with respect to the model at the current value of the model and a particular training example (or a batch of training examples). We use *sum* as a simple stand in for the loss function. Fortuitously, but not entirely coincidentally, the type inference modifications we describe in section 3 also improve the precision of type inference to be able to infer through all of the VGG19 backwards pass. As for the forward pass, the total optimized and unoptimized instruction counts are shown in figure 1. The backwards pass generates significantly more XLA instructions than the forward pass. One of the biggest contributors to the instruction bloat is Zygote's mixed mode broadcast fusion, which computes both the forward pass and the backwards pass in one *map* kernel. Because XLA currently does not support multiple outputs from one map instruction, the function body gets duplicated across multiple map instructions, which XLA's DCE then needs to clean up. In general, our compilation process stresses XLA's handling of the *map* instruction, because of the prevalance of calls to Julia's *map* and *broadcast* functions in generic code. We are in the process of improving XLA's handling of *map* to inline mapped computations providing XLA backends with a form of IR more similar to that generated by other frontends.

4.4 Evaluation on TPUs

In this section, we present preliminary performance results of the code generated via the method in this paper. Note that because we were able to fully offload the function of interest, we expect future performance improvements to be the result of improvements to XLA's high level and backend optimizations, as opposed to modifications to the method in this paper. We note that the XLA developers have not yet had a chance to implement any improvements as a result of our work and we thus expect all XLA performance results to improve in the future. Additionally, while the Cloud TPU profiler does show does the operations of the model running, the output is incorrect. We have filed this bug as TensorFlow issue 22709 and Google has indicated that the issue is with the data transfer, not the actual computation on the TPU (as such a fix should not affect the performance results). With those qualifications in mind, we obtain the results in figure 2.

Batch Size	1	10	100
Flux CPU	0.79s	6.67s	52.4s
PyTorch CPU	1.16s	9.55s	93.0s
FluXLA CPU	12.06s	64.8s	> 600s
FluXLA TPU (total)	0.86s	0.74s	0.93s
FluXLA TPU (compute)	0.02s	0.04s	0.23s

Figure 2: Timings for the VGG19 forward pass for varying batch sizes. Flux CPU is Flux master/Julia master without the XLA compiler. PyTorch CPU is the equivalent model in pytorch on the same CPU. FluXLA CPU is our work against an xrt implementation running on the CPU, FluXLA TPU (total) is end-to-end time as reported by the client (including kernel launch overhead and data transfer back from Google Cloud over the internet - note that as a result of the additional network transfer this measurement had significant variability), FluXLA TPU (compute) is the total compute time on TPUs as reported by the cloud profiler (unlike the total time measurement, this measurement was very stable). All CPU measurements on *Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz* CPUs supporting AVX512. Up to 20 cores were available and CPU benchmarks were not restricted to a single core (though in practice not all CPU benchmarks actually used the available parallism). TPU benchmarks were restricted to a single TPU core. All Timings are minimum of 4 runs (except FluXLA CPU for a batch size of 100 which failed to finish a single run within 10 minutes).

5 Conclusion

In this paper, we discussed how to compile Julia code to XLA IR, thus enabling offload to TPU devices. The described implementation re-uses significant parts of the existing Julia compiler and is thus less than 1000 lines of code, but is nevertheless able to compile both the forward and the backward pass (and the fusion thereof, including the training loop) of models of practical interest such as VGG19 into a single XLA kernel. We have also demonstrated how Julia's multiple dispatch semantics aid in the specification of this transformation. This work suggests that it is possible to not only compile a number of ML models written in Julia to TPUs, but also more general non-ML Julia code (as long as such code is also dominated by linear algebra operations). We hope that this facility may hasten the exploration of non-ML problem areas for which TPUs may be useful.

Acknowledgements

Our work heavily leverages Julia type inference capabilities, which were recently significantly enhanced by Jarrett Revels, Jameson Nash and Jeff Bezanson in support of the Cassette.jl dynamic compiler framework (9). We are indebted to Mike Innes for his work on Flux.jl and Zygote.jl, without which we would not have been able to show the applicability of our method to a model of real world interest. Matt Bauman kindly provided guidance on properly implementing Julia's broadcast semantics against the XLA backend. More generally, we thank the Julia community for their attention to detail and the clarity of Julia's array abstraction that allowed us to achieve our results without requiring significant amounts of code. We gratefully acknowledge Zak Stone, Michael Isard, Mark Heffernan, James Bradbury, Roy Frostig, Eli Bendersky and Chris Leary of Google's TPU and XLA teams for their openness and willingness to answer our questions about TPUs, answer our bug reports and provide assistance in our quest to make this project a reality.

References

[1] 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 *OSDI* (2016), vol. 16, pp. 265–283.

[2] FROSTIG, R., JOHNSON, M. J., AND LEARY, C. Compiling machine learning programs via high-level tracing.

[3] INNES, M. Don't Unroll Adjoint: Differentiating SSA-Form Programs. *ArXiv e-prints* (Oct. 2018).

[4] INNES, M., AND CONTRIBUTORS. Flux.jl: The ml library that doesn't make you tensor, 2017.

[5] JOHNSON, S. G. More dots: Syntactic loop fusion in julia, 2017.

[6] LATTNER, C., AND ADVE, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* (2004), IEEE Computer Society, p. 75.

[7] MIKE INNES, A. P., AND CONTRIBUTORS. Metalhead.jl: Computer vision models for flux, 2018.

[8] RACKAUCKAS, C., AND NIE, Q. Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia. *Journal of Open Research Software 5*, 1 (2017).

[9] REVELS, J., AND CONTRIBUTORS. Cassette.jl: Overdub your julia code, 2018.