Static Automatic Batching in TensorFlow

Ashish Agarwal [1]

Abstract

Dynamic neural networks are becoming increasingly common, and yet it is hard to implement them efficiently. On-the-fly operation batching for such models is sub-optimal and suffers from run time overheads, while writing manually batched versions can be hard and error-prone. To address this, we extend TensorFlow with *pfor*, a parallel-for loop optimized using static loop vectorization. With *pfor*, users can express computation using nested loops and conditional constructs, but get performance resembling that of a manually batched version. Benchmarks demonstrate speedups of one to two orders of magnitude on a range of tasks, from Jacobian computation, to auto-batching Graph Neural Networks.

1. Introduction

Deep learning models are getting increasingly complicated in structure. New applications, like study of molecular structure, genetic data, relational databases, program synthesis, (Chen et al., 2018; Liang et al., 2018; Schlichtkrull et al., 2018) require dealing with inputs that have rich and dynamic structure. Handling such domains requires dynamic computation graphs, where the model structure is dependent on each input. This doesn't work well with frameworks that require creating a static graph structure a priori (Abadi et al., 2016; Jia et al., 2014). As a result, *define-by-run* paradigm has gained popularity (Tokui et al., 2015; Neubig et al., 2017a; Paszke et al., 2017), where the programming model is closer to the host programming language, and constructs like loops and conditionals can be used freely.

Another interesting trend is that systems for running deep learning are getting very powerful, with custom accelerator pods capable of running at petaflops. Leveraging so much compute can easily make host CPU a bottleneck. Running dynamic computation in the host language, especially interpreted languages like python can make this bottleneck worse.

Given the above two trends, frameworks are increasing turning to trace compilation (Agrawal et al., 2019; Moldovan et al., 2018), where the computation gets expressed dynamically, but a combination of static and just-in-time analysis is used to extract static computation graphs which can be dispatched to these accelerators. This approach promises good usability via programming abstractions that are close to the host programming language, as well as the performance advantages of static optimization together with quick dispatch of large chunks of computation to the accelerators.

However there is a caveat. While trace compilation can convert loops and conditionals in the host languages to the corresponding constructs compatible with the accelerators, the body of these loops are unlikely to get good compute utilization. This forces the programmers to write manually batched implementations, which ends up undoing some of the advantages of define-by-run, and forces them to deal with complicated padding and masking logic.

Frameworks like *DyNet* avoid this manual batching by having the runtime perform operator batching on the fly (Neubig et al., 2017b). However this creates runtime overheads, undoing some of the advantages of trace compilation.

Given this need to leverage fast accelerators in the context of dynamic computation and trace compilation, we propose using static analysis on trace compiled graphs to perform automatic batching. This approach gets rid of runtime overheads of dynamic batching. Users express their computation with nested host control flow constructs. Trace compilation then converts these to nested loop structure in some intermediate representation, and then static techniques auto-vectorize these loops.

We use this insight to implement *static automatic batching* in TensorFlow. Firstly, we add an SPMD (Single Program, Multiple Data) programming abstraction to TensorFlow Python frontend. With SPMD, multiple instances of a single program operate on different data. We expose it as parallel-for loop, which we call `pfor` (see §3.1). Here the different iterations of the loop can be viewed as different instances of the program. Secondly we implement an algorithm that takes SPMD loop definitions and rewrites

[1]Google Inc.. Correspondence to: Ashish Agarwal <agarwal@google.com>.

Proceedings of the 36th International Conference on Machine Learning, Long Beach, California, PMLR 97, 2019. Copyright 2019 by the author(s).

them in a loop-free manner. This is done by greedily auto-vectorizing each operation inside the loop body (see §4).

Using this construct of auto-vectorized parallel-for loops, we implement a range of solutions, from performing auto-batching on dynamic computation like Graph Neural Networks (see §5.2), to computing jacobians (see §3.3) and per-example gradients (see §3.4). Benchmarks show speedups of more than an order of magnitude compared to runtime batching, and an even larger speedup compared to iterative approaches (see §5).

2. Related Work

DyNet and TensorFlow Fold perform operation batching at run time. While they show significant improvements over unbatched execution, the approach has many challenges. Firstly, there are high overheads for creating per-example graphs at run time, and to identify and merge similar operations. Merging may also need memory copies for stacking the inputs and unstacking the outputs. Additionally, since control flow lives in python, branching based on values resulting from tensor computations can potentially block the execution, hurting the potential for batching across control flow. In contrast, our approach avoids run time graph construction or unnecessary memory copies, and is also able to batch across nested control flow constructs.

Cavs (Xu et al., 2018) specializes for the case where a statically defined *vertex* function is run on each node of a dynamically defined graph. Such an assumption allows Cavs to statically optimize the former, and reduces the overall dynamism and hence the run time overheads. They show good speedups on dynamic models like Tree LSTM. However their approach is tailored to specific kind of model structures, and still has some run time overheads. Our approach is both more general, and avoids run-time dynamic dispatch.

Parallel-for is a well known programming abstraction and is often implemented by dispatching multiple iterations in parallel to a pool of workers (Berke, 1988; Luszczek, 2009). In fact, TensorFlow's sequential `tf.while_loop` defaults to launching multiple iterations in parallel, making sure any data dependencies across iterations are respected. As our benchmarks show, auto-vectorized pfor is able to provide significant speedups over such parallelism.

Automatic vectorization has been around in the compiler community for a while (Nuzman et al., 2006; Trifunovic et al., 2009; Barik et al., 2010; Karrenberg & Hack, 2011). These approaches perform loop unrolling and tiling, increasing the instruction level parallelism, and then merge scalar instructions into vector variants. Our approach in contrast is able to completely get rid of parallel loops and is able to optimize across deeply nested loops over very large programs. These results are far beyond those of current low level auto-vectorization implementations, to the best of our knowledge.

ISPC (Pharr & Mark, 2012) proposed an SPMD programming model by extending the C language. Their optimization process resembles ours, but parallelization happens on top of a low level IR, and is limited based on the width of vector instructions. We merge across any number of iterations. Also operating at a higher level tensor representation allows us to use mathematical properties of the operations, and leverage loop invariance properties in interesting ways (see §4). We are able to optimize across opaque kernel implementations and by generating code at the same level of abstraction, our generated code is able to call highly optimized kernels.

Matchbox (Bradbury & Fu) is an ongoing effort to add SPMD execution to PyTorch. However it does so by rewriting the Python AST. Given Python's dynamic typing, this becomes challenging and needs heuristics, and often delayed conversion and run time overheads. In contrast, by separating out trace compilation from vectorization, our vectorization avoids such complexity and overheads, and can operate on statically typed IR, like TensorFlow's GraphDef.

3. Implementation

3.1. SPMD Parallel-For Loop

We provide a new Python function, `pfor`, with the following signature:

```
pfor(loop_body_fn, iters)
```

It represents a parallel-for loop with `iters` iterations, where iteration `i` calls `loop_body_fn(i)`. The output is a nested structure of tensors, with the same nesting structure as the output of `loop_body_fn(i)`, resulting from stacking the outputs of the different iterations. Note that in case the shape of an output varies across iterations, it is stacked into a `tf.RaggedTensor`. `loop_body_fn` is a Python function that takes a scalar integer tensor, representing the loop variable, as input, and returns a nested structure of tensors. `iters` is a scalar integer tensor representing the number of `pfor` iterations to run.

As a toy example `pfor(lambda i: i + 1, 2)` returns a tensor that evaluates to `[1, 2]`. Under the covers, the vectorization machinery described in §4 statically rewrites the above code to `tf.range(2) + 1`.

The execution of the `pfor` loop has SPMD semantics. In particular, the result of running this loop is the same as running each instruction in the loop body across all iterations, in lock step. This semantic is important when there are side-effecting operations inside the loop body. This is not common though, and the code is equivalent to sequential execution for a large class of useful applications we tried.

We also provide a high level primitives on top of `pfor`. `vectorized_map(fn, elems)` maps the function `fn` across all rows of `elems`. It is similar to `tf.map_fn` except that it has SPMD semantics and generally runs much faster due to vectorization optimization.

Example 1 shows an example of computation over variable length input, stacked together into `inp` by padding along dimension `1`. Computation involves running `conv2d` and `dense` layers for each example in this batch. A manually batched version is also provided, and requires masking of intermediate outputs. The auto-batched version is much simpler and matches the speed of manual batching quite well.

Example 1 Auto-batched vs manually batched code

```
# inp: Padded input with shape [B, L, D].
# inp_len: length of inp dimension 1. Shape: [B].

def unbatched_model(x, x_len):
 x = x[:x_len, :] # Unpads the input
 x = tf.reshape(x, [1, -1, 1, D])
 x = tf.layers.conv2d(x, D, (3, 1), padding="same")
 x = tf.reshape(x, [-1, D])
 return tf.layers.dense(x, D)

auto_batched_output = vectorized_map(
  unbatched_model, (inp, inp_len))

def batched_model():
  indices = tf.reshape(tf.range(L), [1, L])
  bool_mask = indices < tf.reshape(inp_len, [B, 1])
  mask = tf.cast(bool_mask, tf.float32)
  mask = tf.reshape(mask, [B, L, 1])
  output = inp * mask
  output = tf.reshape(output, [B, L, 1, D])
  output = tf.layers.conv2d(output, D, (3, 1),
    padding="same")
  mask = tf.reshape(mask, [B, L, 1, 1])
  output *= mask
  output = tf.reshape(output, [B, L, D])
  indices = tf.squeeze(tf.where(
    tf.reshape(bool_mask, [-1])), axis=1)
  output = tf.gather(
    tf.reshape(output, [-1, D]), indices)
  output = tf.layers.dense(output, D)
  indices = tf.reshape(indices, [-1, 1])
  output = tf.scatter_nd(indices, output,
    [B * L, D])
  return tf.reshape(output, [B, L, D])

manually_batched_output = batched_model()
```

3.2. Graph Auto-Vectorization

A key contribution of this paper is an implementation of auto-vectorization of the SPMD loop introduced in §3.1. This vectorization happens on top of high level TensorFlow IR (aka GraphDef). See §2 for comparison with prior work, and §4 for how the algorithm works.

The vectorization routine takes a GraphDef representing the body of the loop function, and a scalar Tensor representing the number of iterations, and outputs a new set of GraphDef nodes that implement functionality equivalent to running `pfor` with those arguments. We had the option of plugging this as an optimizing rewrite in the TensorFlow C++ runtime. However, as a first version, we chose to keep this in the Python frontend and to invoke it during graph construction. This allowed us to build this independent of the runtime internals and makes the conversion routine more accessible and extensible for the user.

A call to `pfor` first creates a `tf.placeholder` representing the iteration variable `i`, makes a single call to the `loop_body_fn` to create a graph, and then calls the vectorization routine. It gets back tensors that represent the vectorized version of the outputs of `loop_body_fn` which is returned from the call to `pfor`.

When TensorFlow's *Eager Execution* is enabled, calls to `pfor` return Tensors with concrete values. Internally, the implementation switches to *graph mode* using a `tf.function` wrapper, and then calls the returned vectorized function. Automatic batching together with *Autograph* (Moldovan et al., 2018) compilation allows writing intuitive and pythonic code with the performance of manually batched statically compiled graphs.

3.3. Jacobians and Higher order Gradients

Jacobian is defined as the matrix of all first order partial derivatives of a vector valued function. TensorFlow's `tf.gradients` function computes vector-jacobian product and prior to our work did not have an efficient implementation for computing the full jacobian matrix. Doing so required a sequential loop, with iteration i computing gradients of the i^{th} scalar value in the output w.r.t. the vector inputs, and stacking these gradients into a matrix. Replacing this sequential loop with a `pfor` loop enables our vectorization process to provide a much faster version.

Using this approach, we have added the following functions to TensorFlow: `jacobian(output, inp)` and `batch_jacobian(output, inp)`. The former computes the jacobian of `output` with respect to `inp`, where `inp` can be a nested structure of Tensors. The latter computes the jacobian of each row of `output` with respect to each row of `inp`. The second version is useful when each output of a batch is dependent only on the corresponding input in the batch. It leverages this independence between batch elements to avoid unnecessary computation of zero values, thus reducing compute and memory requirements. These primitives can be further composed to produce higher order gradients, like hessians.

Our new API has enabled research including, (Golub & Sussillo, 2018) for analyzing fixed points of RNNs, (Lee et al.,

2018) for studying eigen-values of jacobians of machine translation decoders, and (Pfau et al., 2018) for computing eigen-functions of linear operators via stochastic optimization.

3.4. Per-Example Gradients

A common approach in stochastic optimization is to compute the sum of losses over a batch of elements, and then compute its gradients w.r.t the parameters. However some optimization strategies require computing the gradient of each scalar loss w.r.t the parameters and then combining these per-example gradients in sophisticated ways to compute the parameter updates, e.g. (Alain et al., 2015).

Computing per-example gradients requires running a sequential loop where each iteration runs the forward and backward pass for a single batch element. This is highly inefficient. Alternatively, one could perform manual surgery on the generated graphs to compute per-example gradients (Goodfellow, 2015).

Expressing this computation is straightforward with `vectorized_map`, by having the map function be the forward and backward computation, and mapping it over the batch of inputs. §5.4 shares some benchmark numbers to show that this provides significant speedups over the iterative approach.

4. Graph Auto Vectorization Algorithm

Here we provide more details about auto-vectorization of `pfor` loops mentioned in §3.2. We first explain the motivation and high level approach, followed by details on how vectorization leverages loop invariance. Next we talk about handling conditionals, loops and stateful operations. Finally we discuss managing memory overheads of vectorized code. For more formal details and correctness of the conversion process, we refer the user to (Agarwal & Ganichev, 2019).

4.1. Motivation and Challenges

Conversion at Tensor level abstraction allows leveraging mathematical properties of the operations instead of inferring them from deeply nested loop nests. Also this conversion works even when the kernel implementations are opaque and proprietary, common with GPU kernels. Output of the vectorization process is itself expressed using TensorFlow operations, which in turn uses highly optimized kernels. For example, vectorization of `tf.mamtul` can generate `tf.batch_matmul` which would leverage advanced optimization, like loop tiling, cache and memory bandwidth utilization, done in the implementation of that kernel. Vectorizing individual low level instructions inside `tf.matmul` is likely to produce inefficient code.

Vectorizing at the level of TensorFlow kernels requires dealing with a much larger and complicated instruction set. These kernels can have complicated semantics like broadcasting, multiple input attributes, and could deal with variable number of inputs with different ranks and shapes. Vectorization needs to deal with this, both for correctness, as well as for optimization specific to those diverse semantics. In addition, it also tracks loop invariance of tensors and generates optimized code based on which combination of inputs is loop invariant (see §4.3). Dealing with mutable state, TensorFlow's data structures, like `TensorArray` and `Stack`, as well as nested control flow constructs, like `tf.cond` and `tf.while_loop` further complicates this conversion.

4.2. Conversion Routine

The algorithm is given a scalar `tf.Placeholder` Tensor that corresponds to the loop iteration counter, i, a list of Tensors corresponding to the outputs of `loop_body_fn(i)`, and a scalar Tensor corresponding to the number of iterations `iters`. The algorithm traverses nodes from i to the outputs in topological order. For each node, η, visited during the traversal, it calls a converter function specific to the kernel of η. The semantics of this converter is to add new nodes to the graph that efficiently implement the functionality of running η `iters` times and stacking the outputs.

Lets revisit `pfor(lambda i: i + 1, 2)` as an example. Here we first create a placeholder node, η_i, then call the body function on it. This creates some extra nodes: η_{const} with value 1 and η_{add} with inputs η_i and η_{const}. Additionally, η_{iters} node with value 2 is created for the number of iterations. The auto vectorization routine is then invoked. It first visits node η_i and adds a new node $\hat{\eta}_i$ which computes $tf.range(\eta_{iters})$. η_{const} is visited and left as is. η_{add} is visited which creates a new node $\hat{\eta}_{add}$ that performs $\hat{\eta}_i + \eta_{const}$. This node is then returned as the output of the conversion process. This node represents the expression `tf.range(2) + 1`.

4.3. Loop Invariance

In the example in §4.2, note that η_{const} was left as is. This is because it was loop invariant. i.e. the output of the node did not depend on the iteration variable i. In general, the body of the `pfor` can reference tensors from outside the loop body, create tensors using `tf.const` or `tf.random_uniform`, or create operations using inputs that doesn't recursively depend on `i`.

We track this dependence on the loop iteration counter `i`, and leverage it inside the converters. For example, when vectorizing `tf.matmul`, if both inputs to `tf.matmul` depend on i, we create a `tf.batch_matmul`, while if only one of them depends on i, we perform some reshapes, and possibly transposes, and call `tf.matmul`. Note that this rule is based on the mathematical properties of the operations.

In general, each registered converter considers the cross product of loop invariance of all the inputs to the operation to generate vectorized code. If all inputs are loop invariant, and the kernel is stateless, then we reuse the output directly.

4.4. Handling Conditionals

TensorFlow conditionals, `tf.cond`, use special nodes like *Switch* and *Merge* to implement the functionality. When traversing the graph, we first convert such forms into equivalent functional forms by separating out the sub-graphs corresponding to the *condition*, and the *then* and *else* blocks. The *condition* is first vectorized and the output vector represents which iterations would go into the *then* and *else* blocks. All external references from inside these blocks are then partitioned to only contain values for those indices. The *then* and *else* blocks are then recursively converted, and the outputs are finally assembled in the correct order. See (Agarwal & Ganichev, 2019, §3.6) for more details on converting conditionals.

4.5. Handling Nested Loops

Now let us consider the case of loops nested inside the `pfor`. If that loop is itself a `pfor` loop, then the compiler is invoked to convert it first. In case of sequential `tf.while`, corresponding converters need to be invoked. If these loops are deeply nested, each converter will in turn invoke compilation recursively which will lead to converting these loops inside out.

Similar to the conditional case, we first extract a functional form consisting of sub-graphs that correspond to the *condition* and the *loop body*. The generated code consists of a `tf.while_loop` which, in each iteration, keeps track of the indices, I, of all `pfor` loops that are still active, and runs the condition and body on only those indices. Similar to the conditional case, this involves subsetting all the vectorized inputs of those blocks to the set I, and having the recursive compilation be aware of the list and count of active iterations. Intermediate outputs for all the `pfor` indices that become inactive are collected in a `TensorArray` indexed based on the complement of I. When the loop is done, the entries in the `TensorArray` are stitched together and returned.

Note that a simpler and more optimized implementation is done for the case where the while loop condition is loop invariant. See (Agarwal & Ganichev, 2019, §3.7) for more details on converting nested loops.

4.6. Handling Stateful Operations

Semantics of stateful operations inside parallel-for loop is dictated by SPMD semantics, and works given that the conversion happens in the topological ordering of the nodes. It is possible for such programs to be invalid, e.g. if multiple iterations try setting the same variable to different values. Our conversion process detects potential collisions pessimistically and raises an error.

First we consider operations that are idempotent. Examples include creating and reading a Variable. These can be run once and outputs can be marked as loop-invariant. Next we consider operations that are commutative and associative. Examples include subtracting or adding a delta into a Variable. This can be done by first reducing all the updates and then applying the reduced value to the mutable state. Note that addition to Variable can be used to implement sum reductions across all parallel iterations, which in turn can be used to implement operations like batch normalization.

See (Agarwal & Ganichev, 2019, §3.8) for more details on converting stateful operations. Vectorization is similarly implemented for operations on data structures like Stacks and TensorArrays. Optimizing these, and reasoning about loop invariance of such operations in deeply nested contexts, raises many challenges, beyond the scope of this paper.

4.7. Memory Usage

Given that all iterations of `pfor` are effectively run in parallel, this can increase the memory usage. To control that, we provide a knob `parallel_iterations`, to the `pfor` function which controls how many iterations are compiled together. Setting this parameter causes the loop to get tiled into two nested loops, an outer sequential `tf.while_loop`, and and internal vectorized `pfor` loop. This allows trading off between memory usage and compute speedup.

In practice, we have been able to run large experiments, without running into memory problems. In cases of automatic batching, `pfor` generates batched code similar to existing manually batched ones, and hence matches its memory usage as well. For jacobians and per-example gradients, we have experimented with very large networks, with reasonable batch sizes and significant speedups.

5. Benchmarks

5.1. Setup

Experiments were run on a 6 core Intel Xeon E5-1650 3.60GHz CPU with 64GB of RAM and a NVIDIA Maxwell Titan X GPU.

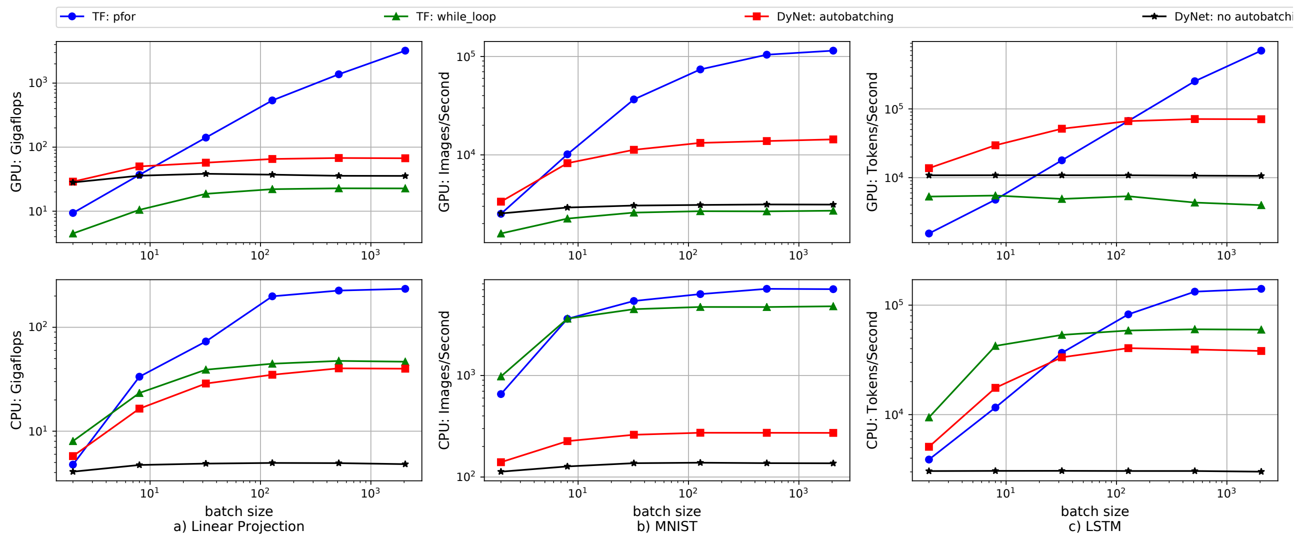

Figure 1. Throughput vs batch size for auto-batching forward pass of three models, one per column. Top row corresponds to GPU, while bottom one is CPU. Different curves are for runs with and without pfor auto-vectorization, and with and without DyNet auto-vectorization.

5.2. Models

Linear Projection is a simple setup which applies linear projection on input data. Inputs are randomly generated float vectors with shape $[768]$. Projection matrix is a constant $[768, 768]$ matrix of floats. Throughput is reported in GFLOPS against batch size.

To test convolution based networks, we use two different architectures. Models using batch norm have been omitted from these benchmarks since our implementation of vectorizing "fused" batch norm kernels in TensorFlow is still experimental.

ConvMnist setup uses a convolutional architecture as described in (tensorflow, 2016). It is a stack of two conv-relu-maxpool blocks followed by a linear-relu-dropout-linear block. Inputs are batches of $[28, 28]$ images and output has shape $[10]$.

VGG16 is as described in (Simonyan & Zisserman, 2014). We picked this model since it supports different input image sizes as well as different number of output classes. Input sizes used are $[48, 48]$ and $[224, 224]$.

For testing dynamic networks on variable length sequences, we use multiple different kinds of models as described below.

LSTM experiments use a single-layer unidirectional RNN based on the LSTM cell described in (Hochreiter & Schmidhuber, 1997). Input is batch of variable length sequences, with each element being a 128 dimensional vector. LSTM state size is 256, except if mentioned otherwise (e.g. in §5.5).

TreeLSTM uses architecture as described in (Tai et al., 2015). Inputs are batches of randomly structured binary trees. To construct each tree, we sample the number of leaves randomly uniformly between 2 and some maximum number (15 or 100), and assign a random embedding to it. A random tree structure is imposed on these leaves as a random bracketing of the list of these leaves. The model works by first embedding the tokens at the leaves. Computation proceeds iteratively from the leaves to the root of each tree. Each node's computation resembles LSTM cell computation, extended to handle recurrent state coming from multiple children. Embedding size and LSTM cell state size are both 128.

GatedGraphNN uses the *Gated Graph Neural Network* architecture as described in (Li et al., 2015). Inputs are batches of randomly structured directed graphs. Each graph can have some maximum number of nodes (20 or 100), and each node has up to some max number of neighbors (respectively 5 or 10). Nodes are embedded into 64 dimensional space. In each step, for each node, its neighbor's embeddings are propagated through an edge specific transform, and then combined using a GRU based network to compute the new embedding for that node.

5.3. Automatic Batching of Forward Pass

We compare the performance of the forward pass of models implemented by looping over the batch elements using `pfor`, or using sequential `tf.while_loop`. Some of these models are also compared against DyNet, with and without auto-batching enabled. We vary the batch size and report throughput (measured as GFLOPS or images/token-

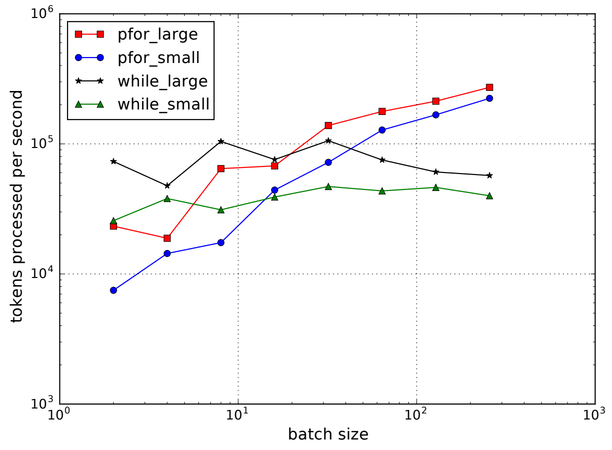

Figure 2. TreeLSTM inference throughput, in leaf tokens / sec vs batch size. Different plots are for runs on CPU, with and without automatic batching, and with small vs large trees.

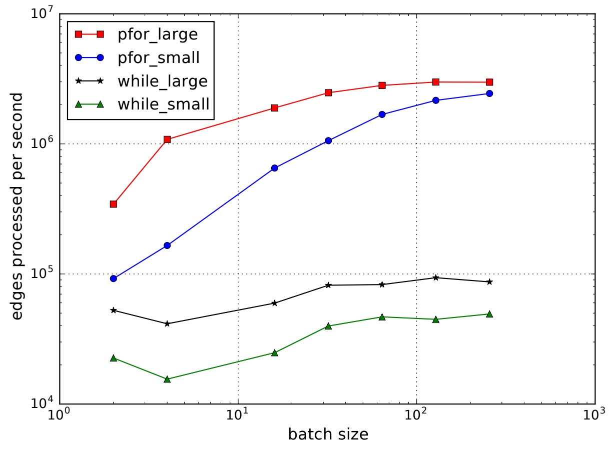

Figure 3. GatedGraphNN inference throughput, in edges / sec vs batch size. Different plots are for runs on GPU, with and without automatic batching, and with small vs large graphs.

s/edges processed per second) against the batch size, in Figures 1, 2 and 3.

5.3.1. Linear Projection

Linear Projection setup is as explained in §5.2. This toy setup allows us to closely study the achieved vs expected FLOPS and also to examine the generated code. Figure 1a shows the throughput as GFLOPS attained vs the batch size.

For this setup, `pfor` is able to rewrite the computation as a matrix multiplication with inputs of size $[batch\_size, 768]$ and $[768, 768]$, thus approaching peak hardware performance as batch size increases. Iterative approaches on the other hand perform a sequence of vector-matrix multiplications and are much slower, by 1 to 1.5 orders of magnitude.

Note that TensorFlow's `tf.while_loop` launches up to 10 iterations in parallel. This parallelism allows good scaling on CPU since different iterations are able to use different CPU threads, and hence use multiple cores. For GPU though, given a stream based execution, kernel execution gets serialized and hence performance of the sequential loop scales poorly.

DyNet performs well at small batch sizes, especially on GPU. In this regime, the computation time is dominated by fixed overheads, which appear lower for DyNet. However it does not scale as well at higher batch sizes likely due to run time overheads of performing batching. Without auto-batching enabled, execution fails to effectively utilize multiple CPU cores.

5.3.2. ConvMNIST

Figure 1b reports the number of images processed per second by the ConvMNIST model described in §5.2.

It turns out that `pfor` is able to match the performance of a manually batched version. Manually examining the generated graph reveals that the generated model is similar to the manual version as well.

Overall trends on this setup are comparable to those of the linear projection benchmark in §5.3.1. One noticeable difference is that on CPU, performance of `tf.while_loop` scales better than that of linear projection benchmark. This is likely because this model puts less pressure on memory bandwidth, given the convolutional structure. `pfor` still outperforms the other three for moderate and large batch sizes.

5.3.3. LSTM

LSTM setup is as described in §5.2. Input sequence lengths for LSTM are sampled uniformly at random between 1 and 100, inclusively.

Figure 1c reports tokens per second processed vs batch size. Here `pfor` needs relatively larger batch sizes to outperform other implementations, and the margins are narrower. This is likely caused by the following two factors. Firstly, our implementation of vectorizing `tf.while_loop` still has overheads, especially related to host to device copies for the GPU case, which we are working on optimizing. For the CPU case, `pfor` performance is close to the manually batched version. Secondly, given that sequence lengths are randomly chosen, iterations of the sequential loop generated by `pfor` progressively operate on smaller batches (i.e. input sequences that are not done), thus lowering the effective batch size, and hence the speedup.

5.3.4. Tree LSTM

Tree LSTM setup is as described in §5.2. Manually batching such computation is hard given that the structure of the trees varies across the inputs. Our implementation has two nested

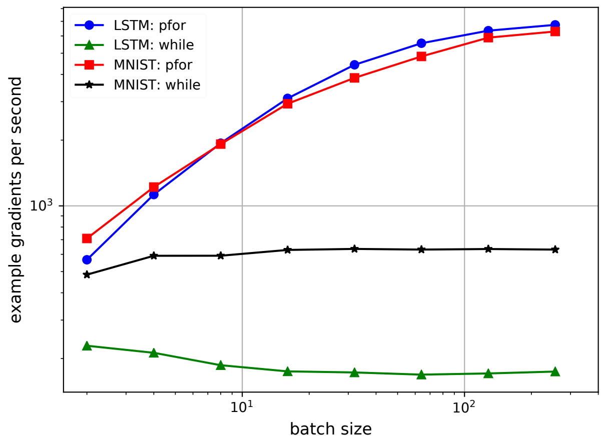

Figure 4. Throughput of per-example gradient computation on ConvMNIST and LSTM with and without automatic batching, on GPU. The x-axis is the number of examples in the batch. The y-axis is the number of per-example gradients that can be computed.

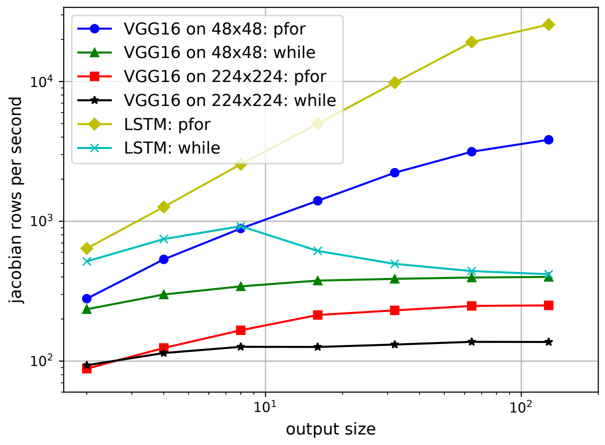

Figure 5. Throughput of computing jacobians (output with respect to input) for VGG16 and LSTM models as output size is varied, with and without auto-batching, on GPU. VGG16 uses two different input image sizes.

loops: a parallel loop over the batch elements, and an inner sequential loop over the depth of each tree, where the former is optimized using `pfor`.

Figure 2 shows the results of running inference on this model with and without auto-batching. We tested this on small vs large trees, respectively with maximum number of leaves per tree of 15 and 100. We vary batch size and report throughput as total leaf tokens processed per second. Large batch sizes and large trees seem to have larger speedups with `pfor`.

5.3.5. Gated Graph Neural Networks

Setup for gated graph neural networks is as described in §5.2. These are implemented by specifying a graph vertex function which is applied in parallel to all nodes in the graph using a `pfor` loop. Figure 3 shows the results of running inference on this model, with and without automatic batching, and with running on small and large graph sizes. We vary the batch size and report throughput as the total number of edges in the graphs processed per second. Automatic batching presents significant speedups.

5.4. Per-Example Gradients

For these benchmarks, we compute per-example gradients as explained in §3.4, for the ConvMNIST and LSTM models (see §5.2). We report example gradients processed per second as we vary batch size.

Figure 4 shows that `tf.while_loop` based implementation achieves low constant throughput for both models while `pfor` based implementation is able to scale much better. At the highest batch size of 256 for the LSTM model, `pfor` outperforms `tf.while_loop` by a factor of 38.

5.5. Jacobians

Here we compare our jacobian API (see §3.3) against an iterative approach. Experiments are run on GPU using *VGG16* and *LSTM* models (see §5.2). However for *LSTM*, we statically unrolled the loop for 10 steps since TensorFlow is currently unable to compute Jacobian of `tf.while_loop` given its Stack based gradient implementation.

Since the iteration is over the output elements, for these experiments, we vary the output size and measure throughput as rows of jacobian processed per second, reported in Figure 5. This metric normalizes the compute done for a given task as we vary the output size. We observe that the iterative implementation scales poorly with output sizes. `pfor` runs faster by up to 60 times at high output sizes.

6. Summary

We have extended TensorFlow to provide a parallel-for loop with SPMD semantics. In addition we have implemented a static optimization algorithm to auto-vectorize such loops. This is novel compared to existing run time operation batching as well as static instruction level auto vectorization approaches. Higher level constructs, like `vectorized_map` and `jacobian`, further enrich TensorFlow's capability. Automatic batching provides a big usability jump for dynamic computation graphs without sacrificing performance. GPU benchmarks show speedups of up to two orders of magnitude compared to TensorFlow's sequential loop and an order of magnitude compared to DyNet with runtime batching. Such speedups have already enabled new research that was not feasible earlier, e.g. (Pfau et al., 2018; Lee et al., 2018; Golub & Sussillo, 2018).

Acknowledgements

I would like to thank Igor Ganichev for insightful feedback and detailed reviews for the design and implementation of the auto-vectorization framework. David Sussillo, Martin Abadi and Derek Murray provided useful feedback on the manuscript. Various members of the Google Brain and DeepMind teams helped by being early adopters and providing valuable feedback.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I. J., Harp, A., Irving, G., Isard, M., Jia, Y., Józefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D. G., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P. A., Vanhoucke, V., Vasudevan, V., Viégas, F. B., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016. URL `http://arxiv.org/abs/1603.04467`.

Agarwal, A. and Ganichev, I. Auto-vectorizing tensorflow graphs: Jacobians, auto-batching and beyond. *CoRR*, abs/1903.04243, 2019. URL `http://arxiv.org/abs/1903.04243`.

Agrawal, A., Modi, A. N., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., and Cai, S. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. In *Proceedings of the 2nd SysML Conference*, 2019. URL `https://www.sysml.cc/doc/2019/88.pdf`.

Alain, G., Lamb, A., Sankar, C., Courville, A., and Bengio, Y. Variance Reduction in SGD by Distributed Importance Sampling. *ArXiv e-prints*, November 2015.

Barik, R., Zhao, J., and Sarkar, V. Efficient selection of vector instructions using dynamic programming. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pp. 201–212, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4299-7. doi: 10.1109/MICRO.2010.38. URL `https://doi.org/10.1109/MICRO.2010.38`.

Berke, W. *ParFOR: A Structured Environment for Parallel FORTRAN*. 1988.

Bradbury, J. and Fu, C. Automatic batching as a compiler pass in pytorch.

Chen, H., Engkvist, O., Wang, Y., Olivecrona, M., and Blaschke, T. The rise of deep learning in drug discovery. *Drug discovery today*, 2018.

Golub, M. and Sussillo, D. FixedPointFinder: A tensorflow toolbox for identifying and characterizing fixed points in recurrent neural networks. *Journal of Open Source Software*, 3(31):1003, 2018. doi: 10.21105/joss.01003.

Goodfellow, I. Efficient Per-Example Gradient Computations. *ArXiv e-prints*, October 2015.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL `http://dx.doi.org/10.1162/neco.1997.9.8.1735`.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R. B., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093, 2014. URL `http://arxiv.org/abs/1408.5093`.

Karrenberg, R. and Hack, S. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pp. 141–150, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL `http://dl.acm.org/citation.cfm?id=2190025.2190061`.

Lee, K., Firat, O., Agarwal, A., Fannjiang, C., and Sussillo, D. Hallucinations in neural machine translation. 2018.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. S. Gated graph sequence neural networks. *CoRR*, abs/1511.05493, 2015. URL `http://arxiv.org/abs/1511.05493`.

Liang, C., Norouzi, M., Berant, J., Le, Q. V., and Lao, N. Memory augmented policy optimization for program synthesis and semantic parsing. In *Advances in Neural Information Processing Systems*, pp. 10015–10027, 2018.

Luszczek, P. Parallel programming in matlab. *The International Journal of High Performance Computing Applications*, 23(3):277–283, 2009.

Moldovan, D., Decker, J. M., Wang, F., Johnson, A. A., Lee, B. K., Nado, Z., Sculley, D., Rompf, T., and Wiltschko, A. B. Autograph: Imperative-style coding with graph-based performance. *CoRR*, abs/1810.08061, 2018. URL `http://arxiv.org/abs/1810.08061`.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang, D., Clothiaux, D., Cohn, T., et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017a.

Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 3971–3981. Curran Associates, Inc., 2017b.

Nuzman, D., Rosen, I., and Zaks, A. Auto-vectorization of interleaved data for simd. *SIGPLAN Not.*, 41(6):132–143, June 2006. ISSN 0362-1340. doi: 10.1145/1133255.1133997. URL http://doi.acm.org/10.1145/1133255.1133997.

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.

Pfau, D., Petersen, S., Agarwal, A., Barrett, D., and Stachenfeld, K. Spectral Inference Networks: Unifying Spectral Methods With Deep Learning. *ArXiv e-prints*, June 2018.

Pharr, M. and Mark, W. R. ispc: A spmd compiler for high-performance cpu programming. IEEE, 2012. ISBN 978-1-4673-2633-9. doi: 10.1109/InPar.2012.6339601. URL https://ieeexplore.ieee.org/document/6339601.

Schlichtkrull, M., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014. URL http://arxiv.org/abs/1409.1556.

Tai, K. S., Socher, R., and Manning, C. D. Improved semantic representations from tree-structured long short-term memory networks. *CoRR*, abs/1503.00075, 2015. URL http://arxiv.org/abs/1503.00075.

tensorflow. models/tutorials/image/mnist/convolutional.py. https://github.com/tensorflow/models/blob/master/tutorials/image/mnist/convolutional.py, 2016.

Tokui, S., Oono, K., Hido, S., and Clayton, J. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.

Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., and Rosen, I. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pp. 327–337, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3771-9. doi: 10.1109/PACT.2009.18. URL https://doi.org/10.1109/PACT.2009.18.

Xu, S., Zhang, H., Neubig, G., Dai, W., Kim, J. K., Deng, Z., Ho, Q., Yang, G., and Xing, E. P. Cavs: An efficient runtime system for dynamic neural networks. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 937–950, 2018.