Memory Safe Computations with XLA Compiler

Anonymous Author(s)
Affiliation
Address
email

Abstract

Software packages like TensorFlow and Pytorch are designed to support linear algebra operations, and their speed and usability determine their success. However, by prioritising speed, they often neglect memory requirements. As a consequence, the implementations of memory-intensive algorithms that are convenient in terms of software design can often not be run for large problems due to memory overflows. Memory-efficient solutions require complex programming approaches with significant logic outside the computational framework. This impairs the adoption and use of such algorithms. To address this, we developed an XLA compiler extension that adjusts the computational data-flow representation of an algorithm according to a user-specified memory limit. We show that k-nearest neighbour and sparse Gaussian process regression methods can be run at a much larger scale on a single device, where standard implementations would have failed. Our approach leads to better use of hardware resources. We believe that further focus on removing memory constraints at a compiler level will widen the range of machine learning methods that can be developed in the future.

1 Introduction

Progress in science is inextricably linked with advances in scientific computing, in terms of both software and hardware. This is particularly noticeable in machine learning through the huge impact of numerical software packages supporting automatic differentiation (Baydin et al., 2018). Packages such as TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), or JAX (Bradbury et al., 2018) greatly accelerated **1)** the implementation of gradient-based optimisation procedures by eliminating error-prone manual differentiation, and **2)** the execution of code by leveraging modern and heterogeneous hardware (e.g. GPU, TPU or IPU). A large portion of this impact is attributable to the accessible and user-friendly form that these features were delivered in. This contributed to the growth in the machine learning community, in terms of methodological researchers, as well as the wider scientific audience and practitioners.

The aforementioned software frameworks work by chaining together efficient implementations of mathematical operations (known as *kernels*). By providing implementations that are tailored to various types of hardware, a speed-optimised implementation can be obtained. While speed is certainly important to pursue, many algorithms face a different challenge: hardware memory constraints. Often, these have a larger impact, as memory constraint violations can lead to the execution terminating before an answer is obtained. This make-or-break property is particularly noticeable on GPUs, where allocating more memory than is physically available leads to an immediate termination of execution, and larger amounts of physical memory comes at a significant cost.

Now that numerical computation frameworks are widely used, they strongly influence what machine learning algorithms are adopted. This happens through hard limitations, as well as usability considerations through what is easily implementable. Currently, the emphasis on optimising runtime causes many algorithms to be severely memory limited, or too cumbersome implement. This is particularly

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

noticeable in methods that rely heavily on matrix and linear algebra computations, e.g. kernel methods (e.g. Titsias, 2009) or nearest neighbour methods for geometric deep learning (Bronstein et al., 2017).

In this work, we aim to remove these limitations, by developing a tool that optimises code to be more memory efficient, with a particular focus on linear algebra operations. This optimisation is transparent to the user, and therefore allows many algorithms to be run at scales that were previously impossible, while leaving implementations as simple as before. This allows a wider range of algorithms to take advantage of "the bitter lesson"—*"General methods that leverage computation are ultimately the most effective"* (Sutton, 2019)—while making them more accessible to the wider community, as computational frameworks have sought to do all along.

Our method is implemented as an extension to the XLA compiler (Leary and Wang, 2017), which we chose due to its wide use and support for optimising computations specified in TensorFlow and JAX. We demonstrate the benefits of our method by scaling algorithms where simple implementations do not scale due to memory bottlenecks, such as k-nearest-neighbours, and Sparse Gaussian process regression (Titsias, 2009). With our extensions, these methods scale to far larger problems, *without changing a single line* of their implementation in Python. Our Gaussian process experiment shows that simply scaling up a 13 year old method can outperform much more recent methods, indicating that older methods may be undervalued in recent literature.

2 Motivation: Memory-Constrained Machine Learning

Since memory overflows cause the execution of code to be immediately halted without producing any result, memory constraints form the key obstacle for scaling many machine learning algorithms. In addition, memory is a scare resource that comes at a considerable cost, particularly in GPUs. This causes memory to be a key limiting factor in researchers and practitioners using machine learning tools at scale. This is particularly noticeable in algorithms where minibatching is undesirable and that rely on pairwise distances, like k-Nearest Neighbours (kNN) or Gaussian processes (Rasmussen and Williams, 2006) (which we particularly focus on in this work). Even in modern deep learning memory constraints cause problems, by limiting batch sizes, layer widths, or sizes of attention mechanisms (Vaswani et al., 2017). In all of these examples, matrix and linear algebra operations cause the bottleneck. For kNN, kernel/GP methods, and transformers the root of the problem is a pairwise matrix needs to be computed between inputs, giving a quadratic memory cost.

Often, more memory efficient implementations *can* be programmed, although at the cost of increased software complexity. This ranges from minor annoyances, for example accumulating minibatch gradients in an outer loop for large-batch training, to complex engineering efforts that have been published as scientific contributions, for example in scaling Gaussian processes to $> 10^5$ datapoints (Gal et al., 2014; Wang et al., 2019; Meanti et al., 2020).

Our goal is to provide a tool that finds memory-efficient ways to execute algorithms, without the need for increasing software complexity. This will allow scientists and practitioners to access the benefits of scale in existing methods more easily, and without incurring the cost of expensive large-memory hardware. For the main demonstration of our approach, we will automatically obtain a memory-efficient implementation of Sparse Variational Gaussian processes (Titsias, 2009), which was previously implemented with considerable difficulty (Gal et al., 2014). The increase in scale makes the method competitive in comparisons where it was previously dismissed as not scalable enough (Wang et al., 2019), showing the value of reducing the barriers to scaling.

3 Related Work

A popular approach to address memory issues is distributing computation across multiple resources like a group of GPUs or a computer cluster with network protocol connectivity between machines (Buyya, 1999; Dean and Ghemawat, 2008). More specifically, sharding allows large tensors to be split up and distributed across multiple devices, which increases the total amount of memory available for an algorithm, but comes at the cost of requiring more hardware resources. Most computational frameworks[1] (Abadi et al., 2016; Shazeer et al., 2018; Bradbury et al., 2018; Paszke et al., 2019) support this, with semi-automatic tools being available for specific settings, while manual

[1]Published under permissive open-source licenses, like Apache or BSD.

distribution across devices is needed in general. This complicates an implementation and requires the user to have wide engineering skillset. Nevertheless, while this approach does allow scaling of certain implementations, it remains wasteful for algorithms that *can* be implemented in a more memory-efficient way, but where it is simply cumbersome to do so.

Compilers have been introduced to allow humans to express programs in an elegant way, while generating programs that actually run well on specific hardware Aho et al. (2006). Our goal of obtaining memory-efficient implementations, while keeping code convenient for humans, is therefore suited to be addressed by adding memory optimisations to a compiler. Compilers are already being used to optimise computational graphs, notably in JAX, TensorFlow and PyTorch by XLA (Leary and Wang, 2017), TVM (Chen et al., 2018) and Glow (Rotem et al., 2018) for PyTorch only. TVM performs similar optimisations to XLA, but unlike XLA, it is not seamlessly integrated into popular frameworks and requires additional user effort.

The optimisations in XLA mainly focus increasing code speed, for example through *common sub-expression elimination* (CSE), *dead code elimination* (DCE), operations *fusion*, and other more specific modifications. The main advantage of XLA is that it optimises computations in a way that is completely transparent to the user who specifies the computational graph. Although XLA and TVM implement low-level memory optimisations, they do not adapt code handling large tensors to satisfy memory constraints. For the matrix and linear algebra tasks that we consider, KeOps (Feydy et al., 2020) currently provides the most efficient memory management. To achieve any benefits, a user must specify a series of computations using KeOps classes, which form a layer above frameworks like TensorFlow or PyTorch. KeOps works similarly to a compiler, by first building a symbolic representation of the computation, which allows the computation to be broken into memory-efficient sections, that are then run with custom CUDA kernels.

In terms of prior work, KeOps is closest in aim and achievement to ours. We aim to address three of its limitations. Firstly, KeOps requires users to reimplement their algorithms using KeOps classes. While the programming interface is elegant, needing to mix KeOps and other computational frameworks does add complexity. Secondly, for KeOps to be able to optimise an operation, it has to be reimplemented within KeOps, which significantly duplicates effort. Finally, because of the former drawback, KeOps does not inherit the support for a wide range of hardware from e.g. JAX/TensorFlow.

4 Memory Efficient Matrix and Linear Algebra Operations in XLA

Compilers are a highly promising way for improving runtime properties of code, without requiring user intervention, and while leaving code elegant. The specific matrix and linear algebra optimisations that we consider have not yet been implemented in any of the frameworks discussed above. However, they *could* be implemented in any of TVM, KeOps, or XLA. We choose to extend XLA over TVM, because of XLA's better integration with common computational frameworks. In addition, we choose to extend XLA over KeOps, because it **1)** does not require algorithms to be rewritten in a separate framework, **2)** can optimise computational graphs in their entirety, rather than just what is implemented in the separate framework, and **3)** can take advantage of the full capabilities that already exist in JAX/TensorFlow.

We introduce several optimisation strategies (known as *optimisation passes* in the XLA codebase) into the XLA optimisation pipeline. We aim to constrain the program's memory footprint with minimal sacrifices in the execution speed. The optimisation passes examine the entire computational data-flow graph (High Level Optimiser Internal Representation, or HLO IR), search for weak spots, and try to eliminate them. Abstractions at a similar level to HLO IR have been shown to be convenient for optimising linear algebra operations (Barthels et al., 2021). We add match-and-replace operations, e.g. to introduce a more efficient distance computation, reshuffling operations for expressions that are invariant to evaluation order, and splitting with large tensors to reduce memory usage.

4.1 Match and replace

The *match and replace* optimisation pass searches for expressions in a data-flow graph for which we know in advance that an equivalent and more efficient version exists. For example, we search for expressions that compute euclidean distance in naive form between vectors of length n and m with a dimension d. The naive euclidean distance computation uses broadcasting over the dimension d and

Listing 1 Chain multiplication example $C = ABv$ for $A, B \in \mathbb{R}^{n \times n}$, and $v \in \mathbb{R}^n$.

```
@jax.jit
def matrix_matrix_vector_mul(A, B, v):
    C = A @ B @ v
    return C
```

creates a temporary tensor with entries $(x_{nd} - y_{md})^2$ of size $n \times m \times d$. This can be replaced with $\sum_d x_{nd}^2 + y_{md}^2 - 2x_{nd}y_{md}$, where the largest tensor has size $n \times m$.

Replacing sub-parts of the graph is a standard procedure in compilers like XLA, although many linear algebra tricks have been missing. While the euclidean distance is the only match-and-replace optimisation we implement, other operations can easily be added, for example, efficiently adding diagonals to matrices without allocating a dense square tensor where only the diagonal is non-zero.

4.2 Reordering

A computational data-flow graph is an ordered sequence of operations, with the order of operations influencing the memory usage. In some cases, reordering sub-parts of the data-flow graph can lead to reductions in the memory footprint. The classical example of reordering is the optimisation of matrix chain multiplications. For example, consider the matrix expression $\mathrm{C} = \mathrm{AB}\mathbf{v}$ for matrices $\mathrm{A}, \mathrm{B} \in \mathbb{R}^{n \times n}$, and $\mathbf{v} \in \mathbb{R}^n$. In the listing 1, the order of operations determines that the matrix multiplications are performed from left to right, i.e. $\mathrm{C} = (\mathrm{AB})\mathbf{v}$, which gives the most inefficient execution order with the runtime complexity $O(n^3)$ and memory complexity $O(n^2)$. Changing the order to $\mathrm{C} = \mathrm{A}(\mathrm{B}\mathbf{v})$ improves time and memory complexities to $O(n^2)$ and $O(n)$ respectively because the intermediate multiplication result of $\mathrm{B}\mathbf{v}$ is a vector not a matrix as in the case of AB multiplication.

The optimisation of matrix chain multiplication is possible due to the associativity of matrix multiplication, such that the result of the matrix multiplication chain does not depend on where parentheses are placed. There are many efficient and sophisticated algorithms for addressing this task (Chin, 1978; Czumaj, 1996; Barthels et al., 2018; Schwartz and Weiss, 2019). We implement a simplified procedure for reordering matrix vector chain multiplications, that detects inefficient matrix multiplication chains, which are guaranteed to reduce in size at the end of the chain.

4.3 Data-flow graph splitting

Often, a part of a computational data-flow graph can be divided into multiple independent copies, such that each copy of the data-flow graph or its part act on a slice of the input tensor, and the results are combined afterwards in some fashion. This splitting approach is also known as a MapReduce technique Dean and Ghemawat (2008), where a computation is divided into smaller and less expensive parts (map) and then combined into the final result (reduce). The splitting technique is common for distributing the computational load. The focus of existing solutions is on exploiting hardware parallelism or utilising multiple devices. Instead, we use the same techniques for reducing total memory consumption, which is possible because the memory for individual map operations can be freed before the whole result is computed.

An optimisation pass starts by running a depth-first search from the final result of the computation. The operations `dot` or `reduce_*` are special, as they often indicate that a computation involving a large tensor can give a smaller result. Once a `dot` or `reduce_*` operation is marked as fully traversed, we recursively search the traversed paths for operands that are impractically large tensors, until we reach operands that are deemed small enough. Along the way, we keep track of which operations are applied, and along which axes they trivially paralellisable. The result is a collection of sub-graphs, that start at operations that produce large tensors, and end at operations that reduce them again, together with candidate axes that they can be split across. According to some heuristics which ensure appropriate sizes for intermediate results, we then turn this entire computation into a while loop, where each iteration computes a manageable part of the final result (fig. 1).

Checking the axis if it is splittable is necessary as not all operations act independently on each dimension. For example, element-wise operations can be split on any axis, whereas the triangular

Algorithm 1 A description for depth-first search visitor-handler that splits the `dot` operation

input `dot` instruction

if `output_size(dot)` $\geq$ `tensor_size_threshold`, i.e. `dot` is splittable **then**
    Continue traversing succeeding operations in the data-flow graph and search for size-reducing operation for the output tensor of `dot`.
if `dot.rhs` and `dot.lhs` operands are *not* splittable **then**
    Continue traversing the data-flow graph.
if `dot.rhs` $\neq$ `dot.lhs` operands are splittable **then**
    Continue traversing the data-flow graph.
Define `operand_to_split` = either `dot.rhs` or `dot.lhs`
Recursively search which dimensions of `operand_to_split` are splittable, i.e. walk back through all operands in the graph which produce `operand_to_split`
Define `split_dims` i.e. all `operand_to_split` dimensions along which `operand_to_split` can be splitted by recursively traversing back visited operations and checking their dimensions.
Define `split_producers`, i.e.

- operations in the graph which produce `split_dims`
- satisfy `output_size(split_producer)` $\leq$ `tensor_size_threshold`
- all operations between `split_dims` and `operand_to_split` are splittable.

if `split_dims` or `split_producers` are empty **then**
    Continue traversing the data-flow graph.
Define `best_split_dim`
Build a while loop instructions module which iterates over `best_split_dim`,

- `split_producers` are inputs to the loop module, which are sliced
- the body of the loop is the copy of sub-graph `split_producers` $\leftrightarrow$ `dot`

Add instructions to combine results of the while loop.
Replace inefficient sub-graph of `producers` and `dot` with new while loop module.
Continue traversing the data-flow graph.

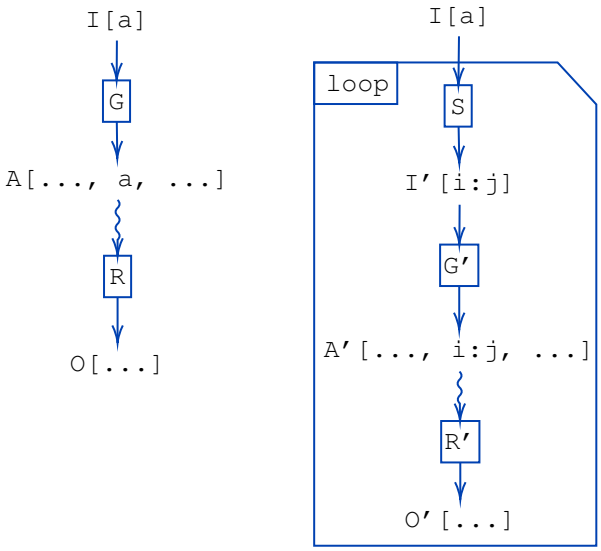

Figure 1: The scheme demonstrates transformation of the sub-graph on the left to the graph on the right. Sub-graph on the left consists of `G` and `R` which a generator and reducer operations respectively, `I` is the input of `G`, `A` is output of `G` and `O` is output of `R`. The following notation `A[..., a, ...]` means that the tensor `A` has a dimension of size `a` and `A` can have other dimensions. The `i:j` is a slicing operation. Splitting procedure converts the left sub-graph into the loop of independent iterations performing the same chain of operations on a small slice `i:j`.

solve operation can be split on "batching" dimensions only. Next, the data-flow graph splitting procedure selects the dimension of the largest size which contributes the most to memory.

As we discussed earlier, the decision about where to split the graph depends on the tensor size. We offer two XLA options to user for out-of-memory mitigation: *tensor size threshold* and *tensor split size upper bound*. Tensor size threshold is a criterion designed for detecting which operations should be marked as candidates for splitting. Tensor split size upper bound serves as a threshold on the largest allowed chunk size for splitting. These options are set equal by default. The following command-line snippet shows how a user would use these options by passing them via an environment variable, and the snippet is indifferent to the machine learning framework used by the script at listing 2. Minimal user effort is required for using our XLA compiler extension. The user is involved only in defining what the suitable threshold and splitting sizes are.

Listing 2 An example of how a user can set options for the extended XLA using the environment variable.

```
XLA_FLAGS="--xla_tensor_size_threshold=1GB --xla_tensor_split_size=500MB" \
python train.py
```

One strong benefit of our compiler-based solution, is that the computational graph represents the whole pipeline of computations, including forward and backward propagation. Our splitting procedure will be applied automatically, regardless of how many derivatives need to be computed. In addition, our procedure encompasses two splitting schemes that the machine laerning literature distinguishes: model-based and data-based splitting schemes of the data-flow graph. The model-based splitting scheme involves partitioning the model over its parameters, whereas the data-based splitting scheme batches over inputs and, therefore, an algorithm. The proposed splitting approach is suited for supporting both schemes out of the box.

4.4 XLA limitations

While we still believe that XLA is the right framework for our extensions, several limitations came to light during implementation.

One limitation that is shared with all current frameworks, is that they only have a weak linear algebra type system, where matrices are represented as arrays without additional properties. Solutions that support stronger type systems (Bezanson et al., 2017; Barthels et al., 2021) may be able to implement a wider variety of match-and-replace optimisations.

Another limitation comes from the default memory allocation manager not being aware of memory limits. Its current behaviour is to execute nodes in the computational graph, and therefore allocate any required memory, as soon as the required inputs have been computed. This means that even if tensors are split to manageable sizes, memory overflows can still occur if several are executed simultaneously. To prevent this from happening, we had to use memory limits that were smaller than our total GPU memory.

5 Experiments

This section shows how existing software packages take advantage of our extension to XLA (eXLA). We demonstrate our optimisations on non-parametric k-nearest neighbours and sparse Gaussian process regression (SGPR) models.

5.1 Matrix-Vector Multiplication

We start by demonstrating the improved efficiency that eXLA offers to large-scale matrix-vector multiplications of the form $y = \mathrm{K}v$, where K is an $n \times n$ kernel matrix, and $y, v \in \mathbb{R}^n$. Such computations are common in Conjugate-Gradients-based Gaussian process approximations (Gibbs and Mackay, 1997; Wang et al., 2019; Artemev et al., 2021), where $\mathrm{K}_{ij} = k(x_i, y_j)$ and k is some kernel function. We choose the common Squared Exponential.

We implement this equation using GPflow (Matthews et al., 2017), a TensorFlow-based package that provides a convenient software interface for Gaussian processes and kernel functions. Without eXLA, the entire K would be stored in memory, leading to a n^2 memory cost. This makes running on large datasets infeasible, where e.g. $n = 10^6$ would lead to a memory requirement of 8TB, which

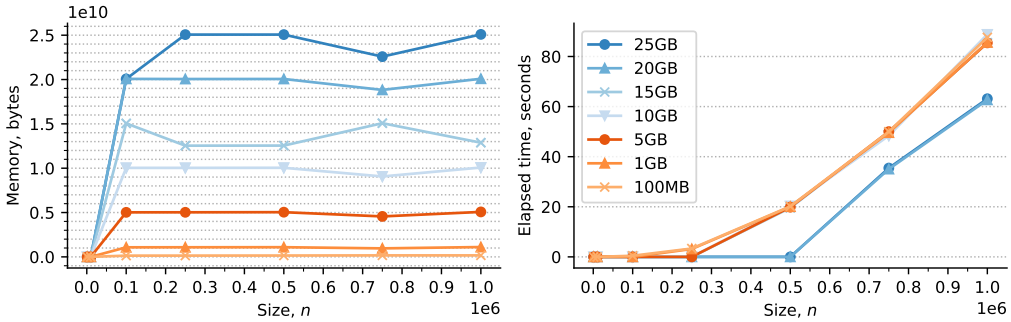

Figure 2: GPU memory consumption and elapsed time of $n \times n$ kernel matrix-vector multiplication.

is impractical even for the largest of modern GPUs with 40GB of memory. A memory efficient split/implicit implementation is necessary to scale to large datasets, as was impressively done by Wang et al. (2019), but is cumbersome.

We ran our implementation with eXLA enabled, which allows a user to control the memory an algorithm. We evaluated the expression in double precision on a Tesla V100 GPU with 32 GB of memory, and applied a range of memory limits. In fig. 2 we report the peak memory consumption and execution time of evaluating the kernel matrix-vector product for different sizes, with different memory limits applied. We see that the memory constraints are not violated, and that dataset sizes are used that are far beyond the 32 GB memory capacity.

5.2 K-Nearest Neighbours

K-nearest neighbours is a fundamental machine learning algorithm, with a similar large memory cost. A kNN query selects k closest data points in the dataset to each query point. Brute-force implementations computate pairwise distances between m query points and n data points, resulting in the distance matrix of size $m \times n$. This is followed by a `topk` operation, which is often naively implemented using column-wise `sort` operation on the distance matrix. Our benchmarks show that eXLA scales the brute-force approach and does not fail for large problems, i.e. large n and m.

We compare TensorFlow and JAX implementations with and without eXLA optimisations, and a KeOps implementation. We use randomly generated data, common benchmarks like MNIST and Fashion-MNIST, and Glove-50, Glove-100 and Glove-200 from the ANN-benchmark toolkit Aumüller et al. (2020). We use $m = 1\text{e}4$ query points in all benchmarks.

Our results are listed in table 1 (see the appendix for a full table that reproduces Feydy et al. (2020, table 3)). In all benchmarks, we set the tensor size threshold for eXLA to 100MB for simplicity, even though this may not be optimal for performance. We observe that eXLA prevents memory overflows in JAX and TensorFlow. In addition, performance is comparable or higher. We acknowledge that KeOps performs significantly better than any JAX or TensorFlow implementation. This is explained by **1)** JAX/TF not having efficient implementations for certain functions (e.g. `topk` runs a full sorting algorithm), and **2)** KeOps having implemented additional optimisations, which could also be added to XLA. However, we note that we also achieved our goal of improving the memory and time performance of a JAX/TensorFlow implementation *without changing the code*.

5.3 Sparse Gaussian Process Regression

Gaussian processes (Rasmussen and Williams, 2006) are considered the gold standard method for performing regression with uncertainty estimates. A straightforward implementation requires taking a matrix decomposition of an $n \times n$ kernel matrix (like those considered in section 5.1), which leads to an $O(n^3)$ time cost, and an $O(n^2)$ memory cost. Scaling Gaussian process is challenging, which is often attributed to the time cost. In reality however, large datasets cause memory overflows far before long runtimes become an obstacle.

Approximate methods have been introduced to deal with both the time and space issues. While there are many, we consider the sparse variational approximation (Titsias, 2009) for which a naive implementation has $O(nm^2 + m^3)$ time cost, and $O(nm + m^2)$ memory cost. Here, m controls the

Dataset	Distance	n	d	KeOps	eJAX	eTF	JAX	TF
Random	L^2	1e4	100	983263	277364	284777	281695	280826
Random	L^2	1e4	3	3662188	292804	294971	288098	294776
Random	L^2	1e6	100	24367	2433	2530	∅	∅
Random	L^2	1e6	3	123765	2512	2605	∅	∅
MNIST	L^2	6e4	784	41084	32290	33455	25544	26138
MNIST	L^1	6e4	784	40697	2356	2985	2498	2988
Fashion	L^2	6e4	784	40399	32382	33428	25558	26128
Fashion	L^1	6e4	784	40982	2357	2984	2498	2989
Glove-50	Cosine	1.18e6	50	3464257	2103	1929	∅	∅
Glove-100	Cosine	1.18e6	100	631420	2053	1871	∅	∅
Glove-200	Cosine	1.18e6	200	398293	1967	1724	∅	∅

Table 1: Query processing rates (queries per second) for kNN. n and d are the number of data points and the data dimension respectively. Runs which failed due to memory overflow are denoted by ∅. Runs with eXLA are denoted eJAX and eTF respectively.

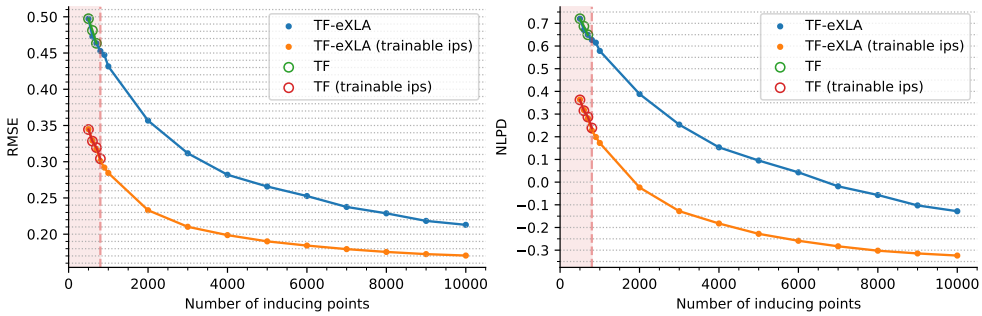

Figure 3: Root mean squared error (RMSE) and negative log predictive density (NLPD) performance test metrics of SGPR for `3droad` dataset as the number of inducing points is increased. The red shaded region emphasizes the capacity of the SGPR model which user can run using standard GPflow and TensorFlow release packages.

quality of the approximation, in terms of the number of inducing variables. While (under certain conditions) $m \ll n$ can give very accurate approximations (Burt et al., 2019, 2020), the memory cost can limit the size of m that can be used, which limits performance. A more compact implementation with a memory cost of $O(m^2)$ exists (Gal et al., 2014), but is so cumbersome to implement that it is not widely used or compared against.

Fortunately, the splitting optimisation we implemented in eXLA can discover the same procedure that was engineered by Gal et al. (2014), without the help of automatic differentiation. Moreover, since eXLA operates on the entire computation graph, it optimises gradients as well as the optimisation objective function with no additional effort. We demonstrate the utility of eXLA by scaling the GPflow (Matthews et al., 2017, 2.3.1 release version) implementation of Sparse Gaussian process regression (SGPR, Titsias, 2009), *without any modifications* of the code.

With our eXLA optimisations, SGPR was able to scale to much larger datasets, with more inducing points. We conduct experiments on a Tesla V100 GPU with 32 GB of memory, and run on two of the largest UCI datasets that are commonly considered in Gaussian process research: `3droad` and `houseelectric`. We primarily compare to Wang et al. (2019), who use a Conjugate Gradients approximation (Gibbs and Mackay, 1997) to achieve the most impressive scaling of a Gaussian process approximation to date, using an impressively engineered implementation that manually splits and distributes parts of the computation.

In fig. 3 we compare GPflow's SGPR implementation with and without eXLA as we increase the number of inducing points. We see that until about 800 inducing points, the normal and eXLA runs

Dataset	Model	RMSE	NLPD	Time (hours)	GPUs
houseelectric	SGPR-1000	$0.048 \pm 2\text{e}{-4}$	$-1.602 \pm 3\text{e}{-3}$	5.01 ± 0.06	1
	SGPR-2000	$0.046 \pm 1\text{e}{-4}$	$-1.651 \pm 3\text{e}{-3}$	18.03 ± 0.09	1
	SGPR-3000	$0.044 \pm 1\text{e}{-4}$	$-1.696 \pm 5\text{e}{-3}$	38.68 ± 0.14	1
	SGPR-4000	$\mathbf{0.043} \pm 1\text{e}{-4}$	$\mathbf{-1.717} \pm 5\text{e}{-3}$	50.00 ± 0.10	1
	Iterative GP*	0.054 ± 0.000	-0.207 ± 0.001	1.55 ± 0.02	8
	Iterative GP**	0.050	∅	79.96	8
3droad	SGPR-1000	0.285 ± 0.002	-0.173 ± 0.004	1.11 ± 0.01	1
	SGPR-5000	0.190 ± 0.002	-0.228 ± 0.002	11.33 ± 0.03	1
	SGPR-8000	0.176 ± 0.001	-0.302 ± 0.004	28.21 ± 0.05	1
	SGPR-10000	0.170 ± 0.001	$\mathbf{-0.322} \pm 0.002$	41.83 ± 0.03	1
	Iterative GP*	0.110 ± 0.017	1.239 ± 0.025	$1.00 \pm 2\text{e}{-3}$	8
	Iterative GP**	**0.106**	∅	7.06	8

Table 2: SGPR performance on houseelectric and 3droad dataset. Iterative GP* and Iterative GP** are trained with lengthscale per dimension and shared lengthscale across dimensions respectively. Iterative GP values are from Wang et al. (2019), with unreported metrics denoted as ∅.

result in the same predictive metrics, as desired. After 800 inducing points, runs without XLA fail with an "out of memory" error, while with eXLA we scaled to 10^4 inducing points. Simply scaling the method in this way, leads to significant performance improvements.

We now compare predictive accuracies directly with the scalable Conjugate Gradients implementation of Wang et al. (2019). In that paper, SGPR was discussed as a method that would not scale, probably due to the difficulty of implementing it in a memory-efficient way as in Gal et al. (2014). Table 2 shows that using eXLA to scale SGPR can improve predictive performance to such a degree that it can outperform the Conjugate Gradients implementation of Wang et al. (2019), without needing additional hardware.

6 Discussion

We showed that our additional XLA compiler optimisation passes (eXLA) could manage memory overflows algorithms with large tensor or linear algebra operations. The developed compiler extension automatically adjusts computational data-flow graphs to control memory utilisation. As demonstrated in the experiments section, we successfully ran machine learning models compiled with eXLA on a greater scale, whereas their out-of-the-box implementations failed with Out of Memory errors. Crucially, we used existing software packages without modifying any code.

In addition to showing that our compiler extensions work as intended, our experiments also provide directly useful empirical results in Gaussian processes. We managed to run an "old" method (SGPR, Titsias, 2009), with unchanged code, to obtain empirical results that outperformed a state-of-the art method (Wang et al., 2019). This corrects earlier observations in the literature that these methods are inaccurate, and shows that—if the methods can be scaled—they may behave according to theory that shows that they should provide very accurate solutions (Burt et al., 2020).

The exciting possiblity of eXLA, is that it opens up the possibility to probe behaviour of machine learning models in regimes that were previously infeasible, and on cheap hardware. For example, one could train very wide neural networks, to empirically compare to behaviour predicted by NTK theory (Lee et al., 2018; Matthews et al., 2018; Jacot et al., 2018; Novak et al., 2020). In addition, transformers (Vaswani et al., 2017) are notoriously memory hungry, and eXLA could help with running them on cheaper hardware, or distributing them across GPUs, without increasing software complexity.

The current implementation of eXLA is still only a demonstration of what compiler optimisations could achieve, and many more optimisations can be added. We believe that increasing the capability of compilers like XLA will greatly increase the efficiency of researchers and practitioners. We hope that community-driven compiler projects will contribute to the community in a similar way to how existing numerical frameworks already do.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). Compilers: Principles, techniques, and tools.

Artemev, A., Burt, D. R., and Van Der Wilk, M. (2021). Tighter bounds on the log marginal likelihood of gaussian process regression using conjugate gradients. In *International Conference on Machine Learning*, pages 362–372. PMLR.

Aumüller, M., Bernhardsson, E., and Faithfull, A. (2020). Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*.

Barthels, H., Copik, M., and Bientinesi, P. (2018). The generalized matrix chain algorithm. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 138–148.

Barthels, H., Psarras, C., and Bientinesi, P. (2021). Linnea: Automatic generation of efficient linear algebra programs. *ACM Transactions on Mathematical Software (TOMS)*.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43.

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM review*.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. (2017). Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42.

Burt, D., Rasmussen, C. E., and van der Wilk, M. (2019). Rates of convergence for sparse variational gaussian process regression. In Chaudhuri, K. and Salakhutdinov, R., editors, *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 862–871. PMLR.

Burt, D. R., Rasmussen, C. E., and van der Wilk, M. (2020). Convergence of sparse variational inference in gaussian processes regression. *arXiv preprint arXiv:2008.00323*.

Buyya, R. (1999). High performance cluster computing. *New Jersey: F'rentice*.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. (2018). {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.

Chin, F. Y. (1978). An o (n) algorithm for determining a near-optimal computation order of matrix chain products. *Communications of the ACM*, 21(7):544–549.

Czumaj, A. (1996). Very fast approximation of the matrix chain product problem. *Journal of Algorithms*, 21(1):71–79.

Dean, J. and Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113.

Feydy, J., Glaunès, J., Charlier, B., and Bronstein, M. (2020). Fast geometric learning with symbolic matrices. *Advances in Neural Information Processing Systems*, 33.

Gal, Y., Van Der Wilk, M., and Rasmussen, C. E. (2014). Distributed variational inference in sparse gaussian process regression and latent variable models. *arXiv preprint arXiv:1402.1389*.

Gibbs, M. and Mackay, D. (1997). Efficient implementation of Gaussian processes. Technical report, Cavendish Laboratory, University of Cambridge.

Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Leary, C. and Wang, T. (2017). Xla: Tensorflow, compiled. *TensorFlow Dev Summit*.

Lee, J., Sohl-dickstein, J., Pennington, J., Novak, R., Schoenholz, S., and Bahri, Y. (2018). Deep neural networks as gaussian processes. In *International Conference on Learning Representations*.

Matthews, A. G. d. G., Hron, J., Rowland, M., Turner, R. E., and Ghahramani, Z. (2018). Gaussian process behaviour in wide deep neural networks. In *International Conference on Learning Representations*.

Matthews, A. G. d. G., van der Wilk, M., Nickson, T., Fujii, K., Boukouvalas, A., León-Villagrá, P., Ghahramani, Z., and Hensman, J. (2017). GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*.

Meanti, G., Carratino, L., Rosasco, L., and Rudi, A. (2020). Kernel methods through the roof: Handling billions of points efficiently. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H., editors, *Advances in Neural Information Processing Systems*, volume 33, pages 14410–14422. Curran Associates, Inc.

Novak, R., Xiao, L., Hron, J., Lee, J., Alemi, A. A., Sohl-Dickstein, J., and Schoenholz, S. S. (2020). Neural tangents: Fast and easy infinite neural networks in python. In *International Conference on Learning Representations*.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*.

Rasmussen, C. E. and Williams, C. K. (2006). Gaussian processes for machine learning. *Gaussian Processes for Machine Learning*.

Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al. (2018). Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*.

Schwartz, O. and Weiss, E. (2019). Revisiting "Computation of Matrix Chain Products". *SIAM Journal on Computing*, 48(5):1481–1486.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., et al. (2018). Mesh-tensorflow: Deep learning for supercomputers. *arXiv preprint arXiv:1811.02084*.

Sutton, R. (2019). The bitter lesson. *Incomplete Ideas (blog)*.

Titsias, M. (2009). Variational learning of inducing variables in sparse gaussian processes. In *Artificial intelligence and statistics*. PMLR.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Wang, K., Pleiss, G., Gardner, J., Tyree, S., Weinberger, K. Q., and Wilson, A. G. (2019). Exact Gaussian processes on a million data points. In *Advances in Neural Information Processing Systems*.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] See section section 6 and section 4.4
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [Yes]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] The code will be open-sourced and available on GitHub.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See section 5
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] XLA
   (b) Did you mention the license of the assets? [Yes] XLA is part of TensorFlow and open-sourced under the same license
   (c) Did you include any new assets either in the supplemental material or as a URL? [N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]