Relational Queries with a Tensor Processing Unit

Pedro Holanda
CWI, Amsterdam, NL
holanda@cwi.nl

Hannes Mühleisen
CWI, Amsterdam, NL
hannes@cwi.nl

ABSTRACT

Tensor Processing Units are specialized hardware devices built to train and apply Machine Learning models at high speed through high-bandwidth memory and massive instruction parallelism. In this short paper, we investigate how relational operations might be translated to those devices. We present mapping of relational operators to TPU-supported TensorFlow operations and experimental results comparing with GPU and CPU implementations. Results show that while raw speeds are enticing, TPUs are unlikely to improve relational query processing for now due to a variety of issues.

1 INTRODUCTION & RELATED WORK

For the last four decades, the use of specialized hardware has been investigated to improve performance of relational query processing [2, 4]. General-purpose processors have reached a plateau with regards to integration density and clock speeds. At the same time, innovations in systems have removed inefficiencies to a large extent [3, 9]. To achieve significant performance improvements, use of specialized hardware has regained some urgency.

A common approach is to re-purpose hardware that is available in large quantities at low cost, most prominently Graphics Processing Units (GPU) [5, 7]. The disadvantage here is that the design of those devices was developed for a very different use-case, 3D computer games, and they only partly benefit the data processing use case. A large number of systems is now openly available that utilize GPUs, for example “BlazingDB” or “OmniSci”.

The rise of Machine Learning (ML) applications has produced a *new* class of hardware devices, so-called Tensor Processing Units (TPU). These devices are built to efficiently support ML workflows by accelerating linear algebra operations such as matrix multiplication. Efficiency in those tasks is achieved with high-bandwidth memory (HBM) and massive parallelism of computation. For example, the third-generation Cloud TPU by Google reportedly achieves 420 teraflops on 128 GB of HBM.

With their massive computation parallelism and high-bandwidth memory, TPUs appear somewhat similar to Graphics Processing Units. Moreover, their intended use case – Machine Learning workflows – is far more related to data management than 3D gaming. Considering their superior performance they are a promising candidate to speed up query processing.

TPUs are controlled through the TensorFlow API. TensorFlow exposes a low-level library of bulk data transformations. The TPUs implement a limited subset of the TensorFlow API [6], ca. 180 operators at the time of writing.

In this experimental paper, we investigate the performance of the third-generation Google TPU for typical analytical query processing tasks. The research question is whether TPUs can benefit analytical query processing.

2 OPERATOR MAPPING

The TensorFlow API does not directly expose relational operators such as selections, projections, joins or aggregations. However, it is possible to combine hardware-supported TensorFlow operators in such a way that the result is equivalent to the corresponding relational operator. Below we will give some examples of how this can be achieved.

Selection. It can be achieved using the `tf.where` and `tf.gather` operations. `tf.where` takes a Boolean vector and returns the indices where the input is true. `tf.gather` takes these coordinates and applies them to the payload vector. If more columns should be projected, we can make use of the rectangular nature of relational tables and re-use the `filter` vector on other columns.

Single-Value Aggregation. The `SUM`, `MAX` and `MIN` aggregate functions can be performed by the reduce TensorFlow operations `reduce_sum`,`reduce_min` and `reduce_max` respectively.

The `COUNT` and `AVG` aggregate functions do not have a one-to-one map with TensorFlow. The `COUNT` can be performed by casting a boolean mask filter, using the `tf.cast` operator, to a `tf.int32` type and executing a `reduce_sum` afterwards. The `AVG` function is composed by the `SUM` divided by the `COUNT` aggregates.

Grouped Aggregation. `Group By` operations are challenging insofar as the possible groups need to be determined beforehand in the TensorFlow API. This can be done using the `tf.unique` operator. However, this operator is not currently supported by the TPU hardware. This requires a two-phase execution where first the groups are determined with CPU code, followed by then the plan generation and execution on the TPU. For grouping, we use the extended form of the `tf.where (cond, true_val, false_val)` operator, which acts very similar to a SQL CASE statement. Using constant vectors of 1 and 0 together with a cast, we can then use `tf.where` again to zero rows that are not in the current group and finally use `tf.reduce_sum` to compute the per-group aggregate. For non-grouped aggregates, the first filtering stage is redundant.

One major drawback of this approach is the pre-computation of the groups using CPU code. If the amount of groups is large, performance will be reduced.

Dimension Join. A form of nested loop join can be implemented by using the `tf.while_loop` operator. The body and condition must be implemented as functions and an index must be defined the `tf.while_loop` call. The condition function is checked before execution of the loop and the body function defines it. For a join, the body is composed of a `tf.gather` operation that is used to loop through the outer relation and a `tf.where` that is used to filter the elements from the inner relation that match the outer relation key. The major drawback of this approach is that TensorFlow generates as many graph nodes as the body loop size times the amount of times it is executed. The creation, compilation and optimization steps of the DAG are heavily dependent on its size. Hence compiling joins for large outer relations will reduce compilation performance.

	SF	CPU	GPU	TPU	HyPer
Selection	1	28	0.68	**0.2**	7.5
	10	380	5.06	**2.7**	16
Single-Value Aggr.	1	1.20	**0.05**	0.06	6.3
	10	3.7	**0.42**	0.5	30
Grouped Aggr.	1	13	1.5	**0.12**	8.2
	10	340	14.4	**1.1**	14
Dim. Join	1	1.2	10.7	**0.06**	1.6
	10	4	12.7	**0.08**	0.9
Top N	1	13.2	5.42	50	**7**
	10	143	47.1	649	**20**

Table 1: Microbenchmark Results

Top N. The `tf.nn.top_k` operator allows to return the sorted, ascending or descendent `n` elements. In case multiple columns are ordered in the order_by function, the columns can be multiplied by powers of 10, summed together and sorted by using the `tf.nn.top_k` operator with `n` equal to the column size. The index generated by the operator can then be applied to all columns using `tf.gather`.

3 EXPERIMENTAL RESULTS

In this section, we provide an evaluation of the TPU using microbenchmarks over TPC-H data. In addition, we provide a comparison of the performance of the TPUs with GPU, CPUs and the commercial RDBMS HyPer (Version 20182.18.1009.2120).

We implemented all queries using the TensorFlow API [1] for Python, which can target CPUs, GPUs and TPUs simultaneously. All experiments were conducted on a Google cloud machine equipped with 360 GB main memory, 96-core Intel(R) Xeon(R) CPU @ 2.00GHz, Nvidia Tesla P100 GPU and a version 3-8 TPU. While not available for sale, it is possible to buy time on Google's TPUs through their Cloud API, currently priced at 8$ per hour. All experiments were repeated five times with the last result recorded. TensorFlow results do not include compilation and data transfer times. Also, the TensorFlow implementations do not perform numeric overflow checking and used the internal floating point type to represent numeric values since this was recommended by the documentation for best performance. Furthermore, since TPUs only supports numeric data, dictionary-encoding of category and date columns was performed for them. We are committed to reproducibility, hence all experimental code is available on-line. [1]

3.1 Microbenchmarks

Selection: Results for selecting a subset of rows show the TPU' to perform best at first glance. The projection of the Boolean selection vector created by the expression to a list of indices used to gather the matching rows is however not supported by the TPU interface, requiring them to be performed in CPU code. Computing these indices also was responsible for ca. 75% of time in the TensorFlow CPU version. This is likely unnecessary considering Hyper's results on this experiment. Time for GPU and TPU scales linearly with data size.

Single-Value Aggregation: TPU and GPU performance is high and similar, owing to the good mapping between the relational task of single-value aggregation and the hardware operations. The CPU version is outperformed by an order of magnitude. Time for GPU and TPU scales linearly with data size.

Grouped Aggregation: The computation of unique groups that is required for the TensorFlow version is not supported by either GPU nor TPU back-ends. We have thus removed it. The TPU shows by far the best performance, outperforming the GPU version by an order of magnitude. Again, the TensorFlow CPU code is outperformed by HyPer (which *does* compute the groups). Time for GPU and TPU scales linearly with data size.

Dimension Join: Results for dimension join show widely differing results. While the TPU performs best, inspection of the TensorFlow execution plans showed three vastly different versions. One problematic observation is that compilation time for the TensorFlow versions highly depends on the amount of rows in the outer relation. For dimension joins, this might be acceptable. In GPUs and TPUs, control flow is necessarily handed over to the hardware to achieve best performance; this is problematic for more complex operations like joins.

Top N: For Top N, the TPU shows the slowest performance. For SF10, HyPer outperforms the TensorFlow implementations regardless of hardware. This might be due to sorting not being a priority in ML workflows and the lack of optimization attention that results from it. Profiling results show that the TensorFlow implementations perform a full sort to implement Top N, which is wasteful.

4 DISCUSSION & CONCLUSION

While the experimental results presented in the previous section might appear promising, there are several serious hurdles to using TPUs to benefit data management. As is customary [8], experiments did not include hardware session management, execution plan generation and data transfer. Additional time required for this is measured in seconds, not milliseconds. Joins and grouped aggregations were particularly problematic with their performance depending on data and group cardinality and unsupported operators. Unfortunately it is precisely the fusing of a large amount of operators that is required to outperform traditional systems like HyPer using specialized hardware such as TPUs. This is particularly true due to the additional cost of transferring data to be processed from main memory into the high-bandwidth TPU memory. TPUs also rely heavily floating point numbers, yet they lead to numeric drift in aggregations. Naturally, the comparison with HyPer is completely unfair [10], since HyPer is fully-fledged system that supports persistent storage, numerous data types, error checking etc. It is most likely possible to improve the usability of TPUs for relational tasks, either through more advanced sequences of TensorFlow operations or through changes to the TPU and their APIs themselves. For example, better integer support would allow for less numeric drift on aggregations, support for selection vector generation with single-argument `tf.where` and unique value generation with `tf.unique`. Given the rapid pace of TPU development, it might not be too unreasonable to hope for those.

[1] https://github.com/pholanda/tpc-tpu

REFERENCES

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 265–283. https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] E. Babb. 1979. Implementing a Relational Database by Means of Specialized Hardware. *ACM Trans. Database Syst.* 4, 1 (March 1979), 1–29. https://doi.org/10.1145/320064.320065

[3] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR* (2007-09-14). 225–237. http://dblp.uni-trier.de/db/conf/cidr/cidr2005.html#BonczZN05

[4] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. 1986. GAMMA - A High Performance Dataflow Database Machine. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB '86)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 228–237. http://dl.acm.org/citation.cfm?id=645913.671463

[5] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. 2007. GPUQP: Query Co-processing Using Graphics Processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. ACM, New York, NY, USA, 1061–1063. https://doi.org/10.1145/1247480.1247606

[6] Google. 2019. Available TensorFlow Ops. https://cloud.google.com/tpu/docs/tensorflow-ops

[7] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. 2004. Fast Computation of Database Operations Using Graphics Processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data (SIGMOD '04)*. ACM, New York, NY, USA, 215–226. https://doi.org/10.1145/1007568.1007594

[8] Chris Gregg and Kim Hazelwood. 2011. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '11)*. IEEE Computer Society, Washington, DC, USA, 134–144. https://doi.org/10.1109/ISPASS.2011.5762730

[9] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (June 2011), 539–550. https://doi.org/10.14778/2002938.2002940

[10] Mark Raasveldt, Pedro Holanda, Tim Gubner, and Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In *Proceedings of the Workshop on Testing Database Systems (DBTest'18)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3209950.3209955