Learning to Fuse

Amirali Abdolrashidi, Qiumin Xu, Shibo Wang, Sudip Roy, Yanqi Zhou
Google Research
{abdolrashidi, qiuminxu, shibow, sudipr, yanqiz}@google.com

1 Introduction

Increasingly many machine learning applications are driven by large and complex neural networks. These models rely on effectively using accelerators (like GPUs and TPUs) to meet their computational requirements. However, effective use of such accelerators is largely determined by device-specific optimizations performed by compilers, like TensorFlow XLA [1], that map the high-level dataflow graph to operations executable on the device.

A standard optimization pass in these compilers is the fusion pass that fuses multiple operations in the dataflow graph and generates a rewritten graph with fused ops that has a lower step time and higher device utilization. The reduction in step time can be attributed to eliminating certain overheads. For example, a fused op avoids writing out intermediate values to device memory by forwarding the values directly to consumer ops. Fusion also improves the temporal data locality and therefore enables a larger optimization space for the compiler. Finally, for memory-bound applications, fusion increases the computation intensity (computation per data load), and thus is one of the best ways to improve the performance. However, fusing too many ops can increase register pressure and increase spill code or have adverse cache effects.

Fusion algorithms typically determine an ordering of nodes to consider for fusion based on heuristics. For example, the current strategy for fusion in XLA chooses nodes based on the potential memory and computation savings achieved if the node is fused with all its consumers. However, this approximation can be inaccurate as the fusion of some subsets of nodes can prevent fusion opportunities for the other nodes. Furthermore, there may be non-trivial interactions between the fusion pass and other stages like placement and scheduling of ops on devices. Therefore, strategies that determine the ordering of nodes holistically instead of through localized decisions can lead to better fusion.

In this work, we proposed a priority-based fusion where an end-to-end trainable neural network influences the order of fusion by assigning priorities to ops. To enable generalization across multiple dataflow graphs, we use a graph neural network to encode the input graphs into a trainable vector representation. This graph network is trained jointly with the policy network that generates priorities. We apply a Proximal Policy Optimization (PPO) [2] algorithm to optimize the policy network and graph neural network.

Our results show the effectiveness of priority fusion on a wide set of real-world workloads, including AmoebaNet [3], NMT [4], RNNLM [5], and Transformer-XL [6]. Compared to no fusion, we achieve an average of 16.1% speedup and a maximum of 31.5% speedup for Transformer-XL. Compared to the default fusion used by Tensorflow, we achieve an average of 3.4% speedup and a maximum of 12.9% speedup for NMT.

2 Background

To process many tasks with a high level of parallelism and computation demand, host processors launch them as GPU kernels or TPU operations onto the accelerator devices (we use ops throughout this paper for simplicity). After an op is invoked by the host, the data and instructions will be transferred from the device global memory to the on-chip memory, and the device will begin the

33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

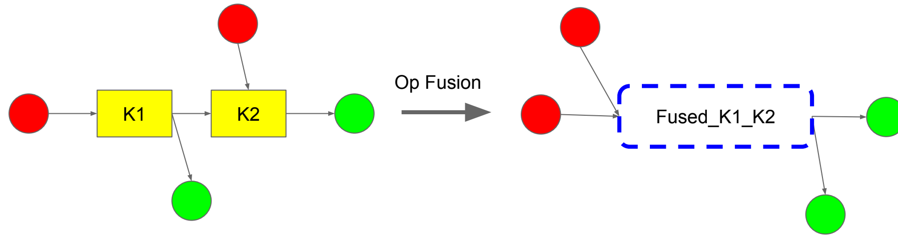

Figure 1: An example of op fusion

computation. When the device has finished with the computation, the outputs are transferred back to the global memory. The memory transactions between the on-chip and the global memory constitute an overhead. Other op launch overheads, depending on the op and device types, may include accessing the device drivers and transferring data between devices and hosts.

One of the many ways to mitigate this problem is to use *op fusion* (also kernel fusion). Op fusion is the process of merging multiple device ops into a single large op. Figure 1 showcases an example of op fusion. In this case, op $K1$ is producing an output which is then consumed by op $K2$. If these two ops are fused, their combined instructions will be sent to the device along with the data for $K1$. When $K1$ finishes on the device, the intermediate data is then immediately used for $K2$, without the need to perform read and write transactions with the global memory, thereby reducing the overhead and boosting the performance.

In addition to the memory traffic reduction, op fusion also improves the data locality on the device, and increases the possibilities for compiler optimization, since the resulting fused kernel has a larger body of instructions [7, 8]. To perform the best fusion in a dataflow graph, we need to have a knowledge of the graph's topology, data size of all the nodes, their output shape, their dependencies and their adjacent nodes. It is also worth mentioning that the maximum number of ops which can fuse together is limited by the device memory. However, the limit may also be imposed by the user in the fusion algorithm to prevent overfusion. Therefore, since the algorithm cannot simply fuse all nodes together, it should carefully choose which nodes to fuse so as to improve the performance as much as possible.

Figure 2 shows an example of how the order of fusion can change the application performance. Let us assume that the fusion algorithm is bound to fuse up to two nodes only. In this case, we have an element-wise multiplication (Mul), a reduction, and a sigmoid function connected to each other as shown. Should the algorithm choose *Reduce* and *Sigmoid* for fusion (left), the performance will not improve much, since the amount of intermediate values transferred to/from the memory will not change significantly, especially if the input tensors of *Mul* are very large, e.g. 10k by 10k. On the other hand, if the fusion algorithm selects *Mul* and *Reduce* (right), the intermediate tensor after the multiplication will stay in the on-chip memory for the *Reduce* op. Therefore, after *Reduce* returns the result, the amount of transferred data has decreased dramatically. Thus, as it can be seen, there should be more priority to fuse the more appropriate nodes. An inefficient fusion algorithm may not improve the performance much and sometimes, might even hurt the performance.

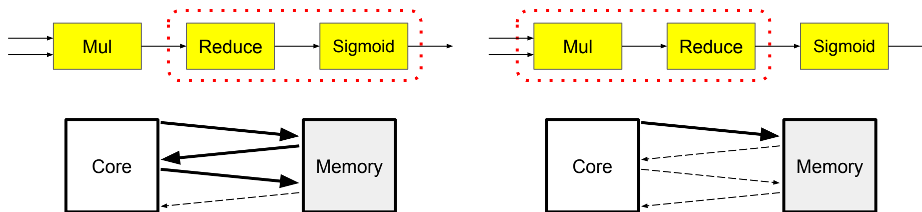

Figure 2: Impact of the order of fusion on memory traffic between the core and the memory: Inefficient fusion, incurring unnecessary memory traffic (left), and better fusion, minimizing the traffic and boosting performance (right)

The baseline fusion scheme is performed using a fixed set of rules, which may or may not work with all the graphs and device topologies. We propose a fusion based on reinforcement learning (RL) which can determine the best fusion procedure and rules during the training on different benchmarks.

3 Design

The process of the baseline fusion in a graph consists of traversing the graph node by node, and attempting to fuse every forward-pass node with one of its input nodes, and backward-pass nodes with one of its output nodes. This process is accumulative, meaning each fused node can fuse with its adjacent nodes again. The baseline fusion traverses the Tensorflow graph in a topological order. It starts from the nodes with no inputs, and after it has processed each node, it will remove its outgoing edges to the other nodes. Any node without any incoming edges left will be put in the queue for processing. This process will continue until all the nodes have been traversed.

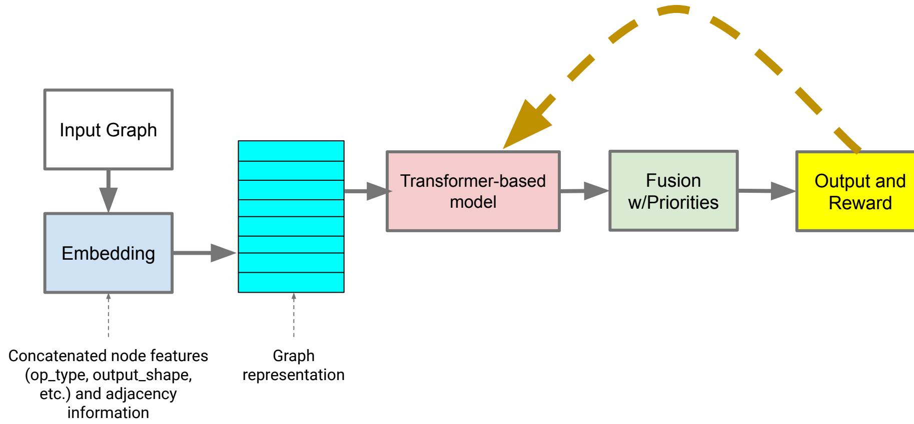

Figure 3: Proposed framework

Since the traversal of the graph determines the order in which nodes will be fused, the resulting fused graph can vary with different traversals. Therefore, an RL which determines the order of the traversal, can change the topology of the fused graph, and subsequently, its performance. Thus, we propose a priority-based RL fusion network, as shown in Figure 3. In this model, the input graph goes through embedding, in which information regarding every node, its features and its neighbors are extracted to be used. The graph representation is then fed to a Transformer-XL attention-based network, which assigns a numerical rank to every node, which represents its priority. The number of priority levels is fixed in the RL network. With more priorities, the ordering of the nodes becomes more precise.

With the priorities determined, we apply priority-based fusion to the whole graph, which involves traversing the graph based on the priority of the nodes, i.e. nodes with the highest priority will be fused first. However, if a node reaches the limit for number of ops (which is a fixed number set in the algorithm to avoid overfusion), it will no longer be allowed to fuse. The process continues until all nodes have been traversed. After the fusion is applied, we estimate the runtime of the resulting fused graph. Based on the runtime, a reward is given back to the model, which will be used to determine new priorities for the nodes.

For the RL network which determines the node priorities for fusion, we used a Proximal Policy Optimization (PPO) [2] algorithm in order to minimize the runtime on a specific topology (using 2 GPUs, 4 GPUs, etc.) during each training step by maximizing the output reward:

$$L_{\pi} = E_{a_{[0:n]} \sim \pi}[\frac{q_{'}(a_n|s_n)}{q(a_n|s_n)} A_{\pi}(s_n, a_n)]$$

$$L_{\pi} = \max_{\pi'} \frac{1}{N} \sum_{n=0, a_n \sim \pi}^{N-1} [\min(\frac{q_{'}(a_n|s_n)}{q(a_n|s_n)}(R - \overline{R}), clip(\frac{q_{'}(a_n|s_n)}{q(a_n|s_n)}, 1-\epsilon, 1+\epsilon)(R - \overline{R}))]$$

4 Evaluation

We use the following benchmarks in our experiments: AmoebaNet [3], Transformer-XL [6], NMT [4], and RNNLM [5]. We use TensorFlow Performance Simulator [1] to estimate the runtime for the following cases: with no fusion, with default fusion, finding the fusion via simulated annealing and via the proposed RL-based priority fusion. Table 1 showcases the resulting speedups normalized with respect to the no-fusion case.

Speedup (normalized)	Default fusion	Simulated annealing	Priority fusion
AmoebaNet (2GPU)	27.38	27.86	**28.3 (+0.92)**
AmoebaNet (4GPU)	25.47	25.59	**26.25 (+0.78)**
NMT (2GPU)	2.82	3	**3.19 (+0.37)**
NMT (4GPU)	-0.89	5.34	**12.03 (+12.92)**
NMT (8GPU)	10.47	10.47	**12.65 (+2.18)**
RNNLM (4GPU)	1.04	1.06	**1.23 (+0.19)**
RNNLM (8GPU)	-2.39	-2.38	**-2.27 (+0.11)**
Transformer-XL (2GPU)	24.27	25.1	**28.51 (+4.24)**
Transformer-XL (4GPU)	17.05	19.32	**19.99 (+2.94)**
Transformer-XL (8GPU)	21.66	26.25	**31.48 (+9.82)**

Table 1: Speedup of each fusion policy normalized to the no-fusion case (reported in %). The number in the parantheses is the improvement of our work over the default fusion.

As it can be seen, our proposed priority fusion outperforms the default fusion policy and simulated annealing. There are cases, however, that the RL does not perform as well, especially in cases where the model is not memory-bound, such as RNNLM.

5 Related Works

Loop fusion is a classical compiler technique that merges multiple loops into one to improve data locality [9, 10, 11, 12, 13]. Inspired by the classical loop fusion techniques, kernel fusion has been proposed on GPUs [7, 8, 14] that fuses two GPU kernels to eliminate redundant data operations across kernels, reduce data movement, and improve data temporal locality. Operator fusion in machine learning systems, such as TensorFLow XLA [1] and TVM [15], fuses ops based on implicit dependency defined by the dataflow graph. Most of the existing work focuses on heuristic-based approaches including cost analysis and critical path analysis to drive fusion decisions [16], and has not yet considered reinforcement learning.

Previous work has shown that machine-learned heuristics can sometimes outperform hand-crafted compiler optimization. Leather et al. [17] uses genetic algorithm to tune features, such as the loop nested level in the loop unrolling pass. Chen et al. [18] proposes to use gradient-boosted tree and TreeGRU to build a domain-specific statistic cost model to optimize the implementation of tensor operators. Recent works demonstrate reinforcement learning as an effective approach for device placement [19, 20] and to reduce peak memory usage [21] on a TensorFlow graph. In this work, we propose an RL-based approach to improve the op fusion.

6 Conclusion

We propose a priority-based operation fusion with an end-to-end deep reinforcement learning model. The proposed network consists of a graph neural network that encodes the input graph and a policy network that generates fusion priorities. We use a sample-efficient PPO to optimize the network and achieve an average of 3.4% speedup compared to Tensorflow default fusion, over a wide range of tasks in computer vision, NLP, and speech.

[1] Publication in progress.

References

[1] Google. XLA: Optimizing Compiler for TensorFlow. `https://tensorflow.org/xla`, 2018.

[2] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[3] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018.

[4] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144, 2016.

[5] Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

[6] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G. Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *CoRR*, abs/1901.02860, 2019.

[7] Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for gpus using kernel fusion/fission. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2433–2442. IEEE, 2012.

[8] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. Automatic kernel fusion for image processing dsls. In *Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems*, pages 76–85. ACM, 2018.

[9] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1982. AAI8303027.

[10] Randy Allen and Ken Kennedy. Vector register allocation. *IEEE Trans. Comput.*, 41(10):1290–1317, October 1992.

[11] G. Gao, R. Olsen, V. Sarkar, and R. Thekkath. Collective loop fusion for array contraction. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 281–295, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[12] Chen Ding. *Improving Effective Bandwidth Through Compiler Enhancement of Global and Dynamic Cache Reuse*. PhD thesis, Rice University, Houston, TX, USA, 2000. AAI9969248.

[13] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, UK, 1994. Springer-Verlag.

[14] Guibin Wang, Yisong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, GREENCOM-CPSCOM '10, pages 344–350, Washington, DC, USA, 2010. IEEE Computer Society.

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 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, Carlsbad, CA, October 2018. USENIX Association.

[16] Guoping Long, Jun Yang, Kai Zhu, and Wei Lin. Fusionstitching: Deep fusion and code generation for tensorflow computations on gpus. *arXiv preprint arXiv:1811.05213*, 2018.

[17] Hugh Leather, Edwin Bonilla, and Michael O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 81–91. IEEE Computer Society, 2009.

[18] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*, pages 3389–3400, 2018.

[19] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *ICML*, 2017.

[20] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. *ICLR*, 2018.

[21] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Regal: Transfer learning for fast optimization of computation graphs. *arXiv preprint arXiv:1905.02494*, 2019.