Defeating the Curse of Dimensionality to Scale JIT Fusion

Jonathan Raiman
Dali
San Francisco, CA
jonathan@dali.ml

Abstract

The growing computational needs of many large scale machine learning applications have motivated the need for better JIT Fusion and graph manipulation tools that are able to improve utilization or stretch the on-chip memory capabilities through algorithmic tricks such as pipelining and recomputation. While there is strong evidence that these techniques can help keep up with the computational demands, they are typically hand written and require domain expertise in the internals of machine learning frameworks, GPU programming, and computational graph manipulation. Recent work, such as AutoTVM, FlexFlow, or Dali have shown that many of these optimizations can be automatically discovered and applied by the framework without human labor or expertise. Unfortunately, many of the proposed algorithms cannot handle fusion across entire deep networks or require hours of auto-tuning, and as we remark in this paper, choosing what operations to fuse is equally challenging. We investigate ways to address limitations to the scaling of JIT Fusion and code generation through the addition of a more robust fusion boundary detector and variable elimination during code generation key to machine learning graph optimizations. We demonstrate the impact of these changes by showing how JIT Fusion and graph optimizations are able to accelerate deep Transformers and RNNs by improving the Dali framework and compiler. . We compare our approach to TensorFlow, TensorFlow with XLA, and PyTorch, and observe a $1.23 - 2.13\times$ speedup over TensorFlow, and $1.08 - 1.92\times$ speedup over PyTorch.

1 Introduction

The exponential rise in computational requirements in the largest machine learning experiments [1] presents an important challenge for future machine learning systems that wish to keep up with demand. One solution to improving hardware utilization comes through the use of optimizations such as pipelining [2] or gradient checkpointing [3, 4], and the use of task-optimized code via autotuning [5] or model-based code generation [6, 7, 8]. In order to benefit from these improvements, human labor and expertise is needed to implement pipelining, tradeoff memory and computation when recomputing, while autotuning requires hours of profiling [7, 5, 9]. Learning or specifying a model has emerged as a portable and fast option for adapting optimizations to new cases without human intervention or significant retraining or search time [10, 7, 6]. However, the search-based techniques in Dali [6] have only been demonstrated on static CNNs, where a greedy JIT Fusion strategy can lead to poor operation coupling and slower execution. Furthermore, code generation relies on an A* [11] algorithm whose complexity grows exponentially relative to search depth.

In this work, we propose to improve Dali's optimization capabilities by introducing a smarter JIT Fusion strategy. We also enable Dali to discover and eliminate redundant variables during code

Workshop on Machine Learning for Systems at 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

generation to shrink the search space. We empirically validate the scalability of these changes on a CNN, Transformer, and mLSTM. Concretely our key contributions are the following:

1. We show how to scale JIT Fusion and code generation to larger models and find that this approach now outperforms PyTorch and TensorFlow. A new fusion strategy lets us find useful boundaries in the computation graphs. This change enables support for deeper networks while finding practical boundaries to prevent naive coupling of operations.
2. We add discovery and elimination of loop variables to code generation A* [11] search process in Dali, enabling it to shrink the solution search space by several orders of magnitude and handle the larger kernels found in Transformers and mLSTMs.
3. We provide results showing cached graph optimization reduces by a factor of $20 - 257\times$ the transformation cost, suggesting graph optimizations are applicable to dynamic graphs ranging from static CNNs to Transformers and RNNs.

2 A* Fused CUDA Kernel Generation

2.1 Approach overview

Fusing operations together and generating task-specific code is a powerful tool for accelerating computation graphs by removing temporary variables and reducing slow memory movement [12, 13, 14]. In this work we build upon the JIT Fusion optimizations in Dali's compiler [6], which rely on having each operation indicate whether they are capable of generating code that can be concatenated and compiled into a larger fused kernel. In this prior work, JIT Fusion operates in two stages: (1) combine a series of operations into a larger compilation unit, (2) make parallelism and design decisions during code generation to compile a fast kernel.

In the first stage, to find a JIT subgraph to compile, Dali considers connected subtrees of the full computation graph composed of JIT-able expressions. If the parent of a JIT-able expression is also JIT-able it will combine the two operations into a single computation unit through fusion. The combined set of operations can now be thought of as a single operation. This process is repeated until all JIT-able expressions are fused to their neighbors.

The second stage collects all loops and code design choices from the nodes inside the JIT subgraph. A model of the GPU's memory and parallelism informs an A* search [11] through different ways of designing the code ranked by a cost function similar to the approach taken by [15, 16].

Through this approach it is possible without human intervention to obtain efficient code for operations composed of many primitives such as Softmax, Layer Norm [17], Batch Norm [18], Attention [19, 20], etc. However, as we remark in this work (Figure 1), combining operations in a greedy fashion (stage 1) leads to two significant problems: (1) the code generation (stage 2) solution space grows exponentially with the number of variables, and (2) the generated code becomes over-constrained and cannot be properly parallelized.

2.2 Fusion strategy

When constructing computation graphs, all parent-child relations involving JIT-able operations are candidates for fusion. Using fusion, we can write $\mathrm{Softmax}$, $\mathrm{Attention}$, $\mathrm{CrossEntropy}$, or any other numerical operations using primitives such as $\mathrm{addition}$, $\mathrm{subtraction}$, $\mathrm{division}$, $\mathrm{reductions}$, etc. and let the code generation system deal with how to plan, reuse, store, and combine these operations to compute the result. However, when planning our computation, we discover many constraints on the parallelism options for each reduction or assignment loop. In cases where we choose to fuse many operations together, we reach over-constrained situations that lead to hardware under-utilization.

Avoiding excessive constraints and choosing the right subsets of operations to fuse is crucial for performance and utilization. In many cases, commonly factored operations such as loss computation, normalization, or affine transformations involve 1-3 inputs. An entire computation graph might actually involve 100s of unique inputs and weight matrices. Since many of these operations are connected through parent-child relationship, we are tempted to fuse 10s to 100s of operations into a single compilation unit to reduce the number of kernels used. We argue that excessive fusion has several downsides: (1) with more stages in the computation, new constraints on parallelism arise,

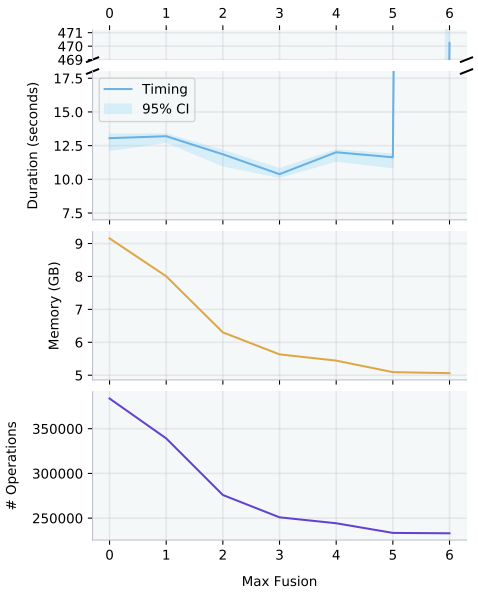

Figure 1: Impact of $F_{\max}$ heuristic on runtime and memory when training a Transformer.

Table 1: Optimization Overhead (% Total)

Step	CNN	Transformer	mLSTM
Optimize	0.64%	0.11%	0.052%
CT[a]	1.88%	1.88%	2.69%
Hash	2.33%	1.56%	1.71%
Compute	95.85%	96.44%	95.54%

[a]Cached Transformation

Table 2: Runtime & Memory Usage (MB)

Framework	Time (s) / Epoch ($\mu \pm \sigma$)	MB / Step
Task	CNN - MNIST	
Dali w/o JIT Fusion	1.29 ± 0.0050	376.62
Dali	1.11 ± 0.0034	363.38
PyTorch	2.14 ± 0.0037	913.00
TensorFlow	2.37 ± 0.0239	1178.00
TensorFlow + XLA	1.53 ± 0.0368	959.00
Task	Transformer - 1 Billion Word	
Dali w/o JIT Fusion	12.518 ± 0.5702	9165.89
Dali	8.735 ± 0.0464	4276.50
PyTorch	9.476 ± 0.2425	4003.76
TensorFlow	11.123 ± 0.1893	4059.63
Task	mLSTM - Amazon Reviews	
Dali w/o JIT Fusion	95.326 ± 0.1528	9581.12
Dali	93.342 ± 0.0889	7102.13
PyTorch	114.868 ± 0.0240	4967.00
TensorFlow	115.120 ± 0.1336	4363.00

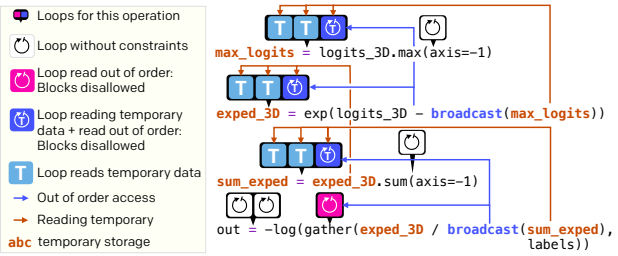

Figure 2: `sparse_softmax_cross_entropy` constraint graph.

leading in some cases to non-performant code, (2) the additional loops and variables increase the search space for code generation dramatically, (3) the cap on CUDA kernel launch arguments places an upper bound on the largest fused kernel, or requires compacting arguments during kernel launch.

We propose a simple heuristic that is computed greedily to find smaller subsets of operations to fuse:

1. Fuse JIT operations bottom-up; record how many unique input arrays they require.
2. If the fusion of a new JIT operation would increase the number of unique inputs past the maximum number $F_{\max}$, create a new compilation unit with this operation as a leaf. Otherwise, continue extending.

We measure empirically the effect of $F_{\max}$ on runtime, memory consumption, and number of executed operations when training a Transformer [20] and report our findings in Figure 1. As expected, increasing $F_{\max}$ leads to more aggregation of JIT operations, and a decrease in the number of executed operations. Fusion decreases memory consumption by allowing chaining of operations without temporary variables. As described earlier, fusing too many operations together can lead to non-performant code as is visible with $F_{\max} = 6$, where runtime jumps from 11 to 470 seconds. Surprisingly, we observe a runtime Goldilocks principle: setting $F_{\max}$ to 3 gives the best results[1].

2.3 Variable Coupling and Elimination

A common problem Dali faces when handling large scale models is generating kernels involving many loops, since loop count exponentially grows the search space. Many of the variables in these loops

[1]We see this effect repeated for the mLSTM and CNN.

have constraints and dependencies that make them redundant, thus by coupling their assignments with other variables we can reduce the search space.

Loop redundancy exists implicitly: two iterations over the same batch dimension of an array will be constrained to run on the same block to respect the order of read and writes. The most efficient use of our GPU resources will involve these loops sharing the same parallelism strategy (e.g. use blocks or threads in both cases). We can make this implicit coupling explicit by looking at all parent-child relationships in our constraint graph and derive which variables are actually redundant[2].

The value of variable coupling and elimination is visible when computing the cross entropy between a set of labels and logits, a commonly used operation. Chaining `probs = softmax(logits)` and `sparse_cross_entropy(probs, labels)` involves only two inputs `logits` and `labels`, so all operations can be fused under the $F_{\max}$ condition. After looking for reuse and using temporary storage, 4 assignments are used, and 14 loops can be parallelized. Dali annotates the loops with data dependencies to discover constraints (see Figure 2). In this example broadcast operations perform out of order access on their input. Because a different block might have written to these inputs during evaluation, "block parallelism" is disallowed in the loops connected to broadcast. The search space starts with 4 million solutions, and drops to 512 after detecting that 7 loops are coupled and can be ignored. Guided search examines 155 candidates and reaches a faster solution than the one written by in PyTorch and TensorFlow, using the same scheduling as one written by human programmers [3].

3 Results

We investigate the scalability and benefits of using a fusion strategy and variable elimination on neural network training tasks through a C++ implementation of Dali [6]. We summarize our timing experiments regarding graph transformation on 3 tasks and architectures: image classification with a CNN on MNIST (28x28 grayscale), subword language modeling with a Transformer [20] , and character language modeling using the mLSTM from [21]. We measure memory usage and runtime per epoch over (1) 100 epochs of training a CNN, (2) 10 epochs of training a Transformer, and (3) 10 epochs of an mLSTM. Full hyperparameters and setup given in Appendix A. We compare six configurations: Dali with and without JIT Fusion, TensorFlow 1.13.1 with and without XLA[4], and PyTorch 1.1 [22]. We report our results in Table 2 and find that we always obtain a speedup over TensorFlow and PyTorch. Specifically we see a $1.92/2.13\times$ speedup over PyTorch/TensorFlow with a CNN, a $1.23/1.23\times$ speedup over PyTorch/TensorFlow with an mLSTM, and a $1.08/1.27$ speedup over PyTorch/TensorFlow using a Transformer.

We time fine-grained steps of the system in Table 1 (exact timings in Appendix B). Hashing plus cached transformation is $\approx$ 20-$257\times$ faster than the initial optimization pass. Optimization overhead remains modest across models.

4 Conclusion

We have shown how improvements to Dali's compiler allow it to scale to deeper networks and outperform existing computational frameworks on a CNN, Transformer, and mLSTM. These gains can be attributed to two new capabilities for scaling JIT Fusion and graph optimizations to large computational graphs where the abundance of fusion opportunities can cripple greedy computation graph compilers. The first capability involves the addition of JIT Fusion boundaries based on a simple heuristic that accelerates the execution of a CNN, Transformer, and mLSTM. We hypothesize that controlling the number of input arrays in a kernel is correlated with the likelihood of introducing incompatible parallelism constraints, which would force the generation of slower code. The second capability is the discovery and elimination of variables in the kernel prior to performing an A* search for the most parallel solution. In prior work [6], the number of variables could grow to 50 or more, leading to intractably large search spaces. We show how detecting loop variable redundancy lets us

[2]We detect loop redundancy by checking if two loops are descendants of each other, have the same symbolic size, and equal access to using threads or blocks.

[3]Leading "batch" dimensions, threads elsewhere.

[4]XLA support is currently experimental, so we were only able to obtain results with XLA on the CNN example. The Transformer and mLSTM segfault after compilation.

shrink the search space by 6 orders of magnitude, enabling code generation in deeper networks where more operations fuse.

As future work, we have shown the importance of fusion boundaries, and we expect a data-driven or learnt policy following [23, 24, 7] would perform even better at choosing which operations to fuse. We also see in similar subgraph detection[5] an opportunity to massively improve optimization reuse.

References

[1] Dario Amodei and Danny Hernandez. AI and Compute. `https://openai.com/blog/ai-and-compute/`, 2018.

[2] Yanping Huang, Yonglong Cheng, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv preprint arXiv:1811.06965*, 2018.

[3] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*, volume 105. Siam, 2008.

[4] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016.

[5] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[6] Jonathan Raiman. Dali: Lazy compilation of dynamic computation graphs. In *Workshop on Systems for Machine Learning and Open Source Software at NeurIPS 2018*, 2018.

[7] 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.

[8] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018.

[9] Paul Barham and Michael Isard. Machine learning systems are stuck in a rut. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 177–183. ACM, 2019.

[10] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. Placeto: Efficient progressive device placement optimization. In *NIPS Machine Learning for Systems Workshop*, 2018.

[11] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[12] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE, 2012.

[13] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. Optimizing cuda code by kernel fusion: application on blas. *The Journal of Supercomputing*, 71(10):3934–3957, 2015.

[14] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing, 2018.

[15] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*, 2013.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: end-to-end compilation stack for deep learning. In *SysML Conference*, 2018.

[17] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *stat*, 1050:21, 2016.

[5] It should be possible to reuse optimization passes done on computation graphs from different RNN unrolls.

[18] Sergey Ioffe. Batch renormalization: Towards reducing minibatch dependence in batch-normalized models. In *Advances in neural information processing systems*, pages 1945–1953, 2017.

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[21] Alec Radford, Rafal Jozefowicz, and Ilya Sutskever. Learning to generate reviews and discovering sentiment. *arXiv preprint arXiv:1704.01444*, 2017.

[22] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration. *PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration*, 6, 2017.

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

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

[25] Ashish Vaswani, Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018.

A Hyperparameters

In this section we include the CNN architecture in Table 3, and hyperparameters for the experiments: CNN in Table 4, Transformer in Table 5, and mLSTM in Table 6. All experiments are run on a 12-core 3.60GHz Intel i7-6850K CPU with an NVIDIA Titan X Pascal, with all frameworks using CUDA 10.1 and CuDNN 7.5.1 primitives [1].

Table 3: CNN Architecture

Layer	Window/Strides	Input channels	Output channels
Convolution + Relu	$(5,5)/(1,1)$	1	64
MaxPool	$(2,2)/(2,2)$	64	64
Conv + Relu	$(5,5)/(1,1)$	64	64
Convolution	$(2,2)/(2,2)$	64	64
FC + Relu		3136	1024
FC + Softmax		1024	10

Table 4: CNN Hyperparameters

Dataset	MNIST 28x28
Batch Size	256
Optimizer	SGD

Table 5: Transformer Hyperparameters

Dataset	1 Billion Word [2][a]
Timesteps	100
Batch Size	32
Optimizer	SGD
Hidden Size	512
Intermediate Size	1024
Attention Heads	4
Number of Layers	8
Activation	GELU [3]
Examples/epoch	2048
Steps/epoch	64

[a] We use the `languagemodel_lm1b32k` problem from tensor2tensor [25].

Table 6: mLSTM Hyperparameters

Dataset	Amazon Reviews [4]
Timesteps	256
Batch Size	32
Optimizer	SGD
Hidden Size	4096
RNN Weight Norm	Yes
Output Weight Norm	No
Vocabulary Size	256
Embedding Size	64
Examples/epoch	2048
Steps/epoch	64

B Timing Results

Timing results when measuring overhead for optimization steps given in Table 7

Table 7: Optimization Overhead Time/Call ($\mu \pm \sigma$)

Step	CNN	Transformer	mLSTM
Optimization	41.11ms $\pm$ 320μs	183.3ms $\pm$ 1.91ms	794.9ms $\pm$ 10.3ms
Cached Transformation	106.31 μs $\pm$ 1.00μs	3.041ms $\pm$ 8.50μs	40.56ms $\pm$ 150μs
Expression hash	53.36 μs $\pm$ 0.160μs	2.520ms $\pm$ 7.03μs	25.80ms $\pm$ 74.5μs
Computation	4.366ms $\pm$ 14.3μs	155.8ms $\pm$ 3.25ms	1.439s $\pm$ 57.8ms

References

[Appendix1] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[Appendix2] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

[Appendix3] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[Appendix4] Julian McAuley, Rahul Pandey, and Jure Leskovec. Inferring networks of substitutable and complementary products. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*, pages 785–794. ACM, 2015.