Dali: Lazy Compilation of Dynamic Computation Graphs

Jonathan Raiman
OpenAI
San Francisco, CA
raiman@openai.com

Abstract

Computation graphs underpin many of the large machine learning projects of our time, from computer vision, to machine translation, speech synthesis, or facial recognition. Graph optimizations offer the ability to make many of these applications run more efficiently, adapt to a variety of hardware platforms, and tighten the iteration cycle from prototype to production. However, generating these graph optimizations requires extensive domain knowledge about hardware and machine learning, and applying these transformations is generally considered too costly to run online.

Due to these difficulties, graph optimizations have only seen applications in static graph frameworks (e.g. TensorFlow, CNTK, Theano, MXNet), while the flexibility and ability to use native control flow has prevented dynamic graph frameworks (e.g. PyTorch, DyNet, Chainer) from benefiting.

We seek to address these challenges by operating on lazily compiled graphs instead. This approach retains the imperative style of dynamic graphs, while also enabling the ability to apply graph optimizations at each evaluation point. We achieve this with Dali through two key components: 1) we amortize the optimization cost by caching our graph optimization passes into Transformation Graphs that are reusable, 2) we generate fused CUDA Kernels using an A* algorithm and model of the GPU's memory which enables fast exploration of the code generation landscape. We offer preliminary results indicating that our cached transformations are inexpensive and can be applied to automatically replace operations by more efficient primitives or dynamically generated CUDA kernels on the fly, and a $1.79\times$ speedup over TensorFlow while training an image classification task.

1 Introduction

Computation frameworks have greatly expanded the ability to transform research ideas into working prototypes and deploy models in production. Moreover, many of these frameworks enable cross-usage of GPUs, CPUs, and other devices, enabling the use of massive computational resources across a variety of application areas without requiring GPU or HPC domain knowledge, or even the ability to port models from one framework to another [1]. Perhaps the most important aspect of a shared computation graph language is the ability to apply graph optimization incorporating domain knowledge from a variety of experts and sources to bear on any problem expressed within this representation. Graph optimizations have proven extremely valuable in a variety of cases by providing memory or speed optimizations [1, 2, 3, 4], or automatic code generation using a JIT compiler and an autotuner (e.g. Tensor Comprehensions, XLA, PyTorch Tracing) [5, 6, 7]. There exists rare cases

[1] Support for ONNX (`https://onnx.ai`) representations can allow programmers to move their graph from one framework to another, or run other optimizations on this intermediate representation.

Preprint. Work in progress.

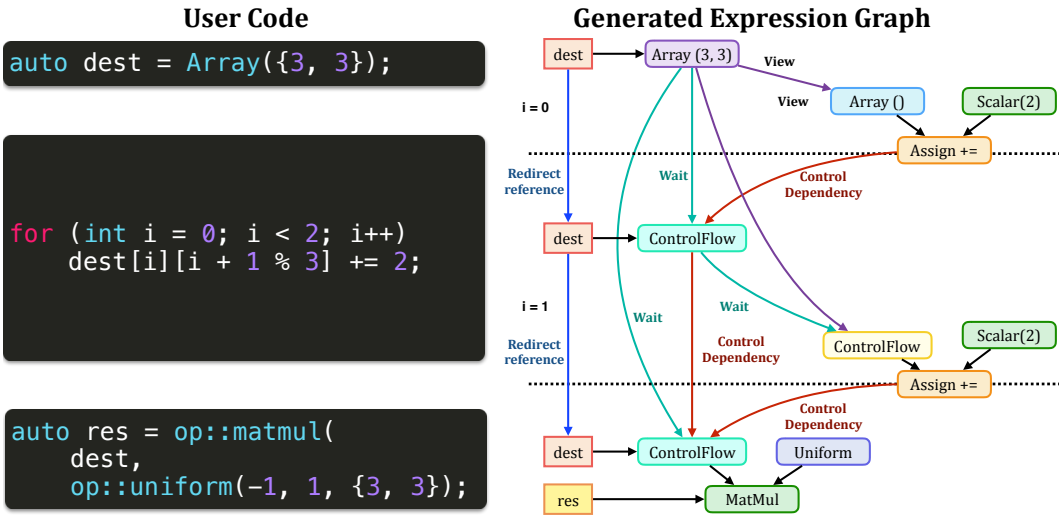

Figure 1: Example of a lazy dynamic graph modifying memory while building up an Expression Graph that tracks a partial ordering of events. The dashed line indicates when the variable `dest` is redirected to reflect a memory transaction.

of on-the-fly optimizations [8], however the vast majority of these graph optimizations appear to be too costly to run online, thereby limiting their applicability to static graphs or during conversion to production.

In this paper we introduce Dali, a lazy compiler for dynamic computation graphs, which enables efficient application of graph optimizations while retaining the imperative style and expressivity of dynamic approaches. Our solution takes advantage of the observation that most machine learning applications rely on a small set of computation graphs that are evaluated $10^6 - 10^8$ times over the course of training: we compute a graph optimization once, and cache a reusable transformation that is cheaply applied on the next occurrence.

Concretely our key contributions are the following:

1. A lazy compiler that integrates a variety of optimization passes in an extensible optimization registry (operation conversion to CuDNN [9], sub-expression elimination, device placement, JIT kernel generation and fusion across reductions, etc.). We overcome the cost of graph optimization by caching our optimization using a method we name Transformation Graphs.
2. We provide preliminary results that our cached graph optimization approach reduces by a factor of $257\times$ the transformation cost, suggesting dynamic graphs are applicable to graph optimizations.
3. We provide a method for generating fused JIT kernels across multiple operations while only exploring a very limited number of solutions. We rely on an A* algorithm [10] that integrates an admissible heuristic for estimating parallelism and constraints that enforce data-consistency in the kernel.

The rest of this paper is structured as follows. In Section 2 we will explain the behavior of Lazy Dynamic Computation Graphs, in Section 3 we introduce Transformation Graphs, in Section 4 we present our search algorithm for CUDA kernel code generation, in Section 5 we present some numerical results that investigate the overhead of the dynamic graph compilation and a comparison with TensorFlow. Conclusions and directions for future work are given in Section 6.

2 Lazy Dynamic Computation Graph

We record all operations in the graph symbolically, by building up an Expression Graph dynamically as depicted in Figure 1.

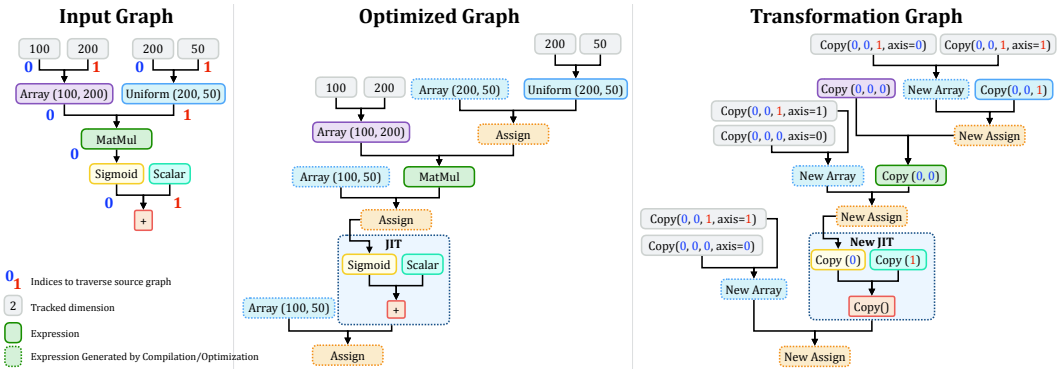

Figure 2: Transformation of an Input Graph into an Optimized Graph, from which we derive a Transformation Graph we cache and can reapply later. In this example we multiply two matrices together, then apply a Sigmoid nonlinearity, and add a scalar to the result. Our compilation/optimization passes ensure that all the operations have a destination to store intermediary results, while also combining the Sigmoid and scalar addition into a single operation marked JIT. From the "Optimized Graph" we form the $Q_{\text{placeholder}}$ graph as described in Section 3.1, thereby assigning to each node a method for reconstructing its expression.

We track operations that modify memory by replacing an object whose memory was modified by the associated transaction operation (assignment, read, synchronization point, etc..) last applied to it. This substitution ensures that the order of events imperatively declared by the user is translated to the same partial order in the computation graph.

We also keep track of the effect of memory view operations on strides and dimensions using symbolic stride and dimension objects. This information can later be used within cached optimization passes as we will demonstrate in the next section.

3 Transformation Graph

Application and testing of many optimizations against a graph can be costly. In our approach we have $O(N \cdot k)$ where N is the number of nodes and k the number of optimizations (applied in order of their priority). If we knew ahead of time the effect of the composition of these k transformations on the input graph, we could sidestep the iterative application. We can achieve this by hashing the graph in bottom-up order, taking care to only include in our hash components that affect the nature of the computation, not its exact contents (e.g. hash the type of operations used, the dimensionality, etc.., but not the memory contents or exact dimensions). In this way we now have a hash for some given input graph P structure. We can now apply transformations iteratively until we have produced some output graph $P \rightarrow Q$.

To recover how Q was constructed given a similar input graph P we must trace the effect of each transformation in the optimization passes. However, tracing through these passes can be error-prone and complex, particularly if new nodes are inserted or deleted without explicitly indicating the transition to the tracer. Our solution instead relies on constructing a placeholder target graph $Q_{\text{placeholder}}$ which has the same graph structure as Q, but with all nodes now containing reconstruction information.

3.1 Reconstruction

Each node of $Q_{\text{placeholder}}$ contains one of two types of reconstruction information:

1. Retrieval from the input graph P: if a node in Q was left unchanged from its state in P, then we store a path from the root of the input graph P to the nodes[2]. This is marked as "Copy" in Figure 2.

[2]Each expression contains zero or more arguments, so a path can be generated by storing a list of indices indicating which argument of a node we must traverse to arrive at the node we want to retrieve.

2. A function that constructs the node anew along with some recovery information. For instance: if temporary storage is created during our optimization pass to store intermediary results, then we save instructions stating “New Array", with a shape given by retrieving the proper strides and dimensions from the input graph P. Since we track the symbolic graph of Dimension operations, we can apply the same reconstruction strategy to re-transform input dimensions from P into our node's shape.

To reapply our transformation to a new input graph P' that has the same hash as P, we can now iterate through each node in $Q_{\text{placeholder}}$, using P' as the input graph, using the reconstruction information we stored when transforming $P \rightarrow Q$. As a byproduct of optimizing using a cached transformation graph, we removed the dependence on the number of optimization passes k in the computational complexity of our optimization: we now take time $O(M)$, where M is the number of nodes in the output graph Q.

4 A* CUDA Kernel Search

Automatic CUDA Kernel code generation is an active area of research, with many recent approaches relying on evolutionary or random search techniques to optimize the code in a kernel in a black-box fashion until they arrive at a task-specific efficient implementation [3, 5]. While autotuning based approaches have shown impressive performance, the associated hour-long search process makes them harder to apply to the online optimization and dynamic graph setting. Therefore we propose to use a search based strategy that does not require compilation to estimate performance, but instead relies on a hand designed heuristic that aims to maximize parallelism[3].

To design our search space we require that any JIT Expression in our graph that can be fused provides information about the loops and data-traversal it will perform over its inputs. We can aggregate the hierarchy of loops, iterations, and traversals to ensure data-consistency, while also maximizing exposed parallelism.

4.1 Kernel Domain Specific Language

Our approach to representing threads, loop, and iteration semantics is similar to the one taken in Halide and TVM [11, 3]. Each JIT expression annotates its Loops with the different ways it can be parallelized (Blocks, Threads, Blocks and Threads, or Sequential) and whether the loop can be fused with another loop (e.g. iterating across 2 or more dimensions simultaneously). By observing the JIT subgraph we can also discover a data access hierarchy that respects the read and write access across blocks.

Iterators are stored as child nodes of Loops and can also be annotated by stating that they will be used non sequentially (indicating the possibility of violation of data-consistency by accessing data generated by a different CUDA block). If this iterator is used to access an intermediary result in the kernel, we disallow block-level parallelism over the Loop that created this iterator.

4.2 Search strategy

Once we have collected all the loops and data-consistency constraints in the problem, we can now apply graph search over the assignment of parallelism types to Loops while avoiding incorrect assignments (e.g. two loops that are nested cannot both rely on blocks to do their iteration, since the sub-expression will only be evaluated when the parent is active, thus the inner loop's block iteration will never fully execute).

Using our heuristic and cost estimate we can now use A* to search through the space of valid solutions and find the best one[10]. In Table 1 we compare the number of heuristic evaluations used, versus the total solutions space, and the number of solutions considered to be data-consistent. As we can see nearly 80% of large composite function's solutions are data inconsistent, and using our search strategy we are able to only explore a small subset of those solutions.

[3]Our heuristic weighs the total number of loops against the number of unique parallelism types exposed, while penalizing the number of loops that have to be run sequentially. Our admissible heuristic sets all unassigned loops to be maximally parallel, thereby providing a valid upper bound on the remaining parallelism opportunities.

Table 1: CUDA Kernel Search

Operation	Variables	# Heuristic calls	# Data-consistent Solutions	Solution Space
Softmax	6	143	214	1024
Layer Norm[12]	7	218	356	1528
Add	1	4	4	4
Strided Add	2	12	16	16

Table 2: Runtime

Framework	Time (s) / Epoch ($\mu \pm \sigma$)	N
Dali	1.32 ± 0.0047	100
TensorFlow	2.39 ± 0.0166	100

While our hand-designed heuristic is benefitial for search, it is not able to adapt to changes in hardware, or observe usage statistics to improve. Therefore we believe future work includes integrating either value-function based learning using policy gradients as was done for device placement by [13, 14], or by simply constructing a bank of statistics that can be regularly updated and used to make the heuristic data-driven. Moreover, we believe that the advantage of our approach is that it can serve as an efficient way of eliminating unwanted candidates before moving to the more costly autotuning approaches for finer grain decisions.

5 Results

To investigate whether dynamic compilation is viable and can be competitive with static or dynamic graph approaches, we have run experiments using a C++ implementation of Dali that includes a lazy dynamic computation graph, Transformation Graphs, and A* based fused JIT kernel generation. In this section we will summarize our timing experiments regarding graph transformation on an image classification Convolutional Neural Network (CNN).

Our model is a CNN with layers described in Table 3. We measure runtime over the last 54,000 images of MNIST (28x28 grayscale images) using a batch size of 256, on a 12-core 3.60GHz Intel i7-6850K CPU with an NVIDIA Titan X Pascal with CUDA 10, CuDNN 7.3.1, using Dali with optimizations, JIT, and CuDNN, and compare to TensorFlow 1.11 [15] with CuDNN.

We run through a 100 epochs of training using both frameworks and observe a $1.35\times$ speedup of Dali over TensorFlow with the same CuDNN primitives in both frameworks (Table 2). We also measure finer grain timings of optimization, cached transformation applications, and hashing in Table 4. In our optimization timing experiments, we note that hashing and transformation is $\approx 257\times$ faster than the initial optimization pass. We also observe that over the course of training, only 3 distinct computation graphs were used (an initial graph that contains the initialization of the weights, one that computed the loss for printing, and one containing the forward and backward passes). These preliminary results suggest that lazy compilation over dynamic graphs can be time efficient given the amortization and repeated use of computation graphs.

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: Optimization Time

Step	Time/Call ($\mu \pm \sigma$)	# Calls/10 Epochs	Percent Total
Optimization	41.11ms $\pm$ 320μs	3	0.64%
Apply Cached Transform	106.31 μs $\pm$ 1.00μs	4217	1.17%
Expression hash	53.36 μs $\pm$ 0.160μs	4220	2.33%
Computation	4.366ms $\pm$ 14.29μs	4220	95.85%

6 Conclusion

We have introduced Dali, a lazy compiler for dynamic computation graphs. We have shown how through the use of caching expressions we are able to take advantage of the repeated occurrence of computation graphs throughout the course of training to amortize the cost of graph optimization by recognizing repeated occurrences. Furthermore we demonstrated that Transformation Graphs can reduce by a factor of $\approx 257\times$ the time taken to apply graph optimizations, while also removing the number of optimization passes k from the computational complexity of the graph transformation. We have also presented a method for generating fused JIT kernels that are aware of data-consistency and can efficiently search over thousands of possibilities, greatly increasing the viability of just in time compilation of kernels without exhaustive search, while reducing the need for hand written CUDA kernels by incorporating knowledge about GPU data-consistency into the search algorithm.

We believe that broadening the applicability of graph optimizations can have a strong impact on a variety of machine learning applications by allowing easy integration of systems domain knowledge without requiring expertise from the end user.

As future work we believe that the graph transformation work presented here could benefit from a smarter search and application strategy, for instance by incorporating data-driven heuristics into the search process discovered either through tuning or by exploiting the value function derived from reinforcement learning the optimization policy following [13, 14]. Similarly, our admissible heuristic for CUDA kernel generation is hand designed, and only seeks to maximize parallelism, while being unaware of the underlying hardware or context: it would also be fruitful to explore how we could retain the ability to prune large portions of the search space, while also using past kernel executions to inform our heuristic.

References

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

[2] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Satish Nadathur, Jakob Olesen, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[3] 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.

[4] Salimans, Tim and Bulatov, Yaroslav. Saving memory using Gradient Checkpointing. `https://github.com/openai/gradient-checkpointing`, 2018. [Online; accessed 19-October-2018].

[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] Tensorflow. XLA Overview. `https://www.tensorflow.org/performance/xla/`, 2018. [Online; accessed 19-October-2018].

[7] Torch Contributors. Torch Script. `https://pytorch.org/docs/master/jit.html`, 2018. [Online; accessed 19-October-2018].

[8] Graham Neubig, Yoav Goldberg, and Chris Dyer. On-the-fly operation batching in dynamic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3971–3981, 2017.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.