Dali: Lazy Compilation & Kernel Fusion in 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 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 reusable Transformation Graphs, 2) we generate fused CUDA kernels using an A* search over a basic cost model of the GPU’s memory and parallelism and an admissible heuristic allowing for rapid exploration of the code landscape. We offer preliminary results on Dali while training an image classification task: 1) we find that cached transformations are inexpensive and can automatically replace operations by more efficient primitives or dynamically generated CUDA kernels on the fly, 2) fusing operations reduces memory usage by 45% and runtime by 9%, 3) we observe a $1.79 \times$ speedup over TensorFlow.

1 Introduction

Computation frameworks have greatly expanded the ability to transform research ideas into working prototypes and deploy models in production. 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 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

Preprint. Work in progress.
Application and testing of many optimizations against a graph can be costly, and typically grows as $O(N \cdot k)$, where $N$ is the number of nodes and $k$ the number of optimizations.
Our solution relies on caching the transformations applied to the input graph, and reapplying them to graphs that have the same structure and operations, but different contents. To achieve this we hash the input graph $P$, then iteratively optimize $P$ into an output graph $Q$.

To recover the transformation that takes $P$ to $Q$ we must trace the effect of each transformation in the optimization passes. Because tracing through passes can be complex or error prone, we instead construct a placeholder target graph $Q_{\text{placeholder}}$ which has the same graph structure as $Q$, but with all nodes now containing reconstruction information. 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 node. Marked “Copy” in Fig. 2
2. A function that constructs the node anew along with some recovery information

To reapply our transformation to a new input graph $P’$ with the same hash as $P$, we iterate through each node in $Q_{\text{placeholder}}$, using $P’$ as the input graph, and the reconstruction information 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 any future application of the same optimization: we now take time $O(M)$, where $M$ is the number of nodes in the optimized output graph $Q$.

3 A* Fused 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]. Autotuning approaches have shown impressive performance, but the hour-long search process makes them impractical for online optimization/dynamic computation graphs. We propose to use a search based strategy that relies on a basic cost model of the GPU’s memory and parallelism, which does not require compilation to estimate performance, but instead uses a hand designed heuristic that aims to maximize parallelism.

To construct the JIT subgraph, we consider each JIT-able Expression which provides information about the for loops and access pattern over its inputs. This information lets us find a safe way of combining multiple reductions and elementwise operations into a single kernel while remaining data-consistent. Loops are annotated with the available ways to parallelize them: Blocks, Threads, Blocks and Threads, or None (e.g. run sequentially).

<table>
<thead>
<tr>
<th>Framework</th>
<th>Time (s) / Epoch ($\mu$ ± $\sigma$)</th>
<th>Allocated MB / Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dali w/o JIT Fusion</td>
<td>1.44 ± 0.0035</td>
<td>513.33</td>
</tr>
<tr>
<td>Dali</td>
<td>1.32 ± 0.0047</td>
<td>343.89</td>
</tr>
<tr>
<td>Dali w/o temp</td>
<td>1.37 ± 0.0069</td>
<td>343.86</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>2.39 ± 0.0166</td>
<td>1178.00</td>
</tr>
</tbody>
</table>

Figure 3: Epoch time as thread vs block usage preference changes.

Table 1: Runtime & Memory Usage

---

1. Hashed 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 operation type, dimensionality, but not the memory contents or exact dimensions).
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.
3. E.g. 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 $P$.
4. 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.
5. Iterators belonging to a loop can be annotated to state they will access data non sequentially, indicating a possible violation of data-consistency if the read data was generated in a different CUDA block.
Table 2: Optimization Time

<table>
<thead>
<tr>
<th>Step</th>
<th>Time/Call ($\mu \pm \sigma$)</th>
<th># Calls/10 Epochs</th>
<th>Percent Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimization</td>
<td>41.11ms ± 320$\mu$s</td>
<td>3</td>
<td>0.64%</td>
</tr>
<tr>
<td>Apply Cached Transform</td>
<td>106.31 $\mu$s ± 1.00$\mu$s</td>
<td>4217</td>
<td>1.17%</td>
</tr>
<tr>
<td>Expression hash</td>
<td>53.36 $\mu$s ± 0.160$\mu$s</td>
<td>4220</td>
<td>2.33%</td>
</tr>
<tr>
<td>Computation</td>
<td>4.366ms ± 14.29$\mu$s</td>
<td>4220</td>
<td>95.85%</td>
</tr>
</tbody>
</table>

Once we have collected all loop variables and data-consistency constraints of the problem, we use A* to find the lowest cost kernel [10]. Empirically, over 80% of the solutions to generating Softmax or LayerNorm [11] are data inconsistent, and our heuristic only evaluates the cost of 13-14% of the total solution space, thereby eliminating many unwanted candidates. We also compared the effect of weighing different aspects of our heuristic and report our results in Figure 3. Specifically we simulate whether threads will parallelize better than blocks, thereby deciding to allocate more variables to use threads instead, and vice-versa. The fastest kernels are found by preferring blocks over threads, while preferring threads to blocks increases runtime $4.23 \times$.

4 Results

To investigate whether dynamic compilation is competitive with current approaches, we use a C++ implementation of Dali with a lazy dynamic computation graph, Transformation Graphs, and A* based fused JIT kernel generation. We summarize our timing experiments regarding graph transformation on an image classification Convolutional Neural Network.

We measure memory usage and runtime per epoch over 100 epochs of training on the last 54,000 images of MNIST (28x28 grayscale) using a batch size of 256, on a 12-core 3.60GHz Intel i7-6850K CPU with an NVIDIA Titan X Pascal. We compare four configurations: Dali with or without JIT Fusion, Dali with no temporary storage in the generated code, and TensorFlow 1.11. We report our results in Table 1 and observe a $1.79 \times$ speedup of Dali over TensorFlow [12] while the same CUDA 10, CuDNN 7.3.1 primitives [9] in both frameworks (Table 1).

We also time finer grain steps of the graph optimization in Table 2. In those measurements, we note that hashing and transformation is $\approx 257 \times$ faster than the initial optimization pass. Only 3 distinct computation graphs were used over the course of training. These preliminary results suggest that lazy compilation over dynamic graphs can be time efficient given the amortization and repeated use of computation graphs.

5 Conclusion

We have introduced Dali, a lazy compiler for dynamic computation graphs. We have shown how we can amortize the cost of graph optimization by recognizing repeated occurrences, reducing transformation time by a factor of $\approx 257 \times$, while also removing the number of optimization passes from the computational complexity of the graph transformation. We have also presented a method for generating fused JIT kernels using search by using a cost model that respects data-consistency, maximizes parallelism. 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 intend to further investigate the performance limitations of the technique on more complex models and tasks. 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], while we also recognize that our search heuristic can also be complimentary to data-driven approaches by dramatically pruning the search space.

---

6 1) Initialization of the weights, 2) Forward pass only, 3) Forward and backward passes.
References


