Data Movement Is All You Need: A Case Study on Optimizing Transformers

Andrei Ivanov*, Nikoli Dryden*, Tal Ben-Nun, Shigang Li, Torsten Hoefler
ETH Zürich
firstname.lastname@inf.ethz.ch
* Equal contribution

Abstract—Transformers have become widely used for language modeling and sequence learning tasks, and are one of the most important machine learning workloads today. Training one is a very compute-intensive task, often taking days or weeks, and significant attention has been given to optimizing transformers. Despite this, existing implementations do not efficiently utilize GPUs. We find that data movement is the key bottleneck when training. Due to Amdahl’s Law and massive improvements in compute performance, training has now become memory-bound. Further, existing frameworks use suboptimal data layouts. Using these insights, we present a recipe for globally optimizing data movement in transformers. We reduce data movement by up to 22.91% and overall achieve a 1.30× performance improvement over state-of-the-art frameworks when training BERT. Our approach is applicable more broadly to optimizing deep neural networks, and offers insight into how to tackle emerging performance bottlenecks.

Index Terms—Data movement, high-performance computing, deep learning, transformers

I. INTRODUCTION

Transformers [1] are a class of deep neural network architecture for sequence transduction [2], similar to recurrent neural networks [3] and LSTMs [4]. They have recently had a major impact on natural language processing (NLP), including language modeling [5]–[7], question-answering [8], translation [1], and many other applications. The significant improvement in accuracy brought by transformers to NLP tasks is comparable to the improvement brought to computer vision by AlexNet [9] and subsequent convolutional neural networks. Transformers have also begun to be applied to domains beyond NLP where RNNs would previously have been used, including speech recognition [10], reinforcement learning [11], molecular property prediction [12], and symbolic mathematics [13].

Training transformers is very compute-intensive, often taking days on hundreds of GPUs or TPUs [14]–[20]. Further, generalization only improves with model size [18], [20]–[22], number of training samples [16], [20], and total iterations [16], [23]. These all significantly increase compute requirements. Indeed, transformers are becoming the dominant task for machine learning compute where training a model can cost tens of thousands to millions of dollars and even cause environmental concerns [24]. These trends will only accelerate with pushes toward models with tens of billions to trillions of parameters [22], [25], their corresponding compute requirements [26], and increasing corporate investment towards challenges such as artificial general intelligence [27]. Thus, improving transformer performance has been in the focus of numerous research and industrial groups.

Significant attention has been given to optimizing transformers: local and fixed-window attention [28]–[32], more general structured sparsity [33], learned sparsity [34]–[36], and other algorithmic techniques [19], [37] improve the performance of transformers. Major hardware efforts, such as Tensor Cores and TPUs [38] have accelerated tensor operations like matrix-matrix multiplication (MMM), a core transformer operation. Despite this, existing implementations do not efficiently utilize GPUs. Even optimized implementations such as Megatron [19] report achieving only 30% of peak GPU flop/s.

We find that the key bottleneck when training transformers is data movement. Improvements in compute performance have reached the point that, due to Amdahl’s Law and the acceleration of tensor contractions, training is now memory-bound. Over a third (37%) of the runtime in a BERT training iteration is spent in memory-bound operators: While tensor contractions account for over 99% of the flop performed, they are only 61% of the runtime. By optimizing these, we show that the overhead of data movement can be reduced by up to 22.91%. Further, while MMM is highly tuned by BLAS libraries and hardware, we also find that existing frameworks use suboptimal data layouts. Using better layouts enables us to speed up MMM by up to 52%. Combining these insights requires moving beyond peephole-style optimizations and globally optimizing data movement, as selecting a single layout is insufficient. Overall, we achieve at least 1.30× performance improvements in training over general-purpose deep learning frameworks, and 1.08× over DeepSpeed [39], the state of the art manually-tuned implementation of BERT. For robustly training BERT [16], this translates to a savings of over $85,000 on AWS using PyTorch. For the GPT-3 transformer model [40] with a training cost of $12M [41], our optimizations could save $3.6M and more than 120 MWh energy. To do this, we develop a recipe for systematically optimizing data movement in DNN training.

Our approach constructs a dataflow graph for the training process, which we use to identify operator dependency patterns and data volume. With this representation, we identify opportunities for data movement reduction to guide optimization. We aim to maximize data reuse using various forms of fusion. Then we select performant data layouts, which is particularly
II. BACKGROUND

Here we provide a brief overview of our terminology, transformers, and data-centric programming. We assume the reader is generally familiar with training deep neural networks (see Goodfellow et al. [48] for an overview).

A. Training Deep Neural Networks

We use the standard mini-batch data-parallel approach to training, wherein a mini-batch of samples is partitioned among many GPUs. During backpropagation, we distinguish between two stages: computing the gradients with respect to a layer’s input (dX), and computing the gradients with respect to the layer’s parameters (dW). Note that the second stage is relevant only for layers with learnable parameters.

B. Transformers

The transformer architecture [1], originally developed for machine translation, is a neural network architecture for sequence transduction, or transforming an input sequence into an output sequence. Transformers build upon a long sequence of work within natural language processing, most relevantly beginning with word embeddings [49]–[51], neural machine translation [52], [53], and sequence-to-sequence learning [54]. A key component is attention [28], [29], which enables a model to learn to focus on particular parts of a sequence.

The transformer makes two key contributions. First, it generalizes attention to multi-head attention, which we discuss below. Second, the transformer neglects recurrent or convolutional mechanisms for processing sequences, and relies entirely on attention. Critically, this enables significantly more parallelism during training, as the model can process every element of a sequence simultaneously, instead of having a serial dependence on the prior element.

1) Multi-head Attention: Multi-head attention (MHA) generalizes attention, and uses h attention “heads” in parallel to attend to different learned projections of a sequence. We provide Python code and an illustration of MHA forward propagation in Fig. 1.
Each attention head is an instance of scaled dot-product attention, and takes three input tensors: queries (q), keys (k), and values (v). Conceptually, attention finds values corresponding to the keys closest to the input queries. Heads are also augmented with learned linear layers that project their corresponding inputs to a lower-dimensional space. The three inputs are first multiplied by weight tensors \( w_q \), \( w_k \), and \( w_v \), respectively, as a learned input projection (we use Einstein-notation sums, or \( \text{einsums} \), for tensor contractions). The query and key tensors are subsequently multiplied together and scaled (stored in \( \beta \)), followed by applying the softmax operation in order to weight and select the most important results. This is then multiplied with \( vv \) to produce the per-head output (gamma). The outputs of all the heads are finally concatenated and linearly projected back to the input dimension size (i).

The respective dataflow graph (Fig. 1b) immediately exposes coarse- (whole graph) and fine-grained (within rectangular nodes) parallelism, as well as data reuse. As every edge represents exact data movement, their characteristics (access sets and movement volume) can be inspected for guided bottlenecks and potential solution analysis.

There are three broad classes of MHA, distinguished by their inputs. General attention uses distinct tensors as the queries, keys, and values. Encoder/decoder attention uses the same tensor for both keys and values (typically produced by an encoder layer). Self-attention uses the same tensor for all three inputs. MHA may also have a masking step, which is used during training to prevent a model from “seeing the future” and using information from a later part of a sequence.

2) Transformer Architectures: BERT [14] is a widely-used transformer for NLP tasks. Fig. 2 illustrates the forward and backward pass for a single BERT encoder layer. The layer essentially consists of MHA (as self-attention) followed by a feed-forward neural network. The feed-forward network consists of two linear layers with bias and ReLU activations after the first layer. Dropout [55], layer normalization [56], and residual connections [57] are also used.

Transformers also incorporate several other layers that we will not discuss in detail: embedding layers for input sequences and various output layers, depending on the task. Other transformer architectures, such as the original Transformer and GPT-2/3 [21], [40] have very similar architectures.

C. Data-Centric Programming

As DNN processing is among the most popular compute-intensive applications today, considerable efforts have been made to optimize its core operators [58]. This has driven the field to the point where optimization today is almost exclusively performed beyond the individual operator, either on the whole network [59], [60] or repeated modules.

Performance optimization on modern architectures consists of mutations to the original code, sometimes algorithmic [61]–[63] but mostly relating to hardware-specific mapping of computations and caching schemes. This includes tiling computations for specific memory hierarchies, using specialized units (e.g., Tensor Cores) for bulk-processing of tiles, modifying data layout to enable parallel reductions, hiding memory latency via multiple buffering, pipelining, and using vectorized operations. It is thus apparent that all current optimization techniques revolve around careful tuning of data movement and maximizing data reuse.

The Data-Centric (DaCe) parallel programming framework [64] enables performance optimization on heterogeneous architectures by defining a development workflow that enforces a separation between computation and data movement. The core concept powering program transformation is the Stateful Dataflow multiGraph (SDFG), which is a graph intermediate representation that defines containers and computation as nodes, and data movement as edges. DaCe takes input code written in Python or DSLs, and outputs corresponding SDFGs. Subsequently, programmers can mutate the dataflow via user-extensible graph-rewriting transformations to change the schedule of operations, the layout of data containers, mapping of data and computation to certain processing elements, or any adaptation to the data movement that does not change the underlying computation.

As opposed to traditional optimizing compilers and deep learning frameworks (e.g., XLA, TorchScript), DaCe promotes a white-box approach for performance optimization. The
framework provides an API to programmatically instrument and explore, e.g., different layouts and kernel fusion strategies, all without modifying the original code. DaCe was shown to map applications to different hardware architectures, including CPUs, GPUs, and FPGAs [64], enabling both whole-program and micro-optimizations of nontrivial applications to state-of-the-art performance [65].

The combination of the separation of the algorithm from the transformed representation and white-box approach for optimization enables us to inspect and optimize the data movement characteristics of Transformer networks. As we shall show in the next sections, this drives a methodical approach to optimizing a complex composition of linear algebraic operations beyond the current state of the art.

### III. Elements of Optimized Transformers

We now apply our recipe to optimize data movement in training, using a BERT encoder layer as an example. We focus on a single encoder layer, since these are simply repeated throughout the network, and other components (e.g., embedding layers) are not a significant component of the runtime. In this section, we discuss dataflow and our operator classification. Sections IV and V discuss our optimizations and Section VI presents end-to-end results for transformers. At a high level, our recipe consists of the following steps:

1. Construct a dataflow graph for the training process and analyze the computation to identify common operator classes.
2. Identify opportunities for data movement reduction within each operator class using data reuse as a guide.
3. Systematically evaluate the performance of operators with respect to data layout to find near-optimal layouts.
4. Find the best configurations to optimize end-to-end performance of the training process.

#### A. Dataflow Analysis

We use SDFGs and the DaCe environment to construct and analyze dataflow graphs. Fig. 2 provides a simplified representation of dataflow in a transformer encoder layer. Each node represents an operator, which is a particular computation along with its associated input and output, which may vary in size. An operator may be implemented as multiple compute kernels, but is logically one operation for our analysis. To produce an SDFG, all that is required is a simple implementation using NumPy. As the goal of this stage is to understand the dataflow, we do not need to optimize this implementation: It is simply a specification of the computations and data movement.

Using DaCe, we can easily estimate data access volume and the number of floating point operations (flop) required for each computation. Fig. 2 is annotated with the number of flop and the ratio of flop to data volume, and we provide a precise comparison with PyTorch in Tab. III. The key observation is that the ratio of data movement to operations performed varies significantly among operators. In many cases, the runtime of an operator is dominated by data movement, rather than useful computation, and this should be the target for optimization.

#### TABLE I: Proportions for operator classes in PyTorch.

<table>
<thead>
<tr>
<th>Operator class</th>
<th>% flop</th>
<th>% Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>△ Tensor contraction</td>
<td>99.80</td>
<td>61.0</td>
</tr>
<tr>
<td>□ Stat. normalization</td>
<td>0.17</td>
<td>25.5</td>
</tr>
<tr>
<td>○ Element-wise</td>
<td>0.03</td>
<td>13.5</td>
</tr>
</tbody>
</table>

#### B. Operators in Transformers

With this analysis, we can now identify high-level patterns that allow us to classify operators. We base our classification both on the data movement to operation ratio and the structure of the computations. This classification is useful as it allows us to consider optimizations at a more general level, as opposed to working on each operator (or kernel) individually. For transformers, we find three classes: tensor contractions, statistical normalizations, and element-wise operations. The border of each operator in Fig. 2 indicates its class and Tab. I gives the proportion of flop and runtime for a BERT encoder layer for each class.

1. **Tensor Contractions △**: These are matrix-matrix multiplications (MMMs), batched MMMs, and in principle could include arbitrary tensor contractions. We consider only MMMs and batched MMMs for simplicity, as these are efficiently supported by cuBLAS. In transformers, these are linear layers and components of MHA. These operations are the most compute-intensive part of training a transformer. For good performance, data layout and algorithm selection (e.g., tiling strategy) are critical.

2. **Statistical Normalizations □**: These are operators such as softmax and layer normalization. These are less compute-intensive than tensor contractions, and involve one or more reduction operation, the result of which is then applied via a map. This compute pattern means that data layout and vectorization is important for operator performance.

3. **Element-wise Operators ○**: These are the remaining operators, and include biases, dropout, activations, and residual connections. These are the least compute-intensive operations.

#### C. Memory Usage Efficiency (MUE)

The MUE metric [66] provides a measure of the memory efficiency of an operation, both with respect to its implementation and achieved memory performance. This provides another method for understanding performance beyond flop/s that is particularly relevant for applications that are bottlenecked by data movement. MUE evaluates the efficiency of a particular implementation by comparing the amount of memory moved ($D$) to the theoretical I/O lower bound [67] ($Q$) and the ratio of achieved ($B$) to peak ($\hat{B}$) memory bandwidth:

$$\text{MUE} = \frac{Q}{D} \cdot \frac{B}{\hat{B}} \cdot 100.$$

If an implementation both performs the minimum number of operations and fully utilizes the memory bandwidth, it achieves MUE = 100. This can be thought of as similar to the percent of peak memory bandwidth.
A significant portion of the runtime in existing transformer implementations is in statistical normalization and element-wise operators (Tab. I). Thus, fusion is a major opportunity for promoting data reuse. We find fusion opportunities with a combination of standard performance engineering heuristics and by using DaCe to identify conformant iteration spaces.

We develop a set of fusion rules applicable to any application with operators similar to the three here. The process consists of two steps: detecting which operations can be fused and then deciding whether it is beneficial to fuse them.

To discover fusion opportunities, we analyze their iteration spaces. Every operator has independent dimensions. Statistical normalization operators also contain reduction dimensions. Tensor contractions additionally have reduction dimensions, and special independent dimensions for the two input tensors.

The type of iteration space determines which tools are used to implement them. Independent dimensions can be implemented using GPU block or thread parallelism, or with "for" loops within a single thread. Reduction dimensions use these techniques but also specify how the reduction is to be performed: accumulating to the same memory location ("for" loops), or as grid-, block-, or warp-reductions.

Two operators can be fused if their iteration space implementations are compatible: They are either the same or the only difference is that one operator performs a reduction. The order and size of dimensions and the implementation for each must match. If the first operator produces output the second uses as input, partial fusion can be done: The outermost independent dimensions can be shared, but the innermost iteration spaces are put in sequential order inside.

When a fusion opportunity is detected, we take it in two cases: First, if we can perform fewer kernel launches by merging iteration spaces. Second, if we achieve less data movement by avoiding loads and stores between operators. Theoretically, the first case could increase memory pressure, but we observe it provides benefits in practice.

We attempt to fuse maximally. There are four structural patterns (Fig. 3) in the dataflow graph for the encoder layer when fusion rules explained above are applied to a pair of non-tensor contraction operators. Using the SDFG, we fuse two adjacent operators whenever we detect these patterns and continue until we cannot fuse further. This means we fuse until either a reduction dimension or iteration space changes. As a further constraint, we only fuse simple element-wise scaling operations into tensor contraction operators (see Sec. IV-C).

A. Implementation

We implement each fused operator as a single custom CUDA kernel and specialize it for a specific data layout using templates to maximize opportunities for compiler optimization. To correctly handle data dependencies, if a reduction is the first operator in a fusion kernel, it is implemented with two loops: the first computes the reduction and the second uses it. Otherwise, each kernel is implemented as a single loop.

Reduction operations in statistical normalizations use a warp allreduce among all threads in a warp, implemented with CUDA intrinsics. If the number of elements to be reduced exceeds the warp size, we perform a sequential reduction over smaller chunks first. Layout-permuting, we use vectorized loads, stores, and arithmetic within a single thread, and fall back to word-wise implementations otherwise. For dropout operators, which must generate a random mask, we use cuRAND for random number generation.

After fusion, we have the following fused element-wise and normalization operations (see Sec. IV-C).

- **AIB**: Attention input bias.
- **BAOB**: Backward attention output bias.
- **BAIB**: Backward attention input bias.
- **SM**: Softmax with scaling and dropout.

![Fig. 3: Examples of operator fusion.](image-url)
B. Results

Tab. II presents our comprehensive results, including operator fusion. In this, we show a detailed breakdown of the required and observed flop, data movement, runtime, and MUE for each operator within the encoder layer, for both PyTorch and our implementation, with our fused operators marked. We can easily observe that while the vast majority of flop are in tensor contractions, much of the runtime is in statistical normalization and element-wise operators (see also Tab. I). These operators are indeed memory-bound.

In forward propagation, every fused operator outperforms PyTorch’s. In backpropagation, this trend generally holds, but EBSB and BAOB are slower due to our configuration selection algorithm choosing a layout that is suboptimal for some operators to optimize the overall performance (see Sec VI).

By studying the MUE and flop/s, we can reason about the bottlenecks behind each operator. For the fused operators, we see that high MUE rates are often achieved. In fact, the MUE from Tab. II and the theoretical flop/IO ratio from Fig. 2 are highly correlated across operators. We say that a kernel is memory bound if its MUE is larger than the achieved peak flop/s, and compute bound otherwise. This insight aids in analyzing the bottlenecks of general DNNs and automated tuning of operators, prior to measuring their performance. We note that for our operators, which involve multiple tensors of different shapes, 100% MUE is potentially unattainable as achieving peak memory bandwidth requires a specific, highly regular access pattern into DRAM.

As for the tensor contraction results, we see that the attained MUE is consistently under 50%. This is acceptable, since the underlying matrix multiplications are generally compute-bound. However, we note that some cases, such as $QK^T$, are both low in flop/s and MUE. A more in-depth look into the dimensions of the contraction reveals that the dimensions are small, which then indicates that the tensor cores are underutilized. This may result from insufficient scheduled threads, or low memory throughput to compute ratio. We thus try to increase hardware utilization by fusing additional operators into the contractions next.

C. Fusing into Tensor Contractions

Because cuBLAS does not support fusing arbitrary operators into (batched) MMMs, we evaluated CUTLASS version 2.1 as an alternative, which does support fusing element-wise operators. We conducted a simple benchmark comparing cuBLAS with a separate bias kernel to CUTLASS for the first linear layer of BERT. We found that the batched MMM in CUTLASS is approximately 40 $\mu s$ slower than cuBLAS.

The reduction in runtime by fusing the bias is less than this. Hence, we only consider cuBLAS for tensor contractions. cuBLAS does support simple scaling operations, which we use to implement the scaling for the scaled softmax operator.

D. Algebraic Fusion

There is an additional opportunity for fusion among certain tensor contraction operators. Using domain knowledge and the dataflow graph, we can identify some operations that can be combined into a single algebraically equivalent operation. For example, there are several different ways to perform the input projections (batched MMMs) in self-attention, since the input queries, keys, and values are the same tensor, $X$:

1) $X$ can be multiplied by each of the projection matrices: $W^Q X$, $W^K X$, and $W^V X$.
2) The $W^Q$ and $W^K$ projection matrices can be stacked and two multiplications performed: $[W^Q W^K]X$ and $W^V X$.
3) Similarly, the $W^K$ and $W^V$ matrices can be stacked.
4) All three can be stacked: $[W^Q W^K W^V]X$.

In backpropagation, the $dW$ and $dX$ computations for each of the projection matrices can be similarly fused: $X[dQ dK dV]$ and $[W^Q W^K W^V][dQ dK dV]$, respectively.

There are different tradeoffs to these approaches, which must be determined empirically. Performing separate MMMs may enable task parallelism. On the other hand, this stacking enables data reuse, since $X$ is used only once.

Tab. II presents results for each of these cases. Fully fusing this batched MMM performs the best. Unfortunately, cuBLAS launches kernels that utilize the entire GPU, so task parallelism is not profitable. This specific example can also be adapted to fuse keys and values in encoder/decoder attention.

V. DATA LAYOUT

We now consider data layout selection, which enables efficient access patterns, vectorization, and tiling strategies for tensor contraction and statistical normalization operators. To study this systematically, for each operator, including the fused operators produced in the prior step, we benchmark every feasible data layout to measure its performance, as well as varying other parameters depending on the specific operator. The best parameterization of an operator is highly dependent on the GPU model and tensor sizes, and may not be obvious a priori; hence it is important to empirically consider this.

Because there are a myriad of potential configurations for each operator, we summarize the distribution of runtimes over all configurations using violin plots. The width of the violin represents the relative number of configurations with the same runtime. This allows us to see not only the best runtime, but to...

<table>
<thead>
<tr>
<th>Operators</th>
<th>Non-fused</th>
<th>QK fused</th>
<th>QKV fused</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward</td>
<td>345</td>
<td>294</td>
<td>275</td>
</tr>
<tr>
<td>Backward</td>
<td>342</td>
<td>312</td>
<td>291</td>
</tr>
</tbody>
</table>
see how sensitive operators are to layouts, an important factor when globally optimizing the layout in Step 4 of our recipe.

A. Tensor Contractions

Using the Einsum notation for tensor contractions, we consider all equivalent permutations of the summation string. The einsum is then mapped to a single cuBLAS MMM or batched MMM call. While this notation allows us to express arbitrary tensor contractions, cuBLAS does not support all configurations, we limit ourselves to these two types.

In addition, we consider every possible cuBLAS algorithm for each layout, as we have observed that the heuristic selection provided by its default algorithm is not always optimal. We use the cublasGemmEx API to manually select algorithms. We support both regular and Tensor Core operations, and perform all accumulations at single-precision, in line with best practices for mixed-precision training [70].

Fig. 4 presents performance distributions over all data layouts, for each layout, as we have observed that the heuristic selection provided by its default algorithm is not always optimal. We use the cublasGemmEx API to manually select algorithms. We support both regular and Tensor Core operations, and perform all accumulations at single-precision, in line with best practices for mixed-precision training [70].

We use the cublasGemmEx API to manually select algorithms. We support both regular and Tensor Core operations, and perform all accumulations at single-precision, in line with best practices for mixed-precision training [70].

Table III: Flop analysis for BERT encoder layer. △ - tensor contraction, □ - statistical normalization, ○ - element-wise. MHA operators are filled black. We bold whichever is greater, % peak (compute-bound) or MUE (memory-bound). The speedup is computed for kernels in isolation, overall speedup may be different due to measurement overheads.
cuBLAS MMVs can be easily swapped, results for both orders are merged into the same tile of the figure. All tiles are labeled with \( M > N \). As expected, in many cases using tensor cores offers significant performance advantages, but interestingly, in several cases (where one of the matrix dimensions is 64) the performance is quite close to the regular floating point units, due to a failure to saturate the tensor cores. Among the tensor core results, we can typically see there are several modes in the performance distributions; these correspond to particularly important axes for data layout. Indeed, for many configurations, one of these is near or contains the best-performing configuration, indicating that many slightly different data layouts could be used with little impact on performance depending on the needs of our global optimization pass. However, this does not mean that any data layout is acceptable; in every case, the majority of the mass does not perform well, illustrating the importance of careful tuning.

We also investigated how well the built-in cuBLAS algorithm heuristics compare to the best-performing configuration. On half precision, we found that the algorithm chosen by cuBLAS’s heuristic was up to 14.24% worse than the best algorithm (in \( QT \) dX1). We also investigated the performance at single-precision and found similar results with up to 7.18% worse performance. This demonstrates the importance of carefully tuning for the particular hardware and workload.

### B. Fused Operators

For our fused operators, we consider all combinations of input and output layout permutations. This enables us to include transposes of the output data as part of the operator, should the next operator perform better in a different layout than the input. The data layout is especially critical for statistical normalization operators, where the appropriate data layout can enable vectorization opportunities, especially vectorized loads and stores for more efficient memory access. Where relevant, we therefore also consider vectorization dimensions, the mapping of dimensions to GPU warps, etc. Our implementation takes advantage of these layouts when feasible.

Fig. 5 presents the runtime distribution for all configurations of our fused operators (note some are used twice). The distributions here are qualitatively similar to those in Fig. 4, except these have much longer tails: A bad configuration is, relatively, much worse, potentially by orders of magnitude.

All kernels support changing the layouts of tensors they use. This change is done via template parameterization, so the compiler can generate efficient code. All kernels support selecting different vectorization dimensions. The BRD and BEI kernels can select the dimension used for CUDA thread distribution; BSB, EBRSB, and BDRB can select the warp reduction dimension, as they reduce over two dimensions.

The most noticeable performance improvement is made by layouts that enable vectorized memory access, showing that the main performance limitation is the amount of accessed data volume. The second notable category are layouts with the same reduce and vector dimensions. Joining these dimensions decreases the number of registers required to store partially reduced values from the vector size (eight at FP16) to one.

We can expect to get good performance restricting ourselves to configurations in the two groups described above. Usually, the best layout discovered follows them. For example, the SM kernel has the same warp and reduction dimensions, and these dimensions are the last and sequential ones for involved arrays. However, this intuition is not always correct. Examining the results in Fig. 5 we find that there are kernel configurations that both satisfy these intuitive rules yet are very slow. For example, the best configuration of AIB takes 65 \( \mu s \), and the worst "intuitively good" configuration takes 771 \( \mu s \).
Fig. 5: Performance of fused kernels for element-wise and statistical normalization operators.

Fig. 6: Example configuration selection graph for SSSP.

VI. END-TO-END OPTIMIZATION

The final step is to assemble fused components and select data layouts for each operator to yield a complete implementation. This is the culmination of the prior steps performing dataflow analysis, fusion, and layout selection. From these, we have performance data for all data layouts as well as algebraic fusion strategies. One cannot simply pick a single data layout a priori, as the benefit of running two operators in different layouts may outweigh the overhead of transposing data.

Our assembled implementation is structured using the SD-FGs produced in Step 1. We integrate it into PyTorch [45] via its C++/CUDA operator API.

A. Configuration Selection

We develop an automatic configuration selection algorithm to globally optimize our implementation using the performance data. We construct a directed graph based on the dataflow graph of the operators. Beginning from the input data and proceeding in the order given by a pre-order depth-first search, we add a node to the graph for each input and output data layout of the operator. An edge is added from the input to the output layout, weighted with the minimum runtime of any configuration with that layout. Determining this simply requires a linear search over the matching performance data. This allows us to select both the best data layout and any other operator knobs (e.g., vectorization dimension). To minimize the size of the graph, we only add a configuration from an operator when it has at least one input and output edge. We then run a single-source shortest-path (SSSP) algorithm from the input to the output in the graph; the resulting path gives our final configuration. Because this graph is a DAG and the number of feasible input/output configurations for each operator is small, SSSP takes linear time asymptotically and seconds for BERT. We illustrate the graph used in Fig. 6.

To simplify the implementation, we make two assumptions on the global configuration space. First, we omit residual connections. Second, we run SSSP only for the forward propagation dataflow graph, and then infer the layouts of the backpropagation operators from those selected (including weight and gradient layouts). Both of these assumptions could be relaxed in a future version of this algorithm. Although this means we are not guaranteed to find a globally optimal data layout, the runtime of our configuration is nevertheless within 4% of the sum of the best possible configuration of each operator (which ignores data layout constraints).

B. Multi-head Attention

We first analyze the performance of multi-head self-attention. While it is a key primitive in BERT, MHA is also used outside of transformers, so understanding its performance
TABLE IV: Multi-head attention performance for BERT.

<table>
<thead>
<tr>
<th></th>
<th>TF+XLA</th>
<th>PT</th>
<th>cuDNN</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward (ms)</td>
<td>1.60</td>
<td>1.90</td>
<td>1.31</td>
<td>1.25</td>
</tr>
<tr>
<td>Backward (ms)</td>
<td>2.25</td>
<td>2.77</td>
<td>6.52</td>
<td>1.86</td>
</tr>
</tbody>
</table>

TABLE V: Full BERT encoder layer performance.

<table>
<thead>
<tr>
<th></th>
<th>PT</th>
<th>TF+XLA</th>
<th>DS</th>
<th>Ours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forward (ms)</td>
<td>3.45</td>
<td>3.2</td>
<td>2.8</td>
<td>2.63</td>
</tr>
<tr>
<td>Backward (ms)</td>
<td>5.69</td>
<td>5.2</td>
<td>4.8</td>
<td>4.38</td>
</tr>
</tbody>
</table>

in isolation can inform other models too. Tab. IV compares our globally-optimized implementation with PyTorch, TensorFlow+XLA, and cuDNN. cuDNN’s (experimental) MHA implementation (cudnnMultiHeadAttnForward and related) supports six different data layouts; we report the fastest. cuDNN’s performance is orders of magnitude worse than the others; as it is a black box, our ability to understand it is limited. However, profiling shows that its implementation launches very large numbers of softmax kernels, which dominate the runtime, indicating additional fusion would be profitable. TensorFlow+XLA finds several fusion opportunities for softmax. However, its implementation does not perform algebraic fusion for the queries, keys, and values, and it uses subpar data layouts for tensor contractions.

Our performance results in Tab. III illustrate the source of our performance advantage over PyTorch: Our data layout and algorithm selection results in faster tensor contractions overall. This is despite the Gamma stage actually being slower than PyTorch’s: Sometimes locally suboptimal layouts need to be selected to improve performance globally.

C. End-to-End Performance

We present overall performance results for the encoder layer in Tab. V. For forward and backpropagation combined, we are 1.30× faster than PyTorch and 1.20× faster than TensorFlow+XLA, including unoptimized framework overheads. At a high level, this is because we perform a superset of the optimizations used by both frameworks, and globally combine them to get all the advantages while minimizing drawbacks. As a general guideline, we use flop and MUE rates as proxies for which operators require the most attention and their corresponding bottlenecks. This ensures a guided optimization rather than tuning all operators aggressively.

We also include performance results from DeepSpeed, which we are 1.08× faster than. This is despite DeepSpeed being a manually-optimized library tuned specifically for BERT. Note also that DeepSpeed modifies some operations, e.g., to be reversible or to exploit output sparsity, and so is not always strictly comparable to the other implementations. This also provides it opportunities for optimization that we do not consider.

The total data movement reduction we attain is \(\sim 22.91\%\) over the standard implementation. We obtain this information from Tab. III where for each fused kernel we omit the interim outputs and inputs that are not part of the overall I/O that the fused kernels perform. TensorFlow+XLA’s automatic kernel fusion reduces data movement similarly to ours. However, the data layouts used for tensor contractions are not optimal, and its BERT encoder implementation does not use algebraic fusion in MHA. PyTorch’s data layouts enable faster tensor contractions and it implements the algebraic fusion, but it has higher overheads for other operators.

Our fusion pass finds all the opportunities that TF+XLA does, plus several additional ones; for example, we implement layernorm as a single fused kernel. Our data layout selection picks better layouts than PyTorch in nearly every individual instance; when it does not, this is because the layout change enables greater performance gains downstream. In Tab. III we also see that PyTorch performs more flop than predicted. Some of this is due to padding in cuBLAS operations, and generic methods performing excess operations. However, we also discovered that some cuBLAS GEMM algorithms, including ones called by PyTorch, incorrectly perform twice as many FLOPs as necessary; our recipe avoids these cases automatically.

We also briefly considered another configuration for training BERT, where we change the batch size to \(B = 96\) and the sequence length to \(L = 128\), and retuned our layout selection. In this case, forward and backpropagation for a single encoder layer takes 18.43 ms in PyTorch, 16.19 ms in DeepSpeed, and 16.22 ms in our implementation. We significantly outperform PyTorch, and match the performance of DeepSpeed in this case (even with its additional optimizations). We believe that with further improvements to our layout selection algorithm, the performance of our implementation will improve further.

Beyond BERT, other transformers have very similar layers, such as decoder layers in GPT-2/3. With very few changes, our recipe and implementations are directly applicable to these. Our implementation can also be extended to support a full training pipeline by stacking our optimized layers.

VII. RELATED WORK

There has been significant work optimizing both transformers in particular and deep learning in general. For a recent comprehensive overview, see Ben-Nun & Hoeffer [58]. To help guide training regimes for transformers, recent work has provided empirical recommendations on model size, batch size, and so on on [23], [25], [74]. Many of the subsequent techniques we review are complementary to our work.

Most directly relevant are other approaches specifically to accelerate transformer training. Distributed-memory techniques, such as ZeRO [25], Megatron [18], and Mesh-TensorFlow [76], scale training to many GPUs to accelerate it. Mesh-TensorFlow also presents a classification of operators similar to ours. Large batches have also been used to accelerate training via LAMB [77] or NVLAMB [78]. None of these directly address the performance of a single GPU as done in this paper. DeepSpeed [39], which we compare with in Section VI-C, is closest to our work, but performs all optimizations and layout selections manually.
Transformer architectures to enable improved training have also been the subject of significant recent work. ALBERT [19] used a combination of weight sharing and factorized embedding layers to reduce memory requirements; however, compute times are unaffected. Transformer-XL [79] caches prior hidden states to learn long sequences. RevNets [80], a variant of ResNets which allow activations to be reconstructed during backpropagation, have been applied to transformers. Notably, Reformer [37] combines reversible residual layers with locality-sensitive hashing to improve the efficiency of multi-head attention. Sparsity optimizations for attention [28–36] reduce memory and compute requirements. We view these as orthogonal to our work: the same principles of data-flow analysis can be applied to optimize for sparsity and reuse.

There has also been much work on optimizing deep learning in general. Many frameworks provide implementations of transformers or their components, such as PyTorch [45], TensorFlow [46], cuDNN [47], and others built atop these [69], [81]. Optimizing frameworks can also be applied to transformers [59], [60], [82–100]. None of these frameworks provide all the optimizations or the systematic study of data movement and its impact on performance. We have specifically compared against some of the most popular production frameworks: PyTorch, TensorFlow with XLA, and cuDNN. Beyond these, TASO [87] targets similar optimizations to ours by using graph substitutions, but considers only inference and does not exhaustively explore the search space.

Other optimizations, including model parallelism [76], [85], [101–105], pipeline parallelism [106–109], microbatching [110], and recomputation for memory reduction [111] are all also applicable. Communication can also be a major bottleneck for training transformers, due to the large model size [13]. Frameworks for inference, including TensorFlow [113], Caffe2 [114], and the ONNX Runtime [115], all help to enable a suite of optimizations primarily applicable during inference. Pruning [116], [117] and distillation [118] has also been used to accelerate inference.

Neural network architecture-specific optimizations have a long history outside of transformers, and have primarily targeted CNNs [9], [119–126]. Notably, Li et al. [127] optimized data layouts for convolution.

In general, data movement reduction is a core component of high-level optimization [128]. Optimizing compilers, most notably components that specialize in polyhedral programs [129], [130], apply loop transformations (e.g., tiling, skewing) that belong to the class of data movement optimization. Other white-box approaches for separation of program definition from data optimization passes include Halide [131], JAX [82], [83], Legion [132], Lift [133], and MLIR [134]. The data-centric approach proposed here enables user-extensible coarse- and fine-grained data movement optimization via the flat (yet parametric) dataflow graph representation [64]. This allows us to perform and tune complex data layout and fusion transformations that span multiple granularity levels, surpassing the optimization capabilities of the aforementioned tools and achieving state-of-the-art performance.

\section{VIII. Discussion}

The recipe we propose in this paper can be directly adopted in other DNN architectures. Additional transformer networks, such as Megatron-LM [18] and GPT-3 [40], only differ by dimensions and minor aspects in the encoder and decoder blocks (e.g., dropout position, biases). Once a data-centric graph is constructed from them, the recipe remains unchanged.

\subsection{A. Beyond Transformers}

The classification of operators into three groups covers a wide variety of operators that span beyond transformers.

Large tensors and their contraction are ubiquitous in modern DNNs. For fully connected networks (MLPs) and recurrent neural networks (RNNs), there is little change, as the core operator types are essentially the same. Convolutions, pooling, and other local spatial operators can be treated similarly to tensor contractions, owing to their arithmetic intensity properties and abundance of optimized implementations. Therefore, the same considerations we take here are just as critical in CNNs. However, as opposed to contractions (see Section \ref{sec: convo}), further fusion is typically considered for convolutions.

Statistical normalization also takes different forms in DNNs. This includes a variety of reductions, as well as Instance, Group, and Batch Normalization, where the latter constitutes the second largest computation in ResNets after convolutions. When varying data layouts, these operators share properties (normalizing a dimension) and are optimized in exactly the same way. Lastly, element-wise operators always exist in DNNs and benefit from the same fusion and bulk data movement optimizations as we perform here. For graph neural networks [135], capsule neural networks [136], and other emerging architectures, the operators change more significantly, but the basic procedure applies.

Due to the prohibitively large search space of configurations in transformers, writing manually-optimized kernels becomes infeasible. Instead, each of our data-centric implementations chooses an optimization scheme (e.g., tiling, vectorization, warp-aggregated reductions) automatically, according to the input data layout and the operator type. Combined with automated configuration selection (Section \ref{sec: automated}), we rely only on the dataflow graph structure to choose the best feasible data layout configuration.

As many networks are bound by data movement rather than compute performance [127]. This leads us to believe that our recipe, regardless of the objective (e.g., classification, regression, RL) and the constituent operators, is generically applicable for optimizing any neural network architecture.

\subsection{B. Hardware Implications}

The implications of data movement reduction extend beyond software. Given that the highest performance for different operators is achieved with different data layouts, there would be significant benefits if future machine learning accelerators included built-in support for fast data layout changes. We confirm this in our MUE results (Tab. \ref{tab: mue}), showing that even

\begin{center}
\begin{tabular}{|c|c|}
\hline
Layer & Layout
\end{tabular}
\end{center}
the most compute-intensive tensor contractions are bounded by the hardware’s capability of transferring data to Tensor Cores.

Hardware trends indicate a similar situation. New architectures are moving towards reducing data format conversion (e.g., TF32 [137]), increased on-chip memory and low-latency interconnects [138], and coarse-grained spatial hardware [139]. For the latter two, the recipe and analysis provided here is crucial to maintain high utilization in pipelined DNN execution.

IX. CONCLUSIONS

Despite the importance of transformer neural networks, training them is memory-bound and underutilizes GPUs. Using our recipe for data movement analysis, we identified bottlenecks and optimizations, yielding improved implementations that outperform the already highly tuned state-of-the-art. As training transformers is already a major compute workload that will only grow larger, our improvements offer significant real-world impacts for both research and industry.

Our approach is applicable more broadly to deep learning: many neural networks easily fit within our operator classification. This is especially important for guiding the optimization of emerging or novel model architectures, which do not benefit from existing acceleration libraries. Our results also highlight the importance of considering data movement at every level of the training stack—from the application down to hardware.

ACKNOWLEDGMENTS

This project received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 programme (grant agreements DAPP, No. 678880, and EPiGRAM-HS, No. 801039). N.D. is supported by the ETH Postdoctoral Fellowship. T.B.N. is supported by the Swiss National Science Foundation (Ambizione Project #185778). Experiments were performed at the Livermore Computing facility.

REFERENCES


