Demystifying BERT: System Design Implications

Suchita Pati[1,2] Shaizeen Aga[2] Nuwan Jayasena[2] Matthew D. Sinclair [1,2]

[1]*University of Wisconsin-Madison*
{spati, sinclair}@cs.wisc.edu
[2]*Advanced Micro Devices Inc.*
{shaizeen.aga, nuwan.jayasena}@amd.com

Abstract

Transfer learning in natural language processing (NLP) uses increasingly large models that tackle challenging problems. Consequently, these applications are driving the requirements of future systems. To this end, we study the computationally and time-intensive training phase of NLP models and identify how its algorithmic behavior can guide future accelerator design. We focus on BERT (Bi-directional Encoder Representations from Transformer), one of the most popular Transformer-based NLP models, and identify key operations which are worthy of attention in accelerator design. In particular, we focus on the manifestation, size, and arithmetic behavior of these operations which remain constant irrespective of hardware choice. Our results show that although computations which manifest as matrix multiplications dominate BERT's execution, they have considerable heterogeneity. Furthermore, we characterize memory-intensive computations which also feature prominently in BERT but have received less attention. To capture future Transformer trends, we also show and discuss implications of these behaviors as networks get larger. Moreover, we study the impact of key training techniques like distributed training, checkpointing, and mixed-precision training. Finally, our analysis identifies holistic solutions to optimize systems for BERT-like models and we further demonstrate how enhancing compute-intensive accelerators with near-memory compute can help accelerate Transformer networks.

1. Introduction

In recent years, rapid advancements in natural language processing (NLP) have enabled important applications that interpret human language, such as intelligent personal assistants, near instantaneous language translation, and more intelligent search engines. This tremendous, transformative rise has been enabled by a virtuous synergy of (1) better hardware systems, (2) larger datasets, and (3) improved deep neural network (DNN) structures and machine learning (ML) algorithms that further benefit from more efficient hardware and larger datasets. For decades, technology scaling has driven Moore's Law and enabled more efficient hardware designs. However, the slowing of technology scaling and Moore's Law [44] presents a crucial challenge to more fully realizing ML's potential. In recognition of the slowing of technology scaling, the community responded by optimizing ML workloads for GPUs [15, 22, 45, 66, 101] and designing accelerators [27, 43, 49, 61, 92]. As a result, ML workloads, especially ML inference and convolutional neural networks (CNNs), are often able to fully utilize on-chip ALUs and execute efficiently [1, 2, 4, 12, 16, 27, 29, 30, 34, 35, 40, 41, 50, 51, 57, 59, 72, 76, 78, 83, 88, 96, 97, 99, 100].

However, more recently Transformer-based networks [87] have become the preferred algorithm for NLP. These networks, along with transfer learning, have given rise to models like the **B**i-directional **E**ncoder **R**epresentation from **T**ransformer **(BERT)** [21], which mark a shift towards deeper knowledge transfer by applying massive pre-trained models to different tasks. BERT accurately trains on billions of words and outperformed its predecessors on several NLP tasks [21]. To do this, BERT uses unsupervised learning to train on large unlabeled datasets. Its deeply bi-directional Transformer-based architecture implies its representation of a token in a given sequence encodes information from all the preceding and subsequent tokens. Furthermore, BERT models are large with 110-340 million parameters. Collectively, these features are important contributors to its power, and accuracy.

Understanding these feature's underlying behaviors is vital to designing efficient accelerators for Transformer-based models. However, thus far there has been little work on a detailed characterization for them. Most prior work that characterizes ML algorithms focuses on CNNs, recurrent neural networks (RNNs), or recommendation models [2, 32, 92, 100]. Moreover, prior works that analyze Transformers miss important details such as manifestation of layer operations and detailed runtime breakdown amongst all operations [89, 92, 98]. Consequently, some works build accelerators with matrix-vector engines for BERT layers which actually perform matrix-matrix operations [33, 36].

Moreover, BERT has also become the basis of several larger (3.9 billion parameters) models [56, 79] and has also inspired other NLP architectures such as ALBERT [52], ERNIE2.0 [85], Google's TransformerXL [19], OpenAI's GPT-2 [68] and GPT-3 [14], RoBERTa [56], and XLNet [93]. Given the explosion in ML algorithms designed for NLP, it is important to further optimize future systems for these algorithms [95]. Although each of these models are worthy of deeper investigation, most Transformer-based models have a similar structure to BERT but have different sizes (discussed in Section 2.3).

Accordingly, in this work we provide a detailed algorithmic characterization of BERT's training to guide the design and optimization of future accelerators for Transformer-based networks. We focus on BERT since it embodies several of the essential features and further study the impact of varying its hyperparameters to capture Transformer

TABLE 1: Summary of takeaways

Takeaway	Algorithmic Explanation	Sec.
LAMB optimizer is very memory intensive & important to optimize for.	LAMB updates 340M BERT parameters and is the second highest training time contributor. It reads data worth 4× the model size and has few element-wise operations. LAMB's runtime scales linearly with transformer layer count & quadratically with layer size.	3.2.1 3.2.3 3.3
GEMMs dominate BERT runtime but have heterogeneity.	BERT processes all input sequence tokens in parallel & thus layers manifest as GEMMs, even if mini-batch is one. Linear and Batched-GEMMs in attention are smaller than in FC & thus may not fully utilize accelerators and may also be memory-bound. GEMM proportion also increases with layer size.	3.2.1 3.2.2 3.3
Non-GEMMs are memory-bound & a considerable proportion of runtime.	These constitute element-wise (add, mul, scale) & reduction operations. Their proportion drops with increasing layer size as they scale only linearly with it (unlike GEMMs & LAMB, which are quadratic).	3.2.3 3.3
Reducing precision makes optimizing memory-intensive operations crucial.	GEMMs speedup more than others in half precision due to faster arithmetic & reduced memory traffic. Non-GEMMs only benefit from reduced footprint of reduced precision data. LAMB uses high (FP32) precision data to maintain accuracy and is unaffected.	3.2.1 3.2.3
Tensor Slicing is bottlenecked by communication.	Communication is serialized with computations in tensor slicing. Its cost increases with device count.	5.2

trends. Although we characterize the pre-training phase, our takeaways hold for fine-tuning as well (details in Section 7). To make our analysis broadly applicable to various types of accelerators and agnostic of any particular hardware, we focus our analysis and takeaways on BERT's operations, their manifestation, size, and arithmetic intensity, as well as on important training techniques used to train the network. Moreover, to demonstrate the impact of changing training techniques and hyperparameters, we analyze the change in relative runtime contributions of different operations and layers using a GPU. However, as discussed in Section 7, these contributions can also be extrapolated for other devices. Finally, using these observations we discuss hardware and software optimizations for BERT, and further demonstrate how leveraging near-memory compute can accelerate BERT-like models. Overall, we make the following contributions:

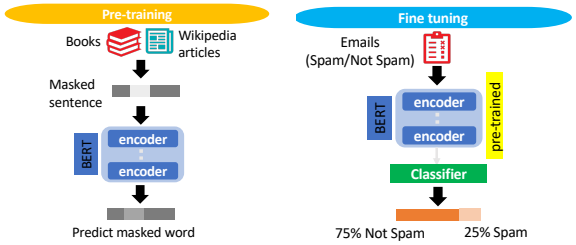
- We characterize BERT training and, using the runtime proportions as well as the arithmetic intensities of operations, identify important components that future accelerators should optimize for. Table 1 summarizes our main takeaways.
- We analyze the implications of these behaviors as networks get larger, deeper, and use different input sizes, to capture future Transformer trends.
- We study the impact of mixed-precision training, checkpointing, and distributed training, that are employed to scale network training.
- Finally, we discuss software and potential hardware optimizations for BERT and further demonstrate that enhancing compute-intensive accelerators with near-memory compute can accelerate BERT by 5-22%.

More broadly, we identify holistic future acceleration solutions for Transformer-based models like BERT.

2. Background

2.1. Transfer Learning & BERT Pre-training

In *transfer learning*, a model trained for a particular task is reused for different tasks. Although widely used in Computer Vision [77, 94], transfer learning was only recently applied to NLP in BERT. As shown in Fig. 1, BERT's training consists of two parts. It has a long *pre-training* phase where the model learns the language using large unlabeled datasets (e.g., Wikipedia), independent of



(a) Pre-training (b) Fine-tuning

Figure 1: BERT training overview

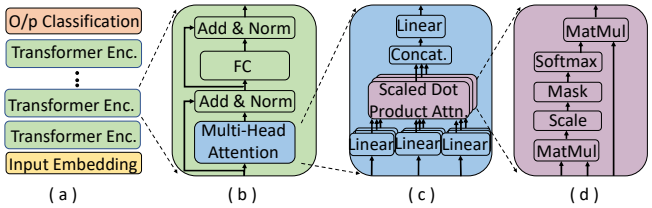


Figure 2: BERT hierarchical model breakdown.

any target task. BERT Large's (one of the larger BERT configurations, detailed in Section 3.1.3) pre-training takes up to four days on 16 cloud TPU Pods (each Pod has four TPU chips and 64 GB RAM [31]). BERT's pre-training is further split into two phases based on the input *sequence length* (n): Phase-1 (n=128) and Phase-2 (n=512). Although larger n is important for accuracy, BERT's runtime is quadratic with n (discussed in Section 2.2). Thus, to reduce training time, Phase-1 has 90% of training iterations, while Phase-2 has only 10% of them. Once pre-trained, BERT is *fine-tuned* (Fig. 1b) during which it is trained on a labeled dataset for a specific task with minimal model changes (e.g., BERT authors tune it separately for 11 different tasks [21]). Fine-tuning is usually inexpensive, taking up to an hour on a single TPU Pod or few hours on a GPU [31].

2.2. Attention

The *attention* network within the transformer layers (Fig. 2(c, d)) is an essential component of BERT and other recent NLP models. Given an input sequence, the attention networks output a representation of the sequence such that each output token of the sequence is encoded with *weighted* information from all (or a subset, for Masked-Attention) other tokens in the sequence (details in Section 3.2.2). Thus, unlike its predecessor, RNN, in which processing of a token

is sequentially dependent on the processing of previous tokens in the sequence, attention processes all tokens independently. However, due to the all-to-all computations, attention layer complexity grows quadratically with n, making it computationally expensive for larger n.

2.3. BERT & Other Transformer Architectures

BERT's basic building block is the Transformer *encoder* layer (Fig. 2(a)). Fig. 2(b) shows a breakdown of the Transformer encoder layer into an attention layer and a fully connected (FC) feed-forward layer, both of which are followed by a residual connection and layer normalization. BERT has an input embedding layer that provides the first Transformer layer with an input representation for every token. It has an output classification layer responsible for two unsupervised tasks: masked word prediction and Next Sentence Prediction. An additional output layer may be added during fine-tuning for the target task. Most Transformer-based models have a similar structure to BERT but different sizes. They use multiple layers of encoder, *decoder* or both depending on target functionality. The decoder (e.g., in GPT-3) is similar to encoder except its attention layer is masked to consider only past tokens. While this causes different inference behavior, it does not affect training (it only zeros certain matrix elements).

2.4. Gradient Descent Optimizer & LAMB

Gradient descent is commonly used to train neural networks. It minimizes an *objective function* (usually the loss) parameterized by the model's parameters. Models use algorithms which optimize gradient descent to converge faster but require computing and tracking additional parameters. Although BERT is compatible with many optimizers, it has recently used *LAMB* [95], which is effective for very large batch-sizes. LAMB updates model weights once every (few) iteration(s) using additional *momentum* (m) and *velocity* (v) parameters. This algorithm (Algorithm 2 [95]) is executed independently for every model layer, each accessing the corresponding layer's data (weights, gradients, and optimizer parameters).

2.5. Distributed Training

Growing model sizes and datasets have made the use of multiple devices to train DNNs commonplace. There are two common techniques to do so. *Data parallelism* replicates the model and partitions the dataset amongst D devices. Each device trains its model (using mini-batch of B) while synchronizing with other devices every iteration.[1] During synchronization, local gradients from all devices are averaged and re-distributed (AllReduce operation), following which each device updates its model. This enables large effective mini-batch ($D * B$) training. Conversely *model parallelism* splits the model across M devices such that each device stores only a subset of parameters and activations. This enables training of larger models. One form of this approach, Megatron-LM [79] for Transformers, uses *tensor slicing* (TS) to split individual layers amongst devices and requires communication/reduction of activations and gradients. Models also use a *hybrid* approach, where the model is split between M devices in a cluster, and replicated across D such clusters, each with a disjoint dataset. This enables training across $M * D$ devices.

1. Asynchronous training avoids this by converting fine-grained synchronization into data accesses but may increase convergence time [20].

2.6. Arithmetic Intensity

An algorithm's *arithmetic intensity* (ops/byte) is defined as the number of operations it performs for every byte of data read. If it performs very few operations on each byte of data, it will likely be bottlenecked by memory bandwidth and vice-versa. It is an important parameter used to gauge if operations benefit from more compute, or higher memory bandwidth.

3. BERT's Algorithmic Behavior

3.1. Experimental Setup

3.1.1. System. Our system consists of an AMD Ryzen™ Threadripper™ CPU [7] and an AMD Instinct™ MI100 GPU [10] with 32GB of HBM2 [42]. Our software stack is built on top of the AMD ROCm™ platform [8] with PyTorch v1.7. Although many accelerators are used to train BERT, we choose GPUs for this study because of their wide availability and popularity for DNN training. However, our takeaways are accelerator agnostic and should be applicable to other GPUs, accelerators, and frameworks suitable for machine learning. Since our goal is to characterize BERT training in a platform independent manner, we focus on relative importance of its operations, as well as the size and nature of operations, which in turn depend on BERT's network architecture, hyperparameters, and selected training mechanism (e.g., mixed precision; model versus data parallelization strategy). Thus, this approach provides fundamental value in guiding architecture design based on deep, algorithmic understanding of the application instead of solely profiling-based analysis, since architectures, both within and across vendors, evolve considerably from one generation to another. We discuss this further in Section 7.

3.1.2. BERT Phases. We analyze the BERT pre-training phase. Since fine-tuning requires only minor model tweaks and is similar to the more intensive pre-training, studying pre-training provides a solid understanding of BERT's overall training behavior while focusing on the costliest - most important to accelerate - part. We focus on Phase-1 (n=128) of pre-training with a mini-batch size (B) of 32 and discuss how Phase-2's (n=512) characteristics differ. Finally, we study both single and mixed precision (MP) training to discuss how bottlenecks shift with reduced precision. Tables 2a and 2c list the acronyms we use to refer to model details and training techniques.

3.1.3. BERT Hyperparameters. Although BERT has several configurations [86], we focus on the largest and most accurate one: *BERT Large*. BERT Large model contains 24 Transformer layers (N) with a hidden state size (d_{model}) of 1024, 16 attention heads (h) and an intermediate dimension (d_{ff}, usually $4*d_{model}$) of 4096. Since these hyperparameters can scale in future models, we also study their impact on

TABLE 2: BERT hyperparameters, GEMMs and acronyms.

(a) BERT hyperparameters

Acronym	Parameter
B	mini-batch size
d_{model}	Hidden Dim.
h	#Attention Heads
d_{ff}	Intermediate Dim.
N	Layer Count
n	Sequence Length

(b) Architecture-agnostic sizes of BERT GEMMs for both training and inference.

Operation	FWD	BWD Grad. Activation	BWD Grad. Weight
Linear	d_{model} x n^*B x d_{model}	d_{model} x n^*B x d_{model}	d_{model} x d_{model} x n^*B
Attn. Score	n x n x d_{model}/h, B=B^*h	n x d_{model}/h x n, B=B^*h	d_{model}/h x n x n, B=B^*h
Attn. O/p	d_{model}/h x n x n, B=B^*h	d_{model}/h x n x n, B=B^*h	n x n x d_{model}/h, B=B^*h
FC-1	d_{ff} x n^*B x d_{model}	d_{model} x n^*B x d_{ff}	d_{model} x d_{ff} x n^*B
FC-2	d_{model} x n^*B x d_{ff}	d_{ff} x n^*B x d_{model}	d_{ff} x d_{model} x n^*B

(c) BERT acronyms

Acronym	Full Form
FC	Fully-connected
EW	Element-wise
LN	LayerNorm
DR	Dropout
SM	Softmax
RC	Residual Conn.
MP	Mixed Precision

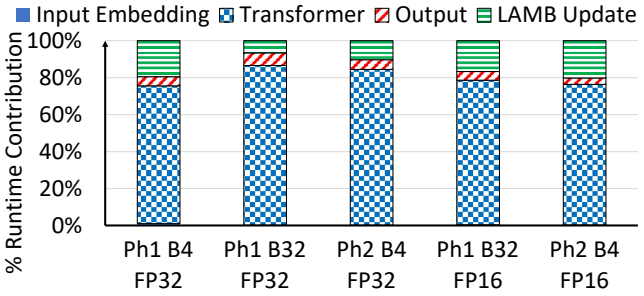


Figure 3: Runtime breakdown of BERT pre-training.

BERT's execution profile in Section 3.3. Throughout the remainder of the paper, we use these symbols to refer to the parameters, as shown in Table 2a.

3.1.4. Profiling Mechanism. Profiling entire BERT pre-training (with the English Wikipedia dataset [21]) can be impractical. Although CNNs can be characterized by profiling a single training iteration [58, 65, 99, 100], NLP model iterations can be heterogeneous due to varying input sequence length [67]. However, BERT's training iterations operate on same-size inputs within a phase. Thus, we profile and study a single training iteration (after a set of warm-up iterations) per pre-training phase. We use rocProf [6] to gather runtime and other performance counter data.

3.2. Compute & Memory Demands of BERT Operations

A BERT iteration performs (a) a forward (FWD) phase to process input sequences to produce an output, (b) a backpropagation (backprop, or BWD) phase to calculate the loss in output prediction and weight gradients, and (c) an update phase to update the weights using the gradients. Table 2b describes three GEMM operations and activation sizes for each important BERT sub-layer: one for FWD and one each for BWD activation and weight gradient calculation. Throughout we represent a matrix as MxN, a GEMM as MxNxK, and a product of two variables as $a * b$.

3.2.1. Runtime Breakdown. We first present a high-level runtime breakdown amongst different network layers and training phases. In all the runtime distribution plots we consider a layer's FWD and BWD phases together and show weight updates separately. Fig. 3 shows this for different phases, B, and precisions: Phi-Bj-FPk, where i represents the phase (Phase-1 or Phase-2), j is the mini-batch size, and k is the number of floating-point (FP) bits used in the experiment. Note that, FP16 here refers to mixed precision [62], where FWD and BWD use FP16 inputs, weights, and gradients, but updates are in FP32 to maintain accuracy.

As expected, for all the configurations, the Transformer layers dominate (68-85%) the runtime since they account for most of the layers. The output layer constitutes only a small proportion (3-7%) and the input embedding layer is negligible. Interestingly, the LAMB optimizer (Section 2.4) is consistently the second highest contributor (7-25%). Its proportion increases as the number of tokens ($n * B$) per training iteration decrease (e.g., Ph1-B4-FP32 and Ph2-B4-FP32 have a higher LAMB proportion than Ph1-B32-FP32). This occurs because the FWD and BWD runtimes depend on token count, while the weight update runtime is only proportional to model size. LAMB's proportion also increases with MP training (in Ph1-B32-FP16 and Ph2-B4-FP16). Reduced precision speeds up computations via faster arithmetic and reduced memory accesses. This helps both FWD and BWD operations speed up by about 2× while the runtime of FP32 LAMB remains constant. Its proportion will further increase with more aggressive quantization [75].

Obs. 1: Transformer layers dominate (68-85%) BERT runtime. Output and embedding layers contribution is negligible.

Takeaway 1: LAMB updates are the second highest contributor (7-10%) to BERT's training time. Their contribution increases (25%) with decreasing token count per iteration.

Takeaway 2: LAMB updates become more important (16-19%) to optimize for with mixed-precision training.

Fig. 4 presents a hierarchical breakdown of Transformer layers for single (Ph1-B32-FP32) and MP (Ph1-B32-FP16) training (labels represent their contribution to overall training time). The second bar, **Transformer**, shows the runtime breakdown among the Transformer layer's components: the attention layer, the Fully Connected (FC) feed-forward layer, as well as the combined dropout (DR), residual connection (RC), and Layer Normalization (LN) layers. Overall, FC layer has a higher runtime contribution compared to the attention layer due to its larger (4×) intermediate dimension (Section 3.1.3). Additionally, the combined DR, RC and LN layers have a smaller, but non-negligible (5% in FP32, 9% in MP) contribution per iteration.

The third bar of Fig. 4, breakdown of the **Attention** layer, shows that a significant portion of the runtime (22% in FP32, 19% in MP) is spent on *linear operations* (linear in Fig. 2(c)). These operations are required to project each of the inputs query, key, and value vectors (of length d_{model}) into h different vectors (of dimension d_{model}/h) to be operated on by h attention heads (detailed in Section 3.2.2). The actual attention operation (Fig. 2(d))

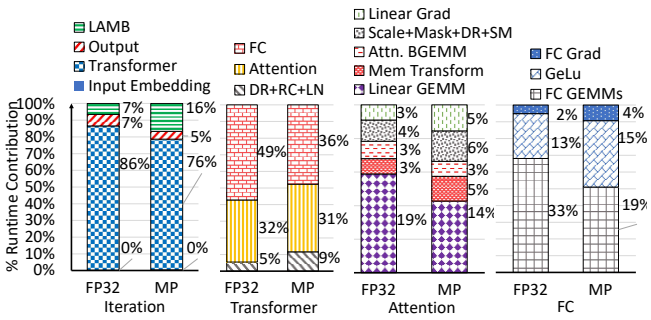


Figure 4: Hierarchical breakdown of BERT pre-training runtime. Labels show contribution to overall training time.

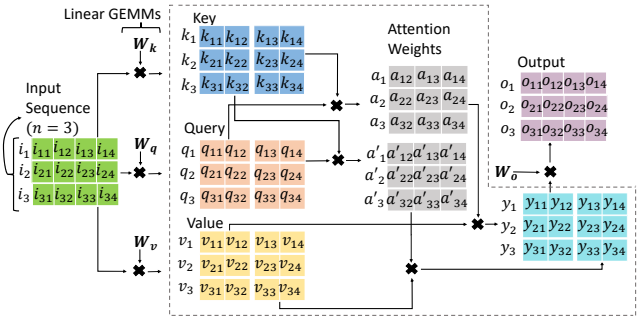


Figure 5: Computations in the Attention layer.

represented by Attn. BGEMM and Scale+Mask+DR+SM, constitute a much smaller proportion (7% in FP32, 9% in MP) of the overall runtime. The feed-forward sub-layer, **FC**, of BERT's Transformer layers (last bar in Fig. 4) consist of two fully-connected connections with a Gaussian Error Linear Unit (GeLu) [38] activation in between. The FC connections (FC GEMMs+Grad) dominate the runtime with GeLu contributing to 13% in FP32 and 15% in MP.

Finally, the proportion of linear and FC layers decreases considerably (from 57% in FP32 to 42% in MP) compared to other operations when executed with MP implying they benefit more from the drop in precision. These layers manifest as GEMMs (details in Section 3.2.2) and their speedup can be attributed to both faster arithmetic (Matrix Core Engine [9]) and smaller memory footprint.

Obs. 2: Linear and FC layers dominate (57%, FP32) BERT runtime. Rest is spent executing several smaller operations.

Takeaway 3: Reducing precision speeds up GEMMs in the dominant linear and FC connections more than other operations, reducing their overall contribution (42% in MP).

Takeaway 4: Attention operations constitute a very small proportion (7% in FP32, 9% in MP) of BERT runtime.

3.2.2. GEMM Operations in BERT. Since GEMMs constitute a large proportion (55% in FP32 and 36% in MP) of BERT's iteration time, we next characterize these and their compute requirements. There are three sets of GEMMs in BERT's Transformer layers, corresponding to the attention computations, the linear transform operation, and the fully connected layers.

As illustrated in Fig. 5 (within the dotted box), the attention head takes the query (q_n) and key (k_n) vectors of all the tokens in the input and calculates the attention score (a_n) between every token pair through the product of their respective query and key vectors using a GEMM. Since there are h independent attention heads working in parallel, there are h GEMMs executed in parallel per input sequence (h = 2 in Fig. 5). Furthermore, since a training iteration operates on a mini-batch (B) of inputs, there are $B * h$ GEMMs invoked as a single batched-GEMM kernel (**Attn. B-GEMM** in Fig. 4 and Attn. Score in Table 2b). The attention scores are then used to calculate the weighted sum (y_n) of all value vectors (v_n) in the input sequence, also invoked as a batched-GEMM (Attn. O/p in Table 2b) with $B * h$ parallel GEMMs. While there are several ($B * h$) parallel GEMMs in this operation, each of them is quite small (dimensions of n and d_{model}/h).

To enable multiple attention heads, the query, key, and value vectors of the tokens are first linearly projected (outside the dotted box in Fig. 5, left) into h smaller (d_{model}/h) feature vectors. All the token vectors of all the input sequences in a mini-batch are usually combined into a single ($B * n$) x d_{model} matrix. Thus, unlike in RNNs, a batch size of one does not lead to matrix-vector operations in Transformers. Using the learned Weights (W_k, W_q, W_v), the tokens are linearly projected via three different GEMMs which dominate the attention layer runtime (**Linear GEMMs** in Fig. 4 and Table 2b). These GEMM outputs are then split to create the query, key, and value vectors for each of the attention heads. The concatenated outputs of the attention heads are also projected back using W_o (outside the dotted box in Fig. 5, right). Finally, the FC layers use their learned Weights (4× the linear weights) to operate on the output of the attention layer. This creates two large **FC GEMMs** (Table 2b FC-1 & FC-2) which dominate the execution time of the FC layer (Fig. 4).

Usually, larger, and squarer GEMMs perform better on modern accelerators by leveraging the highly parallel accelerator's compute power, exploiting data reuse, and better hiding memory latency. However, not all of BERT's GEMMs and B-GEMMs fit this paradigm. To understand better, we plot the arithmetic intensity (ops/byte) of all GEMMs (labeled as $transposeA, transposeB, M, N, K, [batch]$) in a BERT's Transformer layer (Ph1-B-32-FP32) in Fig. 6. While the FC GEMMs are large and extremely compute intensive, the linear transform GEMMs are not, with 4× smaller matrix dimensions and smaller ops/byte ratios. Furthermore, the attention layer's B-GEMM matrices are even smaller, leading to extremely low ops/byte ratio. We further plot their memory bandwidth requirements normalized to the maximum bandwidth achieved by any BERT operation (i.e., element-wise or EW multiply) in Fig. 7. Attn. GEMMs have much higher (70%) memory bandwidth requirements compared to the other GEMMs (only 20%), making them memory-bound in contrast to the commonly occurring compute-bound GEMMs in DNNs.

Takeaway 5: GEMM dimensions in BERT are a multiple of the input token count (i.e., $B * n$), and layer's hidden size (d_{model} or d_{ff}) and scale with these parameters. Unlike RNNs, a B of one does not lead to matrix-vector operations.

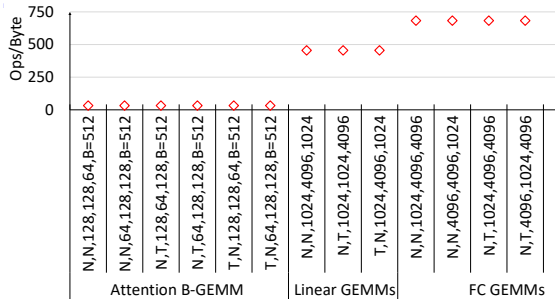


Figure 6: Arithmetic intensity of BERT's training GEMMs. It shows that not all of BERT's GEMMs are equal.

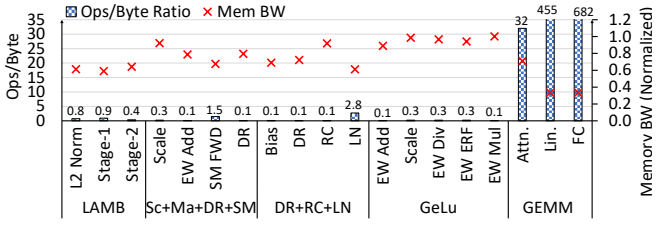


Figure 7: BERT op's arithmetic intensity & bandwidth requirements.

Takeaway 6: Not all GEMMs in BERT are equal. Smaller, skinnier GEMMs in BERT's attention layer are memory-bound and can under-utilize highly parallel accelerators.

3.2.3. Non-GEMM Operations in BERT. In the previous sections, we observed that 45% (FP32) and 64% (MP) of BERT's training time is spent executing non-GEMM operations. Thus, accelerators for BERT-like models must optimize both GEMMs and these operations. There are four parts of BERT pre-training where we observe these operations: (1) LAMB, (2) scale, mask, dropout (DR) and softmax (SM), (3) GeLU activation, and (4) DR, residual connection (RC), and layer normalization (LN). Input and output layer operations are omitted as they are a small proportion, especially as model sizes grow (Section 3.3.2). Fig. 7 includes memory bandwidth requirements and ops/byte ratios of the four phases.

LAMB Updates: The LAMB algorithm described in Section 2.4 is 7-25% of BERT's iteration time, and can further increase with increasing model size (Section 3.3.2), smaller token size per iteration, or MP training. It is implemented as two stages in [62]. LAMBStage1 (Fig. 7) determines the update values and learning rate multiplier using additional momentum (m) and velocity (v) states from past iterations and gradients of the current iteration (all of which are the same size as the model parameters being updated, shown as MxN of the BWD Grad. Weight GEMMs in Table 2b). This stage performs multiple EW add, multiply, divide, scale, and square-root operations on these parameters and therefore, has very low arithmetic intensity (Fig. 7) making it memory intensive. The second stage (LAMBStage2 in Fig. 7) updates model weights with stage 1's output also using multiple EW operations and has similar memory characteristics to stage 1. These two stages are executed for each layer, and access the corresponding layer's data (weights, gradients, and optimizer parameters). Therefore, each set has no data reuse across kernels (its impact on kernel fusion is discussed in Section 6.1.1). Moreover, LAMB must perform the L2 Norm (reduction) across all the model's gradients before it can update any parameter, which serializes the model update with respect to the entire model backprop.

Takeaway 7: The memory-intensive LAMB optimizer reads 4× more data than the model size and has few EW operations.

Scale, Mask, Dropout & Softmax: The attention head generates attention scores between token pairs. These scores are normalized and operated on by a mask, softmax, and dropout functions (**Scale+Mask+DR+SM** in Fig. 4) before being used to calculate the weighted representation of each token in the input. The normalization kernel multiplies each element of the input matrix with a constant value. The mask and DR operations, invoked as separate kernels, involve an EW add and multiply of the activation matrix with a mask and DR matrix, respectively. Therefore, all three perform only a single operation on each data read. Finally, SM performs a series of EW operations on the input matrix, which improves its arithmetic intensity, but it is still not very compute intensive. Thus, these operations have high memory bandwidth requirements (Fig. 7).

GeLU: GeLu activation [38] is executed between two FC GEMMs and consists of a series of EW add, multiply, divide, and ERF (error function) as shown in Equation 1:

$$GELU(x) = x * \frac{1}{2} * [1 + erf(\frac{x}{\sqrt{2}})] \quad (1)$$

When invoked as separate kernels, these operations have very low ops/byte ratios, as shown in Fig. 7. Along with the large input activation size (output of FC GEMM), this makes these kernels memory bandwidth bound.

Dropout, Residual Connection, & Layer Normalization: Outputs of the FC and attention sublayers are applied the DR, RC, and LN (DR+RC+LN in Fig. 4) function as shown in Fig. 2(b) (Add & Norm). DR randomly sets activation elements to zero using an EW multiply. RC does EW addition of the input to the output of a sublayer. Thus, these kernels have an arithmetic intensity of less than one (Fig. 7). Finally, LN [13] is a reduction operation and requires calculation of mean/variance of rows/columns, followed by a few EW operations. However, it still has a very low arithmetic intensity as shown in Fig. 7. Consequently, these kernels are memory bandwidth bound.

While LAMB kernels remain unchanged in MP training (since updates are in FP32), most other memory bandwidth bound kernels speed up by 1.5 – 1.9× in MP. However, this speedup is much smaller than GEMMs, thereby increasing the relative proportion of these operations in MP training. Thus, non-GEMM operations become even more important to optimize for when training with reduced precision.

Takeaway 8: BERT has multiple memory-bound element-wise operations that make up to 30% of its (FP32) runtime.

Takeaway 9: Optimizing memory-bound operations is even more important for BERT's reduced precision training, where they make up 46% of all operations.

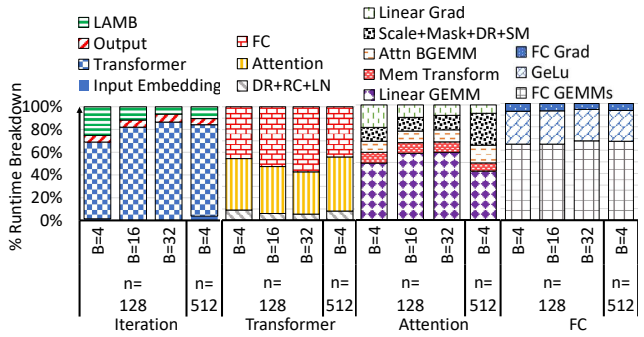


Figure 8: Impact of scaling mini-batch size & sequence length.

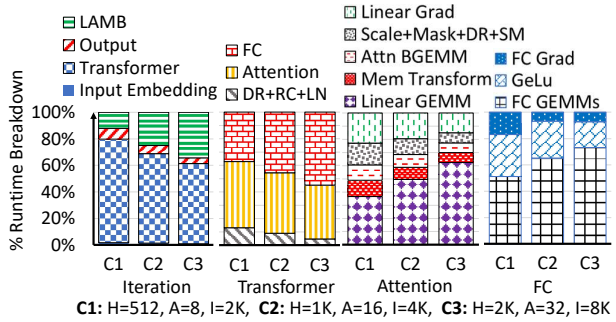


Figure 9: Impact of scaling Transformer layer size.

3.3. Effects of Hyperparameter Sweep

Most Transformer-based NLP models have a similar structure to BERT and vary largely in their model/input sizes (discussed in Section 2.3). However, Transformer training characteristics can change as models get larger and deeper, with evolving hyperparameters like Transformer layer count, hidden dimension, mini-batch and sequence length. Thus, we next analyze and characterize the impact of these hyperparameters on BERT's execution profile.

3.3.1. Input Size: Mini-batch Size (B), Sequence Length (n). B and n impact training convergence and throughput. Increasing B improves throughput but may hurt convergence, especially in data-parallel training (Section 5). Conversely, increasing n improves accuracy but increases training costs. B and n decide the token count processed in a BERT iteration. Thus, increasing them increases the total computations in the forward and backward gradient calculations while keeping the parameter update computation (which only depends on model size) constant. Fig. 8 highlights this: as B ranges from 4 to 32 LAMB updates constitute 25% to 7% of training time.

Within the Transformer layer the input size's impact varies across layers/operations. The impact depends on the layer type and its relationship with the input. For example, a layer with a MxNxK GEMM has operations proportional to $M*N*K$. Thus, increasing any dimension would linearly scale operation count. Since $B*n$ forms one of the Linear and FC GEMM dimensions (Table 2b), their operations scales linearly with B or n. This is similar to its impact on other operations (e.g., EW, reduce) which operate on activations with one of the dimensions as $B*n$. The number of GEMMs in attention B-GEMMs, and thus its runtime, also scale linearly with B (Table 2b). Thus, the breakdowns of the Transformer layers with a constant n (128) but varying B (from 4 to 32) remains largely the same in Fig. 8.

The efficiency of operations at each size also impacts the proportions: for higher Bs, attention and DR+RC+LN proportions drop, while FC's increases. Operations with small matrices may not be able to utilize the accelerator's peak throughput and/or memory bandwidth. A smaller $B*n$, along with 4× smaller hidden dimension, can lead to smaller matrices in attention and DR+RC+LN layer as compared to the FC layer. Accordingly, increasing B (from 4 to 16 in Fig. 8) improves the size of matrices, which improves these layers' throughput more than others, causing their overall runtime proportion to drop. The benefits, however, diminish with further increase in B.

Changing n has a similar impact as B, except attention operations (B-GEMMs in Table 2b and Scale+Mask+DR+SM) scale quadratically with n but only linearly with B. Thus, increasing n from 128 to 512 (and changing B from 16 to 4 to keep token count same) increases their proportion from 7% to 17% (B-GEMMs' proportion increases from 3% to 8%) as shown in Fig. 8. This also implies that, unlike B, Transformer iteration time increases super-linearly with n.

Obs. 3: B impacts all layers similarly due to their linear dependence on it. Increasing it sometimes improves throughput.

Takeaway 10: Higher n makes attention operations important.

3.3.2. Model Size: Layer Count (N), Hidden Dimension (d_{model}). BERT model size is dictated by N and hidden sizes, d_{model} and d_{ff}. Increase in N linearly scales the count of every operation pertaining to a Transformer layer and LAMB update (parameter count also scales linearly). Intuitively, this does not change the Transformer layer breakdown, but runtime proportions of both the Transformer layers and LAMB update slightly increase as operation count in the input and output layers remain constant. Conversely, increasing layer widths (i.e., d_{model} and d_{ff}) increases the size of weight matrices and input to layer operations. Thus, it changes two of MxNxK GEMM's dimensions (see Table 2b) and scales GEMM computation count quadratically. Since other layer operations only scale linearly with d_{model} or d_{ff}, the proportion of linear and FC GEMMs increase with increasing layer size. Fig. 9 highlights this: proportion of these GEMMs in configuration C3 (i.e., similar to Megatron-LM-BERT with 2× higher d_{model} than BERT-Large or C2) is much higher than in C2. Furthermore, the "Transformer" breakdown shows that FC runtime proportion increases compared to the attention layer. This indicates that (similar to changing B) throughput of linear GEMMs increase more than FC GEMMs', causing their runtimes to scale differently. Finally, the proportion of LAMB update increases considerably with larger layers (34% for C3). Unlike the linear scaling with N, parameter count and thus LAMB operations scale quadratically with d_{model} and/or d_{ff} (if d_{model} = 1024, layer parameters = 1024∗1024). Thus, optimizing for complex optimizers like LAMB, which

thus far had not been studied in detail, is increasingly important as Transformer models grow deeper and larger.
Obs. 4: Transformer and LAMB updates scale linearly and remain important as Transformer layer counts increase.
Takeaway 11: GEMM and LAMB runtime proportions increase with larger Transformer layers due to their quadratic relationship with layer size.

4. Effects of Activation Checkpointing

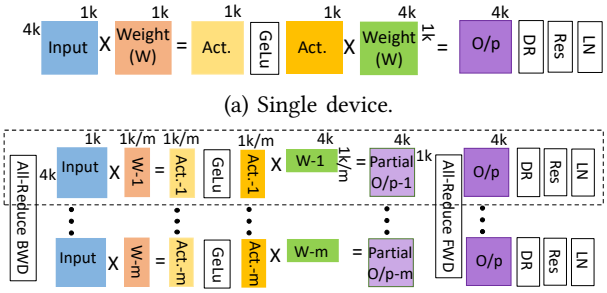
Activation (or gradient) checkpointing helps overcome device memory capacity issues. Instead of saving all layer activations from the forward pass to use in backprop, it checkpoints a limited set of activations and recomputes the others on demand during backprop. This reduces a model's memory capacity requirements and enables training a large model or a model with larger B on a single device. It, however, adds considerable recomputation overheads. We executed BERT Large training with activation checkpointing, which checkpoints activations at four ($\sqrt{N}$) different points and recomputes activations after backprop of every six Transformer layers. This increases kernel count by ∼33% and runtime by ∼27%. However, the breakdown within Transformer layers remains similar. Furthermore, since LAMB remains unaffected, its proportion drops.

5. Effects of Multi-device Training

Although studying BERT training on a single device is important and reveals interesting computational behaviors, BERT is usually trained in a multi-device environment using data parallelism (DP), a form of model parallelism called tensor slicing (TS), or both (Section 2). Thus, we first describe the analytical model we use to generate the profiles. Next, we characterize training BERT Large on 128 GPUs using DP and TS (Megatron-LM [79] with 2-way and 8-way TS) approaches.

5.1. Modeling Multi-device Training

We construct per-device execution profiles in a distributed setting by building an analytical model from a single GPU's data. We use an analytical model because the publicly available BERT implementations are not optimized for multiple devices. For example, they do not overlap gradient computation and communication in DP training, without which network communication becomes a bottleneck. Thus, to avoid drawing incorrect conclusions we instead model the behavior analytically to take optimizations like these into account. This model also allows us to study different multi-device configurations and can project performance for hypothetical GPU/network improvements. We briefly describe how we model DP and TS training below:
Modeling Data Parallelism: Since DP training replicates the model on every device, the per-device computation matches single-device training. Additionally, an AllReduce operation gathers each device's gradients (during backprop). To estimate AllReduce's communication costs, we use the gradient sizes and Ring AllReduce [28]. To estimate communication time, we divide the gradient sizes by the communication bandwidth assuming PCIe™ 4.0. Finally, since the communication and computations of different



(a) Single device.

(b) Multi-device, m-way Tensor Slicing in Megatron-LM.

Figure 10: FC layer computation in BERT.

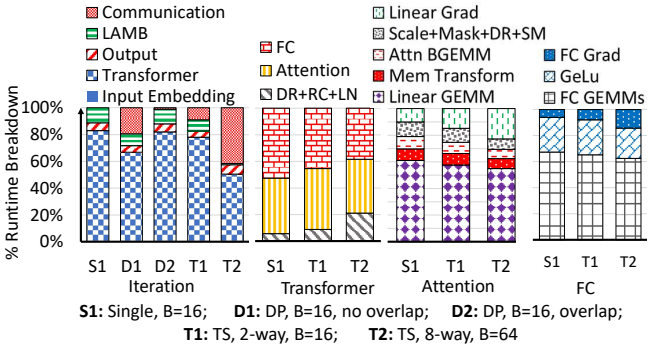


Figure 11: BERT iteration breakdown in a multi-GPU setup.

layer's gradients are independent, they can be overlapped (e.g., layer L's gradients are communicated while the device calculates gradients for layer $L-1$). We model this overlap by taking the maximum of the computation and communication times for every pair of consecutive layers.
Modeling Tensor Slicing: Megatron-LM splits most of the layer's operations across all devices. Some involve splitting of weight matrices horizontally, while others are split vertically. The remaining layers (e.g., LN) are replicated across devices to reduce communication overheads. Fig. 10 illustrates this change for the FC layer operations. The operations within the dashed line illustrate the dimension of operations on each device after a m-way split. To model TS computations, we execute BERT operations with the expected input dimensions after splitting and replicating the layers. Since each device only updates a fraction ($/m$) of the weight matrices, the LAMB operations are also split equally amongst the GPUs. Finally, there are four AllReduce operations executed per forward and backward pass of a Transformer layer. We estimate this communication time using the approach described above. However, unlike in DP, these AllReduce operations cannot be overlapped with computations due to data dependencies.

5.2. Multi-GPU Training Profile

Fig. 11 compares the execution breakdown within a single GPU participating in different distributed training mechanisms.
Data Parallel: The per-GPU execution profiles of BERT's DP approach with overlap, D2 (DP, B=16 w/ overlap) in Fig. 11, is similar to a single GPU training, S1 (w/ B=16). This is unsurprising as each GPU has a copy of the model and

independently computes the entire forward, backprop, and update phases. Although DP requires additional inter-GPU communication of local gradients, this cost (except for the first layer) can be hidden by overlapping computations and using a fast channel such as PCIe 4.0™. D1 (DP, B=16, w/o overlap) uses the same data parallel approach as D2 but communicates gradients after the entire backprop, highlighting this cost. Consequently, a significant portion of D1's runtime (19%) is spent communicating gradients. Recent work has also shown that these communication overheads and redundant updates could potentially be reduced by making each device gather a reduced copy of a subset of gradients and only update the corresponding subset of parameters [69]. However, certain optimizers such as LAMB require normalization of all the layers' gradients at the beginning of the algorithm, thus requiring at least a single device to have a copy of all gradients.

Tensor Slicing: Fig. 11 shows the per-GPU runtime breakdown for TS implementations: T1 (TS, 2-way, B=16) and T2 (TS, 8-way, B=64). The reduction in parameters going from 2-way to 8-way TS enables the increase in B. The high-level iteration breakdown of T1 is similar to S1, single-GPU training with the same B (16). However, there are two differences. First, T1 spends considerable time (9%) communicating activations and gradients. Second, LAMB's proportion scales by half as each device is responsible for half of the model's parameters. These changes are more prominent in T2 which uses eight devices. The communication costs increase to about 42% due to the larger volume of data communicated (due to its larger B). Moreover, the proportion of LAMB is negligible in T2 with 8-way partitioning of parameters and is unaffected by an increase in B. As device count continues to increase, this trend continues since the total data traffic increases with device count while the per-device computations scale down proportional to device count and any scaling of B to improve per-device computations would also scale the communication volume. Finally, T2 also highlights that the proportion of replicated layers (DR+RC+LN), which are memory-intensive, increases with device count.

Obs. 5: The compute and memory-bound operation breakdown in a data-parallel, multi-GPU setting is similar to single-GPU training due to data parallel's ability to overlap most communication with computation.

Takeaway 12: Proportion of memory-bound LAMB updates drops for model-parallel, multi-GPU training as parameter count per device scales inversely with device count.

Takeaway 13: The communication volume (and runtime) increases with tensor-sliced devices due to a larger B.

To validate our analytical model, we compared our observations to prior work and found the takeaways to be similar [71, 79]. Megatron-LM observed near-linear scaling as they increase the number of devices in the DP training of BERT, implying little impact from synchronization and communication. Similarly, ASTRA-SIM show that using a DP approach for some ML algorithms can provide a near-perfect overlap of communication and computation, although this can change if compute speeds up much faster than communication. These observations are in line with our Obs. 5. Furthermore, Megatron-LM also shows that BERT training's scaling efficiency drops with more TS devices due to increased communication overheads. This is also in line with our Takeaway 13.

Although we assume a homogeneous topology and network bandwidth, our takeaways also hold for non-homogeneous networks. Communication costs will not change for DP since communication and compute are overlapped. Although TS is more sensitive to communication, algorithms are often optimized for the underlying substrate (e.g., TS is usually employed within a node for higher bandwidth). Furthermore, while non–homogeneous networks within a node can change absolute communication cost (bottlenecked by the slowest connection), increasing cost with additional TS devices would still hold.

6. Potential Optimizations for BERT

Since roughly half of BERT's training time is spent executing GEMMs, a straightforward way to improve its throughput would be to improve the accelerator FLOPs. Thus, most prior works accelerate GEMMs. However, our work highlights that not all GEMMs can utilize highly parallel accelerators (Section 3.2.2), the importance of non-GEMM (optimizer update, other element-wise) operations (Section 3.2.3), and how reducing precision and increasing layer size (Section 3.3.2) can make optimizing for non-GEMMs more important. Moreover, as GEMMs speed up, the remaining memory-intensive operations will become the bottleneck. Thus, we next discuss several common software and hardware optimizations to help overcome this bottleneck.

6.1. Software Mechanisms

6.1.1. Kernel Fusion. Fusion combines two or more consecutive GPU kernels, potentially with a producer-consumer relationship, into a single one. It improves performance by reducing duplicate memory accesses, and kernel launch overheads [11, 25, 26, 55, 63, 82, 84, 90]. However, there are some important considerations while fusing kernels:

Data reuse across operations is directly correlated with improved performance from their fusion. Kernel fusion prevents data from being flushed into global memory between kernel calls [5, 37, 48, 80, 81]. Thus, data intensive phases like GeLu, DR+Res+LN, and Scale+Mask+DR+Soft (Section 3.2.3) in which the output of one operation feeds into the next are perfect scenarios for applying kernel fusion. LAMB operations of a single layer are already fused in PyTorch [62] (into LAMBStage1 and LAMBStage2 kernels of Fig. 7). There is little benefit from fusing LAMB operations of different layers given they access independent data. This is evident in Fig. 12 which shows the impact of fusion on kernel counts, runtime and memory accesses. Fusion is very effective for LayerNorm; runtime and memory traffic scale similar to kernel count (by $6-8\times$) implying high data-reuse opportunities across the unfused kernels. However, for Adam,[2] the reduction in memory accesses

2. Adam is an alternate to LAMB; we chose Adam for this study because its unfused and fused versions were publicly available.

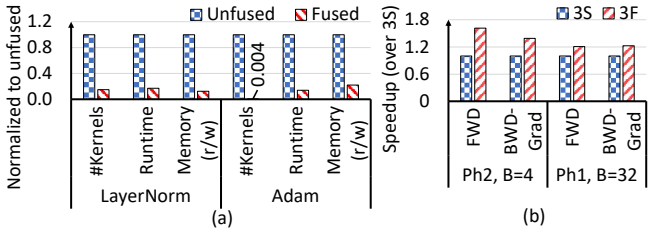


Figure 12: Impact of fusing (a) kernels, (b) 3 Linear GEMMs (3F vs. non-fused serial, 3S, execution).

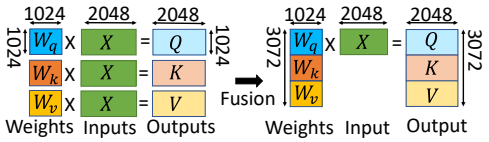


Figure 13: Fusion of Attention linear GEMMs in BERT.

and runtime ($6-8\times$) is not proportionate to that of kernel count ($\approx 250\times$). This is because Adam's unfused kernels had fewer data re-use opportunities: some operations access independent data (gradients, parameters, and optimizer states) corresponding to different layers and do not benefit from fusion.

6.1.2. GEMM Fusion. Fusing multiple smaller, independent GEMMs with a common input matrix into a single, large GEMM is another common optimization. Fig. 13 shows how the independent linear transform GEMMs of the attention layers can be fused. Since these GEMMs operate on the same input matrix and their respective weight matrices, W_q, W_k, and W_v (Fig. 13, left), they can be fused such that the weight matrices are concatenated, and the input matrix is read once (Fig. 13, right). The output of this GEMM is simply the output of the three individual GEMMs concatenated, which can be split for subsequent use. Fig. 12(b) examines the impact of this fusion on both FWD and BWD Grad. GEMMs of the linear layer. Fusion improves performance by up to 62% by enabling reuse of the common input matrix and increasing parallelism by using a larger matrix dimension. Its impact is higher when the input matrices are small (smaller token count or hidden dimension). Along with data layout improvements, such fusion can speed up BERT training considerably [39].

6.2. Hardware Mechanisms

6.2.1. Near-Memory Computing. Near-memory compute (NMC) can help accelerate BERT's memory-intensive phases. NMC performs operations using specialized ALUs that are part of the main memory. Thus, NMC avoids data movement between the main memory and GPU [3, 46] and improves performance and energy efficiency. It also provides very high bandwidth accesses to data from the ALUs in memory.

We examine such a system where the compute-intensive phases such as GEMMs are executed on the GPU, as is done today, and only the memory-intensive operations are offloaded to NMCs. While Transformers have several memory-intensive operations, we focus on the optimizer algorithm (LAMB) as it consists of a sequence of EW operations (Section 3.2.3) and is invoked at the end of the training iteration after all memory updates. Thus, executing LAMB with NMC does not require frequent, expensive synchronization between the NMC and GPU compute units. Moreover, these operations do not benefit from GPU kernel fusion as it cannot further reduce data accesses to the memory (Section 6.1.1).

DRAM is internally organized hierarchically with the lowest level being 2D groupings of memory cells. Several such groupings form a sub-array and multiple sub-arrays form a memory bank. A typical DRAM chip contains several such banks. In an NMC design, placement of ALUs at each sub-array provides extremely high bandwidth by enabling access to all (or many) sub arrays in parallel. However, such a design incurs high complexity, area, and power costs as well as limited data accessibility due to the large number of ALUs and each ALU being associated with a relatively small amount of memory. Placing ALUs at each bank, enabling parallel access to all (or many) banks, leads to fewer ALUs and, thus, reduced cost and increased memory capacity accessible by each ALU. Further reductions in ALU count can be achieved by sharing ALUs among multiple banks. However, reduced ALU count limits performance as fewer operations can occur in parallel. A more thorough discussion of design tradeoffs can be found in prior works focused on NMC [3, 36, 46, 54]. Here, we consider a balanced design point with ALUs at each bank. This design is similar to recent proposals by major memory vendors [46, 53, 54]. The NMC ALUs operate on commands broadcast from the host GPU to all banks and support arithmetic and logic operations. Similar to prior work [3], we assume the data structures are allocated such that data for an NMC ALU operation is placed in the associated bank.

To evaluate the benefits of this system, we model the execution of LAMB on NMC units using DRAM timing parameters from prior works [3, 46, 54]. We compare it against an optimistic GPU model in which LAMB's execution comprises of only (minimal) data reads and writes to GPU memory. The higher bandwidth and parallelism provided by NMC units can speed up memory-intensive LAMB updates on millions (higher for larger Transformers) of BERT parameters by $3.8\times$, which improves BERT training's overall performance by 5-22%.

6.2.2. GEMM Accelerators. A dynamically configurable accelerator would be well positioned to tackle BERT's GEMM diversity (Section 3.2.2) and address a spectrum of GEMM behaviors.

6.2.3. In-Network Processing. Compute capabilities in network switches can potentially eliminate the interference between computation and communication operations and help accelerate collective operations such as All-Reduce [47], used in distributed training (discussed in Section 5).

7. Discussion

Other Accelerators: While our analysis largely focused on a GPU, by focusing on platform-independent analysis, our takeaways can also guide BERT analysis on other

devices or accelerators. Most of the observations (1,4,5) and takeaways (1,2,5-7,11-13) are architecture-agnostic, only depending on BERT's architecture and the manifestation, size, computational complexity, and arithmetic intensity of its training operations. Thus, they hold regardless of the profiled accelerator. For example, analysis of BERT inference on CPUs shows that Obs. 1, which is purely based on model architecture, is also applicable to CPUs (differences between BERT training and inference are discussed below) [23]. Although some takeaways (e.g., 8) about operation runtime distribution might differ across accelerators, one can *approximately* extrapolate these proportions to another device by comparing the device's compute and memory bandwidth ratios. For example, the measured proportion of memory-bound and GEMM operations on an AMD Instinct™ MI100 GPU are similar to other commercial GPUs with similar compute and bandwidth ratios [39]. While differences in GPU architectures can also impact this distribution, we believe they would be small enough to not alter the application's compute- or memory-boundedness. This demonstrates the value of architecture-independent takeaways and using any particular device only secondarily. Finally, since compute generally improves faster than memory, takeaways (7,8,9) involving the memory boundedness of BERT operations will either hold or be amplified in current and future accelerators.

BERT Fine-tuning & Inference: Although we focus on BERT's pre-training phase, our takeaways also hold for fine-tuning since the latter uses the same training techniques and model with changes only to the output layer (which is often simpler and thus negligible). For example, the output layer of SQUAD (Q&A) [70] is simpler than tasks BERT is pre-trained for, requiring fewer GEMMs and thus making it a negligible component of SQUAD fine-tuning. Importantly, just like pre-training, the Transformer layers still dominate the runtime. BERT's inference differs from pre-training since the former does not require backpropagation and parameter updates. Since backpropagation has approximately 2× more operations as a forward pass with similar properties, the breakdown of the Transformer layer's execution during inference would remain similar to pre-training. However, the high-level breakdown of an inference iteration would not include LAMB updates.

Other NLP Models: Although several Transformer-based models have been proposed after BERT (discussed in Section 1), we focus on BERT as it embodies several of the essential trends that are important when optimizing accelerators for these networks (discussed in Section 2.3). Furthermore, our analysis on the impact of larger and deeper models as well as of different input sizes (Section 3.3) capture future Transformer trends.

8. Related Work

Characterizing DNNs: Prior work characterize ML workloads, especially CNNs, RNNs and recommendation models. Although most focus on inference [73, 89, 98], some also characterize training [32, 60, 100]. However, Transformer-based models, an important optimization target for modern systems, have received less attention; especially the expensive pre-training phase we focus on. Works that include Transformer characterization either do not provide detailed runtime breakdown amongst operations, only focus on its FC layers, or focus on inference rather than training [33, 89, 92, 98]. Instead, we focus on detailed end-to-end breakdown which helps in identifying LAMB or optimizer updates as one of the important candidates for Transformer acceleration. We also show how Transformer operations scale with varying hyperparameters and when employing different training techniques, which can be useful to project bottlenecks when training future Transformer models. While some works [23, 64] examine the impact of sweeping input size on throughput, they either do not include in-depth characterization that explains the behavior or are focused on inference in CPUs.

Optimizing Transformers: Recent work has also designed accelerators for Transformer-based networks. However, the relative lack of comprehensive characterization of Transformers has led these works to overlook important characteristics of self-attention. For example, recent works design both efficient matrix-vector [33, 36] and matrix-matrix engines [17] to accelerate BERT even though BERT does not execute matrix-vector operation the majority of the time, as our work shows. Unlike in RNNs where tokens are processed one at a time, Transformer layers process all the tokens of the input sequence in parallel. This leads to matrix, rather than vector, operations in Transformer layers even if mini-batch is one (e.g., during inference) as illustrated in Fig. 5. Although some prior work acknowledges this property when comparing their accelerator against GPUs [33], it did not influence the accelerator's design. This confusion about matrix-vector operations in BERT underscores the necessity of our work – understanding DNNs at an algorithmic level – before building efficient accelerators for them. Very few works optimize for non-GEMM operations or data-intensive phases [39], which we show have a significant runtime contribution that increases with reducing precision and increasing layer size. Amongst non-GEMMs, complex optimizers (e.g., LAMB), used in modern NLPs, have received little attention; we highlight LAMB's bandwidth-intensive characteristics and demonstrate how near-memory computing can help accelerate it. Finally, other works optimize Transformer inference [18, 23, 24, 91] or memory management [74].

Near-Memory Computing for Optimizers [46]: GradPIM [46] evaluated NMC for optimizers. However, they only evaluate simple momentum-based optimizers and focus on CNNs, which have an order of magnitude fewer parameters to update compared to NLP models.

9. Conclusion

BERT has been a groundbreaking innovation in NLP. Its accuracy stems from its Transformer architecture, millions of parameters, and its ability to train on enormous, unlabeled datasets. Its success has also inspired several popular models that are larger but have a similar structure to BERT. However, training these models is expensive due

to their large compute and memory requirements. They pose challenges to system designers that must be met through deeper understanding of algorithmic behaviors as the waning of Moore's Law changes the virtuous synergy that has helped propel prior transformative improvements of ML and NLP. Thus, we focus on BERT's most expensive component, pre-training, analyze its execution, and provide a detailed characterization that acts as an exemplar for optimizing Transformer networks. Moreover, we further analyze how these characteristics change with evolving hyperparameters, and training techniques, including mixed precision and in a distributed setting. Our analysis also identifies future acceleration opportunities and we demonstrate how enhancing compute-intensive accelerators with near-memory compute helps accelerate Transformer networks.

10. Acknowledgements

AMD, AMD Ryzen, AMD Radeon, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] D. Abts, J. Ross, J. Sparling, M. Wong-VanHaren, M. Baker, T. Hawkins, A. Bell, J. Thompson, T. Kahsai, G. Kimmell, J. Hwang, R. Leslie-Hurd, M. Bye, E. R. Creswick, M. Boyd, M. Venigalla, E. Laforge, J. Purdy, P. Kamath, D. Maheshwari, M. Beidler, G. Rosseel, O. Ahmad, G. Gagarin, R. Czekalski, A. Rane, S. Parmar, J. Werner, J. Sproch, A. Macias, and B. Kurtz, "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads," in *ISCA*, 2020, p. 145–158.

[2] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks, "Fathom: Reference Workloads for Modern Deep Learning Methods," in *IISWC*, 2016, pp. 1–10.

[3] S. Aga, N. Jayasena, and M. Ignatowski, "Co-ML: A Case for Collaborative ML Acceleration Using near-Data Processing," in *MEMSYS*, 2019, p. 506–517.

[4] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing," in *ISCA*, 2016, pp. 1–13.

[5] J. Alsop, M. S. Orr, B. M. Beckmann, and D. A. Wood, "Lazy Release Consistency for GPUs," in *MICRO*, 2016, pp. 26:1–26:13.

[6] AMD, "AMD ROCm Profiler," rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html, 2019.

[7] ——, "AMD Ryzen™ Threadripper 2950X Processor," https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-2950x, 2019.

[8] ——, "ROCm, a New Era in Open GPU Computing," rocm.github.io/, 2019.

[9] ——, "AMD CDNA ARCHITECTURE," amd.com/system/files/documents/amd-cdna-whitepaper.pdf, 2020.

[10] ——, "AMD Instinct™ MI100 Accelerator," "https://www.amd.com/en/products/serveraccelerators/instinct-mi100", 2020.

[11] J. Appleyard, T. Kociský, and P. Blunsom, "Optimizing Performance of Recurrent Neural Networks on GPUs," *CoRR*, vol. abs/1604.01946, 2016.

[12] O. M. Awad, M. Mahmoud, I. E. Vivancos, A. H. Zadeh, C. Bannon, A. Jayarajan, G. Pekhimenko, and A. Moshovos, "FPRaker: A Processing Element For Accelerating Neural Network Training," *CoRR*, vol. abs/2010.08065, 2020.

[13] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *CoRR*, vol. 1607.06450, 2016.

[14] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," in *NeurIPS*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33, 2020, pp. 1877–1901.

[15] N. Chatterjee, M. O'Connor, D. Lee, D. R. Johnson, S. W. Keckler, M. Rhu, and W. J. Dally, "Architecting an Energy-Efficient DRAM System for GPUs," in *HPCA*, 2017, pp. 73–84.

[16] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks," in *ISCA*, 2016, pp. 367–379.

[17] B. Y. Cho, J. Jung, and M. Erez, "Accelerating Bandwidth-Bound Deep Learning Inference with Main-Memory Accelerators," *CoRR*, vol. 2012.00158, 2020.

[18] Y. Choi, Y. Kim, and M. Rhu, "LazyBatching: An SLA-aware Batching System for Cloud Machine Learning Inference," in *HPCA*, 2020, pp. 493–506.

[19] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context," *CoRR*, vol. 1901.02860, 2019.

[20] C. De Sa, M. Feldman, C. Ré, and K. Olukotun, "Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent," in *ISCA*, 2017, pp. 561–574.

[21] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2019, pp. 4171–4186.

[22] G. Diamos, S. Sengupta, B. Catanzaro, M. Chrzanowski, A. Coates, E. Elsen, J. Engel, A. Y. Hannun, and S. Satheesh, "Persistent RNNs: Stashing Recurrent Weights On-Chip," in *ICML*, 2016, pp. 2024–2033.

[23] D. Dice and A. Kogan, "Optimizing Inference Performance of Transformers on CPUs," *CoRR*, vol. abs/2102.06621, 2021.

[24] J. Fang, Y. Yu, C. Zhao, and J. Zhou, "TurboTransformers: An Efficient GPU Serving System for Transformer Models," in *PPoPP*, 2021, p. 389–402.

[25] J. Filipovič, M. Madzin, J. Fousek, and L. Matyska, "Optimizing CUDA Code by Kernel Fusion: Application on BLAS," *The Journal of Supercomputing*, p. 3934–3957, 2015.

[26] J. Fousek, J. Filipovič, and M. Madzin, "Automatic Fusions of CUDA-GPU Kernels for Parallel Map," *SIGARCH Comput. Archit. News*, p. 98–99, Dec. 2011.

[27] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-scale DNN Processor for Real-time AI," in *ISCA*, 2018, pp. 1–14.

[28] A. Gibiansky, "Bringing HPC techniques to deep learning." https://github.com/baidu-research/baidu-allreduce, 2017.

[29] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. Baghsorkhi, and J. Torrellas, "SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs," in *MICRO*, 2020, pp. 796–810.

[30] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, and J. Torrellas, "SparseTrain: Leveraging Dynamic Sparsity in Software for Training DNNs on General-Purpose SIMD Processors," in *PACT*, 2020, p. 279–292.

[31] Google, "Google Research: BERT," https://github.com/google-research/bert, 2020.

[32] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G. Wei, H. S. Lee, D. Brooks, and C. Wu, "DeepRecSys: A System for Optimizing End-To-End At-Scale Neural Recommendation Inference," in *ISCA*, 2020, pp. 982–995.

[33] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee *et al.*, "Aˆ3: Accelerating Attention Mechanisms in Neural Networks with Approximation," in *HPCA*, 2020, pp. 328–341.

[34] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *ISCA*, 2016, pp. 243–254.

[35] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *HPCA*, 2018, pp. 620–629.

[36] M. He, C. Song, I. Kim, C. Jeong, S. Kim, I. Park, M. Thottethodi, and T. Vijaykumar, "Newton: A DRAM-maker's Accelerator-in-memory (AIM) Architecture for Machine Learning," in *MICRO*, 2020, pp. 372–385.

[37] B. Hechtman, S. Che, D. Hower, Y. Tian, B. Beckmann, M. Hill, S. Reinhardt, and D. Wood, "QuickRelease: A Throughput-Oriented Approach to Release Consistency on GPUs," in *HPCA*, 2014, pp. 189–200.

[38] D. Hendrycks and K. Gimpel, "Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units," *CoRR*, vol. abs/1606.08415, 2016.

[39] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data Movement Is All You Need: A Case Study on Optimizing Transformers," in *MLSys*, vol. 3, 2021, pp. 711–732.

[40] S. M. A. H. Jafri, H. Hassan, A. Hemani, and O. Mutlu, "Refresh Triggered Computation: Improving the Energy Efficiency of Convolutional Neural Network Accelerators," *ACM TACO*, vol. 18, no. 1, 2021.

[41] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist: Efficient Data Encoding for Deep Neural Network Training," in *ISCA*, 2018, pp. 776–789.

[42] JEDEC, "High Bandwidth Memory DRAM (HBM1, HBM2)," jedec.org/standards-documents/docs/jesd235a, 2019.

[43] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017, pp. 1–12.

[44] S. W. Keckler, "Life After Dennard and How I Learned to Love the Picojoule," Keynote at MICRO, 2011.

[45] F. Khorasani, H. A. Esfeden, N. Abu-Ghazaleh, and V. Sarkar, "In-Register Parameter Caching for Dynamic Neural Nets with Virtual Persistent Processor Specialization," in *MICRO*, 2018, pp. 377–389.

[46] H. Kim, H. Park, T. Kim, K. cho, E. Lee, S. Ryu, H.-J. Lee, K. Choi, and J. Lee, "GradPIM: A Practical Processing-in-DRAM Architecture for Gradient Descent," in *HPCA*, 2021.

[47] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An In-Network Architecture for Accelerating Shared-Memory Multiprocessor Collectives," in *ISCA*, 2020, pp. 996–1009.

[48] K. Koukos, A. Ros, E. Hagersten, and S. Kaxiras, "Building Heterogeneous Unified Virtual Memories (UVMs) Without the Overhead," *ACM TACO*, vol. 13, no. 1, pp. 1:1–1:22, Mar. 2016.

[49] S. Kumar, V. Bitorff, D. Chen, C. Chou, B. A. Hechtman, H. Lee, N. Kumar, P. Mattson, S. Wang, T. Wang, Y. Xu, and Z. Zhou, "Scale MLPerf-0.6 models on Google TPU-v3 Pods," *CoRR*, vol. abs/1909.09756, 2019.

[50] S. Kumar, J. Bradbury, C. Young, Y. E. Wang, A. Levskaya, B. Hechtman, D. Chen, H. Lee, M. Deveci, N. Kumar, P. Kanwar, S. Wang, S. Wanderman-Milne, S. Lacy, T. Wang, T. Oguntebi, Y. Zu, Y. Xu, and A. Swing, "Exploring the limits of Concurrency in ML Training on Google TPUs," *CoRR*, vol. 2011.03641, 2020.

[51] Y. Kwon and M. Rhu, "A Disaggregated Memory System for Deep Learning," *IEEE Micro*, vol. 39, no. 5, pp. 82–90, 2019.

[52] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," in *ICLR*, 2019.

[53] S. Lee, K. Kim, S. Oh, J. Park, G. Hong, D. Ka, K. Hwang, J. Park, K. Kang, J. Kim *et al.*, "A 1ynm 1.25 v 8gb, 16gb/s/pin gddr6-based accelerator-in-memory supporting 1tflops mac operation and various activation functions for deep-learning applications," in *ISSCC*, vol. 65, 2022, pp. 1–3.

[54] S. Lee, S.-h. Kang, J. Lee, H. Kim, E. Lee, S. Seo, H. Yoon, S. Lee, K. Lim, H. Shin *et al.*, "Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology: Industrial Product," in *ISCA*, 2021, pp. 43–56.

[55] A. Li, B. Zheng, G. Pekhimenko, and F. Long, "Automatic Horizontal Fusion for GPU Kernels," *CoRR*, vol. 2007.01277, 2020.

[56] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *CoRR*, vol. 1907.11692, 2019.

[57] S. Lym and M. Erez, "FlexSA: Flexible Systolic Array Architecture for Efficient Pruned DNN Model Training," *CoRR*, vol. 2004.13027, 2020.

[58] S. Lym, D. Lee, M. O'Connor, N. Chatterjee, and M. Erez, "DeLTA: GPU Performance Model for Deep Learning Applications with In-depth Memory System Traffic Analysis," in *ISPASS*, 2019, pp. 293–303.

[59] M. Mahmoud, I. Edo, A. H. Zadeh, O. Mohamed Awad, G. Pekhimenko, J. Albericio, and A. Moshovos, "TensorDash: Exploiting Sparsity to Accelerate Deep Neural Network Training," in *MICRO*, 2020, pp. 781–795.

[60] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. A. Patterson, H. Tang, G. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. M. Hazelwood, A. Hock, X. Huang, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. J. Reddi, T. Robie, T. S. John, C. Wu, L. Xu, C. Young, and M. Zaharia, "MLPerf Training Benchmark," *CoRR*, vol. abs/1910.01500, 2019.

[61] T. Norrie, N. Patil, D. H. Yoon, G. Kurian, S. Li, J. Laudon, C. Young, N. P. Jouppi, and D. Patterson, "Google's Training Chips Revealed: TPUv2 and TPUv3," in *IEEE Hot Chips 32 Symposium*, 2020, pp. 1–70.

[62] NVIDIA, "Apex (A PyTorch Extension)," "https://nvidia.github.io/apex/", 2018.

[63] ——, "NVIDIA cuDNN: GPU Accelerated Deep Learning," "https://developer.nvidia.com/cudnn", 2018.

[64] ——, "NVIDIA FasterTransformer," "https://github.com/NVIDIA/FasterTransformer/", 2020.

[65] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, "Timeloop: A Systematic Approach to DNN Accelerator Evaluation," in *ISPASS*, 2019, pp. 304–315.

[66] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks," in *ISCA*, 2017, pp. 27–40.

[67] S. Pati, S. Aga, M. D. Sinclair, and N. Jayasena, "SeqPoint: Identifying Representative Iterations of Sequence-based Neural Networks," in *ISPASS*, 2020, pp. 69–80.

[68] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[69] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models," *CoRR*, vol. 1910.02054, 2019.

[70] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "SQuAD: 100, 000+ Questions for Machine Comprehension of Text," *CoRR*, vol. abs/1606.05250, 2016.

[71] S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "ASTRA-SIM: Enabling SW/HW Co-Design Exploration for Distributed DL Training Platforms," in *ISPASS*, 2020, pp. 81–92.

[72] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators," in *ISCA*, 2016, pp. 267–278.

[73] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "MLPerf Inference Benchmark," in *ISCA*, 2020, pp. 446–459.

[74] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning," in *HPCA*, 2021, pp. 598–611.

[75] C. Sakr, S. Dai, R. Venkatesan, B. Zimmer, W. Dally, and B. Khailany, "Optimal Clipping and Magnitude-aware Differentiation for Improved Quantization-aware Training," in *ICML*, 2022, pp. 19 123–19 138.

[76] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-situ Analog Arithmetic in Crossbars," in *ISCA*, 2016, pp. 14–26.

[77] A. Sharif Razavian, H. Azizpour, J. Sullivan, and S. Carlsson, "CNN Features off-the-shelf: an Astounding Baseline for Recognition," in *CVPR*, 2014, pp. 806–813.

[78] Y. Shen, M. Ferdman, and P. Milder, "Maximizing CNN Accelerator Efficiency Through Resource Partitioning," in *ISCA*, 2017, pp. 535–547.

[79] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro, "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism," *CoRR*, vol. abs/1909.08053, 2019.

[80] M. D. Sinclair, J. Alsop, and S. V. Adve, "Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models," in *MICRO*, 2015, pp. 647–659.

[81] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *HPCA*, 2013, pp. 578–590.

[82] M. Sivathanu, T. Chugh, S. S. Singapuram, and L. Zhou, "Astra: Exploiting Predictability to Optimize Deep Learning," in *ASPLOS*, 2019, p. 909–923.

[83] Z. Song, B. Fu, F. Wu, Z. Jiang, L. Jiang, N. Jing, and X. Liang, "DRQ: Dynamic Region-based Quantization for Deep Neural Network Acceleration," in *ISCA*, 2020, pp. 1010–1021.

[84] M. Springer, P. Wauligmann, and H. Masuhara, "Modular Array-Based GPU Computing in a Dynamically-Typed Language," in *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, 2017, p. 48–55.

[85] Y. Sun, S. Wang, Y. Li, S. Feng, H. Tian, H. Wu, and H. Wang, "ERNIE 2.0: A Continual Pre-training Framework for Language Understanding," *CoRR*, vol. 1907.12412, 2019.

[86] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-Read Students Learn Better: On the Importance of Pre-training Compact Models," *CoRR*, vol. abs/1908.08962, 2019.

[87] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *NeurIPS*, 2017, p. 6000–6010.

[88] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "ScaleDeep: A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in *ISCA*, 2017, pp. 13–26.

[89] S. Verma, Q. Wu, B. Hanindhito, G. Jha, E. B. John, R. Radhakrishnan, and L. K. John, "Demystifying the mlperf training benchmark suite," in *ISPASS*, 2020, pp. 24–33.

[90] G. Wang, Y. Lin, and W. Yi, "Kernel Fusion: An Effective Method for Better Power Efficiency on Multithreaded GPU," in *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, 2010, p. 344–350.

[91] H. Wang, Z. Zhang, and S. Han, "SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning," in *HPCA*, 2021, pp. 97–110.

[92] Y. Wang, G.-Y. Wei, and D. Brooks, "A Systematic Methodology for Analysis of Deep Learning Hardware and Software Platforms," *MLSys*, pp. 30–43, 2020.

[93] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized Autoregressive Pretraining for Language Understanding," *CoRR*, vol. 1906.08237, 2020.

[94] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How Transferable Are Features in Deep Neural Networks?" in *NeurIPS*, 2014, p. 3320–3328.

[95] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, and C.-J. Hsieh, "Reducing BERT Pre-Training Time from 3 Days to 76 Minutes," *CoRR*, vol. abs/1904.00962, 2019.

[96] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism," in *ISCA*, 2017, pp. 548–560.

[97] A. H. Zadeh, I. Edo, O. M. Awad, and A. Moshovos, "GOBO: Quantizing Attention-Based NLP Models for Low Latency and Energy Efficient Inference," in *MICRO*, 2020, pp. 811–824.

[98] A. H. Zadeh, Z. Poulos, and A. Moshovos, "Deep Learning Language Modeling Workloads: Where Time Goes on Graphics Processors," in *IISWC*, 2019, pp. 131–142.

[99] B. Zheng, N. Vijaykumar, and G. Pekhimenko, "Echo: Compiler-based GPU Memory Footprint Reduction for LSTM RNN Training," in *ISCA*, 2020, pp. 1089–1102.

[100] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "TBD: Benchmarking and Analyzing Deep Neural Network Training," in *IISWC*, 2018.

[101] M. Zhu, M. Rhu, J. Clemons, S. W. Keckler, and Y. Xie, "Training Long Short-Term Memory With Sparsified Stochastic Gradient Descent," https://openreview.net/forum?id=HJWzXsKxx, 2016.