Training Giant Neural Networks Using Weight Streaming on Cerebras Wafer-Scale Systems

Stewart Hall, Rob Schreiber, Sean Lie, Cerebras Systems, Inc.

Abstract

State-of-the-art language models have grown in parameter count by three orders of magnitude over the last two years. This growth has presented challenges for training both in terms of storage and compute requirements. In this paper, we survey existing approaches used to scale training to clusters of compute units and explore the limitations of each in the face of giant models. One thing that all these approaches have in common is the storage of model parameters on the compute units, which we find to be a primary driver of complexity and communication overhead. We present a new paradigm for giant model training, called weight streaming, based on the disaggregation of model storage and compute, and describe an implementation of this paradigm using Cerebras wafer-scale systems. Our weight streaming architecture enables the training of models two orders of magnitude larger than the current state-of-the-art, with a simple scaling model. Combined with built-in support for weight sparsity, our solution can make training giant networks tractable for the first time.

Table of Contents

Trends in Large Neural Network Models

State-of-the-art natural language processing (NLP) models have increased in size by three orders magnitude in the past two years, and we expect this trend to continue (Figure 1). The recent GPT-3 paper[1] reports that in the 2018-2020 period, networks presented in the literature grew rapidly, with, successively, 0.1, 0.3, 1.5, 8, 11, and 17 billion parameters. The GPT-3 language model greatly exceeds these in size, with 175 billion parameters. There were significant benefits to size, especially a demonstrated rapid specialization of the pre-trained model to a narrow context with only a few additional training examples. Networks, such as the Switch Transformer,[2] are now pushing past the trillion-parameter mark.

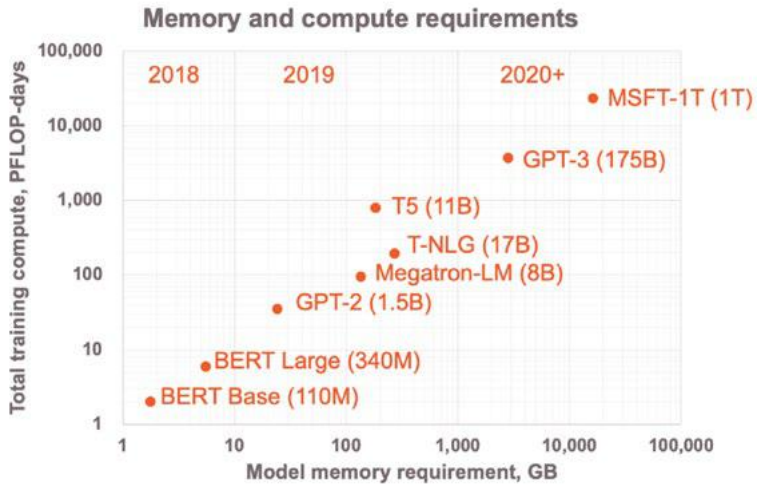

Figure 1. Growth of NLP models over the past three years.

Because of the growth of network size, training is taking more memory and more time. GPT-3 was estimated to require 3.15×10^{23} total floating-point operations to train and requires 2.8TB of memory just to store weights and optimizer state (weights, gradients, and momentum terms). An additional 7.5TB would be required to store activations for a single training batch of 3.2 million tokens, assuming that one batch of activations is stored for each of the 96 layers. The number of floating-point operations required to train GPT-3 is one billion times greater than the peak FLOPS delivered by an NVIDIA® A100 GPU, meaning that it would take over 30 years to train the model on a single GPU at full utilization! Moreover, the memory footprint of training, including parameters and activations, is 125x greater than the memory capacity of a single A100. Clearly the key to training models of this size is to scale out and harness the compute speed and memory capacity of many devices.

With models growing in computational complexity and outpacing the capabilities of individual compute units, we must scale training to multiple compute units. The computational workload is commonly distributed by splitting the batch or the model over many compute units, such as GPUs. NVIDIA® estimates that GPT-3 could be scaled to 1,000 GPUs, reducing the training time to about 34 days[3]. Distributing training to a cluster of this size is complicated, but training on a small cluster is impossible due to the memory required to store model parameters and activations. This is because common approaches to distributed training require the model to be stored entirely in the memory of

the compute units, what we refer to as *stored weight training*. Even if it were possible to work around memory constraints, a training time lasting years would be prohibitive. In this paper, we describe a new paradigm for training giant neural networks, called *weight streaming*, where model parameters are not stored in the memory of the compute units. Weight streaming allows the cluster size to scale independent of model size. Using *weight streaming* to harness the power of the Cerebras CS-2 system, we can reduce training time and simplify training at scale.

Concepts in NLP Model Training

This section gives a brief overview of the structure and process of training NLP models and defines related terminology used throughout the paper. Readers who are already familiar with these concepts may skip this section. NLP models process sequences of tokens which commonly represent text as a series of words. We use S to represent the number of tokens (words) in each sequence. Each token is encoded as an integer value corresponding to a word in the model's vocabulary. An NLP model converts a source sequence into a target sequence through a series of transformations, beginning with an embedding table lookup. The embedding table contains a vector of features, called a hidden state vector, for each word in the model's vocabulary. The number of features in each hidden state vector, which we refer to as H, is a property of the model and typically numbers in the thousands. The per-token hidden state vectors propagate through the model's layers, each of which transforms the input vector to an equal-length output vector. Layers use sets of trainable parameters to transform the hidden state vectors with operations such as matrix multiplications and element-wise additions. We represent the number of layers in the model as L, also referred to as the depth. The total number of parameters in the model is represented as P, which includes the embedding table and each layer's parameters. Throughout this paper, model size is used to refer to the number of parameters. The hidden state of an input sequence at any point in the model is represented by an activation tensor, which consists of a hidden state vector for each token of the input sequence. As a shorthand, we use T to represent the activation tensor size for one sequence:

$$T = H \times S$$

Training an NLP model is performed by feeding labeled inputs, called training samples, through the model to compute gradients which can be used to update the model's parameters. The output of the model for each sample is compared with the label to compute an error measure, which is differentiated to compute an activation gradient. The gradient is backpropagated through the layers of the model to compute both a weight gradient and new activation gradient at each layer. A training job consists of multiple steps, called training iterations. In each training iteration a subset of the training dataset, called a batch, is used to compute weight gradients for each layer. These weight gradients are consumed by an optimizer algorithm to update the model's weights between each training iteration. Stochastic gradient descent (SGD) and Adam are commonly used optimizer algorithms. Training iterations are performed until the model has converged, i.e. when an error measure levels off.

Methods for Scaling Training with Stored Weights

Training can be distributed to many compute units using data parallelism or model parallelism (Figure 2). Data parallelism splits the batch of training samples over N_d compute units with each compute unit processing all layers of the model for its subset of the batch. Model parallelism splits the model over multiple compute units, with each compute unit processing a subset of the layers for all samples in the batch. The model can either be split by placing a subset of each layer on each of N_m compute units, called tensor model parallelism, or by placing a subset of the layers on each of N_p compute units, called pipeline model parallelism. Both modes of parallelism can be combined, for example by creating N_d instances of the model, sharding the batch into N_d shards, and distributing each model instance across a set of N_m compute units, thereby using $N_d \times N_m$ compute units.

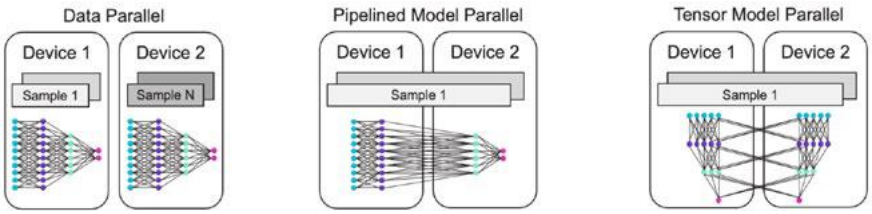

Figure 2. Approaches to parallelism in neural network training.

This section quantifies memory and communication needed by each of the scaling approaches. The memory and communication requirements of these common scaling approaches are summarized in Table 1. Memory requirements are calculated based on the size of model weights and per-layer activation tensors stored on a single compute unit. The communication models account for transfers of weight gradients and per-layer activation tensors into and out of a single compute unit. Although communication is also needed to transfer each batch of training samples to the compute units, we do not account for this in our models because the inputs to NLP models, which are the focus of this paper, are very small relative to gradients and activation tensors. Inputs consist of a single integer per token, representing a word ID in the vocabulary, whereas activation tensors consist of a H-length vector per token.

Data Parallelism

Data parallelism places a copy of the model on each of N_d compute units and shards each batch of training samples over the compute units. The number of training samples processed by each compute unit, b_d, is:

$$b_d = \frac{B}{N_d}$$

After each unit has computed partial weight gradients using its shard of the batch, gradients are summed between all compute units to get the gradient for the full batch. Each compute unit uses the summed gradient to update its copy of the weights, guaranteeing that the replicas of the model stay in sync.

Memory is needed on each compute unit to store model parameters and activations. Each compute unit stores an entire replica of the model and an activation tensor per-layer for each sample in its shard of the batch, needed for gradient computations in the backward pass. The total memory requirement per compute unit is:

$$Memory = P + LTb_d$$

At the end of every training iteration, compute units communicate in order to reduce gradients prior to each weight update. They perform an all-reduce operation, where each compute unit receives a copy of the sum of the original, partial gradient tensors. Commonly used algorithms for all-reduce require each node to send and receive the entire gradient for each layer's tensor once or twice, so total communication per compute unit is:

$$Communication\ volume\ = O(P)$$

Model Parallelism

Model parallelism involves distribution of the model's layers over multiple compute units. Individual layers can be split over multiple compute units, or each compute unit can contain groups of entire layers, also known as pipeline model parallelism. Parameters are not replicated when model parallelism is used alone: each compute unit stores a subset of the model parameters corresponding to the computations which it performs. Activations are communicated between compute units as a batch of training data progresses through the model.

Tensor Model Parallelism

One method of model parallelism, known as tensor model parallelism, splits each layer over N_m compute units. For example, a fully connected layer can be split on its input features, its output features, or both. When input features are split, each compute unit computes partial sums for all output features, then a reduction is required to compute output features. When output features are split, input activations are broadcast to each compute unit where a subset of output features are computed for each sample. Storage of parameters and activations, for gradient computations, is distributed, so memory required per compute unit is:

$$Memory = \frac{P}{N_m} + \frac{LTB}{N_m}$$

Communication among compute units is required, at the transition between finishing a layer and beginning the next layer, to reduce partial sums or redistribute the output tensor so that it matches the distribution of weights used by the next layer. When weights are split on either the input feature or output feature dimension, the communication required for each layer is proportional to the activation tensor size:

$$Communication\ volume\ = O(LTB)$$

If weights are split on both input and output feature dimensions, then the minimal communication required decreases proportional to the square root of compute units:

$$Communication\ volume\ = O\left(\frac{LTB}{\sqrt{N_m}}\right)$$

Pipeline Model Parallelism

Pipeline model parallelism is a different method of model parallelism where each compute unit is responsible for a subset of the model's layers. Each subset of layers constitutes a stage in a large pipeline. The training batch is split into shards, of size b_p, to keep the pipeline full, with each stage operating on a different shard of the training batch at any given time. We denote the number of pipeline stages as N_p, so the number of layers allocated to each compute unit is roughly equal to:

$$Layers\ per\ compute\ unit = \frac{L}{N_p}$$

There is a latency to fill and drain the pipeline between each weight update, so the number of batch shards, equal to B / b_p, should be high relative to N_p. Cerebras has employed pipeline model parallelism effectively for moderate sized networks, since the company's Wafer-Scale Engine (WSE) can operate efficiently on a batch size of one in each pipeline stage ($b_p=1$). Memory is distributed across compute units and is used to store parameters and activations which are stored for each layer processed by the compute unit. Activations need to be stored for the duration of time between when a batch shard is processed in the forward pass and when it is processed in the backward pass. This means that the number of samples buffered for each layer depends on the layer's location in the network pipeline, with the first pipeline stage storing activations for $2N_p-1$ batch shards and the last pipeline stage storing activations for only one batch shard. Increasing the number of compute units increases the pipeline depth, which increases the aggregate memory required for activation buffers. As a result, the percentage of memory used for activation buffers increases with the number of compute units. The worst-case memory required on a single compute unit for parameters and activations is:

$$Memory = \frac{P}{N_p} + \left(2N_p - 1\right)\frac{L}{N_p}Tb_p$$

The bandwidth required to move activations between pipe stages for the full batch is:

$$Communication\ volume\ = O(TB)$$

Scaling Approach	Activation Memory	Parameter Memory	Communication Volume	Main Limitations
Data Parallel (DP)	$\frac{LTB}{N_d}$	P	P	Parameter memory does not decrease with N.
Tensor Model Parallel (TMP)	$\frac{LTB}{N_m}$	$\frac{P}{N_m}$	$\left(\frac{LTB}{\sqrt{N_m}}\right)$	Communication does not scale as well as compute.
Pipeline Model Parallel (PMP)	$\left(2N_p - 1\right)\frac{L}{N_p}Tb_p$	$\frac{P}{N_p}$	TB	Activation memory and communication do not decrease with N.
DP+TMP	$\frac{LTB}{N_d N_m}$	$\frac{P}{N_m}$	$\frac{P}{N_m} + \frac{LTB}{\sqrt{N_m}}N_d$	Communication does not scale as well as compute. Complexity.
DP+PMP	$\left(2N_p - 1\right)\frac{L}{N_p}Tb_p$	$\frac{P}{N_p}$	$\frac{P}{N_p} + \frac{TB}{N_d}$	Sub-linear activation memory and communication scaling. Complexity.
DP+TMP+PMP	$\left(2N_p - 1\right)\frac{L}{N_p}\frac{T}{N_m}b_p$	$\frac{P}{N_m N_p}$	$\frac{P}{N_m N_p} + \frac{LTB}{\sqrt{N_m}}N_d N_p$	Complexity.

Table 1. Overview of memory and communication requirements of common scaling approaches. The last column highlights the primary limitation when scaling for a giant model.

Limitations of Scaling Stored-Weight Training

When scaling to multiple compute units, data parallelism is commonly chosen for its simplicity: each compute unit performs the same computations and participates in one communication step, to synchronize gradients, once per training iteration. However, data parallelism alone fails for giant models due to its memory requirement per compute unit. The activation component of this memory requirement can be reduced by increasing N_d, which decreases b_d. The model parameter component of this memory requirement is not reduced as N_d increases, which means model size is limited by the compute unit's memory capacity. For GPT-3, the weights alone require 700GB of memory, which is an order of magnitude greater than the capacity of a typical GPU.

Tensor model parallelism allows for the memory requirement per compute unit to be reduced as N_m grows, but it introduces significant communication overhead. Activation tensors must be communicated between compute units for each layer. The compute requirement per compute unit for a fully connected layer is $O(H^2SB/N_m)$ and the amount of communication required for activations is $O(HSB/\sqrt{N_m})$. Training on a large cluster with tensor model parallelism alone is likely to be communication bottlenecked since compute units often provide orders of magnitude more FLOPS than network bandwidth. Unlike data parallelism, where batch size can be increased to amortize this overhead, this bottleneck cannot be as easily overcome because the batch size also factors into the volume of communication needed for activations. Due to the frequency of communication, latency can also become a critical component of training time if the cluster is large enough.

Both the frequency of communication and required parameter memory are addressed by pipeline model parallelism, but the aggregate memory needed for activations grows with N_p. Earlier layers in the network need to store more activation buffers as the number of pipeline stages increases, which counters the effect of distributing the layers to more compute units. Since the percentage of memory used for activations increases with the number of compute units, scaling out with pipeline model parallelism for the purpose of fitting a larger model has diminishing returns. Pipeline model parallelism also introduces implementation complexities. Each compute unit may be executing different computations and communicating different types of tensors. The network needs to be distributed such that each pipeline stage completes in the same amount of time. Furthermore, as the pipeline depth increases, the overhead of filling and draining the pipeline between weight updates becomes more significant, requiring a larger batch size.

These three approaches to parallelism can be combined to make training of giant neural networks feasible, both in terms of compute time and memory required per compute unit. However, the complexity introduced by these hybrid approaches puts them out of reach for many users. Furthermore, all of these stored-weight solutions share one problem: that the number of compute units required is partially dictated by the number of parameters in the network. The total number of compute units used by a specific scale-out configuration is:

$$N = N_d \times N_m \times N_p$$

For a neural network like GPT-3 with its large parameter count P, there is a smallest value of N below which the model cannot be trained in the stored-weight paradigm due to per-compute unit memory requirements. This is problematic because the compute requirement does not always scale with the model's parameter count. Workloads such as fine-tuning require storage for the entire set of parameters but involve far less compute than pre-training. Allocating the same amount of compute power to both workloads does not make sense. Architectural changes to the neural network, such as the use of sparse attention, similarly reduce compute without reducing storage required for parameters.

Clearly a new solution to scaling is needed to enable efficient training of giant neural networks. The new solution should:

1. Not impose constraints on model size based on the memory available on individual compute units,
2. Be able to scale compute throughput with the computational requirements of the model, and
3. Achieve scaling without complicated hybrid approaches to parallelism.

To accomplish this decoupling of compute performance from model size, we have developed a radical new approach to training giant neural networks. It is based on three key points:

1. The replacement of CPU and GPU processing by wafer-scale accelerators such as the Cerebras CS-2 system. This change reduces the number of compute units needed to achieve an acceptable compute speed.
2. To meet the challenge of model size, we employ a system architecture that disaggregates compute from model storage. A compute service based on a cluster of CS-2 systems (providing adequate compute bandwidth) is tightly coupled to a memory service (with large memory capacity) that provides subsets of the model to the compute cluster on demand. As usual, a data service serves up batches of training data to the compute service as needed.
3. An innovative model for the scheduling and coordination of training work across the CS-2 cluster that employs data parallelism, layer at a time training with sparse weights streamed in on demand, and retention of activations in the compute service.

We will give below the details of an implementation of this new approach, based on the Cerebras CS-2 system, and we will assess its effectiveness for training GPT-3 as well as order-of-magnitude smaller and larger models.

Introducing Weight Streaming

Our new approach to training is called weight streaming. It is based on the disaggregation of compute and storage. Similar solutions have been developed which spread model weights over multiple compute units and insert communication as necessary[4], but our approach takes this a step further. In weight streaming, model weights are not stored in the memory of the compute units at all, as has been traditionally done. Instead, we move the weights to a separate memory service characterized by a high storage capacity with relatively low compute capabilities. This means that each component of the solution can be optimized for its specific role. Compute units can be optimized for high floating-point throughput on linear algebra operations and the memory service can be optimized for capacity and bandwidth. Specialization is commonly applied in existing distributed training solutions for storage of the training dataset. In like manner, we store the weights of a giant model in a separate memory service and stream them to the compute units as needed. The dataset service and memory service have different requirements, so they are separate components of the solution. The most important difference is the bandwidth requirement, which is generally lower for the dataset service when training NLP models. As a result, we focus only on the compute unit and memory service for the remainder of this paper.

Weight streaming can be combined with existing approaches for parallel training. The training batch can be sharded over N_d compute units to leverage data parallelism, requiring weights to be broadcast from the memory service to all compute units and weight gradients to be reduced between compute units. Tensor model parallelism can be used with weight streaming by splitting activations on their feature dimension over N_m compute units and pairing separate memory services with each compute unit. Pipeline model parallelism can similarly be combined with weight streaming by splitting activations by layer over N_p compute units and pairing separate memory

services with each compute unit. Weight streaming, in all cases, requires bandwidth proportional to P for streaming weights to the compute units and gradients from the compute units. This matches the bandwidth required for stored-weight data parallelism, so does not introduce extra bandwidth requirements. Combining weight streaming with model parallelism increases bandwidth requirements since data movement is required for activations, between compute units, in addition to weights and gradients between the memory service and compute units. For this reason and reduced implementation complexity, we have focused on data parallelism *($N_m = N_p = 1$ and $N = N_d$)* in the Cerebras weight streaming implementation. Using this approach to scaling, an interconnect is needed to link the single memory service to the cluster of compute units.

The Parameters of the Problem and Requirements on the Services

We designed the Cerebras weight streaming solution to enable scalable training of the largest existing neural network models and to power the development of future models. Specific examples of these models inform the requirements for the compute units, memory service, and interconnect. Observing trends in the sizes and parameters of these models allows us to further refine the requirements to best support the needs of future research. NLP models have seen the most growth in parameter counts recently, requiring large-scale training, so these are our initial focus. Table 2 details several of the largest transformer-based language models introduced over the past three years.

Model	Parameters (P)	Layers (L)	Features (H)	Sequence Length (S)
GPT-2 XL [5]	1.5×10^9	48	1,600	1,024
Megatron-8.3B [6]	8.3×10^9	72	3,072	1,024
Turing-NLG [7]	1.72×10^{10}	78	4,256	1,024
GPT-3 [1]	1.75×10^{11}	96	12,288	2,048
Megatron T-NLG [8]	5.3×10^{11}	105	20,480	2,048

Table 2. Evolution of large transformer-based language models over the past 3 years. Note that the features (H) is also sometimes referred to as d_{model}.

The Required Performance

The compute units in the cluster should be capable of training these large models in a reasonable amount of time. To estimate the compute requirement for the cluster, we calculate a target floating-point operation rate which the cluster would need to deliver to train each model in one week. The number of floating-point operations required to train each model can be estimated given the number of tokens needed to train each model to convergence. Forward propagation uses each model parameter once per token. Accounting for weight and activation gradient computations, each model parameter is used three times per token. In all three cases, the weight is used for a multiply-accumulate operation, resulting in 6 total floating-point operations per

parameter per token. The number of operations needed to train the model is dominated by the fully connected layers, so we can approximate the operation count for each model, shown in Figure 3 and Table 3, by multiplying 6 × tokens × parameters. Dividing the estimated total operations by 604,800, the number of seconds in one week, gives a target for the massive floating-point performance required. For comparison, the target for Megatron T-NLG of 1,420 petaFLOPS is more than three times greater than the performance, 442 petaFLOPS, of Fugaku, currently the world's largest supercomputer!

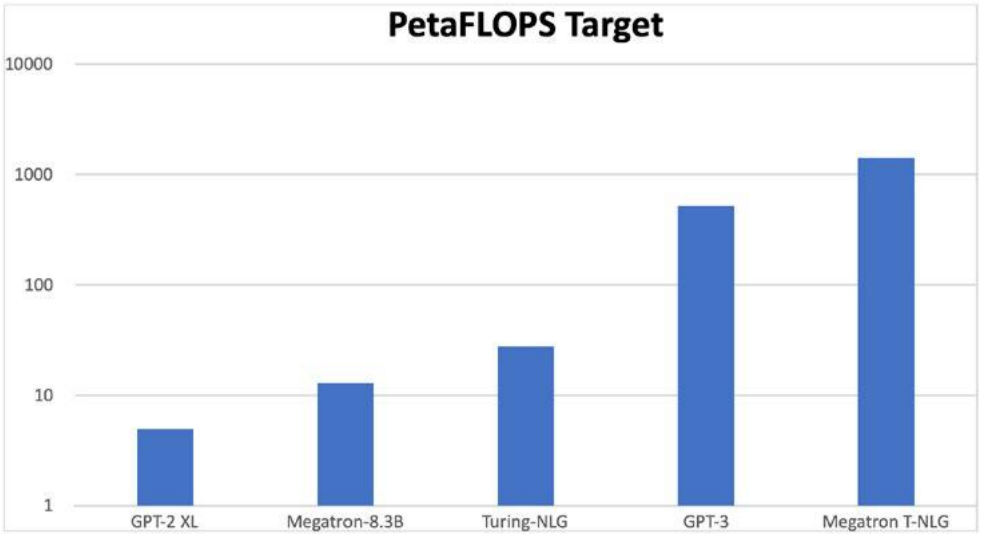

Figure 3. PetaFLOPS required to train each model in 1 week.

Model	Tokens to Train	Total Operations	PetaFLOPS Target
GPT-2 XL	3.00×10^{11} (est.)	2.7×10^{21}	5
Megatron-8.3B	1.57×10^{11}	7.82×10^{21}	13
Turing-NLG	1.57×10^{11}	1.62×10^{22}	28
GPT-3	3.00×10^{11}	3.15×10^{23}	520
Megatron T-NLG	2.7×10^{11}	8.59×10^{23}	1,420

*Table 3. From left to right, (1) tokens to train the model to convergence or batch size multiplied by total training steps (2) total floating-point operations used to train the model based on 6*tokens*parameters (3) target petaFLOPS for a cluster to train in one week.*

The Required Memory Service Capacity

The main requirement of the memory service is capacity. Enough capacity is needed to hold the weight, gradient, and optimizer state for each parameter in the model. Adam is a popular optimizer choice, which requires two momentum terms per parameter. Assuming that weight updates are computed in single-precision floating point format (FP32), the memory service must store 16 bytes per model parameter to contain the weight, gradient, and two momentum terms. We further round this up to 20 bytes per weight to account for a sparse working copy of the nonzero weights, which includes a 2-byte column index and FP16 copy of the weight. Capacity required of the memory service can then be computed by multiplying a model's parameter count by 20 bytes (Figure 4 and Table 4).

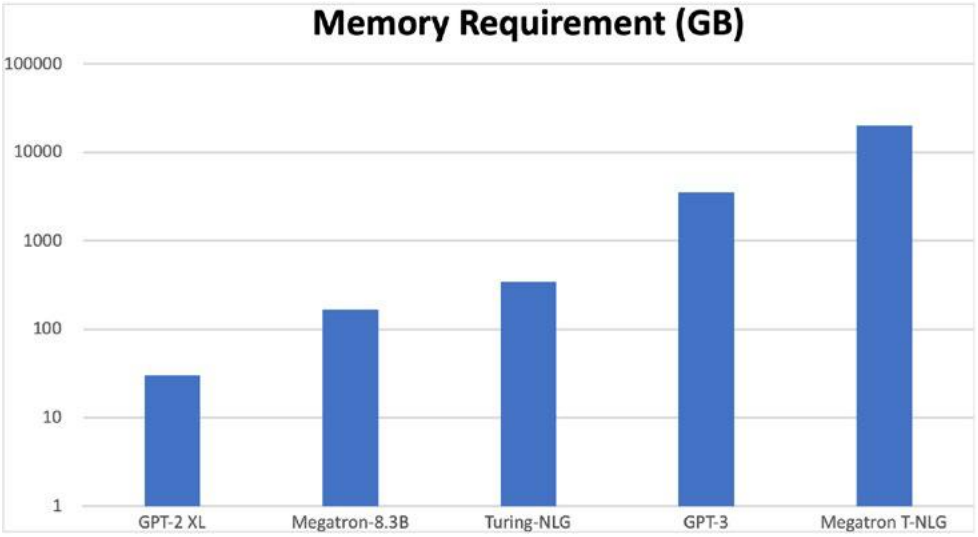

Figure 4. Memory required to store model parameters and optimizer state.

Model	Total Parameters	Memory Requirement
GPT-2 XL	$1.5x10^9$	30GB
Megatron-8.3B	$8.3x10^9$	166GB
Turing-NLG	$1.72x10^{10}$	344GB
GPT-3	$1.75x10^{11}$	3.5TB
Megatron T-NLG	$5.3x10^{11}$	10.6TB

Table 4. Total parameters and storage size of model parameters and optimizer state.

The Required Interconnect Bandwidth

The interconnect between the memory service and the compute units should provide enough bandwidth to support both the flow of weights to the compute units and the flow of weight gradients from the compute units at a rate determined by the compute speed. Compute units must be supplied with weights fast enough to avoid bubbles (unused processor cycles). During the backward pass, weight gradients must be streamed out of the compute units as fast as they are computed. We compute the bandwidth requirement by dividing the volume of data fed into and

out of the compute units by the target compute time of one week. Weights are represented by half-precision floating-point (FP16) values in our implementation. They are fed into the compute units once during the forward pass, for computation of activations, and once during the backward pass, for computation of activation gradients. Weight gradients are sent out of the compute units in FP32 format during the backward pass. In total, 32 bits of data are sent in each direction per parameter in the model per training iteration (Figure 5 and Table 5). (As is now commonplace, gradients are computed and weights are updated at 32-bit precision, which preserves accuracy over the many learning steps during training. On streaming to the compute units, they are rounded to 16-bit precision for use there.)

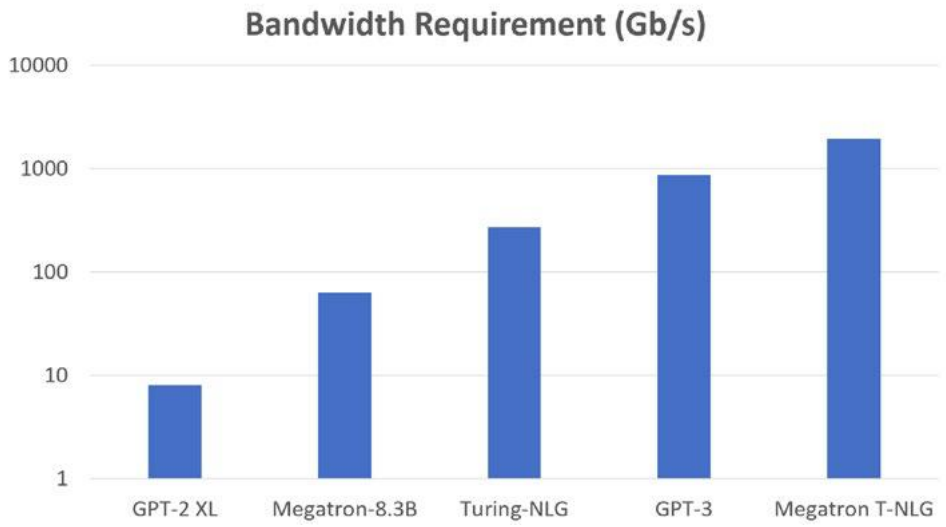

Figure 5. Bandwidth required for weight and gradient communication to train each network in 1 week.

Model	Total Parameters	Tokens to Train	Batch Size	Training Iterations	Bandwidth Requirement
GPT-2 XL	1.50×10^{9}	3.00×10^{11} (est.)	5.24×10^{5}	1×10^{5} (est.)	8Gb/s
Megatron-8.3B	8.30×10^{9}	1.57×10^{11}	1.1×10^{6}	1.43×10^{5}	63Gb/s
Turing-NLG	1.72×10^{10}	1.57×10^{11}	5.24×10^{5}	3.00×10^{5}	273Gb/s
GPT-3	1.75×10^{11}	3.00×10^{11}	3.2×10^{6}	9.37×10^{4}	868Gb/s
Megatron T-NLG	5.30×10^{11}	2.70×10^{11}	3.9×10^{6}	6.92×10^{4}	1.94Tb/s

Table 5. From left to right, (1) Total parameters in the model (2) Total tokens needed to train the model (3) Batch size in tokens used to train the model based on published values (4) Total training iterations calculated by dividing tokens to train by batch size (5) bandwidth required in each direction computed by multiplying number of parameters by number of iterations by 32 bits and dividing by seconds per week.

The estimates in Tables 3, 4, and 5 provide bounds for the compute, memory, and bandwidth capabilities required of our weight streaming implementation. For example, the compute units should scale from a combined compute throughput of tens to thousands of petaFLOPS based on the floating-point operations required to train each model. The memory service should support a minimum of 4TB capacity to enable training of GPT-3 sized models.

Components of the Cerebras Solution: Wafer-Scale Engine, MemoryX, SwarmX

The Cerebras weight streaming implementation is built around our second-generation Wafer-Scale Engine (WSE-2), which lies at the heart of the CS-2 system. Storage for model parameters and optimizer state is disaggregated from compute into our memory service, called the MemoryX service. The MemoryX service provides persistent storage for the parameters. It accepts weight gradients and uses stored optimizer parameters to compute weight updates between training iterations. The Cerebras weight streaming architecture allows for training to be scaled, without replication of the model, to a cluster of CS-2 systems, each served by a single MemoryX service. Weights and gradients are shuttled between the single MemoryX service and multiple CS-2 systems by our SwarmX interconnect fabric.

Compute: The Wafer-Scale Engine

The Cerebras weight streaming implementation harnesses the power of the WSE-2 to compute activations, activation gradients, and weight gradients for each batch of training samples. The WSE-2 is an optimal compute unit for training giant neural networks because it can parallelize computation of massive layers over its 850,000 cores. The on-wafer network of the WSE-2 offers an aggregate 220Pb/s of bandwidth, which is three orders of magnitude more than the aggregate bandwidth between GPUs in an NVIDIA DGX™ server. This bandwidth is uniform between cores, meaning that the WSE-2 can truly be treated as a single massive compute unit. This allows giant layers in our weight streaming implementation to be executed extremely quickly on a single compute unit, avoiding the need to employ tensor model parallelism.

Activations computed during the forward pass and intermediate activation gradients computed during the backward pass are stored in the WSE-2's on-wafer memory. With 40GB of SRAM distributed across 850,000 cores, the WSE-2 has enough activation storage for massive layers and provides 20PB/s of aggregate memory bandwidth for ultra-fast access to activations during computation. The activation tensors stored on-wafer are used for computation of the subsequent layer's activations as weights arrive from the MemoryX service. During the backward pass the stored activation and activation gradient tensors are used for computation of weight gradients which are sent back to the MemoryX service. The WSE-2's 1.2Tb/s of I/O bandwidth is used for receiving weights from and transmitting gradients back to the MemoryX service.

Exploiting weight sparsity

While the WSE-2 allows each layer to be distributed over hundreds of thousands of cores, reducing computation time, parallelism is not the only way to reduce training times. In response to the explosive growth in model size, researchers have been developing smarter models which make more efficient use of their parameters. Weight sparsity is one promising way to reduce parameter count, and in recent tests large drops in training floating-point operations were reported, as shown in Table 6. The Lottery Ticket Hypothesis[9] for example, showed that model parameters can be pruned by 90% without a reduction in accuracy.

Technique	Sparsity	FLOP ↓	Reference
Fixed Sparse Training	90%	**8x**	Lottery Ticket [MIT CSAIL]
Dynamic Sparse Training	80%	**2x**	Rig the Lottery [Google Brain, DeepMind]
Scaling-up Sparse Training	90%+	**10x+**	Pruning scaling laws [MIT CSAIL]
Monte Carlo DropConnect	50%	**2x**	DropConnect in Bayesian Nets [Nature]

Table 6. Some Existing Sparsity Research Examples.

Realizing a performance improvement from this kind of unstructured sparsity, while historically difficult due to hardware limitations, will be essential to affordably training larger models. The WSE-2's unmatched performance on sparse linear algebra operations is key to achieving our targeted training times.

Using the WSE-2 for weight streaming allows us to reduce training time by automatically taking advantage of weight sparsity. The WSE-2 is the only processor that handles unstructured sparsity at the silicon level. The WSE accelerates sparse computation using dataflow scheduling where computational work is triggered by the arrival of data over the fabric. Zeros are omitted when a tensor is transmitted over the fabric, which results in a reduction of computation time proportional to the sparsity of the tensor. In the weight streaming execution mode, model weights are transmitted over the fabric, reducing compute proportional to the sparsity of the weights. Cores in the WSE take advantage of the enormous memory bandwidth available to operate at full utilization on a single weight value at a time, providing a speedup even for weights with unstructured sparsity. Sparsity can reduce wall-clock time of both weight communication and all four computational phases: activation, activation gradient, weight gradient, and weight update. This allows the Cerebras weight streaming solution to train giant networks faster without a bottleneck. Figure 6 uses projections based on measured utilization to show how weight sparsity can be leveraged to further reduce training time. Models the size of GPT-3 can be trained in a single day.

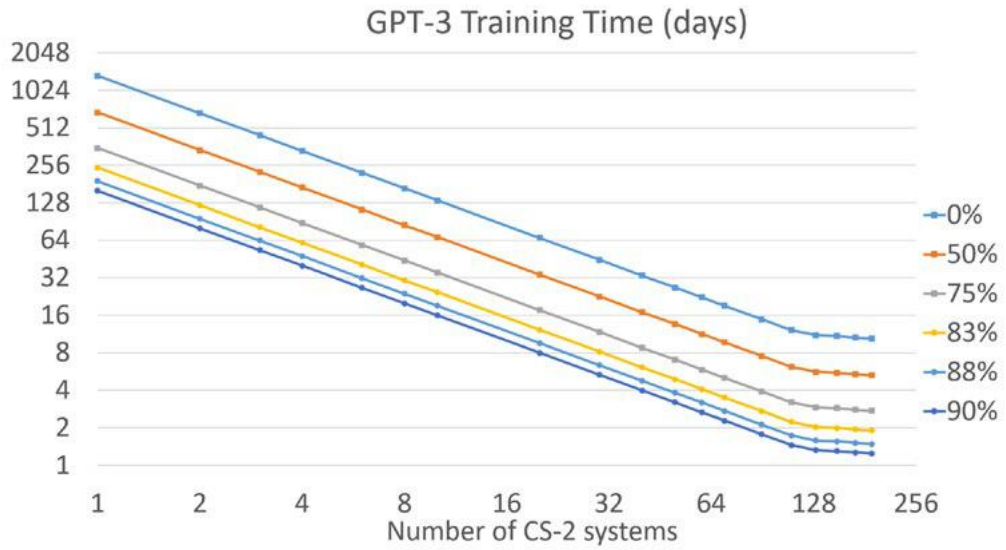

Figure 6. Time needed to train GPT-3 plotted against number of CS-2s at different levels of weight sparsity.

Weight Storage: The MemoryX Service

Model parameters and optimizer state are stored using the MemoryX service, where they are also updated between each training iteration. The capacity of the MemoryX service can scale from 4TB to 2.4PB, allowing the solution to support models with up to 120 trillion parameters. Internally, the MemoryX architecture uses both DRAM and flash storage in a hybrid fashion to achieve both high performance and high capacity. Achieving full compute utilization requires enough network and memory bandwidth to feed weights into the compute units as fast as they are consumed for computations. Both the storage and I/O interface of the MemoryX service can match or exceed the I/O bandwidth of a CS-2 system.

It is important that parameters can be accessed by the compute units with minimal latency, to avoid bubbles during the training process. The process of streaming weights for each layer can be pipelined since weights for most layers can be accessed before computations of the previous layer complete. The one exception to this occurs at the boundary between training iterations, when weights are updated. This means that latency can be hidden for most of the training iteration and has a minimal impact on performance.

Weight updates are performed by the MemoryX service, which provides flexible compute capable of supporting any optimizer algorithm, such as SGD or Adam. The amount of compute required for weight updates is relatively low compared to the compute used to calculate activations and gradients. This is because the number of weight update operations is proportional to the number of parameters, O(P), but activation and gradient compute increases linearly with the batch size, O(BSP). For the same reason, it is possible for the compute provided by the MemoryX service to support any size of CS-2 cluster. Weight updates must be computed at least as fast as weights are streamed out to the CS-2s to avoid a compute bottleneck. Each weight is streamed out of the MemoryX service twice between each weight update, once in the forward pass and once in the

backward pass. The MemoryX service delivers a FLOP/s rate three orders of magnitude greater than its I/O bandwidth, which allows for execution of thousands of floating-point operations per weight on each training iteration. This is plenty of compute power to support any commonly used optimizer algorithm.

Linking Weights to Compute: The SwarmX Fabric

For each training iteration, for each network layer, a copy of the layer weight tensor is sent from the MemoryX service to each CS-2 system, once for computing activations in forward propagation, and again for computing activation gradients during backwards propagation. Weight gradients computed by each CS-2 system are also sent back to the MemoryX service where they are used for weight updates. We use the SwarmX fabric to connect the MemoryX service to each CS-2 system, facilitating the broadcast of weights and aggregation of gradients (Figure 7). The MemoryX service sends a single copy of the weights to the SwarmX fabric, which handles the broadcast to each CS-2 system. On the backward pass, the MemoryX service receives a single copy of the gradients from the SwarmX fabric.

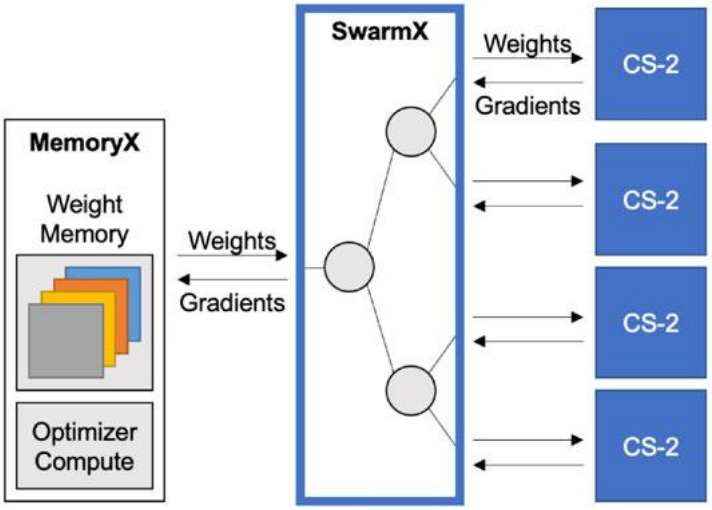

Figure 7. Connectivity between the MemoryX service and a CS-2 cluster using the SwarmX fabric.

Since each CS-2 system is computing a partial gradient for the entire training batch, the gradients from each CS-2 system need to be accumulated prior to the weight update. We chose to perform this reduction inside of the SwarmX fabric, which requires the fabric to perform just one add per element per CS-2. Reducing along the way has the benefit of making the bandwidth requirement symmetrical for weight and gradient communication. One copy of the gradient tensor propagates back through the SwarmX fabric, over each link, to the MemoryX service. Moving the reduction into the SwarmX fabric also has the benefit of providing a clean abstraction. From the perspective of the MemoryX service, the SwarmX fabric looks just like a single CS-2 and from the perspective of the CS-2, the SwarmX looks just like the MemoryX service. This abstraction allows us to use the same compute unit and memory service regardless of cluster size.

The SwarmX fabric is composed of broadcast-reduce nodes each containing a set of 100Gb/s network interfaces. Each broadcast-reduce node provides enough bandwidth to perform either 1:4 broadcast-reduce operations or a pair of 1:2 broadcast-reduce operations. The nodes can be configured in several different modes, which provides flexibility to meet the needs of a particular cluster. Each node provides enough compute to perform the floating-point gradient reductions at line-rate, allowing reductions to occur as gradients flow back to the MemoryX service.

SwarmX nodes are connected in a bidirectional tree topology which minimizes the overall bandwidth and latency required to perform the broadcast and reduction operations (Figure 8 and Figure 9). Each CS-2 system has 1.2Tb/s of I/O bandwidth and, in the worst case, needs weights to be delivered at this rate to keep the compute units busy, so the aggregate bandwidth required from the SwarmX fabric increases linearly with N, the number of CS-2 systems. To satisfy this requirement, the number of nodes composing the SwarmX fabric scales linearly with N. Since tree reductions are work-efficient, the compute required also increases linearly with N, and is delivered by the compute in each broadcast-reduce node. A tree topology also has the benefit of reduced latency, with the latency between the MemoryX service and the CS-2 systems growing logarithmically with N.

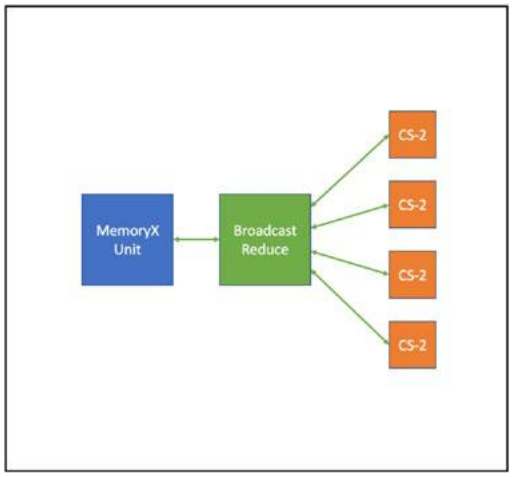

Figure 8. SwarmX fabric connectivity for a cluster of 4 CS-2 systems.

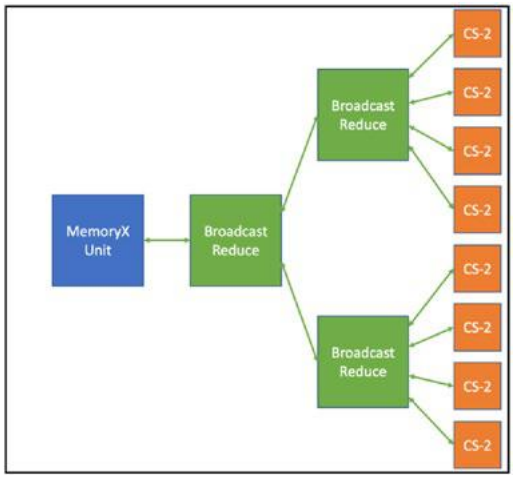

Figure 9. SwarmX fabric connectivity for a cluster of 8 CS-2 systems.

Principles of Operation

The Cerebras Graph Compiler (CGC) integrates with machine learning frameworks, such as TensorFlow and PyTorch, to compile user models into binaries that can execute on a CS-2 system and supporting cluster. The CGC now natively supports the new weight streaming execution mode. All details of the workload mapping and distribution are handled by the CGC, allowing users to easily bring up existing models on a cluster of CS-2s. As explained above, the weight streaming approach streams weights (one layer at a time) from the MemoryX service to a CS-2 cluster, which computes weight gradients and streams them back to the MemoryX service where weights are updated. One batch of activations remains resident in the CS-2 cluster while the forward and backwards propagation passes occur; then this process is repeated with a new batch streaming in from the data storage service.

Execution on the Wafer-Scale Engine

At the start of each training iteration, a unique shard of the training batch is downloaded from the data service to each CS-2 system, and then to the WSE-2. This shard serves as the activation input to the first layer in forward propagation. The WSE-2 is responsible for computing activations, activation gradients, and weight gradients for its local shard. Layers run one at a time in the forward and then in the backward pass. For each layer, the sparse weights arrive from the MemoryX service twice, first to trigger compute operations with the activations in the forward pass, and again for the backward pass when the activation gradient from the following layer becomes the input activation gradient for the previous layer. The activations for the local shard are stored on the wafer throughout the training iteration. Stored activation shards, one per layer, that are generated during the forward pass are consumed in the backward pass to compute weight gradients. Weight gradients computed by the WSE-2 in the backward pass are sent to the MemoryX service through the SwarmX fabric, where they are added to the weight gradients of all the other shards, so that the sum of the per-shard weight gradients is finally presented to the MemoryX service.

Data Layout on the Wafer

Activation and activation gradient tensors are stored in the memory of the WSE-2. Each of the 850,00 cores on the WSE-2 contains 48kB of memory, which is used to store these tensors. The method of distribution of these tensors over the wafer, what we refer to as *data layout*, is an important aspect of execution on the wafer. This is because the arrangement of data determines how much work each core performs, and which communication operations are required. On the WSE-2, we use a rectangular array of interconnected cores to perform the tensor operations found in neural network layers. The matrix multiplication of a fully connected layer is an example. A WSE-2 implementation of a tensor operation specifies its own layout for its weight and activation tensors as well as the local computation and the inter-core communication needed for the implementation. Activation tensors are often distributed across the cores such that an individual activation tensor element is stored on one core. Weight tensors are received from the IO interface along one edge of the rectangle of cores and trigger local computations on each core. The communication often entails broadcast of input tensors across rows or columns of cores and the sum-reduction of partial outputs, also across rows or columns of cores. The software stack allows for each implementation of a tensor operation to specify its desired data layout, but it is important

to use a consistent strategy to reduce data movement as the network executes.

Activation tensors in models like GPT-3 have three dimensions: batch, sequence, and hidden (feature). Our general method of distributing tensor data on the wafer involves splitting one or more of these tensor dimensions over the wafer's x and/or y dimension. For example, if we split the tensor's hidden dimension over the wafer's x dimension, it means that each core in a row of cores has a contiguous chunk of features from the logical tensor, with adjacent cores containing adjacent chunks of features. One goal of this activation layout strategy is to reduce data movement during and between compute operations. Compute operations involving activations or activation gradients are performed on the core containing the corresponding elements of the tensor. This means that elementwise operations can be performed without any data movement. Elements are read from local memory, the compute operation is executed, then the resulting elements are written back to local memory. However, reductions over dimensions of these tensors do require communication between cores containing each chunk of the reduced dimension. Our data layout strategy for NLP models, depicted in Figure 10, optimizes for reductions over the hidden and sequence dimensions since these are the most common. The feature dimension is split over the wafer's x dimension, and the sequence and batch dimensions are split over the wafer's y dimension, with elements from each sequence mapped to adjacent sets of cores. Reductions over the batch dimension are needed for computation of weight gradients, but these operations also require a reduction over the sequence dimension, both of which can be efficiently accomplished with a single sequential reduction over the wafer's y dimension.

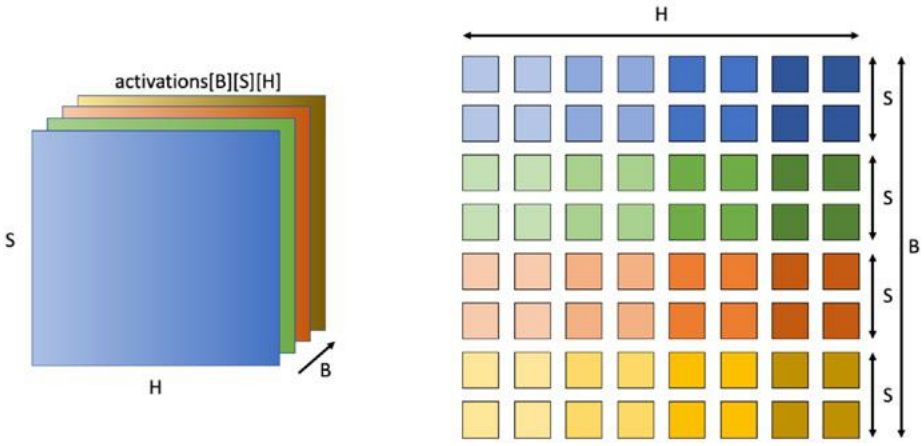

Figure 10. Layout of an activation tensor on the wafer. Chunks of the H dimension are distributed over rows of cores. Chunks of the S and B dimensions are distributed over columns of cores with sequential chunks of the S dimension on adjacent cores.

Activation and activation gradient computations for fully connected layers involve matrix multiplication operations which reduce over the hidden dimension. For an activation computation, where X_0 is the input activation tensor, W is the weight matrix, and X_1 is the output activation tensor:

$$X_1[B \times S][H_1]^T = W[H_1][H_0] \times X_0[B \times S][H_0]^T$$

Where H_0 is the hidden dimension of the input and H_1 is the hidden dimension of the output. This data layout allows for reductions over the hidden dimension to occur between adjacent sets of cores in each row, reducing the bandwidth requirement. Weight gradient computations for fully connected layers involve matrix multiplication operations which reduce over the batch and sequence dimensions:

$$dW[H_1][H_0] = dX_1[B \times S][H_1]^T \times X_0[B \times S][H_0]$$

Our chosen data layout also allows for reductions over these dimensions to occur between adjacent sets of cores in each column.

Our array of cores is roughly square, but these tensor dimensions are not; the feature dimension used by GPT-3 is 12k, and batch shards typically contain hundreds of thousands of tokens. To better map these disparate dimensions to the wafer, we can also split the batch dimension over the fabric x dimension as shown in Figure 11. This allows us to evenly spread the activation and activation gradient tensors to the entire wafer, while giving each core a roughly square subregion of the tensor. Features from the same sequence in the batch are still placed on adjacent cores within the row, but each row of cores contains data from more than one sequence in the batch.

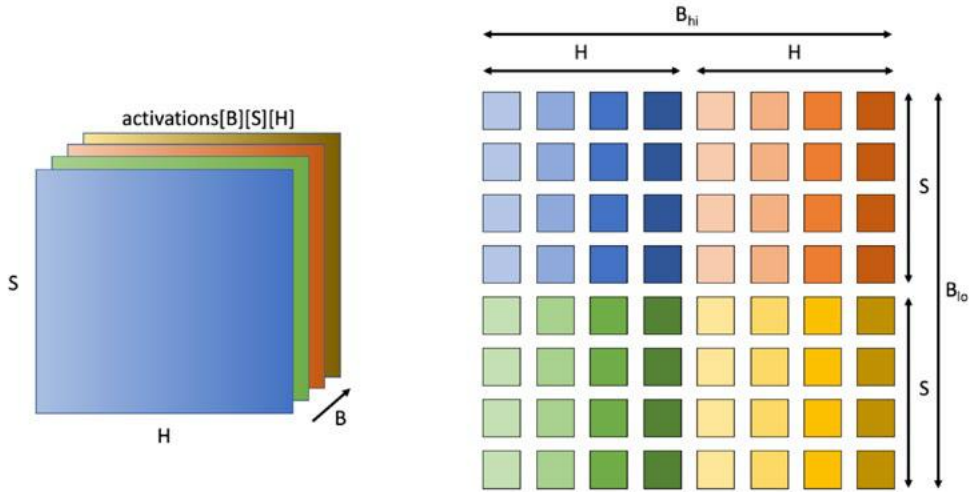

Figure 11. Layout of an activation tensor which splits the B dimension over both spatial dimensions on the wafer. This allows the per-core sub-tensor to be roughly square even when the number of tokens is much greater than the number of features.

Data Layout for Weights and Gradients

Weight and gradient tensors are not stored persistently on the WSE-2 but are streamed on and off the WSE-2 from and to the MemoryX service. Weight data arrives over the on-wafer network connections which link each core to its four adjacent neighbors. Weight data is split over these links such that each core only receives weights needed for computations on its local activations. The number of columns, H_0, in a weight matrix corresponds to the hidden dimension of the input activations. When computing activations for the next layer, the H_0 dimension of the weight matrix is split over the wafer's x dimension to produce the same distribution as is used for the H dimension of the input activations. This is depicted in Figure 12. During the backward pass computations of activation gradients, the number of rows, H_1, in the weight matrix corresponds to the hidden dimension of input activation gradients. As a result, we split the H_1 dimension over the fabric X dimension for this operation. When weight gradients are computed in each column of cores, they are transmitted out of the column with similar distribution to the weights. Gradients are computed one row at a time to match the order of activation computations, so each row of gradients is split along its H_0 dimension over the wafer's x dimension. Computing gradients in the same order as forward pass activations means that the weights of the first layer can be updated in the same order as they are streamed out, allowing for more efficient pipelining.

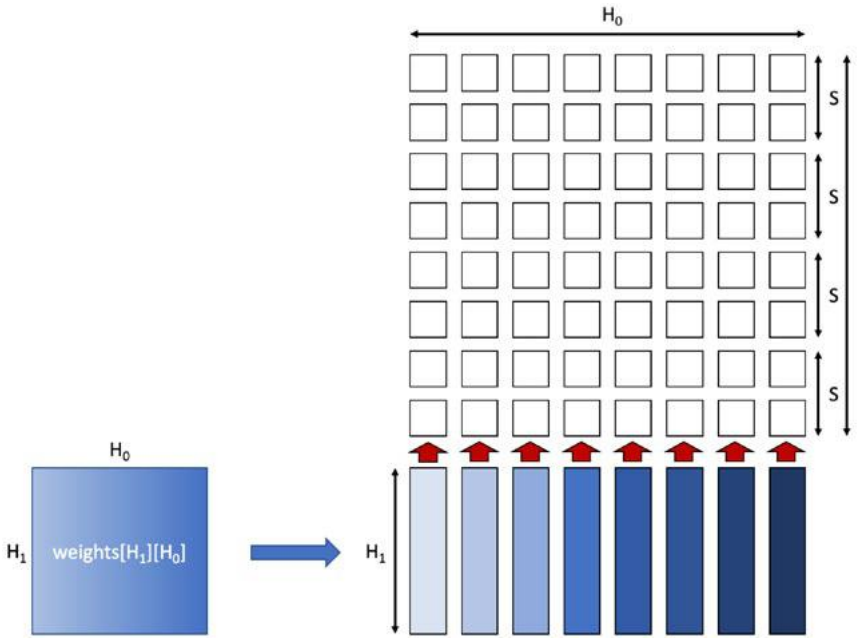

Figure 12. Data layout used for weights arriving during activation computations.

Wafer Scale Matrix Multiplication

Matrix multiplication operations dominate the compute used by NLP models such as GPT-3. They are used for all three phases of fully connected layer execution: activation, activation gradient, and weight gradient. We have achieved high utilization on these operations by avoiding bandwidth bottlenecks, through our data layout strategy, and minimizing overheads. There are two flavors of matrix multiplication that are needed to train sparse models: one which supports a single sparse input and dense output and another which supports two dense inputs and a sparse output.

One Sparse Input

The first flavor of matrix multiplication is targeted at activation and activation gradient computations. A dense activation tensor, stored in local memory, is multiplied with a sparse weight tensor streamed into the WSE-2 from the MemoryX service. The resulting dense activation tensor is stored to local memory.

This matrix multiplication operation is broken down into a series of matrix vector multiplications. Each matrix vector operation multiplies one row of the weight tensor with the entire activation tensor in local memory and results in a single vector of the output tensor. As described in the previous section, a row of sparse weights is streamed onto the wafer with the row split such that each weight arrives on the column of cores containing all corresponding activations. Weights are broadcast to all cores in the column using the on-wafer network whose routers support multicast in hardware. As weights arrive at each core, they are multiplied with the corresponding feature for each token in the core's subset of the batch and accumulated into a temporary buffer. After all weights for the row have been processed, each core contains a partial sum which must be reduced with all cores in the row to compute the result. Partial sums are reduced over a ring using the on-wafer network, with the result landing on the column of cores which should store the feature corresponding to the received row of the weight matrix. These broadcast and reduction communication patterns are shown in Figure 13.

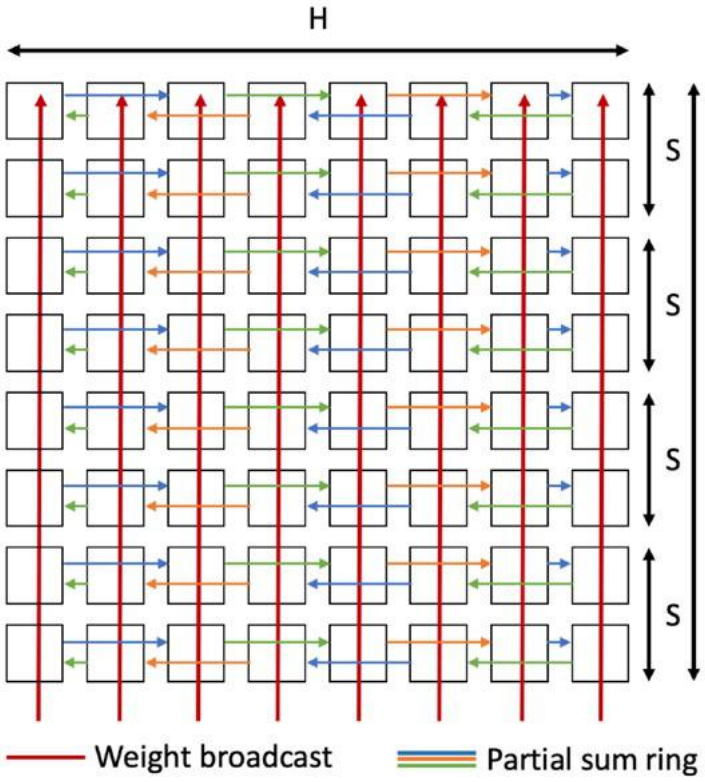

Figure 13. Inter-core communication patterns used for matrix multiply.

Tensor operations on the wafer are driven by the arrival of weight data or commands. The arrival of weight data triggers a floating point multiply-accumulate operation between the received weight data and corresponding row of activations. Commands are sent by the MemoryX service's coordinator, as shown in Figure 14, which drives execution of each tensor operation, and are used to trigger other operations such as the partial sum reduction. When the cores receive a partial sum command, they initiate communication with their upstream neighbor, reducing incoming partial sums with the values in their local accumulators and transmitting the results to their downstream neighbor. The fully reduced values are received and stored on one column of cores indicated by a special argument to the partial sum command sent to that column. Commands can be used to trigger other computations on the wafer such as nonlinear functions or normalization operations. This system allows us to support all tensor operations required by NLP models.

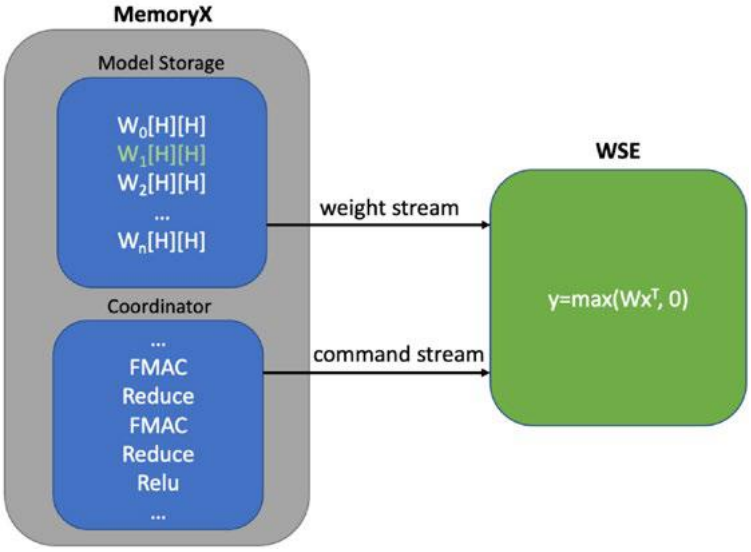

Figure 14. Inputs from the MemoryX service to the WSE which drive execution of each tensor operation.

Sparse Output

The second flavor of matrix multiplication is intended for computing weight gradients. A dense activation tensor and dense activation gradient tensor, both stored in wafer memory, are multiplied to compute a sparse weight gradient tensor which is sent off-wafer.

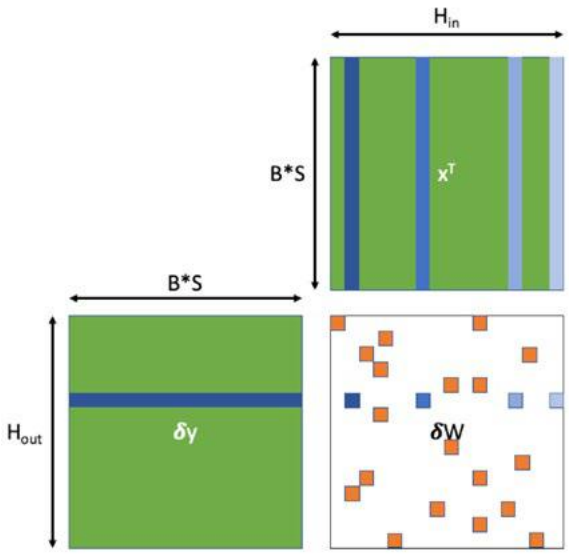

Figure 15. Matrix multiplication with a sparse output.

Since the Cerebras weight streaming implementation supports unstructured weight and gradient sparsity, this matrix multiplication operation is computed one output element at a time on each core. Computation of each weight gradient involves a dot product between the corresponding activation and activation gradient vectors, as shown in Figure 15. The MemoryX service sends a sparsity mask to the WSE-2 as input, which instructs the wafer to compute a gradient for each masked position. Gradients are computed for each row of the weight matrix in order, so this mask is received one row at a time. Each row of the weight matrix corresponds to a single output feature. The row of the activation gradient tensor corresponding to this feature is broadcast to all cores in each row of cores using the blue and green routes shown in Figure 16. An element of the output gradient tensor is then computed by multiplying this broadcasted vector with the feature from the activation tensor corresponding to the weight gradient's column index. The sparsity mask input is split over the fabric the same way that the weight matrix is split, such that masked elements are received on the core containing the corresponding activation feature data. This means that the broadcasted gradient vector can be multiplied with activation data from local memory to compute a partial sum of the gradient element. A chain routing pattern, also shown in Figure 16, over the on-wafer network is used to accomplish the reduction of these partial sums between cores in the same column, containing the same feature for different tokens in the batch.

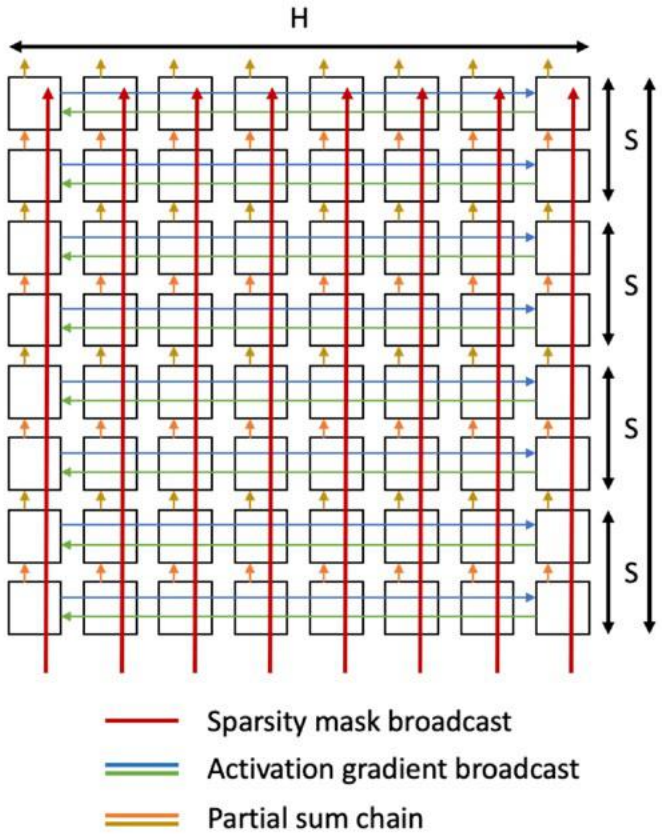

Figure 16. Inter-core communication patterns used for gradient variant of matrix multiply.

Execution on the MemoryX Service

Model parameters and optimizer state are stored persistently on the MemoryX service during training in weight streaming mode. The MemoryX service performs three primary functions corresponding to the three phases of training: weight streaming, gradient receipt, and weight update. These operations and the weight and gradient streaming interfaces are shown in Figure 17. During the forward and backward passes, the MemoryX service streams weights out to the WSE-2. In the backward pass, the MemoryX service sends a sparsity mask for each layer as a request for the wafer to compute gradients. It then receives gradients back from the WSE-2 and updates the weights before the next training iteration begins.

The MemoryX service stores weights in both dense and sparse formats. A dense copy of the weight and optimizer state is needed for most sparsity algorithms, since they need the ability to regrow the weights. The dense format includes FP32 values for all weights and per-weight optimizer states such as momentums. A compressed sparse row (CSR) representation is used for the sparse copy of the weights, using FP16 for the weight values and a 16-bit delta-encoded integer for the index. The dense FP32 format is used for performing weight updates, but only the active sparse weights are updated. Updated weights are converted into the sparse format by applying a sparsity mask and rounding values from FP32 to FP16. During the forward and backward passes, it is the sparse format of the weights which is sent to the WSE-2.

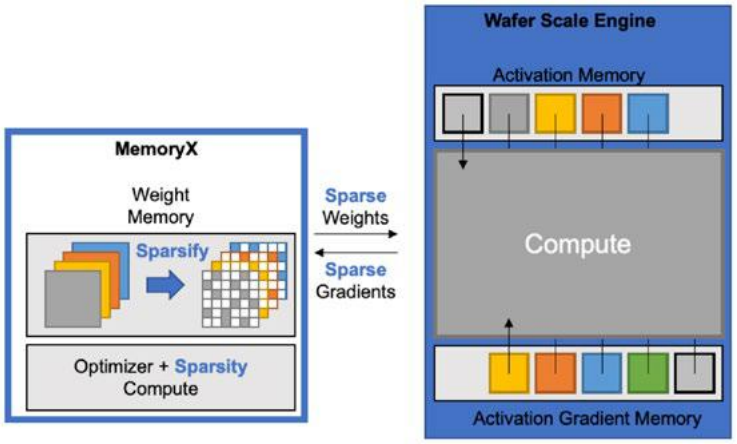

Figure 17. Data stored on the MemoryX service and the WSE and communication between the two.

Weights can be streamed out of the MemoryX service in either forward or backward order. As described in the previous section, the WSE-2 computes GEMM operations, corresponding to activation or activation gradient computations, as a series of GEMV operations. One feature of the output tensor is computed at a time. In the forward pass, the output features correspond to rows of the weight matrix, so the MemoryX service streams sparse weights in row-major order so that the wafer receives full rows of the matrix one at time. This order is flipped in the backward pass when the WSE-2 is computing one row of the activation gradients, corresponding to the previous layer's features, at a time. Supporting both orders requires the MemoryX service to transpose the sparsity pattern of the weight matrix each time the sparsity pattern is updated. The forward and transpose orders both provide views to the same underlying data, so that only one copy of weights needs to be updated.

During the backward pass, weight gradients are requested and received from the WSE-2 by the MemoryX service. Computation of weight gradients on the WSE-2 is triggered by arrival of a sparsity mask indicating which gradient values should be computed. The sparsity mask is sent by the MemoryX service and has the same format as the index component of the forward-order weights. However, the MemoryX service can choose to use a different sparsity mask than that of the weight matrix to trigger computation of gradients corresponding to zero-valued weights. This can be useful when trying to change the sparsity of the weight matrix. For example, the MemoryX service can change a zero-valued weight to a non-zero when its gradient value exceeds some threshold. FP32 gradient values are sent back from the WSE-2 to the MemoryX service in the order that the sparsity mask was sent.

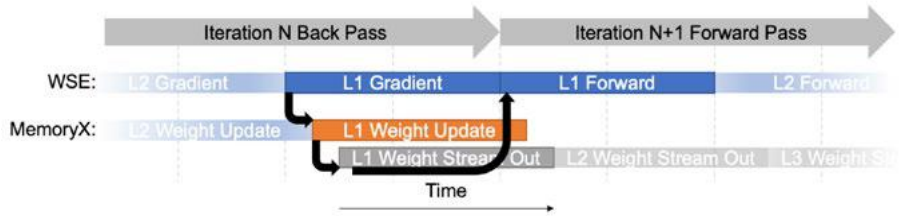

Figure 18. Pipelining of weight and gradient communication between the MemoryX service and the WSE-2.

The MemoryX service performs weight updates after gradients have been received. Scheduling in the MemoryX service is designed to avoid a latency bottleneck in the training loop. While storing model parameters farther from the compute nodes results in a higher communication latency, this is hidden by pipelining which is depicted in Figure 18. In the forward pass, the MemoryX service begins streaming weights for a layer before the WSE-2 has completed activation computations for the previous layer. The same approach is used in the backward pass, but in reverse with transposed weights. While the MemoryX service is streaming weights in transpose order for the backward pass, it is also collecting gradients being streamed back from the WSE-2. Weight updates can be applied as gradients are received, such that the new weights are ready for the next training iteration as soon as gradients are received for the first layer. The critical path is for the weights of the first layer, as these are the last to receive gradients and the first needed in the forward pass of the next training iteration. To mitigate this, the WSE-2 computes gradients for weights in the same order as weights are transmitted for forward pass operations. This means that updated weights for the next training iteration can be streamed out before the WSE-2 has completed gradient computations for the first layer. The round-trip latency between the MemoryX service and the WSE-2 can be completely hidden when the compute time of the first layer is greater than the round-trip latency.

Summary

We have presented weight streaming as a new paradigm for training giant models. Weight streaming disaggregates the storage of parameters from the compute units. We described an implementation, i.e. the Cerebras weight streaming architecture, that is based on wafer-scale compute units; a new system, the MemoryX service, for parameter storage and update; and a novel interconnection, SwarmX fabric, between parameter memory and compute. We believe that because of the storage volume provided by our MemoryX service, our weight streaming solution provides the only way currently known to run models with hundreds of trillions of parameters. The architecture is designed around the Cerebras WSE-2 system which contains a wafer-scale processor that provides enough compute power and on-wafer SRAM to support layer sizes an order of magnitude greater than those used in today's state-of-the-art models. Since each WSE-2 can support massive layers, our architecture is able to use a scale-out model based on pure data parallelism, which can support a cluster delivering more floating-point performance than the current largest supercomputer in the world for this class of workloads. The WSE is also a preferred platform because, unlike units that prefer dense matrix multiplication, it can fully exploit sparsity in the weight tensors, for a one to two orders of magnitude reduction in computational work and runtime. We achieve runtime reduction nearly linear in number of nonzero weights for unstructured weight sparsity.

We contend that the combination of effective sparsity, compute units capable of storing full layers, and memory disaggregation gives researchers the only practical way to train models with trillions of parameters.

References

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[2] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity, 2021.

[3] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM, 2021.

[4] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale Model Training, 2021 .

[5] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners, 2019.

[6] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism, 2019

[7] Corby Rosset. Turing-NLG: A 17-billion-parameter language model by Microsoft. https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/, 2020

[8] Paresh Kharya and Ali Alvi. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, the World's Largest and Most Powerful Generative Language Model. https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/ , 2021

[9] Jonathan Frankle and Michael Carbin. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, 2018

CEREBRAS SYSTEMS, INC.
1237 E. ARQUES AVE, SUNNYVALE, CA 94085 USA
CEREBRAS.NET