An Efficient 2D Method for Training Super-Large Deep Learning Models

Qifan Xu
QifanXu@mednet.ucla.edu
University of California, Los Angeles
Los Angeles, California, USA

Yang You
youy@comp.nus.edu.sg
National University of Singapore
Singapore, Singapore

ABSTRACT
Since the rise of Transformer [22] and BERT [6], large language models [7, 12] have been proposed and shown unprecedented performance in tasks like translation, classification, and text generation. However, due to the memory constraint, model parallelism must be used to split the model across multiple processors. Inter-layer partition, intra-layer partition, and sparse activation are the major approaches to achieve model parallelism. Among them, inter-layer partition [10, 11] often requires the model to be explicitly expressed as a stack of sub-modules, the number of which equals to the number of processors, and would introduce either gradient overhead. In this work, we leverage SUMMA [21] and propose Optimus, a highly efficient and scalable paradigm for training super-large language models. In Optimus, activations and gradients are partitioned and distributed along processors all the way forward and backward propagations, with hardly any memory redundancy. The isoefficiency of communication in pure model parallelism improves from \( W \sim p^3 \) for Megatron-LM, to \( W \sim (\sqrt{p} \log p)^3 \) for our Optimus. This framework is implemented with open-source deep learning framework, PyTorch, and consolidates existing techniques such as mixed precision training [13], activation checkpointing [5], and data parallelism. In experiments on TACC Frontera supercomputers, Optimus shows 1.48x the speed for training, 1.78x speed for inference, and 8x the maximum batch size over Megatron-LM on 64 GPUs in pure model parallelism; and 1.73x speed for training, 2.32x speed for inference with data parallelism size equaling 2 on 128 GPUs. In pure model parallelism, Optimus surpasses Megatron-LM in weak scaling efficiency by a great margin, and shows an extraordinary increasing strong scaling efficiency. Optimus would facilitate the scaling of language models and serve as a strong thrust in the space exploration of artificial intelligence.

CCS CONCEPTS
- Computing methodologies → Massively parallel algorithms; Information extraction.

KEYWORDS
matrix-matrix multiplication, distributed training, neural networks, natural language processing

1 INTRODUCTION
State-of-the-art deep learning models such as Transformer [22] and BERT [6] surpasses their predecessors like RNN and LSTM, and have shown their extraordinary power in the field of natural language processing and computer vision. By using these well-trained models, realistic images are generated given a short piece of description, and essays can be composed as if they were written by humans. The reason for their impressive performance is the increase in model sizes and computational capability. The number of parameters in a state-of-the-art NLP model has increased from 110 million (OpenAI GPT) to 340 million (BERT [6]), 1.5 billion (GPT-2), 8.3 billion (Megatron-LM [18]), 17 billion (Turing-NLP [16]), 175 billion (GPT-3 [3]), 600 billion (Google MoE [12]), and 1.6 trillion (Google Switch Transformer [7]). Based on this observation, we believe the trend of training large language models will be going on for many years ahead.

Larger models have imposed challenges on both hardware and software. In addition to building supercomputers, novel algorithms are being designed by researchers to tackle the problems. Activation checkpointing [5] is proposed as a method to trade computation for memory. In checkpointing method, only a part of middle activations is buffered, while the rest is not buffered but recomputed during backward propagation, which can significantly reduce the memory overhead. Mixed precision training [13] with dynamic loss scaling replaces 32-bit float tensors with 16-bit half tensors during training, which can reduce both the memory overhead and the computational cost while preserving the convergence performance. Besides trying to fit a model within a single processor, attempts have been made to accelerate the training on multiple processors. Data parallelism is proposed to split a mini-batch across multiple processors.

In addition to these methods, another important paradigm is model parallelism. It is employed to distribute the model parameters among multiple processors in case a model does not fit in the memory of a single processor. Pipeline parallelism [10, 11], as
an inter-layer parallelism method, partitions the whole model by layer in a serial manner, so that the input batch is processed on one processor at a time, and then sent to the next processor. Though straightforward, due to the pipeline nature, the gradients in [10] are obtained iterates after the corresponding forward propagation, degrading the training convergence. In [11], to avoid the gradient staleness, bubble overhead is introduced as a compromise in processor utilization.

Mesh-TensorFlow [17], as a pioneering work in intra-layer parallelism, is built upon TensorFlow. It arranges the physical processors into an n-dimensional mesh. Each tensor is assigned to the processors with an injective partial map from the tensor dimensions to the mesh dimensions. When a tensor dimension is assigned to a mesh dimension, the tensor is split over this tensor dimension, and distributed across the corresponding mesh dimension. It provides users with an elegant way for tensor partitioning and computation parallelization. However, if a mesh dimension is not assigned to any tensor dimension, the tensor is replicated along the mesh dimension, causing memory redundancy. Besides, it is primarily designed for TPU, and cannot be easily deployed on GPU servers.

Megatron-LM [18] adopts a similar idea as Mesh-TensorFlow. However, instead of providing a framework for the parallelization of a general class of user-specified models, it is built for a narrow class of language models, with Transformer as the backbone. Besides, it is built on PyTorch and for GPU servers, making it accessible to general researchers. Essentially, the basic idea of both Mesh-TensorFlow and Megatron-LM is one-dimensional matrix partition, which relies on all reduce to obtain the final result when the contraction tensor dimension of einsum is partitioned. This approach, though straightforward, suffers from memory redundancy, in other words, all processors along the reduction dimension store duplicate tensors, or tensors of the same size as the result, forming a memory bottleneck. Besides, one-dimensional partitioning is also sub-optimal in terms of communication efficiency.

ZeRO [15] is essentially an upgraded version of data parallelism, which decreases the memory consumption of model states (parameters, gradients, optimizer states) by d times, where d is data parallel size, at the cost of a 1.5x the communication volume of the vanilla data parallelism. Based on the observation that model states are replicated along processors in a data parallel group, which are not required all the time during training, the authors propose to have each processor hold and update its own partition of parameters. Before the forward propagation of each layer, ZeRO uses all gather to combine the partitions into the entire parameter matrix; after the backward propagation, reduce scatter is used to shard the gradient partitions to corresponding processors for parameter updates. When used alone, though it can reduce the memory demand for model states, it faces the same problem as Megatron-LM, i.e., it requires each processor to store the whole activations, hindering the scaling of training. When CPU offload is enabled to alleviate the memory limit, the data swap between CPU and GPU introduces great overhead in communication. In fact, ZeRO is orthogonal to, and usually used together with model parallelism approaches.

However, with the help of matrix-matrix multiplication algorithms utilizing 2-dimensional partition strategy like SUMMA [21] or Cannon’s algorithm [4], it is possible to both alleviate the memory bottleneck and speedup training and inference. In this work, we propose Optimus, a novel paradigm for distributed model training. Optimus fully distributes parameters, activations, and gradients, achieving both excellent communication efficiency as well as optimal memory performance. Optimus is implemented with PyTorch and can be used together with techniques mentioned above in large scale neural network training. Optimus is able to achieve an iso-efficiency function of $W \sim (\sqrt{pq})^3$, which is much better than that of Megatron-LM ($W \sim p^3$). We believe the mesh topology of newly emerging supercomputers is to further release the power of Optimus. The advantage of Optimus over previous intra-layer parallelism would facilitate the development of high-performance deep learning models as model size and number of processors continue to increase.

In general, our contributions are as follows:

- We design and implement an efficient training paradigm for super-large models based on the idea of 2D partition strategy, addressing both memory bottleneck and communication efficiency.
- We add upon existing activation checkpointing method pre-allocated buffers for communication data, intermediate activations, activation gradients, and parameter gradients, to optimize memory management and avoid fragmentation.
- We achieve an improvement for isoefficiency from $W \sim p^3$, $p \leq 18$ for Megatron-LM to $W \sim (\sqrt{pq})^3$. Experimental result gives improvements of $1.48 \times$ speed for training, $1.78 \times$ speed for inference, and $8 \times$ increase in maximum batch size over baseline Megatron-LM model with pure model parallelism on 64 GPUs. An extraordinary increase in strong scaling efficiency is observed. When data parallelism is enabled, we achieved $1.73 \times$ for training and $2.32 \times$ for inference over the baseline.
- We train a BERT-small equivalent model and achieve comparable convergence performance with baseline in both pre-training and accuracy in downstream tasks.
- Our work is open-sourced 1 and integrated in Colossal-AI 2.

2 BACKGROUND

2.1 Transformer

From here, we use the following conventions: $b$, batch size; $s$, sequence length; $h$, hidden size; $n$, number of attention heads; $v$, vocabulary size; $p$, number of processors per model parallel group; $q$, SUMMA dimension; $d$, data parallel size; $N$, number of Transformer layers. Though SUMMA allows the communication of a tensor partition to be divided in multiple transactions and does not require the processors to be arranged in a square mesh, we broadcast the whole tensor partition in one transaction, and set the mesh to be square, or $p = q^2$ to reduce the number of communication transactions. In the text below, we use a pair of square brackets to express the shape of a tensor.

The overall structure of Transformer is shown in Figure 1. “Tokens” is an integer tensor of shape $[b, s]$, comprising $b$ sequences, each of $s$ tokens. Each token is an integer, standing for a word piece. “Embedding” layer embeds each token into a vector with a look-up table of shape $[v, h]$. The “Tokens” are then embedded into

---

1 https://github.com/xuqifan897/Optimus
2 https://colossalai.org/
hidden states, a float tensor of shape \([b, s, h]\), which then undergoes a sequence of Transformer layers of identical structure but with different parameters. Each layer preserves the shape of its input. For pre-training, the “output” could branch into two losses, a token-wise one and a sequence-wise one.

Each Transformer layer consists of a self-attention layer and a multi-layer perceptron. The self-attention layer firstly projects the input tensor of hidden size \(h\) into query (\(Q\)), keys (\(K\)), and values (\(V\)), all of shape \([b, s, h]\). They are then individually divided over the hidden dimension \((h)\) into \(n\) submatrices. Corresponding submatrices from \(Q, K, \) and \(V\) form an attention head. Per each attention head, \(Q, K, \) and \(V\) are all of shape \([b, s, h/n]\). Attention scores (\(A\)) are calculated as the softmax of the matrix-matrix multiplication of \(Q\) and \(K^T\) of the same attention head, giving a tensor of shape \([b, s, s]\), as the cross-talk between different tokens. \(A\) is then multiplied with \(V\) to restore the shape \([b, s, h/n]\) for a single attention head. Results from all \(n\) attention heads are concatenated. For Megatron-LM, necessitated by the one-dimensional matrix partition strategy, the concatenated tensor is then multiplied with an additional parameter matrix of shape \([h, h]\) to get the result of self-attention layer.

Multi-layer perceptron (MLP) projects the hidden dimension of input tensor into a higher dimension (4 times the hidden size in Megatron-LM), applies a non-linear activation, and then projects it back to original dimension.

### 2.2 Megatron-LM

Megatron-LM is a distributed training framework for language models, which utilizes both data and model parallelism. More specifically, the model parallelism adopts intra-layer partition. In brief, it first arranges all GPUs into a 2-dimensional mesh. Each row of GPUs along one dimension forms a model parallel group, while each row of GPUs along the other dimension forms a data-parallel group. GPUs within a model parallel group divide the parameter matrix of each layer, while GPUs within a data parallel group store duplicated parameters. For each training iteration, GPUs within a model parallel group collectively compute the activations, loss function, and finally gradients for a single batch. Different model parallel groups process distinct batches. After that, corresponding parameters within the same data parallel group are updated synchronously. In other words, data parallelism is equivalent to training with a larger batch size equaling to batch size per model parallel group times data parallel size.

The model parallelism employed by Megatron-LM is illustrated in Figure 2. There are two parameter matrices in MLP; the first one is partitioned along its column, and the other one along its row. During forward propagation, a copy of input is stored by each processor, which is then multiplied with one partition of the first matrix on the same processor, yielding intermediate activations partitioned along its column. Partitioned intermediate activations on each processor then undergoes a non-linear activation and is multiplied with a partition of the second matrix on the same processor, then all reduced across the model parallel group to get the output of MLP.

There are also two parameter matrices in self-attention, partitioned along column and row, respectively. Inputs are again duplicated among processors. Here, each processor is in charge of \(n/p\) attention heads. The input in each processor is multiplied with the first matrix partition, then divided into \(n/p\) \(Q's, K's, \) and \(V's.\) Each attention head computes \(\text{softmax}(QK^T)V.\) Results from \(n/p\) attention heads within a processor are concatenated and multiplied with the second matrix partition, then again all reduced across the model parallel group to get the output of self-attention.

Communication happens when the all reduce operation is performed to get the output in the forward pass and to calculate the gradients in the backward pass (not shown in Figure 2). Detailed analysis of communication is discussed later.

### 2.3 ZeRO

The authors of ZeRO [15] classify the memory consumption into two groups, 1) model states, including optimizer states, gradients, and parameters; 2) residual states, comprising activations, temporary buffers, and fragmented memory. They discovered that during model training, most of the memory is consumed by the model states. Besides, they observed that 1) The existing model parallelism partitions the model states across processors, while having activations replicated. 2) Data parallelism has better scaling efficiency over model parallelism due to the low communication granularity of the former. 3) Data parallelism is memory inefficient because model states are replicated for every processor. 4) Not all model states are required all the time during training. Based on these observations, they tackle the memory redundancies in model states and residual states with ZeRO-DP and ZeRO-R, respectively.

There are three phases in ZeRO-DP: \(P_a\), optimizer states partitioning, \(P_g\), gradients partitioning, and \(P_p\), parameters partitioning. Aggregating all the three phases, i.e., \(P_{a+g+p}\), they reduced the memory consumption of model states to \(\frac{2}{3}\) the original. In this sense, model states, when not required for computation, are fully partitioned across all data parallel processors. Communication happens in three circumstances: 1) Before the computation of the forward propagation of a certain layer, all gather is used to broadcast the partitions of corresponding parameter matrix so that every processor has an entire parameter matrix. After the computation, the weight can be discarded. 2) The same process happens during the recomputation in the backward propagation. 3) After the backward propagation, the gradients are reduce scattered so that each processor holds a partition of gradients corresponding to the parameter partition it updates. Both all gather and reduce scatter incur a communication volume of the size of the parameters; while an all reduce used in the vanilla data parallelism incurs a communication volume of twice the size of the parameters. In other words, ZeRO-DP trades \(1.5\times\) communication overhead for \(dx\) model states memory reduction. For ZeRO-R, the authors have each model-parallel group to store a partition of activations, which are again all gathered before the recomputation in the backward propagation. The authors declare that ZeRO-R only introduces a communication overhead of less than 10% of the baseline model parallelism in Megatron-LM.

### 2.4 SUMMA

SUMMA stands for Scalable Universal Matrix Multiplication Algorithm. It considers four forms of matrix-matrix multiplications. In here, it suffices to only introduce three of them: \(C = AB\), \(C = AB^T\), and \(C = A^T B\). We notice that these three products form a closed set.
in terms of differentiation:

\[ C = AB \rightarrow A_{\text{grad}} = C_{\text{grad}}B^T, B_{\text{grad}} = A^T C_{\text{grad}}. \]  

\[ C = AB \rightarrow A_{\text{grad}} = C_{\text{grad}}B^T, B_{\text{grad}} = A^T C_{\text{grad}}. \]  

\[ C = AB^T \rightarrow A_{\text{grad}} = C_{\text{grad}}B, B_{\text{grad}} = C^T_{\text{grad}}A, \]

where the superscript \( T \) means transpose, and the subscript \( \text{grad} \) means gradients. We assume the processors physically form a mesh of dimension \( q \times q \). As an example, we give the pseudo-code for \( C = AB \) in algorithm 1, which is illustrated in Figure 3. SUMMA has the following merits over one-dimensional matrix partition method:

1) it fully distributes the operand and result matrices among all processors, instead of having them replicated. 2) It has superior communication efficiency.

**Algorithm 1** \( C = AB \)

1. **Input:** \( A_{ij}, B_{ij} \)
2. **Output:** \( C_{ij} \)
3. \( C_{ij} = 0 \)
4. for \( l = 0 \rightarrow q - 1 \) do
5. \( \text{broadcast } A_{ij} \) within any row
6. \( \text{broadcast } B_{lj} \) within any column
7. \( C_{ij} = C_{ij} + A_{il}B_{lj} \)
8. end for
9. return \( C_{ij} \)

Figure 1: The structure of Transformer.

Figure 2: Megatron-LM 2(a) MLP and 2(b) self-attention. Green squares are 3-dimensional activations with dimension \( b \) omitted and blue ones are 2-dimensional parameters. The numbers 1 to \( p \) indicate model parallel ranks.

2.5 Collective communications

Reduce and broadcast are used within a row or column in the SUMMA algorithm. The communication costs are determined by the volume of data, bandwidth, and latency of the network. We use \( \alpha \) to denote the latency, or delay due to data travel time, and \( \beta \) to denote the inverse bandwidth, or time to transfer a 32-bit float. Though real-world cases are more complicated, often involving the specific network topology, in here, for simplicity, we assume homogeneity among communications between different processors. Though more complicated and efficient communication procedures exist, here we give the sub-optimal estimates of time spent in broadcast and reduce in a group of \( q \) processors:

\[ T_{\text{broadcast}} = T_{\text{reduce}} \leq \log(q)(\beta B + \alpha), \]

where \( T_{\text{broadcast}} \) and \( T_{\text{reduce}} \) are times spent in broadcast and reduce, respectively, and \( B \) is the volume of data in a single communication. This formula comes from the naive case where the 1st processor transmits data to the 2nd processor in the first round, and the 1st and 2nd processors transmit data to the 3rd and 4th processors in the second round, respectively, and so on. Latency \( \alpha \) is negligible compared to the transmission time for large volume of data, which is omitted in the following sections.

Ring \( \text{all reduce} \) is employed in Megatron-LM, in which all processors in a model parallel group get the same reduced activations or gradients. The time spent in \( \text{all reduce} \) within a group of \( p \) processors is given below:

\[ T_{\text{allreduce}} = \frac{2(p - 1)(\beta B + \alpha)}{p} \]
3 OPTIMUS

3.1 SUMMA-style operations

To ensure fairness and support an apple-to-apple comparison, the network architecture of Optimus is identical to that of Megatron-LM, and the only difference is tensor partition and computation parallelization. Like Megatron-LM, Optimus supports both data and model parallelism. The model parallelism of a single Transformer layer is illustrated in Figure 4. To fit the Transformer layer into SUMMA pattern, activations, gradients, and parameter tensors must be uniformly split into \( q \times q \) sub-blocks. In this sense, in contrast to Megatron-LM, activations and gradients of Optimus are distributed together with parameters, largely alleviating the memory bottleneck.

Another important module in language models is the embedding layer. In Optimus, the input token matrix is firstly converted into a one-hot sparse matrix. The embedding process is then a SUMMA matrix-matrix multiplication between the sparse matrix and the embedding table. Apart from that, the embedding table is also used in producing the token-wise logits, by multiplying the final output of Transformer with the transpose of the embedding table, which is still in the SUMMA paradigm.

3.2 Non-SUMMA-style operations

Besides SUMMA-style matrix-matrix multiplications, Transformer also involves other operations like bias-add and the affine part in layer normalization. In these operations, parameters are stored by processors in row 0 in each model parallel group. In forward propagation, the parameters are broadcast along columns. While in backward propagation, the gradients of the parameters are reduced to the corresponding processors in row 0. This is to ensure that the same parameters are stored and updated in a single processor. Compared to SUMMA-style operations, the communication overhead for non-SUMMA-style ones is negligible.

Layer normalization is another important operation. Besides the affine transform, the core layer normalization is determined as follows:

\[
\hat{X} = \frac{X - E[X]}{\sqrt{\text{Var}[X]} + \epsilon},
\]

where \( E[X] = \frac{1}{n} \sum_i X_i \), \( \text{Var}[X] = E[X^2] - (E[X])^2 \), and \( \epsilon \) is a small constant to avoid zero-division. The gradient of input \( X \) is:

\[
\frac{1}{\sqrt{\text{Var}[X]} + \epsilon} \left[ \frac{1}{h} \left( \sum_j \hat{X}_j \frac{\partial J}{\partial \hat{X}_j} \right) \right] + \frac{1}{h} \left( \sum_j \left( \frac{\partial J}{\partial \hat{X}_j} \right) \right).
\]

In forward propagation, \( X \) and \( X^2 \) are first summed locally then all reduced along rows. \( \hat{X} \) is calculated with \( X \) and \( X^2 \) and saved together with \( 1/\sqrt{\text{Var}[X]} + \epsilon \) for backward propagation before being returned as output; while in backward, \( \hat{X} \frac{\partial J}{\partial \hat{X}} \) and \( \frac{\partial J}{\partial \hat{X}} \) are also firstly summed locally and reduced along the row. The communication here only involves several bytes.

Cross entropy is utilized to calculate the token-wise loss function, which can be implemented analogously. The original form of cross entropy is \( H(\xi, \eta) = -\sum_i \frac{\xi_i}{\eta} \log \eta_i \), where \( \xi \) and \( \eta \) are ground-truth and predicted probabilistic distributions, respectively. For one-hot ground-truth, we assume \( \xi_i = 1 \), \( \xi_i \neq 1 = 0 \), and the loss reduces to \( H(\xi, \eta) = -\log \eta_i \), where \( \eta_i \) is the softmax of input logits \( x_i \). Therefore, we can obtain:

\[
H = \log \left( \sum_{i=1}^n e^{x_i} \right) - x_i.
\]

As \( x \) spans a SUMMA row of \( q \) processors, we firstly calculate the local sum and then perform an all reduce operation along a SUMMA row. We use \( \sum_{i=1}^n e^{x_i} \) again to calculate the softmax:

\[
\eta_j = e^{x_j} / \sum_{i=1}^n e^{x_i}
\]

and save it for backward. The gradient of input can be easily obtained as

\[
x_i^{\text{grad}} = \eta_j \text{, } x_i^{\text{grad}} = \eta_i - 1.
\]

Although the all reduce seems to be more efficient than broadcast and reduce in the sub-optimal cases, in the following discussion, we show the theoretical advantage of Optimus over Megatron-LM even with the unfavorable setting.

As Mesh-TensorFlow is primarily designed for TPU and could not be easily deployed on GPU servers, and it essentially adopts one-dimensional matrix partition, the same strategy as Megatron-LM, we use Megatron-LM as the baseline of intra-layer parallelism in the following theoretical analysis and experimental comparison.
3.3 Memory pre-allocation and checkpointing

Though the memory demand by intermediate activations can be alleviated by activation checkpointing, memory allocation in every checkpointed forward propagation would potentially cause fragmentation. Inspired by the memory management techniques in ZeRO and the fact that Transformer layers are identical in structure, in Optimus, we choose to employ pre-allocation, including buffers for data duplication, intermediate activations, activation gradients, and parameter gradients, rather than relying on the PyTorch-native memory recycling scheme. Among them, buffers for intermediate activations and activation gradients are reused by each layer, which means that they should be reset before they are used by the next/previous layer. For this, we have to use a conjunction buffer to duplicate the results of input gradients before the gradient buffer can be safely reset. In SUMMA, matrices should be firstly duplicated before they can be communicated. Buffers for data duplication are then used to store the cloned matrices for communication.

3.4 Analysis and comparison

The communication and computation costs with activation checkpointing are shown in Table 1. Costs in computation are in number of float-float multiplications per processor, while costs in communication are in numbers of floats concurrently transferred. Latency is negligible. It is noted that, for Megatron-LM, the model parallelism part of the backward communication \( \left( \frac{2(p-1)}{p}bsh \right) \) is twice the forward; while that for Optimus (the first term in each entry in Optimus communication) is three times the forward. This is because communications in Megatron-LM and Optimus happen in different patterns. The \( all \ reduce \) operations in Megatron-LM happen at the end of self-attention or MLP, while the broadcast and reduce operations in Optimus happen together with the matrix-matrix multiplications, and the computation in the native backward propagation (without recomputation) is twice the forward. Though the communication cost of the model parallelism part of Optimus is based on the sub-optimal estimations above, when the problem size is fixed, it still shows superior scaling properties, i.e., from \( O(1) \) to \( O(\log(p)/\sqrt{p}) \).

To characterize the scalability of a parallel system, isoefficiency function [8] is proposed, which indicates the growth rate of problem size with respect to the number of processors to keep the efficiency fixed. In accordance with the weak scaling experiments below, we assume \( b \) and \( n \) scale proportionally to \( h \), while \( s \) and \( N \) are kept constant. Given the communication and computation costs in Table 1 and the efficiency equation:

\[
E = \frac{W}{pT_p} = \frac{1}{1 + \frac{pT_{comm}}{W}},
\]

where \( W \), \( T_p \), and \( T_{comm} \) are serial execution time, parallel execution time, and communication time, respectively, and we have \( T_p = W/p + T_{comm} \). By fixing the efficiency, or \( pT_{comm}/W \), we get the isoefficiency functions for model parallelism of Megatron-LM and Optimus, \( W \sim h^3 \sim p^3 \) and \( W \sim h^3 \sim (\sqrt{p} \log p)^3 \), respectively.

4 EXPERIMENTS

4.1 Scaling experiments

We firstly demonstrate through experiments the advantage of Optimus in communication efficiency over Megatron-LM, as both Optimus and Megatron-LM do not entail duplicate computation. Specifically, we examine the weak scaling efficiency [9] and strong scaling efficiency [2]. Strong scaling concerns the speedup of a problem of a fixed size with respect to the number of processors, which is governed by the Amdahl’s law: speedup = \( 1/(a + b/p) \), where \( a \) is the portion of execution time spent on the serial part, \( b \) is the portion of execution time spent on the part that can be parallelized, and \( p \) is the number of processors. By definition, we have \( a + b = 1 \). Weak scaling concerns the speedup of a problem which scales with the number of processors \( p \), which is governed by Gustafson’s law: speedup = \( a + b \times p \). Again we have \( a + b = 1 \). For simplicity, we refer to \( b \) as strong/weak scaling efficiency in the context of strong/weak scaling. Please note that scaling efficiency may have different definitions in other literatures.
Table 1: Communication and computation costs per layer per processor

| entry \ framework | Megatron-LM | Optimus | Megatron-LM | Optimus |
|-------------------|------------|---------|------------|---------|
| forward communication | $\frac{8(p-1)}{p}bh$ | $\frac{\log(p)}{\sqrt{p}}(7bh + 12h^2)$ | $\frac{8(p-1)}{p}bh + \frac{12(d-1)}{d}h^2$ | $\frac{\log(p)}{\sqrt{p}}(7bh + 12h^2) + \frac{12(d-1)}{d}h^2$ |
| backward communication | $\frac{8(p-1)}{p}bh + \frac{24(d-1)}{d}h^2$ | $\frac{\log(p)}{\sqrt{p}}(21bh + 36h^2) + \frac{24(d-1)}{d}h^2$ | $\frac{8(p-1)}{p}bh + \frac{24(d-1)}{d}h^2$ | $\frac{\log(p)}{\sqrt{p}}(21bh + 36h^2) + \frac{24(d-1)}{d}h^2$ |
| forward computation | $\frac{1}{p}(12bh^2 + 2bs^2h)$ | $\frac{1}{p}(12bh^2 + 2bs^2h)$ | $\frac{1}{p}(12bh^2 + 2bs^2h)$ | $\frac{1}{p}(12bh^2 + 2bs^2h)$ |
| backward computation | $\frac{1}{p}(36bh^2 + 6bs^2h)$ | $\frac{1}{p}(36bh^2 + 6bs^2h)$ | $\frac{1}{p}(36bh^2 + 6bs^2h)$ | $\frac{1}{p}(36bh^2 + 6bs^2h)$ |

Experiments are carried on TACC Frontera RTX nodes. Each node has two Intel Xeon E5-2620 v4 CPUs, 128 GB / 2400MHz DDR4 synchronous memory, and 4 NVIDIA Quadro RTX 5000 GPUs. Each CPU has 16 cores and a frequency of 2.10 GHz. Each GPU has 3072 CUDA parallel processing cores, 16 GB GDDR6 memory, and an FP32 performance of 11.2 TFLOPS. GPUs are interconnected with Mellanox InfiniBand HDR, which supports connectivity at 200 GB/s between switches and 100 GB/s between GPUs. Experiments are run on 1, 4, 9, and 16 nodes for model parallelism part.

For both strong and weak scaling experiments, we fix $N = 24$ and $s = 512$, $n$ is kept proportional to $h$. Activation checkpointing is enabled. 32-bit float data type is used. We only test the consecutive Transformer layers (i.e., without the embedding layer or subsequent loss calculation layers) that is common to a variety of language models. For now, we only test model parallelism. For $h = 2048$, the number of parameters reaches 1.2 billion, or 4.5 GB in memory. For other settings, the number of parameters is proportional to $Nh^2$. The scaling jobs for Optimus and Megatron-LM for the same number of GPUs is submitted in one batch to exclude the difference of communication network structure. It should also be noted that, the baseline ($p = 1$) cases for Optimus and Megatron are the same, for strong/weak scaling, respectively. The baseline cases are implemented with Megatron-LM. When $p = 1$ and $d = 1$, there is no communication, and Optimus and Megatron-LM reduce to the same workflow, so their difference should be minor.

For weak scaling, we fix the number of parameters per GPU. For Optimus, we also fix the size of hidden activations per GPU. In other words, we fix $h/p$ and $b$. For Megatron-LM, however, due to the memory limit, we cannot use the same batch size as Optimus, as it must store the entire activations. Here we point out that, for Megatron-LM, smaller batch size does not result in an unfair comparison, as both computation and communication costs are proportional to batch size with other dimensions fixed. Data for weak scaling is shown in Table 2.

For strong scaling, we fix the problem size. More specifically, we fix $h = 3072$ and $b = 24$ for $p = 4, 16, 64$. For $p = 36$, to fulfill the divisibility requirement for both Megatron-LM ($p$, $n$, $h$) and Optimus ($q$, $p$, $n$, $h$), we use $h = 3096$ and $n = 72$. It is noted that $n$ does not contribute to either computation or communication volume, and that the alteration in $h$ is negligible, making the problem size nearly constant. Data for strong scaling is shown in Table 3.

Weak and strong scaling efficiencies for pure model parallelism are shown in Figure 6. It is noted that, in both weak scaling and strong scaling, when the number of GPUs $p = 4$, Optimus is inferior to Megatron-LM, despite SUMMA having a better parallelism strategy than one-dimensional partition. The reason is two-fold: 1) SUMMA has a superior iso-efficiency does not necessarily mean that it can perform better in all circumstances, especially when the number of processors is too few. 2) In Optimus, all parameters, activations, and gradients are within the SUMMA scheme and should be communicated, while Megatron-LM only has to communicate activations and gradients. In other words, there is a fixed overhead for Optimus to process a batch, and it can only show its advantage when activations are at least comparatively as large as parameters.

Another point to notice is that, the strong scaling efficiency for Optimus increases from $p = 16$ to $p = 36$ and $p = 64$, while it should typically decrease with $p$. It can be explained from the Amdahl’s law. Given speedup $= 1/(a + b/p)$ and $a + b = 1$, we can get $b = p(speedup - 1)/(p-1)$. The term $p/(p-1)$ is approximately unity for $p > 16$. However, for Optimus, as shown in Table 1, the communication time decreases as $p$ increases, resulting in an improved speedup. No wonder strong scaling efficiency increases. Besides, the strong scaling efficiency for Optimus decreases from $p = 4$ to $p = 16$, contrary to the theory prediction. This could be explained by the communication network topology: GPUs within one node might have a higher bandwidth than those spanning multiple nodes.

In Table 4, we show the results with data parallelism and $d = 2$. It shows a similar pattern as shown in Table 2, and achieves $1.73 \times$ speed in training and $2.32 \times$ speed in reference over the baseline.

4.2 Memory performance

As mentioned in the introduction, One of the main purposes of model parallelism is to address the problem that one model can not fit in the memory of a single processor. To measure and compare the memory efficiency, for each GPU configuration (number of GPUs), we keep the neural network model the same as the settings in weak scaling (Table 2), and increment $b$ until out of memory error occurs for any of the GPUs within 10 iterations of training. It manifests that Optimus can run when $b = 480$, $s = 512$, and $h = 8192$ on 64 GPUs, in which case the entire activations or the input of each Transformer layer should be of shape $[480, 512, 8192]$, or 7.5 GB in memory, impossible for Megatron-LM to accommodate on a GPU with 16 GB memory. Detailed data is shown in Figure 5. As
expected, the maximum batch size Optimus can achieve increases with the number of GPUs, approximately proportionally to $q$, while that of Megatron-LM decreases. On 64 GPUs, Optimus can achieve a batch size 8 times of Megatron-LM.

### 4.3 Transformer-style BERT training

To demonstrate the correctness of Optimus, here we train a BERT-equivalent language model, one of the networks supported by Megatron-LM, and downstream tasks and show comparable results with Megatron-LM. Here we choose a BERT-small equivalent architecture ($N = 4$, $h = 512$, $n = 8$, $s = 128$, $v = 29184$) weighing 28 million parameters or 107 MB. Besides the embedding layer and Transformer layers, the Transformer output splits into two branches, one to compute the inner product of the final Transformer outcome with the transpose of the embedding table to produce the token-wise logits (lm loss), the other to compute the sentence-wise classification (sop loss). In order to test the correctness of distributed training, we use model parallel size equaling 4 for Optimus. For Megatron-LM, we use model parallel size equaling 1 and data parallel size equaling 4 for faster training. Total batch sizes for both frameworks are equal. Loss curves are shown in Figure 7, which show comparable results between Optimus and Megatron-LM. We use MultiNLI and Quora Question Pairs (QQP) as downstream tasks to further validate the effectiveness. We only train once for each task and framework. Accuracy results are shown in Table 5. In the table, MNLI m/mm stands for MultiNLI development set for matched/mismatched sentence pairs. Higher values are better for both the two tasks. Despite the minor difference in pre-trained model parameters, Optimus shows comparable results with Megatron-LM.
Table 4: Weak scaling settings with data parallel size equaling 2.

| #nodes | #GPUs | batch size/#mp groups | hidden size | #attention heads | forward time/batch size(s) | backward time/batch size(s) | throughput (#sequences/s) | inference (#sequence/s) | GPU utilization |
|--------|-------|------------------------|-------------|----------------|--------------------------|----------------------------|----------------------------|-----------------------|-----------------|
| 2      | 8     | 60                     | 2048        | 32             | 0.0406                   | 0.1348                     | 5.7019                    | 24.6346              | 0.3280          |
| 8      | 32    | 60                     | 4096        | 64             | 0.1205                   | 0.2949                     | 2.4073                    | 8.3013               | 0.1357          |
| 18     | 72    | 40                     | 6120        | 72             | 0.2260                   | 0.5446                     | 1.2979                    | 4.4239               | 0.0721          |
| 32     | 128   | 30                     | 8192        | 128            | 0.2630                   | 0.5535                     | 1.2247                    | 3.8020               | 0.0683          |

Figure 5: The memory limits of Megatron-LM and Optimus. The data points are labeled in forms of \( b_1 / b_2 \), which means that Optimus or Megatron-LM can run with \( b = b_1 \), but cannot run with \( b = b_2 \).

Table 5: Downstream tasks accuracy. For both tasks, we use batch size \( b = 4 \), warmup ratio 0.064, and train for 5 epochs. Other hyperparameters are kept the same as the pre-training (Figure 7).

| framework/task | MNLI m/mm accuracy | QQP accuracy |
|----------------|--------------------|--------------|
| Megatron-LM    | 72.28/72.06        | 88.71        |
| Optimus        | 72.68/72.59        | 88.34        |

5 DISCUSSION

5.1 Hyperparameters

It is acknowledged that the choosing of specific values of \( b, s, \) and \( h \) does affect the performance of Optimus, which is reflected in weak and strong scaling experiments as shown in Figure 6. In weak scaling, relatively large batch sizes are used, resulting in larger volume ratios of activations to parameters, and Optimus outperforms Megatron-LM in 3 out of 4 data points; While in strong scaling, smaller batch sizes are used, so Optimus could not outperform Megatron-LM until on 64 GPUs. This result is consistent with the analysis in Table 1, as in model parallelism, Optimus requires the communication of both parameters and activations as well as their gradients, while Megatron-LM only communicates activations and their gradients. Intuitively, one can think of the communication of parameters (and their gradients) as a fixed cost for processing a batch. So the higher the ratio between activations to parameters, the more efficient Optimus is.

In this sense, one may concern that weak/strong scaling comparisons are not fair, as Optimus can only outperform Megatron-LM under specific hyperparameters. Here we legitimize the hyperparameters by comparing them with those used in the training of recent large language models, as shown in Table 6. It manifests that the volume ratios of activations to parameters vary a lot among these models, as reflected by \( b_s/h \). It also indicates a great margin for improving \( b \), which is in favor of Optimus. Besides, when we compare the volume ratios within a model parallel group \( b_s/h_d \), it manifests that our settings for weak scaling experiments match that of MT-NGL (280 DGXs), both having \( b_s/h_d = 24 \). And our settings for strong scaling experiments are of the same level as Megatron-LM (BERT/GPT-2).

5.2 Distributed matrix-matrix multiplication algorithms

Though even more advanced matrix-matrix multiplication algorithms \([1, 20]\) exist, SUMMA, as a two-dimensional algorithm, is shown to have near-optimal results over a wide range of settings. Also, SUMMA can keep the sizes of input and output matrices scale proportionally to the number of processors, while the 3D \([1\) and 2.5D \([20] methods cannot. Besides, the two-dimensional structure naturally fits the topology adopted by state-of-the-art super computers for neural network training. Some studies have focused on...
Table 6: Hyperparameter comparison. This table includes representative models for comparison. 1) MT-NLG used a "3D parallel system" combining data, pipeline, and tensor-slicing (intra-layer) based parallelism together. Specifically, they used 8-way tensor-slicing within a node and 35-way pipeline parallelism across nodes. $d$ is inferred from the number of DGX A100 servers they used, each having 8 NVIDIA A100 80GB Tensor Core GPUs. $s$ is given for 560 DGXs case and assumed the same for other cases. During training, the batch size was gradually increased by 32, starting at 32, until it reaches 1920. 2) In Turing-NLG, they used a 4-way Megatron-LM style intra-layer parallelism and 64-way ZeRO style parallelism. 3) In GPT-3, $b$ is inferred from the number of tokens per batch given in the paper. Batch size was linearly increased from a small value (32k tokens) to the full value over the first 4-12 billion tokens of training. 4) In Megatron-LM (BERT), $s$ is not explicitly mentioned in the paper. It is instead inferred from the training script from github. 5) In BERT, the authors used sequence length of 128 for 90% of the steps, and 512 for the rest 10% of the steps to speedup training. BERT$_{BASE}$ was trained on 4 TPU pods (16 TPU chips), and BERT$_{LARGE}$ was trained on 16 TPU pods (64 TPU chips). The exact configurations for data/model parallelism were not given.
frameworks based on other algorithms. Future works should address the problem of choosing the most suitable one for a specific task or machine.

5.3 Different parallelisms
Leveraging SUMMA, parameters and activations are distributed all the way during training, which is a major difference from existing model parallelism methods. Besides, fixing the problem size, the communication volume of model parallelism decreases with $p$, resulting in an increasing strong scaling efficiency. However, better scaling performance does not guarantee its superiority under all circumstances. It is noted that, the model parallelism of Megatron-LM only communicates activations, while ZeRO alone only communicates parameters and their gradients. Also, as shown in Figure 6, when the number of processors is small, Megatron-LM has a higher efficiency than Optimus. This work, together with Megatron-LM, is orthogonalexceptingpointing [5], mixed precision training [13], as well as ZeRO [15], a data parallelism alternative. In fact, in more recent works [14] in large model training, different parallelisms are combined to achieve desired performance. Optimus provides an strong alternative to existing intra-layer parallelism methods.

6 CONCLUSION
In this work, we demonstrate an original intra-layer model parallelism paradigm which successfully eliminates the memory bottleneck and increases communication efficiency over its baseline, thus paving the path for developing super large language models. As the applications of deep language models have been more and more prevailing, we believe this work would have a huge impact on the language model training.

REFERENCES
[1] R. C. Agarwal, S. M. Balle, F. G. Gustavsson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. IBM Journal of Research and Development 39, 5 (1995), 575–582. https://doi.org/10.1147/rd.395.0575
[2] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computer capabilities. https://doi.org/10.1145/1465482.1465560
[3] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Savry, Amanda Askell, et al. 2020. Language models are few-shot learners. arXiv preprint arXiv:2005.14165 (2020).
[4] Lynn Elliot Cannon. 1969. A cellular computer to implement the Kalman filter algorithm. Ph.D. Dissertation. Montana State University-Bozeman, College of Engineering.
[5] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training Deep Nets with Sublinear Memory Cost. arXiv:1604.06174 [cs.LG]
[6] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]
[7] William Fedus, Barrett Zoph, and Noam Shazeer. 2021. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. arXiv:2101.03961 [cs.LG]
[8] Anamh Graha, Anshul Gupta, George Karypis, and Vinip Kumar. 2003. Introduction to parallel computing. Addison Wesley.
[9] John L. Gustafson. 1988. Reevaluating Amdahl’s law. https://doi.org/10.1145/412441.42415
[10] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. 2018. PipeDream: Fast and Efficient Pipeline Parallel DNN Training. arXiv:1806.03377 [cs.DC]
[11] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Max Xu Chen, Dehao Chen, Hyunjoong Jo, Jiaqin Nijam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. arXiv:1811.06965 [cs.CV]
[12] Dmitry Lepikhin, Hyunjoong Jo, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. arXiv:2006.16668 [cs.NE]
[13] Paulius Micikevičius, Sharun Narang, Johnon Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Olegski Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed Precision Training. arXiv:1710.03740 [cs.AI]
[14] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitrii Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. 2021. Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM. https://doi.org/10.48550/ARXIV.2104.04473
[15] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO Memory Optimizations Toward Training Trillion Parameter Models. arXiv:1910.02054 [cs.LG]
[16] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deep-speed: System optimizations enable training deep learning models with over 100 billion parameters. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 3505–3566.
[17] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Pernporn Koanantakool, Peter Hawkins, Hyunjoong Jo, Minghsheng Hong, Cliff Young, Ryan Sepassi, and Blake Heckman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. arXiv:1810.04805 [cs.LG]
[18] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.00853 [cs.CL]
[19] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shiramai Prabhumoye, George Zervesan, Vijay Korthikanti, Elton Zhang, Rewon Child, Reza Yazdani Aminabadi, Julie Bernauer, Xia Song, Mohammad Shoeybi, Yuxiong He, Michael Houston, Saurabh Tiwary, and Bryan Catanzaro. 2022. Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model. https://doi.org/10.48550/ARXIV.2201.11990
[20] Edgar Solomonik and James Demmel. 2011. Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms. In Euro-Par 2011 Parallel Processing, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 90–109.
[21] Robert A. van der Geijn and Jerrel Watts. 1998. SUMMA: scalable universal matrix multiplication algorithm. Concurrence and Computation Practice and Experience 9 (1998), 255–274.
[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]