2.5-dimensional distributed model training

Boxiang Wang *
National University of Singapore
Singapore
boxiangwang059@gmail.com

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

Zhengda Bian
National University of Singapore
Singapore
zbian@comp.nus.edu.sg

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

Abstract

Data parallelism does a good job in speeding up the training. However, when it comes to the case when the memory of a single device cannot host a whole model, data parallelism would not have the chance to do anything. Another option is to split the model by operator, or horizontally. Megatron-LM introduced a 1-Dimensional distributed method to use GPUs to speed up the training process. Optimus is a 2D solution for distributed tensor parallelism. However, these methods have a high communication overhead and a low scaling efficiency on large-scale computing clusters. To solve this problem, we investigate the 2.5-Dimensional distributed tensor parallelism. Introduced by Solomonik et al., 2.5-Dimensional Matrix Multiplication developed an effective method to perform multiple Cannon’s algorithm at the same time to increase the efficiency. With many restrictions of Cannon’s Algorithm and a huge amount of shift operation, we need to invent a new method of 2.5-dimensional matrix multiplication to enhance the performance. Absorbing the essence from both SUMMA and 2.5-Dimensional Matrix Multiplication, we introduced SUMMA2.5-LM for language models to overcome the abundance of unnecessary transmission loss resulted from the increasing size of language model parallelism. Compared to previous 1D and 2D model parallelization of language models, our SUMMA2.5-LM managed to reduce the transmission cost on each layer, which could get a 1.45X efficiency according to our weak scaling result between 2.5-D [4,4,4] arrangement and 2-D [8,8,1] arrangement.

1 Introduction

Deep Neural Network models [8, 9, 11, 14] are developing at a high speed, and the development of chip is not capable to keep up with it. A natural remedy is to scale the training to multiple chips or servers. A straightforward way is to use data parallelism, which could not only solve the issue of insufficient memory on each GPU, but also speed up the training process. It is to have multiple devices each holding an identical model process distinct mini-batches of input. Gradients are accumulated among all devices after back propagation and parameters are updated synchronously. By applying data parallelism, each processor would be able to store parameters within their limited storage, and thus offset the issue of limited memory. It is equivalent to training a model with a large batch size. To alleviate the poor generalization performance resulting from large batch, researchers proposed optimizers LAMB [17] and LARS [16].

*equal contribution

Technical Report of NUS HPC-AI Lab (https://ai.comp.nus.edu.sg)
Data parallelism \[18\] does a good job in speeding up the training. However, when it comes to the case when the memory of a single device can not host a whole model, data parallelism would not have the chance to do anything. Activation checkpointing \[3\] alleviated the memory constraints by using more computation. Mixed precision training \[7\] reduced both the memory and computation requirement. ZeRO-infinity \[10\] proposes to use CPU memory and NVMe to extend the GPU memory while preserving decent throughput by largely overlapping computation and communication. Another branch is model parallelism. GPipe \[6\] and PipeDream \[5\] split the model vertically. Each GPU hosts a partition of the whole model, takes input from the previous GPU and send the output to the next GPU during forward propagation, and does the reverse during backward propagation. However, they suffer from either gradient staleness or GPU idleness. Another option is to split the model by operator, or horizontally. In 2019, Megatron-LM \[11\] introduced a 1-Dimensional distributed method to use GPUs to speed up the training process. Optimus \[15\] leveraged SUMMA \[13\] and took a step further to improve both the memory and communication efficiency. From their works, language model with growing amount of parameters could be trained on separated processors whose memory could not afford the training procedure independently. While at the same time, due to the disadvantages of SUMMA, the communication within a \([p, p]\) shape processor array takes up to \(2p^3\) times of transmissions among its column and row, together with the increment of \(p\), the increment of transmission loss will also be insufferable and thus reduce the efficiency of SUMMA. Due to this consideration, we looked for the 2.5-dimensional matrix multiplication algorithm which could reduce this effect and save the unnecessary transmissions, from our experiment, we find out that with same number of processors and input matrix, our 2.5-dimensional would have less transmission loss on each layer and thus reach a increment of efficiency.

Introduced by Solomonik et al., 2.5-Dimensional Matrix Multiplication \[12\] developed an effective method to perform multiple Cannon’s algorithm \[2\] at the same time to increase the efficiency. With many restriction of Cannon’s Algorithm and huge amount of shift operation, we need to invent a new method of 2.5-dimensional matrix multiplication to enhance the performance. Absorbing the essence from both SUMMA and 2.5-Dimensional Matrix Multiplication, we introduced SUMMA2.5-LM for language model to overcome the abundance of unnecessary transmission loss result from increasing size of language model parallelism.

In order to suit Transformer with our SUMMA2.5-LM, we developed the parallelization for feed forward section, attention section and non-matrix-multiplication sections separately. For feed forward layer we applied two SUMMA2.5 algorithm to get the output, the algorithm will store the parameter matrices inside each processor for next computation to avoid waste of transmission. To realize the distributed multi-head attention layer, we compute corresponding \(Q, K, V\) matrices and then attention output respectively, in this section, the attention would be computed separately on each processors and to obtain \(Q, K, V\) matrices the algorithm will perform matrix multiplication among the whole layer. For non-matrix-multiplication sections like layer normalization, the algorithm will broadcast the matrix along column, let the distributed processors to work at the same time. Compared to previous 1D and 2D model parallelization of language models, our SUMMA2.5-LM managed to reduce the transmission cost on each layer.

## 2 Preliminary and related work

### 2.1 Cannon’s Algorithm

Cannon’s Algorithm described by Cannon in 1969 is widely used for matrix multiplication on distributed systems. Cannon’s algorithm applied master/slave mode and divide and concur mode. It’s zeroth processor will arrange the I/O for all other processors, it controls the broadcast and the reduce procedures. In another prospective, it divides the multiplication of two input matrices into small pieces, after the calculation on each processor, the final result will be the combination of results from all processors. Cannon’s algorithm does not only parallelizes the matrix multiplication, but also reduces the storage needed for each processors. With number of arithmetic operations per process equals to \(\frac{n^3}{p}\), memory size per process scales with \(\Omega\left(\frac{n^2}{p}\right)\). The lower bound on communication time and estimated lower bound of latency are:

\[
W = \Omega\left(\frac{\text{number of arithmetic operations}}{\sqrt{\text{memory size}}}\right) = \Omega\left(\frac{n^2}{\sqrt{p}}\right).
\] (1)
\[ S = \Omega\left(\frac{\text{number of arithmetic operations}}{(\text{memory size})^{3/2}}\right) = \Omega(\sqrt{p}). \] (2)

The idea of Cannon’s algorithm is described in Algorithm 1. Assume there are \( p = q^2 \) processors in \([q, q]\) shape: Firstly, shift matrix \( A \)'s \([q, q]\) partitioned matrices left by their corresponding row number. Secondly shift matrix \( B \)'s \([q, q]\) partitioned matrices up by their corresponding column number. After these two steps, the algorithm will start to compute the matrices and get corresponding \( C_{ij} \). Then all partitioned matrices \( A_{ij} \) will be shifted left by one, all partitioned matrices \( A_{ij} \) will be shifted up by one, and add the calculated \( C_{ij} \) to previous result, repeat this procedure by \( n \) times. Output \( C \) by combining all \( C_{ij} \) accordingly.

**Algorithm 1: 2D matrix multiplication Cannon’s Algorithm**

Input : Matrix \( A \) with size \( a \times b \); Matrix \( B \) with size \( b \times c \)

Output : Matrix \( C = A \times B \) with size \( a \times c \)

split \( A, B \) in to \( p \) parts to match the processor shape;

store \( A_{ij}, B_{ij} \) into \( p_{ij} \) accordingly;

store \( C_{ij} = 0 \) into \( p_{ij} \) accordingly

\[ \text{for } i, j \in \{0, \ldots, q - 1\} \text{ do} \]

\[ \quad \text{shift } A_{ij} \text{ to } p_{i(j-i)}; \]

\[ \quad \text{shift } B_{ij} \text{ to } p_{i(j-j)}; \]

\[ \quad \text{for } t = 0 \rightarrow q - 1 \text{ do} \]

\[ \quad \quad C_{ij} = C_{ij} + A_{ij} \ast B_{ij}; \]

\[ \quad \quad \text{shift the submatrix } A \text{ owned by } p_{ij} \text{ to } p_{i(j-1)}; \]

\[ \quad \quad \text{shift the submatrix } B \text{ owned by } p_{ij} \text{ to } p_{(i-1)j}; \]

end

end

combine all \( C_{ij} \) accordingly to \( C \);

return \( C \)

**2.2 SUMMA**

The Scalable Universal Matrix Multiplication Algorithm (SUMMA) provides a more effective and efficient algorithm for the 2D matrix multiplication. For SUMMA, Algorithm 2 is the pseudo code for \( C = A \ast B \). The corresponding differentiation for \( A', B' \) can be calculated as:

\[ A' = C' \ast B'^T, B' = A'^T \ast C' \] (3)

By arranging the \( p \) processors into a \( \sqrt{p} \ast \sqrt{p} \) mesh, the matrices \( A \) and \( B \) are also partitioned to \( p \) parts accordingly. After the the partitions of \( A \) and \( B \) are sent to the corresponding processors, the SUMMA algorithm allows each processors to calculate in parallel. At the end of the computation, the algorithm returns the resulting matrix \( C \), distributed among the processors, in the same manner as \( A \) and \( B \) are partitioned.

**Algorithm 2: 2D matrix multiplication SUMMA**

Input : Matrix \( A \) with size \( a \ast b \); Matrix \( B \) with size \( b \ast c \)

Output : Matrix \( C = A \ast B \) with size \( a \ast c \)

split \( A, B \) in to \( p \) parts to match the processor shape;

store \( A_{ij}, B_{ij} \) into \( p_{ij} \) accordingly;

\[ \text{for } i, j \in \{0, \ldots, p^{1/2} - 1\} \text{ (concurrently) do} \]

\[ \quad C_{ij} = 0; \]

\[ \quad \text{for } t = 0 \rightarrow \sqrt{p} - 1 \text{ do} \]

\[ \quad \quad \text{broadcast } A_{it} \text{ in } p_{it} \text{ to } p_{ij}; \]

\[ \quad \quad \text{broadcast } B_{tj} \text{ in } p_{ij} \text{ to } p_{ij}; \]

\[ \quad \quad C_{ij} = C_{ij} + A_{it} \ast B_{tj}; \]

end

end

combine all \( C_{ij} \) accordingly to \( C \);

return \( C \)


2.3 2.5-Dimensional Matrix Multiplication

In 2011, E Solomonik et al. introduced a 2.5D matrix multiplication method to reduce communication time used in Cannon’s Algorithm. This method is named as 2.5D because it has special cases of both 2D and 3D matrix Multiplication. It uses multiple processors with a shape of $\sqrt{p/d} \times \sqrt{p/d} \times d$ where $p$ represents the number of processors and $d$ represents the depth of the processor group. Compared with 2D Cannon’s Algorithm and PDGEMM by ScaLAPACK, 2.5D algorithm could speed up the calculation with less communication cost. The lower bound on communication time is

$$W = \Omega\left(\frac{\text{number of arithmetic operations}}{\sqrt{\text{memory size}}}\right) = \Omega\left(\frac{n^2}{\sqrt{dp}}\right)$$ (4)

The estimated lower bound of latency is

$$S = \Omega\left(\frac{\text{number of arithmetic operations}}{(\text{memory size})^{3/2}}\right) = \Omega\left(\frac{p^{1/2}}{d^{3/2}}\right)$$ (5)

Where number of arithmetic operations per process equals to $\frac{n^2}{p}$, memory size per process scales with $\Omega\left(\frac{dn^2}{p}\right)$. In special cases like $d = 1$, the 2.5D algorithm degenerates to Cannon’s algorithm; when $d = p^{1/3}$, it becomes 3D algorithm.

2.4 Transformer

Transformer was published by Google in 2017. Before that, natural language processing (NLP) was dominant by recurrent neural networks like LSTM. Due to the sequential nature of these models, it was difficult to train with large batch size. Transformer broke the serial dependency, and allowed each hidden vector to attend to any preceding ones. In this manner, the training of Transformer models could be elegantly formulated into basic matrix-matrix multiplications, thus became scalable.

The original Transformer consists of an encoder and decoder. And encoder and decoder are again composed of multi-head attention and feed forward layers. In Megatron-LM, the architecture is adapted in a manner that the whole model consists of multiple identical Transformer layers. Each Transformer layer consists of a self-attention module and a multi-layer perceptron (MLP). MLP is simply two linear layers with an activation function in the middle, the first projecting the hidden vector to a higher dimension while the second projecting it back to the original hidden size. Self-attention module first uses a linear layer to project the original hidden vector into $n$ queries ($Q$), keys ($K$) and values ($V$). Multi-head self attention is calculated as:

$$A = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$ (6)

The $n$ $A$’s are then rearranged back to the hidden size, and undergoes another linear layer to produce the output of self-attention module. This design makes self-attention module and MLP each have 2 linear layers, facilitating the row-column partitioning.

3 SUMMA2.5 - LM

3.1 SUMMA2.5

We designed a SUMMA-like matrix multiplication method in 2.5 dimensional arrangement of matrix and processors. In this section, notations of parameters are:

- SUMMA2.5 dimension: $q$
- SUMMA2.5 depth: $d$
- number of processors: $p$
- batch size: $b$
- hidden size: $h$
- sequence length: $s$
• number of Transformer layers: $N$

Where $p = dq^2$ and $1 \leq d \leq q$ where $d = 1$ makes SUMMA2.5 a 2D algorithm like original SUMMA, and $d = q$ makes SUMMA2.5 a 3D algorithm. The $p$ processors will be arranged in a shape of $[q,q,d]$ as shown in Figure 1. Similar to SUMMA algorithm, SUMMA2.5 splits input matrix $A$ with a shape of $[a,b]$ and matrix $B$ with a shape of $[b,c]$ into partitions to match the arrangement of $p$ processors, after the calculation, and outputs the matrix $C$ with a shape of $[a,c]$ combined from all $C_{ijk}$ on different processors. The procedure of our matrix multiplication is described in Algorithm 3. The method to split and combine the matrices is shown in Fig.2. After calculation of the partitioned matrix multiplication between size $[a/dq,b/q]$ and $[b/q,c/q]$, the respective result matrix will in shape $[a/dq,c/q]$ which is stored in respective processors just like input matrix $A$, thus the algorithm could use broadcast and reduce in the same way for matrices $A$ and $C$.

For calculation of $C = A \ast B^T$, we uses a algorithm which broadcasts $B_{ijk}$ within its column and computes $C_{ijk} = A_{ij} \ast B_{ijk}^T$, then the algorithm reduces the $C_{ijk}$ within its column to $C_{itk}$. Similar for function $C = A^T \ast B$, the algorithm broadcasts $B_{itk}$ within its row, computes $C_{ijk} = A_{it}^T \ast B_{itk}$, then reduces the $C_{ijk}$ within its column to $C_{tjk}$.

In this arrangement, the processors could work on $d$ SUMMA-like matrix multiplication separately, thus reach the target to reduce the computation time used. Compared to the 2.5D matrix multiplication mentioned in section 2.3, our algorithm uses less memory on each processors and less transmissions between processors. Compared to 2-D SUMMA algorithm, our work could make the matrices with different depth conduct matrix multiplication concurrently and relatively independent (except necessary communication for parameter matrices), which could reduce the required time for matrix multiplication with huge amount data. As the lower bound of communication time and latency mentioned in 2.5-D algorithm, $W = \Omega(\frac{n^2}{\sqrt{dp}}), S = \Omega(\frac{p^{1/2}}{d^{3/2}})$, when $d > 1$, we have lower communication time and latency compared to 2-D algorithm. We could get the conclusion that with same amount of processors, greater $d$ could lead to less communication time and lower latency. In special case $d = p^{1/3}$, we have $W = \Omega(\frac{n^2}{p^{1/3}}), S = \Omega(1)$, where the SUMMA2.5 could yield best efficiency.

For computation of matrices’ gradients, the function (3) is applied in SUMMA2.5. For matrix $A, C$, the $dq^2$ partitioned matrices will return $dq^2$ partitioned gradient matrices, but for matrix $B$, the $q^2$ partitioned matrices will return $dq^2$ partitioned gradient matrices, in order to get a correct shape of gradients, our algorithm applied all – reduce function after the computation of $B'$ on processors with same row and column but different depth.

Figure 1: $p$ processors in a 2.5D arrangement of shape $[p,p,d]$
Figure 2: Method to split input matrices $A, B$, and method to combine output matrix $C$ assuming processors’ shape $[q = 2, q = 2, d = 2]$. The blue region representing a layer among the processors with a shape of $[q = 2, q = 2]$. (a) Matrix $A$ with shape $[a, b]$ will be split into $dq^2$ partitioned matrices with shape of $[a/qd, b/q]$, $[q, q]$ partitioned matrices will be stored in each layer. (b) Matrix $B$ with shape $[b, c]$ will be split into $q^2$ partitioned matrices with shape of $[b/q, c/q]$, $[q, q]$ partitioned matrices will be stored in each layer. (c) $dq^2$ partitioned matrices with shape of $[a/qd, b/q]$ will be combined into matrix $C$ with shape $[a, b]$.

Algorithm 3: 2.5D matrix multiplication SUMMA2.5 ($p$ processors in $[q, q, d]$ shape)

**Input**: Matrix $A$ with size $[a, b]$; Matrix $B$ with size $[b, c]$

**Output**: Matrix $C = A \times B$ with size $[a, c]$

split $A, B$ into partitioned matrices with shape of $[a/qd, b/q]$ and $[b/q, c/q]$ accordingly:

For $i \in \{0, ..., qd - 1\}, j \in \{0, ..., q - 1\}$ do

- $h = i\% p$;
- $k = i/p$;
- store $A_{ij}$ into $p_{kjh}$;
- $C_{ij} = 0$;
- store $C_{ij}$ into $p_{kjh}$;

end

For $i \in \{0, ..., p - 1\}, j \in \{0, ..., p - 1\}, k \in \{0, ..., d - 1\}$ do

- store $B_{ij}$ into $p_{ijk}$;

end

For $i, j \in \{0, ..., p - 1\}, k \in \{0, ..., d - 1\}$ (concurrently) do

- for $t \in \{0, ..., p - 1\}$ do

  - broadcast $A_{itk}$ in $p_{itk}$ to $p_{ijk}$;
  - broadcast $B_{tjk}$ in $p_{tjk}$ to $p_{ijk}$;
  - $C_{ijk} = C_{ijk} + A_{itk} * B_{tjk}$;

end

end

combine all $C_{ijk}$ accordingly to $C$;

return $C$

3.2 SUMMA2.5 on language model

In our work, we applied our SUMMA2.5 on Transformer. There are encoders and decoders in Transformer, where encoder consists of a multi-head attention layer and a feed forward layer (fully
To compute the corresponding $E_X$, the procedure of our parallelized multi-head attention layer is shown in the Fig. 3 (b). In our work, we applied SUMMA2.5 to parallelize the matrix multiplication. As shown in the Figure [3], we present both the multi-head attention layer and feed forward layer. In our work, there are two different types of operations, defined as matrix multiplication related and non-related.

### 3.2.1 Matrix multiplication procedure

**Feed forward layer** For feed forward layer which could be understood as a simple multi-layer perceptron layer, will take input matrix with a shape of $[b, s, h]$ by default, multiply two parts of parameters in shape of $[h, 4h]$ and $[4h, h]$ accordingly, and the output of feed forward layer would also be in the shape of $[b, s, h]$. In the Transformer, the purpose of using feed forward layer is to process the output matrix from one multi-head attention section to fit the next attention section as an input matrix. With the property of no communication with other input tokens nor inference issue, feed forward layer could be parallelized and thus improve the performance.

In our work, we split the input matrix into partitioned matrices with shape of $[b/dq, s, h/q]$, and split the matrices into shape of $[h/q, 4h/q]$ and shape of $[4h/q, h/q]$ respectively. As shown in the Figure [3], our work applied SUMMA2.5 to parallelize the matrix multiplication. As the output, the parallelized feed forward will return partitioned matrices with shape of $[b/dq, s, h/q]$ just as the input partitioned matrices.

**Multi-head attention layer** In this attention section, the input matrix with shape of $[b, s, h]$ multiplies with a $[h, 3h]$ weight matrix and get a $QKV$ matrix consisting of queries ($Q$), keys ($K$) and values ($V$) in the shape of $3 * [b, s, h]$. Then the $Q, K, V$ matrices will be partitioned into $n$ attention heads, the result matrices of $Q, K, V$ will be in shape $[s, h/n]$ for each sequence. Attention section gets an attention score $A$ in shape $[s, s]$ by performing $Q * K^T$, then gets the output of a single attention head by performing $A * V$. By gathering all the output of attention heads, the output will be in the shape of $[s, h]$. For all the sequences, the shape of corresponding matrix is $[b, s, h]$, after the matrix multiplication with parameter matrix in shape $[h, h]$, the output shape will be $[b, s, h]$, just like the input shape. Similar to the feed forward section, with no communication with other position’s tokens, the attention part is also parallelizable.

The procedure of our parallelized multi-head attention layer is shown in the Fig. 3 (b). In our implementation of the parallelized attention layer, we partitioned the input matrix into $[b/dq, s, h/q]$ matrices, with the partitioned $[h/q, 3h/q]$ parameter matrices, the resulted $Q, K, V$ matrices will be in shape of $3 * [b/dq, s, h/q]$. There will be $n/q$ attention heads on each processors, and the received $Q, K, V$ matrix for each attention head has a shape of $[s, h/n]$. After the same procedure between $Q, K, V$ matrices, the output matrices will be combined together in the shape of $[b/dq, s, h/q]$. After the matrix multiplication with $[h/q, h/q]$, the distributed processors will combine all the $[b/dq, s, h/q]$ matrices into the output $[b, s, h]$ matrix.

### 3.2.2 Other procedure

Besides feed forward and attention section, there are sections in Transformer that are not suitable to apply parallelized operation, for example, residual connection includes add and normalization operations. These kinds of sections will conduct operations locally on individual GPUs. For the bias-add operation, the matrices will be broadcast all each column for forward process, and the backward process drives the gradients to be reduced back to the processor on row 0. As mentioned above, layer normalization is used in each residual connection, for better presentation, the result of layernorm function could be described as:

$$
\hat{X} = \frac{X - E[X]}{\sqrt{Var[X]} + \epsilon}.
$$

To compute the corresponding $E[X] = \frac{\sum X_i}{n}, Var[X] = E[X^2] - (E[X])^2$, the processors will compute $X, X^2$ respectively and then run $all_{-reduce}$ function on each row. For the computation of
Figure 3: SUMMA2.5-LM’s procedure on both feed forward layer and multi-head attention layer, in this example, we used $q = 2, d = 2$ for presentation.

The gradient of $X$, the function could be described as:

$$X' = \frac{\delta J}{\delta X} - \frac{\left(\sum_i \frac{\delta J}{\delta \hat{X}_i}\right) \hat{X} + \sum_j \frac{\delta J}{\delta \hat{X}_j}}{n \sqrt{\text{Var}[X] + \epsilon}},$$

(8)

for calculation, the processor will use stored $X, X^2$ and $\frac{1}{\sqrt{\text{Var}[X] + \epsilon}}$, the calculation of will $\frac{\delta J}{\delta \hat{X}_i}, \frac{\delta J}{\delta \hat{X}_j}$ take place similar to $X, X^2$. Due to the size of parameters, the communication loss in this process is negligible compare to the matrix multiplication part.

4 Experiment

Our experiments are conducted on rtx section of TACC’s frontera server. In this section, there are 90 GPU nodes with 4 NVIDIA Quadro RTX 5000 per node. Our experiments are conducted with 1, 2, 4, 8, 16 nodes accordingly, with setting: changing SUMMA2.5 depth, fixed SUMMA2.5 dimension; changing SUMMA2.5 dimension and depth, fixed model parallel size respectively, and apply strong scaling and weak scaling respectively. Due to the restriction of the GPU usage by TACC, our experiments planned with more than 20 GPU nodes have not conducted, we are trying to find other possible solutions to continue our planned experiment to further evaluate our work. Due to the time limitation of our experiments, we chose random generated input matrix to check the algorithm and Xavier initialized parameter matrices, after the generation of matrices, we compute the matrix multiplication result directly and the result using our SUMMA2.5 method respectively, to guarantee the output is the same, in this case we could compare our SUMMA2.5-LM’s time usage with that of other parallelized language model.

4.1 Strong scaling

In this part of experiment, we used a fixed total parameter size described as strong scaling. As shown in Table[1] we chose the batch size = 16, hidden size = 1024, and number of attention heads = 32 for our strong scaling experiment due to the limitation of memory on each processor. From the results of the experiments, we could conclude that: With same SUMMA2.5 dimension, bigger SUMMA2.5 depth will reduce the forward/backward time per batch size, but not in inverse proportion.
Table 1: Strong scaling setting experiments

| No. of nodes | No. of GPUs | SUMMA2.5 shape | batch size | hidden size | No. of attention heads | forward time (s) | backward time (s) |
|--------------|-------------|----------------|------------|-------------|-----------------------|-----------------|-------------------|
| 1            | 1           | [1, 1, 1]      | 16         | 1024        | 32                    | 1.2583          | 3.2063            |
| 1            | 4           | [2, 2, 1]      | 16         | 1024        | 32                    | 0.8320          | 2.3102            |
| 2            | 8           | [2, 2, 2]      | 16         | 1024        | 32                    | 0.5859          | 1.4507            |
| 4            | 16          | [4, 4, 1]      | 16         | 1024        | 32                    | 0.6584          | 1.8494            |
| 8            | 32          | [4, 4, 2]      | 16         | 1024        | 32                    | 0.6217          | 1.8687            |
| 16           | 64          | [4, 4, 4]      | 16         | 1024        | 32                    | 0.5857          | 1.7133            |
| 16           | 64          | [8, 8, 1]      | 16         | 1024        | 32                    | 0.6536          | 1.8788            |

Table 2: Weak scaling setting experiments

| No. of nodes | No. of GPUs | SUMMA2.5 shape | batch size | hidden size | No. of attention heads | forward time (s) | backward time (s) |
|--------------|-------------|----------------|------------|-------------|-----------------------|-----------------|-------------------|
| 1            | 1           | [1, 1, 1]      | 24         | 1024        | 16                    | 1.4322          | 4.0159            |
| 1            | 4           | [2, 2, 1]      | 48         | 2048        | 32                    | 4.1692          | 12.2519           |
| 2            | 8           | [2, 2, 2]      | 96         | 2048        | 32                    | 4.1776          | 12.3237           |
| 4            | 16          | [4, 4, 1]      | 96         | 4096        | 64                    | 11.2124         | 33.8762           |
| 8            | 32          | [4, 4, 2]      | 192        | 4096        | 64                    | 11.2316         | 33.7722           |
| 16           | 64          | [4, 4, 4]      | 384        | 4096        | 64                    | 11.1911         | 33.7059           |
| 16           | 64          | [8, 8, 1]      | 192        | 8192        | 128                   | 34.9476         | 100.4571          |

since the communication loss will also increase. Besides, from the example between [4,4,4] and [8,8,1] arrangements, with same number of processors, bigger SUMMA2.5 depth could lead to less forward/backward time.

In the first situation, due to the increase of communication cost, double the depth could not result in a half forward/backward time as expected. While the outcome of experiment in strong scaling between [4,4,4] arrangement and [8,8,1] arrangement could because of the high communication loss, as the later arrangement supposed to have smaller matrices to compute.

4.2 Weak scaling

In this part of experiment, we used a fixed parameter size per processor described as weak scaling. As shown in the Table 2, we set the corresponding batch size, hidden size and number of attention heads to get a fixed size of input parameters, and the setting conforms $[b/dq, n/q, h/n] = [24, 16, 64]$ (change of $n$ does not affect the result) due to the memory of each GPU used.

5 Conclusion

In our work, we designed a SUMMA-like 2.5-Dimensional matrix multiplication method named as SUMMA2.5, and we applied this work on deep language model Transformer. By using the SUMMA2.5, we split the input matrices and parameter matrices according to the shape of the arrangement of processors in the group, thus reduced the time used on calculation. Besides, compared to 2.5-Dimensional Cannon’s Algorithm matrix multiplication, our SUMMA2.5 requires less memory on each GPU and it requires less transmission of data among GPUs, which in return could leads to a higher efficiency and less communication loss. For future enhancement, we will experiment with more processors to evaluate the overall performance of our SUMMA2.5-LM. Besides, we will try to find better parallelization method to further reduce communication loss.
References

[1] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.

[2] Lynn Elliot Cannon. A cellular computer to implement the Kalman filter algorithm. PhD thesis, Montana State University-Bozeman, College of Engineering, 1969.

[3] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[5] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training, 2018.

[6] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[7] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[8] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training.

[9] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. OpenAI blog, 1(8):9, 2019.

[10] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021.

[11] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[12] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, Euro-Par 2011 Parallel Processing, pages 90–109, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[13] Robert A. van de Geijin and Jerrel Watts. Summa: scalable universal matrix multiplication algorithm. Concurrency and Computation Practice and Experience, 9:255–274, 1998.

[14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[15] Qifan Xu, Shenggui Li, Chaoyu Gong, and Yang You. An efficient 2d method for training super-large deep learning models, 2021.

[16] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks, 2017.

[17] Yang You, Jing Li, Sashank Reddi, Jonathan Hseu, Sanjiv Kumar, Srinadh Bhojanapalli, Xiaodan Song, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. Large batch optimization for deep learning: Training bert in 76 minutes, 2020.

[18] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. Imagenet training in minutes, 2018.