Optimization on Sparse Matrix Multiplication

Joyce Yang
jingyi-yang@uiowa.edu
University of Iowa
Iowa City, Iowa, USA

Zhongrun Xiang
zhongrun-xiang@uiowa.edu
University of Iowa
Iowa City, Iowa, USA

Fred Morsy
fred-morsy@uiowa.edu
University of Iowa
Iowa City, Iowa, USA

ABSTRACT

The following work aims to investigate existing SpMM implementations for Deep Learning applications. Current SpMM implementations are well suited for scientific computing applications but, fail to bring speedup in many cases for Deep Learning applications. Throughout the following work we explore issues that arise in SpMM and discuss potential solutions such as Blocking, Loop Interchange, using various data structures and data formats. We use a real world dataset from the NLP model Transformer as our deep learning example and present numerous results.

ACM Reference Format:
Joyce Yang, Zhongrun Xiang, and Fred Morsy. 2020. Optimization on Sparse Matrix Multiplication. In *Iowa City '20: High Performance and Parallel Computing, October 15–16, 2020, Iowa City, IA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Artificial neural networks (ANNs) are a type of machine learning model that have been used in many fields for decades. It was known in early studies as multi-layer perception, a basic structure with multiple dense layers. However, earlier ANN models had a limited number of hidden layers due to algorithmic or computational limitations. In recent years, this problem has been overcome with algorithmic advances and accelerated GPU computing. With recent new approaches such as ReLU, mini-batch training, which have brought a revolution in computer vision, researchers have achieved significant improvements in applications for language translation, speech recognition, the game of Go, social media data analysis, intelligent systems, and many other fields.

Among these deep learning applications, Natural Language Processing (NLP) is one of the major branch of artificial intelligence that deals with the interaction between computers and humans using natural language. The Transformer is a deep learning model architecture that was designed in 2017 to handle sequential data, such as natural language for tasks such as machine translation. The reason why Transformer has been successful in the NLP field is due to the use of the multi-head attention layers. However, there are many issues related to Transformer, such as the high demand for computing ability. Especially, the Multi-head attention calculation, the main part in Transformer, takes an N x N attention matrix for every attention head, which can consume large amounts of memory as shown in 1. Thus, researchers have been developing the sparse Transformer models by introducing sparsity in the input [2], layers [5], or post pruning on weights [4].

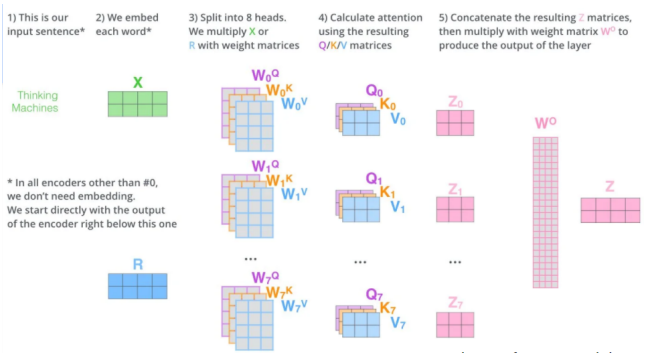

Figure 1: An illustration of the multi-head attention in Transformer model.

2 PROBLEM IDENTIFICATION

The following project aims to investigate how to accelerate unstructured sparsity on CPUs and GPUs for deep learning inference. Previous works have discussed computations that arise in neural network pruning and have attempted to handle these computations on GPUs. In a DNN pruned via unstructured pruning or the Sparse Transformer model, the core operation at inference time is SpMM (Sparse Matrix-Matrix Multiplication). The main downside of deep learning models is that they tend to over fit data. Previous works have discovered that one may prune a model by only including useful weights while preserving classification accuracy. Here the process of pruning is to set useless weights to zero and only include useful weights. After this we are left with a sparse weight matrix. The main operation in many pruning methods for Deep Neural Networks is Sparse Matrix Matrix Multiplication. SpMM is a linear algebra operation in which a sparse matrix in multiplied with a dense matrix. Current SpMM solutions are catered to scientific computing applications, this is where the dimensions of the sparse matrix are massive compared to the dense matrix dimension and only achieve significant speed up when the sparsity ratio is very high (99.0-99.9%). However the dimensions of the dense matrix may be larger than the sparse matrix dimension for SpMM problems in deep learning. Existing implementations often fail to bring speedups in these cases. Our project aims to improve existing SpMM implementations for Deep Learning using optimization techniques mentioned in class. In particular we explore how to model and improve the movement of data. Our discussion is driven by

Iowa City '20, October 15–16, 2020, Iowa City, IA

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

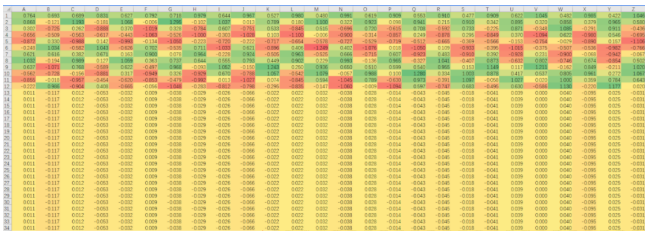

Figure 2: An illustration of the dense matrix A. Row 13 to 34 are the same due to 0-padding.

implementing algorithms and trying to improve performance using techniques such as blocking/tiling and Loop interchange. In other words we would like to take advantage of data locality. Since most entries in the Sparse matrix are 0s, we are able to reduce computation and storage by considering other data formats such as CSR and COO in order to reduce memory overhead. Other techniques include Multi-threading, Load Balancing, Shared Memory on GPU are also addressed by recent studies[4]. This study will try multiple techniques to improve the running time as our performance metric.

3 DATA

In this project, the NLP model Transformer is used as the deep learning example. The sparse Transformer from the Github project Sparse Variant of Transformer[1] is used to generate the sparse Transformer weights. In this project, only the first layer of the multi-head attention layer is extracted from the transformer model. In our project, the WMT 2016 English-to-German dataset (en2de) is used. There are 29,000 samples in training set, 1024 samples in test set.

In this project, we trained a sparse Transformer model using the 29,000 English-to-German sentences. As a general setting, the dimension is 512 in Transformer models for multi-head attention layer. Since the longest word in the test sentences is 34, after the zero-padding in Transformer, the dimension of our input sentences after the word embedding and position embedding is 34*512. This is our dense matrix. The sparse matrix used in this study is the weights in the Transformer layer, which is 512*512. In our study, we set the sparsity level of the sparse matrix at 90%. In our project, we only evaluate our methods on one sample for speed evaluation since they are similar.

Thus, the task we are going to optimize is the matrix-matrix multiplication AxB, where A is a dense matrix with dimensions 34*512 representing the sentences in the NLP models, and B is a sparse matrix with dimensions 512*512 representing the sparse Transformer layer weights at a 90% sparsity level.

As is shown in Figure 2, the dense matrix A has 34 rows. However, this example shows a sentence of 12 words only, and the rows from 13 to 34 are all the same as the 0-padding results. Figure 3 shows the sparse matrix B with 0s in green and is clearly unstructured.

4 ALGORITHMS

In this project, we used the blocking, loop interchange, and 1D/2D representation of the matrix on CPU and shared memory on GPU.

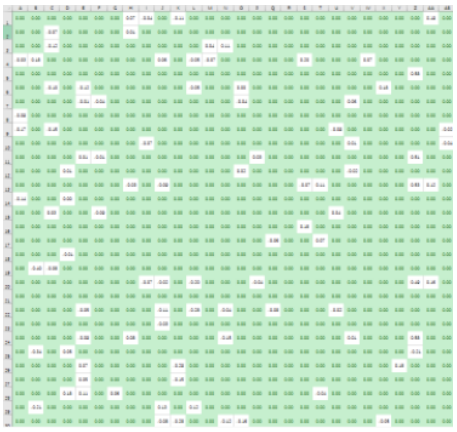

Figure 3: An illustration of the sparse matrix B.

These technologies have been widely used in many matrix multiplication acceleration studies [3, 4]. For the specific data in our task of Transformer matrix multiplication, we also included the data reuse and load balancing [4] as pre-treatments. All the algorithms of blocking, loop interchange, and shared memory are not new so we do not show algorithms here. In our study, we implemented our own algorithms of data reuse in Algorithm 1 for the dense matrix A since there are many repeated rows, and load balancing in Algorithm 2 since they are not even.

Algorithm 1: Data Reuse Algorithm for Dense Matrix A

```
Input: A=dense matrix A in m*k, B=sparse matrix B in k*k
Output: C=dense matrix in m*k
initialize: C in m*k
A_LastRow = A[−1, :]
for i = 0 to m do
    if A[i, :] == A_LastRow then
        A_Unique = A[: i, :]
        Exit
    end
end
C_Unique = A_Unique x B
for j = 0 to m do
    if j <= i then
        C[j,:] = C_Unique[j, :]
        j>i C[j,:] = C_Unique[i, :]
    end
end
```

5 RESULTS

With the knowledge of the data structure and our intuitions, we code the program in C++ and calculate the run time of each experiments. For all experiments, we run a total of 5 times and calculate the average and standard deviation of run time from the 5 times in order to reduce the randomness.

Algorithm 2: Load Balancing Algorithm for Sparse Matrix B

Input: A=dense matrix A in m*k, B=sparse matrix B in k*k
Output: C=dense matrix in m*k
initialize: C in m*k, $B_{LoadBalancing}$ in k*k
Calculate sparsity level of each column in B
Sort the sparsity level list of B as B_{List}
for $i = 0$ *to* $k/2$ **do**
  $B_{LoadBalancing}[:, i*2] = B[:, B_{List}[i]]$
  $B_{LoadBalancing}[:, i*2+1] = B[:, B_{List}[-i]]$
  Exit
end
$C_{LoadBalancing}$ = A x $B_{LoadBalancing}$
C = Reverse the columns of $C_{LoadBalancing}$ based on B_{List}

Table 1: 1D array and 2D array

	1D array	2D array
average	.07982	.08556
standard deviation	.00311	.00256

5.1 1D array V.S. 2D array

With our intuition on the memory location and usage in data storage, we tried the run time between 1D array and 2D array. According to the Table 1, the run time of 1D array is about 6% shorter than the run time of 2D array, which proves our intuition mentioned in the previous section.

5.2 Loop Interchange

Since we consider about the loop interchange, we first do an analysis on the loop interchange shown in Table 2. As you can see, the initial loop with order i, j, k has a maximum number of N^3 cash missing. However, by changing the order into i, k, j, we can see the maximum number of cash missing decreased to $\frac{N^3}{B}$. Therefore, we run an experiment with loop order i, k, j and compared the result with the original loop order i, j, k to see whether our intuition is correct or not. Table 3 proves our intuition. We can see a huge improvement with loop interchange. For 1D array, the loop interchange speeds up the run time by approximately 66%. Regarding to 2D array, the loop interchange boost the speed to about 68%. With Table 3, we can also observe the trend between 1D array and 2D array. 1D loop interchange result is 1% faster than 2D loop interchange result on average, although the difference is not very obvious.

5.3 Blocking

With the knowledge from class, we simply apply blocking on the matrix multiplication. Table 4 illustrates the results of blocking in 1D array and 2D array. For 1D array, a 71% of increasing in run time performance can be observed.Similarly, with 2D array, the performance improved by 72%. The trend that 1D array operates faster than 2D array can also be observed in blocking.

Table 2: Loop Interchange Analysis

	A	B	C
i	N	N	N
j	N	N	$\frac{N}{B}$
k	$\frac{N}{B}$	N	1
	$\frac{N^3}{B}$	N^3	$\frac{N^2}{B}$

	A	B	C
i	N	N	N
k	$\frac{N}{B}$	N	N
j	1	$\frac{N}{B}$	$\frac{N}{B}$
	$\frac{N^2}{B}$	$\frac{N^3}{B}$	$\frac{N^3}{B}$

	A	B	C
k	N	N	N
i	N	N	N
j	1	$\frac{N}{B}$	$\frac{N}{B}$
	N^2	$\frac{N^3}{B}$	$\frac{N^3}{B}$

	A	B	C
k	N	N	N
j	N	$\frac{N}{B}$	N
i	N	1	N
	N^3	$\frac{N^2}{B}$	N^3

	A	B	C
j	N	N	N
i	N	N	N
k	$\frac{N}{B}$	N	1
	$\frac{N^3}{B}$	N^3	N^2

	A	B	C
j	N	N	N
k	N	N	N
i	N	1	N
	N^3	N^2	N^3

Table 3: Loop Interchange Results

	1D array	1D loop	2D array	2D loop
average	.07982	.02739	.08556	.02776
standard deviation	.00311	.00007	.00256	.00005

Table 4: Blocking Results

	1D array	1D blocking	2D array	2D blocking
average	.07982	.02276	.08556	.02353
standard deviation	.00311	.00200	.00256	.00014

Table 5: CSR Results

	1D array	1D CSR	2D array	2D CSR
average	.07982	.00379	.08556	.00363
standard deviation	.00311	.00001	.00256	.00000

5.4 Data Structure

Since our problem is focused on sparse matrix, different data structures may cause different usage of memory storage. Therefore, We tried compressed sparse row (CSR) as an example. As you can see in Table 5, the run time is shortened by a factor of almost 10x in both the 1D and 2D cases. However, the trend discovered between 1D and 2D arrays is not consistent here. 1D CSR proceeds slightly slower than 2D CSR. We consider this issue is caused by the huge improvement of CSR. The numerous improvements through CSR data reformat offsets the benefit of data storage. The multiplication in index computing weighs more in the run time calculation. For example, if we would like to access the data in row 2 column 3, then we need to calculate 2 * (of columns) + 2 for the index in order to access the data in 1D array. And this multiplication takes time. The time of multiplication may not be long compared to 0.02s but may seem long compared to 0.002s. This is just one guess for the cause of the issue.

Table 6: Shared Memory Result

	Shared Memory
average	.00070
standard deviation	.00002

Table 7: Data Reuse Results

	1D array	1D reuse	Shared memory	Shared & reuse
avg	.07982	.03616	.00070	.00063
std	.00311	.00436	.00002	.00003

Table 8: Load Balancing Result

	Shared Memory	Shared & Balancing
average	.00070	.00066
standard deviation	.00002	.00002

5.5 Shared Memory in GPU

After various attempts with multiple CPU methods, we now consider some GPU methods. Due to the time limitation, we directly implemented a shared memory GPU implementation with multi-threading on CUDA, which should be the best implement currently. Table 6 shows the results for the shared memory implementation. By comparing the results in Table 6 and the best results in Table 5, we can still see a 10x improvement.

5.6 Data Reuse

As mentioned in the previous section, the last few lines remain the same because of the source that it comes from. Therefore, we only calculate lines with different values and the one that repeats several times. Then we copy-paste the result from that one line and use it for all other lines that have the same value, both on CPU and GPU. From Table 7, we can see that with about 60% repeated values in the dense matrix, we can see a 2x improvement in the run time performance for 1D CPU data, while, for GPU shared memory, the improvement is only 10%. However, the improvement is still considerable.

5.7 Load Balancing

In load balancing, we reformat the sparse matrix in advance to make each block perform approximately the same amount of work. By combining the line with max none-zero elements and the line with least none-zero elements, we obtain a relatively similar amount of work in each block. Then we load the data with the calculation of shared memory. Table 8 shows the improvements of load balancing. Here there is a slight improvement in run time.

5.8 Summarizing

Since load balancing and data reuse are forms of data prepossessing, we applied these two methods together to the algorithm with best performance, shared memory. Table 9 shows our final result for run time optimizations.

Table 9: Summary

	Shared & Balancing & reuse
average	.00062
standard deviation	.00003

6 FUTURE WORK

Due to the time limitation, there are still many ideas that we failed to implement in this paper.

- Blocking with CSR data structure
- Loop interchange with CSR data structure
- Pure Multi-threading such as OpenMP or pthread API

We hope to implement these in the future with more time and resources.

REFERENCES

[1] jgcbrouns. 2019. Introducing-Sparsity-in-the-Transformer. *github* (2019). https://github.com/jgcbrouns/Introducing-Sparsity-in-the-Transformer.
[2] Alec Radford Ilya Sutskever Rewon Child, Scott Gray. 2019. Generating Long Sequences with Sparse Transformers. *arXiv* (2019). arxiv.org/abs/1904.10509.
[3] Kaya K. Saule, E. and Ü. V. Çatalyürek. 2013. Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. *International Conference on Parallel Processing and Applied Mathematics* (2013). arxiv.org/pdf/1302.1078.pdf.
[4] Ziheng Wang. 2020. SparseRT: Accelerating Unstructured Sparsity on GPUs for Deep Learning Inference. *arXiv* (2020). arxiv.org/abs/2008.11849.
[5] He Zhang and Vishal Patel. 2016. Sparse Representation-based Open Set Recognition. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE,* (2016). https://arxiv.org/pdf/1705.02431.pdf.