Exploring sparse inference with SuiteSparse:GraphBLAS

Deepak Suresh
Department of Computer Science
Texas A&M University
College Station, TX

Tim Davis
Department of Computer Science
Texas A&M University
College Station, TX

***Abstract*—As AI models grow in size and complexity the resource demands are also growing. Sparse inference is an approach to take advantage of sparsity in large AI models to make them run faster thereby lowering the resource requirements. SuiteSparse:GraphBLAS, an open-source implementation of GraphBLAS, offers a robust framework tailored for leveraging sparsity in matrix computations. This work explores the application of SuiteSparse:GraphBLAS in performing inference tasks, particularly focusing on the language model BERT. By showcasing the capabilities of SuiteSparse:GraphBLAS in handling sparse computations and explaining its underlying formulation using semirings, this explores its application in improving inference efficiency. This work implements a complete inference pipeline using SuiteSparse:GraphBLAS, compares its performance with traditional frameworks like PyTorch, and identifies areas for improvement. Through this investigation, the study aims to highlight the strengths and limitations of SuiteSparse:GraphBLAS in AI computations.**

***Index Terms*—Machine Learning, sparsity**

I. Introduction

Over the past few years, AI models have significantly increased in size, with some large language models now containing trillions of parameters. This growth has led to higher compute requirements and increased power consumption, posing a significant challenge for AI infrastructure. Sparse inference has emerged as a solution to address the challenges posed by the increasing size of AI models. Sparsity in deep learning refers to designing models that primarily utilize a select few important features or neurons, while ignoring the less important ones. This concept reflects how the human brain functions, as it does not activate all neurons simultaneously. Rather, it employs a sparse array of neurons tailored to the task at hand. Sparsity helps prevent overfitting and concentrates on essential features. Models that are sparse tend to be simpler to interpret since they utilize fewer, but more important, features. By using fewer resources, sparse models are more memory and computation efficient, which also makes them more eco-friendly. By leveraging sparsity, sparse inference can outperform dense inference by skipping unnecessary computations, leading to faster inference times. Additionally, the lower power and resource requirements make sparse inference particularly well- suited for edge inference scenarios. In addressing the need for efficient sparse inference, the use of SuiteSparse:GraphBLAS, an open-source implementation of GraphBLAS, has been explored. This framework is specifically designed to leverage sparsity in matrix computations. By demonstrating the inference of a language model in suitesparse, we aim to showcase its capabilities and explain the underlying formulation of inference in language models using semirings. Furthermore, this work aims to shed light on the specific features and capabilities that SuiteSparse should possess in order to better support AI processes. By identifying and addressing these requirements, SuiteSparse can become a more effective and efficient tool for leveraging sparsity in AI computations.

This work was supported by funding from JuliaHub and the National Science Foundation. Corresponding author e-mail: deepaksuresh@tamu.edu

II. Related work

As transformer based language models have grown from millions of parameters, e.g., BERT-Large [1], to billions of parameters as in Megatron-Turing [2], there has been growing interest in sparse inference to improve inference efficiency. Consider deploying a LLaMA-2-70B model with 70 billion parameters. If the weights are stored in FP16 it will take 140GB of VRAM requiring 2 NVIDIA A100s and it will take about 100ms to generate a token. This necessitates optimizing either the model or the inference engine for faster and efficient inference. Broadly, the optimizations can be divided into optimizing a)The model and b)The inference engine. Optimizing the model involves techniques like sparsifying the activations, and sparsifying the model through weight pruning and quantization. Sparsity is generally divided into two categories: unstructured and structured sparsity. Unstructured sparsity in neural networks involves randomly pruning individual weights, typically removing those with smaller magnitudes first. This method offers flexibility and a significant reduction in the number of model parameters, which can decrease storage needs. However, the major drawback is that standard hardware and software are not optimized for handling such randomly sparse matrices, causing inefficient memory access and slower computation speeds. Structured sparsity in neural networks involves systematically pruning entire rows, columns, filters, or layers, maintaining a regular structure that aligns well with standard hardware for efficient processing. This method predictably reduces the computational workload and memory usage but can also result in a greater loss of network performance due to the significant reduction of network capacity. The systematic nature of structured sparsity makes it compatible with typical hardware architectures, facilitating easier implementation and integration. Our work exploits unstructured sparsity that results from weight pruning. We consider Mixture of Experts(MoE) [12] to be a case of dynamic sparsity or sparsity in activations. MoEselects different parameters for each incoming example, resulting in a sparsely activated model with a large number of parameters. This approach allows for more flexibility and specialization in handling different inputs. However, MoE models have faced challenges related to complexity, communication costs, and training instability. Switch Transformers [8] were proposed as a solution to the challenges faced by MoE models. They simplify the MoE routing algorithm and design improved models with reduced communication and computational costs. We consider these as dynamic networks that introduce sparsity in activations. Another popular approach is weight pruning. Weight pruning drops weights that are not important while maintaining the model's accuracy. Pruning has been proven to be an effective way of reducing model size while maintaining the similar model quality. Techniques like movement pruning [9] adaptively prunes weight during the fine tuning process. Blockwise pruning [10] extends movement pruning by considering blocks of any size and allows the pruning of weights to a predefined sparsity pattern. The second approach is to optimize the inference engine. Our work falls in this realm. Intel® Deep Learning Boost [3] is a software accelerator for sparse matrix - dense matrix multiplication (commonly abbreviated as SpMM) on CPUs. The SpMM kernel outperforms sparse libraries like oneMKL, TVM, and LIBXSMM. It has also been shown to have 1.5x speedup over NeuralMagic's DeepSparse. SparseDNN [11] is a sparse deep learning inference engine that targets CPUs to accelerate pruned neural networks. GraphBLAS [4] is an open source general purpose package for sparse computations. As opposed to other works, we are not creating a software runtime catered to inference, rather using an existing general purpose framework for inference. This work has been done in python-graphblas [13] which is a Pythonic interface to SuiteSparse:GraphBLAS [5], a comprehensive implementation of the GraphBLAS standard. This standard specifies a range of sparse matrix operations across an extended algebra of semirings, accommodating a vast array of operators and types. GraphBLAS is an OpenMP-based implementation that is designed to offer users a set of objects along with their methods and operations, enabling them to represent graph algorithms through the use of linear algebraic operations on sparse adjacency matrices across various semirings.

III. Inference with SuiteSparse:GraphBLAS

In this section we cover the formulation of layers in the language of semirings. Before neural network layers can be implemented using the SuiteSparse:GraphBLAS, the layers have to be formulated in terms of semirings. This reformulation is necessary since SuiteSparse relies on semirings for expressiveness as well as to perform its optimized graph and matrix operations efficiently.

Dense layer: When an input is fed into a dense layer, each neuron in the layer performs a weighted sum of the inputs. Mathematically, this can be described by the equation

$$y = Wx + b$$

, where x is the input vector W is the weight matrix, b is the bias vector, and y is the output vector of the layer. This means that the input vector is multiplied by W and a bias vector is added to each row of the result. The matrix multiplication is the standard plus_times semiring. The addition of bias utilizes a clever insight derived from how matrix operations work with diagonal matrices. Specifically, when an n-by-n matrix A is multiplied on the left by a diagonal matrix diag(a_1, ..., a_n), each row i of matrix A is scaled by the corresponding ai. Conversely, multiplying matrix A on the right by diag(a_1, ..., a_n) scales each column i by ai. This manipulation alters the scale and may also alter the shape of matrix A, whereas uniform scaling of the matrix occurs only when it is multiplied by a scalar matrix. The bias vector is transformed into a diagonal matrix and addition is performed with the plus_plus semiring.

Algorithm 1 Dense layer

```
// Layer initialization
1. load weight as GrB_Matrix
2. load bias as GrB_Vector
3. convert bias vector to diagonal matrix

// Forward pass
1. initialize output GrB_Matrix
2. multiply weight and input with plus_times semiring
3. add bias with plus_plus semiring
```

Layer norm: Layer normalization is a technique employed in neural networks to normalize the activations of each neuron across a layer. Its primary purpose is to stabilize the distribution of values within each layer, thereby enhancing training efficiency and generalization. The operation involves computing the mean and standard deviation of the activations across each training example and applying normalization independently for each neuron, typically by subtracting the mean and dividing by the standard deviation. By ensuring consistent activation distributions across neurons, layer normalization mitigates the problem of internal covariate shift and facilitates smoother gradient flow during backpropagation. When performing matrix-vector multiplication between a matrix X with dimensions NxN and a dense vector with all entries initialized to 1/N, the result yields the mean per row of matrix X. This operation corresponds to the plus_times semiring on matrix-vector operations. Computing the variance follows a similar approach to calculating means.

Softmax: The softmax layer is utilized to transform a vector of arbitrary real-valued scores into a probability distribution over multiple classes or categories. It applies the softmax

Algorithm 2 Layer normalization

```
// input matrix is X
$v \leftarrow 1/N$ {N is the number of columns in input matrix}
$w \leftarrow X \cdot v$ {where $\cdot$ is matrix vector multiplication}
// $w$ is now a vector that holds mean of each row
$\mu \leftarrow diag(w)$ {w is converted to a diagonal matrix}
// next step subtracts mean from each row of input
$X\_ \leftarrow X \odot \mu$ {$\odot$ is the plus_minus semiring}
// next 2 steps to calculate variance
$\mu\_2 \leftarrow$X_ $* X\_$
$\sigma\_2 \leftarrow \mu\_2 \cdot v$ {where $\cdot$ is matrix vector multiplication}
$\sigma\_2 \leftarrow \mu\_2 \cdot v$
$\sigma \leftarrow \sqrt{\sigma\_2}$
$X\_ \leftarrow X\_ \div \sigma$
// at this point input is mean normalized
// X_ is then passed to dense layer with layer norm weight
and layer norm bias
```

function to each element of the input vector. Mathematically, the softmax function is defined as: where represents the ith element of the input vector, and N is the total number of elements. The function exponentiates each score and divides it by the sum of all exponentiated scores, ensuring that the resulting values are between 0 and 1 and sum up to 1, forming a probability distribution. For numerical stability the max value is subtracted from all entries. reduce_rowwise method can take the "max" keyword to find max value per row. Similar to the diagonal trick in bias addition, we convert the max values to a diagonal matrix and use plus_minus semiring to subtract max values. Element wise exponentiation is performed with unary.exp. For the denominator, we perform unary exponentiation followed by reduce_rowwise with the plus keyword.

Algorithm 3 Softmax

```
// input $X$ is attention score
$maxes \leftarrow max\_reduce\_rowwise(X)$ {Reduce rowwise to
return max in each row}
$max\_mat \leftarrow diag(maxes)$ {maxes to diagonal matrix}
$X\_ \leftarrow max\_mat \odot X$ {$\odot$ is the plus_minus semiring}
$X\_exp \leftarrow exp(X\_)$ {unary exponentiation}
$sums \leftarrow sum\_reduce\_rowwise(X\_exp)$ {Reduce row-
wise to returns sum of each row}
$sum\_mat \leftarrow diag(sums)$ {sum to diagonal matrix}
$X\_probs \leftarrow X\_exp \div sum\_mat$ {$\div$ is the plus_divide
semiring}
```

Self-attention: Query, Key, and Values of tokens are the result of passing the tokens through the corresponding weights. This is performed using dense layers which were defined earlier. For example, the tokens are passed through a dense layer which has the Query weights and biases. The operation for Key and Value follows the same approach. Attention scores are the result of matrix multiplication between Key and Query outputs, which is a plus_times semiring. This is then passed through softmax to create attention weights. Another plus_times semiring on Value output and attention weights is the final output of the self attention layer. Frameworks like PyTorch, TensorFlow etc works with tensors, however SuiteSparse:GraphBLAS supports only matrices. This imposes a limitation on what kind of operations can be performed and how they are implemented. Processing a batch of inputs is not considered in this work. Batch processing is inherently harder because the input starts off as a tensor. To keep the complexity in check we decided to stick with processing one query at a time. Attention layer still has tensor operations in the calculation of attention weights. This is circumvented using a block diagonal trick. Tensor multiplication can be expressed as block diagonal matrix multiplication under certain conditions. Let A and B be tensors of shapes $(n_1, n_2, .. n_m)$ and $(m_1, m_2, .. m_n)$ respectively. They can be represented as block diagonal matrices where A_i and B_j are matrices of size (n_i, n_{i+1}) and (m_j, m_{j+1}). Let C be the result of the tensor

$$\mathbf{A} = \begin{bmatrix} A_{(1)} & 0 & \cdots & 0 \\ 0 & A_{(2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & A_{(m)} \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} B_{(1)} & 0 & \cdots & 0 \\ 0 & B_{(2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & B_{(n)} \end{bmatrix}$$

Fig. 1. Tensors as block diagonal matrices

multiplication. Then $C = A * B$ can be expressed as block diagonal matrix multiplication where the ith block of C is the matrix multiplication of $A(i)$ and $B(i)$. In the same vein if L were the sequence length of the input the query and key matrices can be represented in the block diagonal form as given on the right. Then, can attention scores matrix S can be computed as

$$S = Q \cdot K^T$$

Here, the block diagonal multiplication implicitly computes the dot products between each query vector Q_i and each key vector K_i resulting in the attention scores matrix S. This formulation allows us to compute attention scores efficiently leveraging matrix multiplication operations.

These are essentially the basic constructs required to build an inference pipeline for BERT.

IV. EXPERIMENTAL SETUP

In this section we describe the experimental setup including hardware settings, sparse model, and framework.

A. Hardware Settings

The experiments were conducted on CPU instance with Intel(R) Xeon(R) CPU E5-2698 v4 @ 2.20GHz. It has 40 cores and 250GB memory. Both SuiteSparse:GraphBLAS and the reference implementation are tested on this device.

B. Sparse Model

To demonstrate the utilization of SuiteSparse:GraphBLAS as a backend for inference, a sparse model was needed. The Bert model from ”Prune Once for All” [6] was deemed ideal. This approach introduces a novel method for training sparse pre-trained Transformer language models by integrating weight pruning and model distillation. These sparse pre-trained models retain their sparsity pattern and can be effectively applied for transfer learning across various tasks. Starting with a BERT-base model that has 110 million parameters, it is pruned to achieve 90% sparsity, rendering it an optimal choice to showcase the capabilities of SuiteSparse:GraphBLAS.

C. Framework

We wanted to run the core inference in SuiteSparse:GraphBLAS and have a way to perform on-to-one comparison with an established approach. Our solution was to use Huggingface transformers library as the reference implementation and swap out pytorch-based inference pipeline with a SuiteSparse:GraphBLAS based one. This gave us an easy way to compare both implementations quantitatively and qualitatively. We start with the transformer implementation of BERT inference and replicate that in SuiteSparse:GraphBLAS. The input preprocessing and output postprocessing are left untouched, focusing solely on the core inference process without altering the surrounding setup.

D. Task

BERT was trained on two major text collections: Wikipedia, which includes approximately 2.5 billion words, and Google’s BooksCorpus, with around 800 million words. These extensive datasets have enabled BERT to develop a deep understanding of the English language. Additionally, they have provided BERT with a broad knowledge of the world. Given this we decided to choose masked language modelling as the particular task our model would perform. In this task the model is provided with a sequence of tokens with a masked token. The model is expected to use its knowledge to fill in the masked token. In the input sequence $Paris\,is\,the\,capital\,of\;[MASK]$.. The model is expected to replace $[MASK]$ token with $France$ in the output.

V. Results

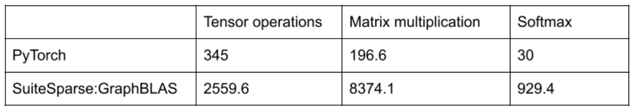
A. Inference pipeline

We replicate the transformers BERT inference pipeline in SuiteSparse:GraphBLAS, and a standalone demo[1] of our engine. The demo also links to the source code for the project. The output of the demo proves that our engine produces the same results as the reference engine.

[1]https://colab.research.google.com/drive/13YwlILu4FNA2aXbTF86T991XAZYorsBq

VI. Performance comparison

SuiteSparse:GraphBLAS takes a total of 0.42s whereas PyTorch takes 0.02s to generate the output. This means PyTorch is 20 times faster. This slowdown can be traced to the attention layer. The operations in attention layer can be divided into three. The first kind is matrix operations that happen when tokens are passed through Query, Key, and Value layers.



	Tensor operations	Matrix multiplication	Softmax
PyTorch	345	196.6	30
SuiteSparse:GraphBLAS	2559.6	8374.1	929.4

Fig. 2. Time spent in attention layer (in microseconds)

The second is softmax operation to calculate attention weights. The third is tensor operations to get raw attention scores. This categorization is based on the distribution of time spent in the SuiteSparse:GraphBLAS. Figure 2 shows a breakdown of inference time in both frameworks. The matrices in this BERT model are of size $(9, 768)$ where 9 is the sequence length and 768 is the input dimension. As described previously, the block diagonal formulation is used for tensor operations. There is a significant overhead to setting up the tensor as a block diagonal matrix. Furthermore, the matrices aren’t large enough and overheads will dominate. The calculation of attention weights with softmax is similar to tensor operations. Given this BERT model has 12 attention heads the attention scores are of size $(12, 9, 9)$. In block diagonal form this becomes a $(108, 108)$ matrix which is not large enough to exploit sparsity. The calculation of attention scores and softmax operations are not areas where SuiteSparse:GraphBLAS excels due to the lack of tensor support. There’s little that can be done to address slowdown in tensor operations. An area where SuiteSparse:GraphBLAS can outperform PyTorch is in matrix multiplications, which are found in both the attention and dense layers. Therefore, we shift our focus to matrix multiplication. The dense layer multiplies the input by a weight matrix and adds a bias. We isolate the matrix multiplication, which involves the input (dense matrix) and the weight (sparse matrix), and conduct experiments to highlight conditions where SuiteSparse can perform this multiplication faster. The matrices we consider and synthetic equivalents of the ones found in the sparse BERT used in the demo. We create weight matrices with similar sparsity patterns and evaluate input matrices under both sparse and dense conditions. Various matrix sizes and sparsities are taken into account to replicate the dense layer in larger and/or sparser models. This series of experiments help determine the scenarios in which SuiteSparse:GraphBLAS outperforms PyTorch. We consider three cases and in each case we perform matrix multiplication to compare the performance of PyTorch(using torch.sparse) and SuiteSparse:GraphBLAS on sparse-sparse and sparse-dense matrix multiplication.

A. Sparse-sparse matrix multiplication on synthetic matrices with constant sparsity

We consider matrices of different sizes and perform square matrix multiplication(sparse-sparse) for each size. The size of the matrix is varied from 10 thousand to 10 million while keeping sparsity constant. This is to mimic the matrix multiplication in a dense layer of the neural network. At every stage the matrix is 90% sparse i.e. only 10% of the entries are non-zero. The operation that is timed is sparse-sparse matrix multiplication. There is a trend that as matrix size increases we gain more performance. One million seems to be that threshold where the performance gains are much higher than the overheads of launching multiple threads. This demonstrates that the larger the matrix, the greater the benefit from using SuiteSparse:GraphBLAS. The error bars show 95% confidence interval.

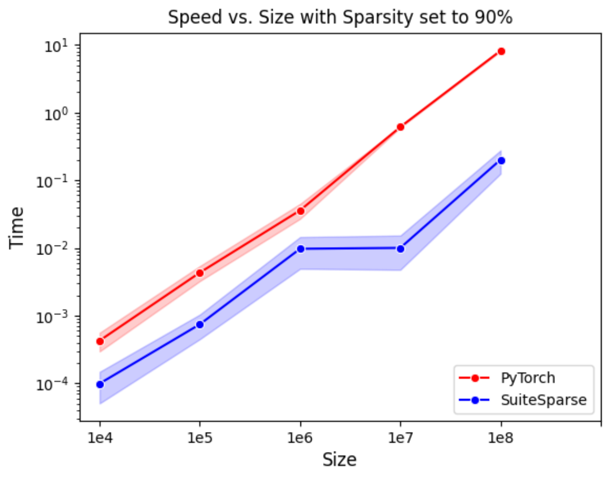


Fig. 3. Sparse-sparse matrix multiplication on synthetic matrices(constant sparsity)

B. Sparse-sparse matrix multiplication on synthetic matrices with constant size

In this experiment we vary sparsity percentage from 85% to 99%. This is again a case of sparse-sparse multiplication. We see a similar trend as in the previous section. In case of a neural network this is the case where input to the network and weights are sparse. Networks with dynamic sparsity and activations like ReLU can result in cases where input matrix is also sparse.

C. Dense-sparse matrix multiplication on synthetic matrices

In this case we look at sparse-dense matrix multiplication. This is closest to the BERT model that we are using in this study. The weight matrix is sparse and the input matrix is dense. This is the common form of sparsity setting that is found in neural networks. We treat the input matrix as a function of the sequence length. Sequence length or context window refers to number of tokens in the input. A token is

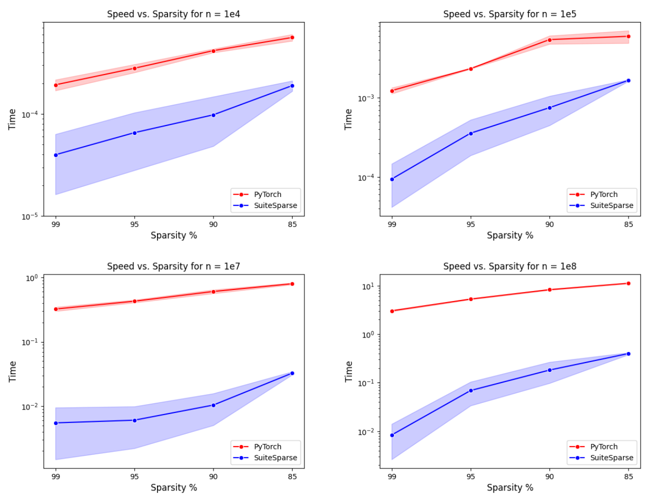


Fig. 4. Sparse-sparse matrix multiplication on synthetic matrices(constant size)

usually 2 to 3 characters. Some of the largest models like mixtral [?] has a context window or sequence length 32000. In this case the input matrix is dense and the weight matrix is sparse. SuiteSparse:GraphBLAS demonstrates superior speed even when one of the inputs is dense, particularly with sufficiently large matrices, as indicated by the graphs. Our sequence length in the demo is only 9 and this explains why even dense layers in the SuiteSparse:GraphBLAS inference engine is slower than PyTorch. It's only around sequence length of 10 thousand that we see performing better. Hence models with large context window like Mixtral [7] that has a context window of 32,000 are better suited for SuiteSparse:GraphBLAS.

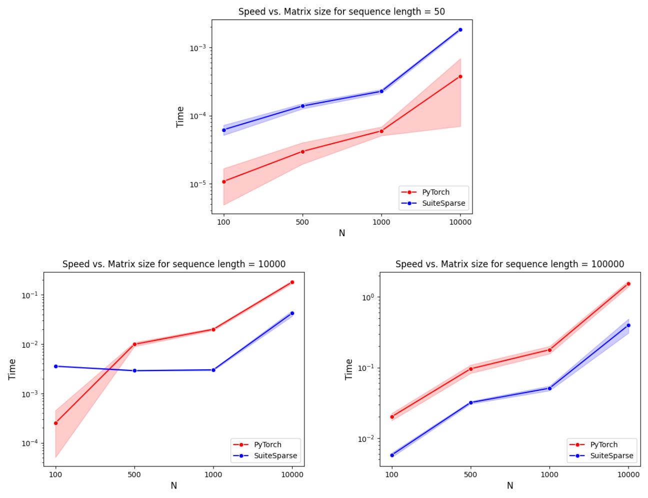


Fig. 5. Dense-sparse matrix multiplication on synthetic matrices(constant sequence length)

In light of these results we can conclude that SuiteSparse:GraphBLAS will shine if the model is large and sparse enough. This is obvious in the scenario where model(or weight matrix) and input are sparse. We have also demonstrated that even when the input is dense, SuiteSparse:GraphBLAS will be faster if the weight matrix is large and sparse.

D. Operator fusion

One of the most expressive features in SuiteSparse:GraphBLAS is the ability to fuse operators. In a transformer the feedforward layer is where the non-linearity is applied. We need non-linearity in a network so that it can learn interesting patterns. The non-linearity in this context is called is GeLU and uses the erf function. A straightforward implementation of GeLU has 5 calls and with it comes the cost of data movement [14]. In an attempt to speed up the feedforward layer we defined a custom semiring with GeLU baked into it as the multiplicative monoid.

$$Y = W * f(X)$$

where f is GeLU. The new multiplicative monoid is GeLU followed by the conventional times operator. This fuses application of GeLU with weight multiplication. A fused operator on its own would be faster but in this case since GeLU is in the semiring it gets coupled with the W matrix. The W matrix is also accessed every time GeLU is performed and GeLU itself is expensive due to the erf function in it. Moreover, W is not large enough or sparse enough to pay off in terms of speed. So we did not see a speed up from operator fusion. In general terms if $T = f(X)$ where f is a non-linearity the complexity will be $O(n)$ where n is the number of entries in X. But in this case, $Y = W * f(X)$ the complexity becomes $O(|W| + |X|)$ and W is not sparse enough in this BERT model for fusion to pay off.

VII. Summary and Future Work

We have been able to demonstrate that SuiteSparse:GraphBLAS can be used to build inference engines for AI. It can be even more suited with support for tensor operattions. PyTorch offers extensive tensor support and this makes a lot of computations easier and faster. It also has all layers and abstractions like dense layer, layer normalizartion, dropout available and that means its easier to build inference pipelines. There are plenty of reference implementations available thanks to the large community of PyTorch users. In terms of expressiveness SuiteSparse:GraphBLAS will let you fuse operations to form custom kernels.These fused unary operations can be used to form custom semirings. This is useful because it can help you reduce memory traffic. Operations like bias addition are performed using broadcasting in PyTorch. This can be expressed as a plus_plus semiring in SuiteSparse:GraphBLAS, that is it offers a mathematical expressivity that closely aligns with formal mathematical concepts. In terms of speed, Pytorch is 20 times faster than SuiteSparse:GraphBLAS, the BERT model that was used to test the inference engine uses small matrices and the sequence length is not large enough. As demonstrated in the results section this is not the ideal scenario for SuiteSparse:GraphBLAS to shine. Another bottleneck was the attention layer which has tensor multiplications. So BERT has 12 encoders and each encoder has an attention block. Each attention block has 2 tensor multiplications which is a dense-dense tensor multiplication In Pytorch the tensor multiplication takes 625 microseconds, the same is performed as block diagonal matrix multiplication in SuiteSparse:GraphBLAS and it takes about 4000 microseconds. But setting up the tensors as block diagonal matrices is expensive and the total time for attention block in SuiteSparse:GraphBLAS is 140 milliseconds opposed to 7 milliseconds in Pytorch. So that subsumes everything else. The dense layers and matrix multiplication as we have seen should be faster on larger models with long context window A complete solution to the performance problem would be to add tensor support but that is a tall order. Or better use both SuiteSparse:GraphBLAS and a tensor library like Pytorch so that each can shine and we get the best of both worlds.

References

[1] J. Devlin, M. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” arXiv, 2019.

[2] S. Smith, M. Patwary, B. Norick, P. LeGresley, S. Rajbhandari, J. Casper, Z. Liu, S. Prabhumoye, G. Zerveas, V. Korthikanti, E. Zhang, R. Child, R. Yazdani Aminabadi, J. Bernauer, X. Song, M. Shoeybi, Y. He, M. Houston, S. Tiwary, and B. Catanzaro, “Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model,” arXiv, 2022.

[3] H. Shen, H. Meng, B. Dong, Z. Wang, O. Zafrir, Y. Ding, Y. Luo, H. Chang, Q. Gao, Z. Wang, G. Boudoukh, and M. Wasserblat, “An Efficient Sparse Inference Software Accelerator for Transformer-based Language Models on CPUs,” arXiv, 2023.

[4] B. Brock, A. Buluç, T. Mattson, S. McMillan, and J. Moreira, “The GraphBLAS C API Specification (v2.0),” Tech. Rep., 2021.

[5] T. A. Davis, “Algorithm 1037: SuiteSparse:GraphBLAS: Parallel Graph Algorithms in the Language of Sparse Linear Algebra,” ACM Trans. Math. Softw., vol. 49, no. 3, Art. no. 28, pp. 1–30, Sep. 2023.

[6] O. Zafrir, A. Larey, G. Boudoukh, H. Shen, and M. Wasserblat, “Prune Once for All: Sparse Pre-Trained Language Models,” arXiv, 2021.

[7] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, “Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer,” arXiv, 2017.

[8] W. Fedus, B. Zoph, and N. Shazeer, “Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity,” arXiv, 2022.

[9] V. Sanh, T. Wolf, and A. Rush, “Movement Pruning: Adaptive Sparsity by Fine-Tuning,” in Advances in Neural Information Processing Systems, vol. 33, 2020, pp. 20378–20389.

[10] F. Lagunas, E. Charlaix, V. Sanh, and A. M. Rush, “Block Pruning For Faster Transformers,” arXiv, 2021.

[11] Z. Wang, “SparseDNN: Fast Sparse Deep Learning Inference on CPUs,” arXiv, 2021.

[12] Mistral, “Mixtral of experts: A high quality Sparse Mixture-of-Experts.,” 2023.

[13] E. Welch and J. Kitchen, “Python library for GraphBLAS: high-performance sparse linear algebra for scalable graph analytics.,” 2024. https://python-graphblas.readthedocs.io/en/stable/user_guide/fundamentals.html

[14] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, “Data Movement Is All You Need: A Case Study on Optimizing Transformers,” arXiv, 2021.