Attention Algorithm Implementation in MagmaDNN Framework

Wing-Lim Lau
Department of Mathematics
The Chinese University of Hong Kong
1155158234@link.cuhk.edu.hk

Xiao-Yang Li
Department of Computer Science
City University of Hong Kong
xyli45-c@my.cityu.edu.hk

Abstract—MagmaDNN is a neural network library in C++ aiming at optimizing towards heterogeneous architectures, i.e., multi-core CPUs and GPUs. Currently, no implementation of the multi-head attention layer, which is a core component of transformer models, is provided by MagmaDNN library, despite the popularity and significance of transformer models in various tasks including vision tasks such as medical segmentation [1], [2], image recognition [3], semantic segmentation [4], and natural language processing tasks such as machine translation [5].

To bridge the gap, we present an implementation of the multi-head attention layer in MagmaDNN framework. Our implementation improves the prediction loss by **20.41%** compared with Tensorflow implementation, despite consuming extra training time (epoch = 1000, learning rate = 10^{-3}, batch size = 8, input size = $[3 \times 8 \times 8]$). Compared with Pytorch implementation, our method also outperforms it by a clear margin in terms of prediction loss.

Index Terms—MagmaDNN, attention mechanism, transformer model

1 Introduction

Developed by Innovative Computing Laboratory, Matrix Algebra on GPU and Multi-core Architectures (MAGMA) [6], [7], [8] is a set of linear algebra libraries for heterogeneous computing. Thanks to its support for common linear algebra packages and standards such as LAPACK [9] and BLAS [10], MAGMA allows easy transfer of linear algebra-reliant software components to heterogeneous computing systems. Relying heavily on efficient and parallel linear algebra computations, deep learning is one of the many areas which largely benefit from MAGMA.

This naturally leads to MagmaDNN [11], a MAGMA-driven deep learning library which aims at providing a simple, modularized framework for deep learning accelerated for heterogeneous computing systems. Currently, several deep learning models including some CNN variants (UNet [12]) and RNN variants (LSTM [13]) have been supported by MagmaDNN. Nevertheless, the multi-head attention layer, despite being the guts of transformer models, has not yet been supported by MagmaDNN.

This paper presents an implementation of the multi-head attention layer, the key module of the increasingly popular transformer model [14] in the MagmaDNN framework.

This project was funded by the National Science Foundation, The Chinese University of Hong Kong and City University of Hong Kong.

In terms of training speed, our implementation is generally faster than PyTorch and TensorFlow implementation for 3000 epochs, given one single batch input of size $[3 \times 4 \times 4]$, $[3 \times 8 \times 8]$, $[3 \times 16 \times 16]$. When the input size is equal to $[3 \times 32 \times 32]$, our implementation is only mildly slower than TensorFlow and PyTorch implementations. In terms of prediction loss, our implementation is better for all types of input size in predicting zero tasks (epoch = 1000, learning rate = 10^{-3}, batch size = 8), except for inputs of size $[3 \times 4 \times 4]$, which is rare in real-world scenario. However, with more epochs, the prediction losses of PyTorch and TensorFlow multi-head attention layers show possibility of decreasing. It is worth noting that it takes many more epochs for our competitors to get close to our best epoch prediction loss.

2 Background

2.1 Multi-head Attention

Multi-head attention [14] is a variant of the attention mechanism used in deep learning models for natural language processing and other tasks. In this approach, the input is linearly transformed into multiple representations, each of which is processed by a separate attention mechanism in parallel. The outputs of these multiple attention mechanisms are then concatenated and passed through a linear layer to produce the final output.

The idea behind multi-head attention is to enable the model to attend to different parts of the input simultaneously, allowing it to capture more complex relationships between the input tokens. Each head of the attention mechanism can attend to a different aspect of the input. In natural language processing, for example, such aspects might be the positional information, syntax, or semantic content.

Multi-head attention has been applied in various deep learning models, including the transformer architecture used in the SOTA language models such as BERT [15]. In language models, the purpose of multi-head attention is to attend to the input sequence at different levels of abstraction, allowing the model to capture both local and global dependencies between the input tokens. Compared with other methods, multi-head attention has better interpretability. In the multi-head attention layer of language

Method	Complexity	Sequential Operation	Maximum Path Length
Self-Attention	$O(n^2d)$	$O(1)$	$O(1)$
Convolutional	$O(knd^2)$	$O(1)$	$O(\log_k(n))$
Recurrent	$O(nd^2)$	$O(n)$	$O(n)$

TABLE 1
Complexities, sequential operations and maximum path lengths of different methods, where n denotes sequence length, d denotes embedding dimension and k denotes the kernel size for convolutional layer. (Adapted from [14])

models, the self-attention mechanism is applied multiple times with different sets of parameters, which explains the reason why the model can capture both local dependencies between adjacent tokens and global dependencies between distant tokens.

2.2 Sequence Transduction

Let $X = (x_1, \ldots, x_n)$ and $Y = (y_1, \ldots, y_m)$, where $x_i, y_i \in \mathbb{R}^d$. The problem of mapping the sequence X of symbol representations to another sequence Y is called a sequence transduction, which can be seen as an analogy to a hidden layer in a typical sequence transduction encoder or decoder. Related works include convolutional layer and recurrent layer. In table 1, the three types of layers are compared in terms of layer complexity, sequential operation and maximum path length for $n = m$. Path length refers to the length of the paths forward and backward signals have to traverse in the network. Generally, as suggested by [16], the shorter the paths in X and Y, the easier it is to learn long-range dependencies.

With the assumption that the sequence length n is smaller than the representation dimension d, self attention has the advantage over convolutional and recurrent layers in all three aspects, as demonstrated in table 1. This assumption generally holds in most of the cases, as in, for instance, the sentence representations used by SOTA models in machine translations [17].

3 Methodology

In this section, we review the classic structure and formulation of the multi-head attention mechanism [14] and elaborate the implementation details of multi-head attention layer in the MagmaDNN framework.

3.1 Preliminaries

The classic attention mechanism can be thought of as matching a query against a set of keys associated with certain values, as in a retrieval process. Let $Q \in \mathbb{R}^{N_q \times d_q}$, $K \in \mathbb{R}^{N_k \times d_k}$ and $V \in \mathbb{R}^{N_v \times d_v}$ be the query, key and value matrices, each row of which contains one single query, key, or value, respectively. A (single-head) attention is computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\alpha}\right)V \tag{1}$$

where $\alpha \in \mathbb{R}$ is some scaling parameter (chosen as $\sqrt{d_k}$ in [14]). Here, we require $d_q = d_k$ and $N_k = N_v$ in order for Q, $K^\top$ and V to be compatible for matrix multiplication.

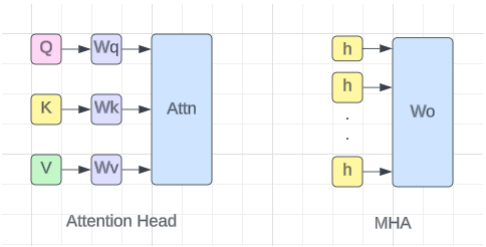

Fig. 1. Structure of multi-head attention, h denotes Attention Head module on the left hand side.

The output of an attention is then an $N_q \times d_v$ matrix, each row of which is a probabilistic linear combination of the values, representing the value obtained from matching the corresponding query against the keys. The (i, j)-entry in $QK^\top$ can be interpreted as a measure of the similarity between q_i and k_j.

In multi-head attention (MHA), we linearly project Q, K and V via learnable weights before passing into the attention function and projecting onto the output dimension. More precisely,

$$\text{MHA}(Q, K, V) = [h_1, \ldots, h_n]W^O \tag{2}$$

$$h_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3}$$

where $W_i^Q \in \mathbb{R}^{d_q \times d_{\text{proj}_q}}$, $W_i^K \in \mathbb{R}^{d_k \times d_{\text{proj}_k}}$, $W_i^V \in \mathbb{R}^{d_v \times d_{\text{proj}_v}}$, and $W^O \in \mathbb{R}^{nd_{\text{proj}_v} \times d_{\text{proj}_o}}$ are learnable weights. Similarly, we require $d_q = d_k$ and $N_k = N_v$ in order for the matrices to be compatible. Each matrix $h_i \in \mathbb{R}^{N_q \times d_{\text{proj}_v}}$ is called a head. The computation of h_i is performed in parallel, yielding d_v-dimension output. At the end, all heads are concatenated and projected onto a d_{proj_o}-dimensional space.

3.2 Implementation

The overview of multi-head attention implementation and the flowchart are demonstrated in the figure 2 and 3 respectively. We mainly use cuDNN APIs to perform the parallel computations. The APIs require that Q, K, V be 4-dimensional tensors, each dimension equal to either the beam size (`BEAM_DIM`), batch size (`BATCH_DIM`), number of sequences (`TIME_DIM`), or the size of each sequence (`VECT_DIM`). cuDNN allows tensors of 6 different shapes and we choose to use the shape (`BEAM`, `BATCH`, `TIME`, `VECT`). See [18] for details. Our multi-head attention layer requires that the inputs Q, K, V be concatenated into one single tensor to maintain consistency with other layers in MagmaDNN.

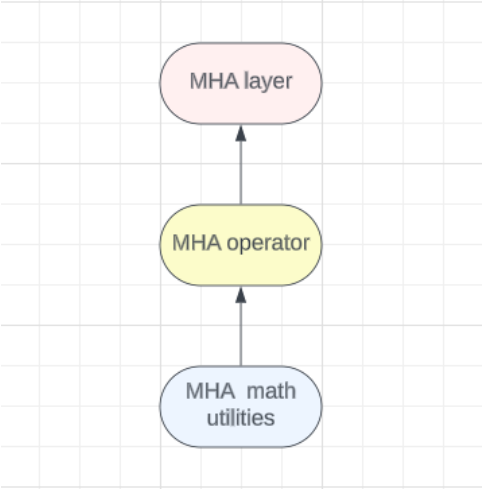

Fig. 2. Big picture of multi-head attention implementation.

(1) Initialization: Before the forward calculation, all options and configurations such as the number of attention heads, input dimensions, projection dimensions and dropout rate are initialized. According to the configurations, the memory space required is allocated and initialized. Meanwhile, since arrays are linearly allocated in the memory space, tensor descriptors are therefore needed to define the shapes and sizes of tensors.

We utilize Xavier initialization [19] to set up our initial weights for the projection, which is widely used by deep learning frameworks such as PyTorch. By Xavier initialization, biases are initialized as 0 and weights are initialized from the uniform distribution $W_{ij} \sim U[-\beta, \beta]$, where $\beta = \sqrt{\frac{6}{\text{fan}_{\text{in}}+\text{fan}_{\text{out}}}}$, in which fan_{in} and fan_{out} denote the size of previous layer and the size of the current layer, respectively. The purpose of the initialization is to initialize the weights such that the variance are stable across every layer, which prevents gradient explosion and vanishing.

(2) Forward pass: At the stage of forward pass, the concatenated input tensor is divided into three components, namely the Q, K, V tensors, via the function `separate_tensor()`. We pass Q, K, V along with the initialized weights, settings, and the placeholder output tensor to the function `mha_device()`, which then calls the cuDNN API `cudnnMultiHeadAttnForward()`.

(3) Backpropagation: The main trainable parameters of multi-head attention layer are the linear projection weights W_i^Q, W_i^K, W_i^V and W^O. According to the chain rule, the gradient of attention output with respect to projection weights is given by

$$\begin{aligned}\frac{\partial out}{\partial W} &= \frac{\partial out}{\partial[\hat{Q},\hat{K},\hat{V}]} \times \frac{\partial[\hat{Q},\hat{K},\hat{V}]}{\partial[W^Q,W^K,W^V]} \\ &= \frac{\partial out}{\partial[\hat{Q},\hat{K},\hat{V}]} \times \frac{\partial[\hat{Q},\hat{K},\hat{V}]}{\partial W},\end{aligned}$$

where $\hat{Q}$, $\hat{K}$ and $\hat{V}$ are the projected Q, K and V respectively. In our implementations, we introduce two functions, `mha_grad_data_device()` and `mha_grad_data_device_weights()`, which call the cuDNN functions for computing gradients with respect to data and weights, namely the function `cudnnMultiHeadAttnBackwardData()` and the function `cudnnMultiHeadAttnBackwardWeights()`, to calculate the two terms above simultaneously.

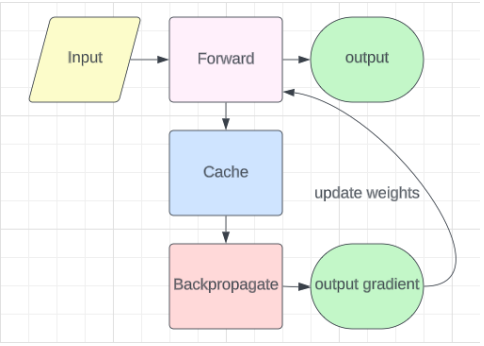

Fig. 3. Multi-head attention flowchart.

(4) Cleanup: The cleanup process, which is omitted in the flow chart, is to delete relevant pointers and free memory allocated for tensors and tensor descriptors. This step is exceedingly important in real practice in improving time efficiency in training mode. To achieve this, we mainly use cuDNN functions with the prefix `cudnnDestroy`.

4 Experiment

4.1 Training Setup

All the experiments are conducted on one single GPU NVIDIA GeForce GTX 1650. We tested different settings of input sizes ($= [3 \times 4 \times 4], [3 \times 8 \times 8], [3 \times 16 \times 16], [3 \times 32 \times 32]$), learning rates ($= 10^{-3}, 10^{-4}, 10^{-5}$) and batch size ($= 1, 4, 8$) for our, PyTorch and TensorFlow implementation of the multi-head attention layer. We emphasize that the setting of batch size 1 is only for pseudo training time counts. All the experiments related to prediction are conducted with batch size 4 or 8.

Identical inputs sampled from a uniform distribution ($X \sim U[-1.0, 1.0]$) are used as training data whereas all-zero outputs are used as ground truth in order to test on predicting zero tasks. We use mean square error as the loss function, which can be formulated as:

$$\text{MSE}(P, G) = \frac{1}{N}\sum_{i=0}^{n-1}\sum_{j=0}^{m-1}\sum_{k=0}^{l-1}\|P_{i,j,k} - G_{i,j,k}\|_2,$$

where P denotes the prediction mask, G denotes the ground truth mask and n, m, l denotes the batch size, projection dimension, sequence length, respectively.

The purpose of the experiments is to compare the training speed and prediction loss of different implementations for predicting all-zero in various settings.

4.2 Result

In this section, we comprehensively study the training speed, convergence and performance of different implementations in various settings.

We conduct pseudo training experiments to compare the average training speed of different implementations for **one**

single batch input of size $[3\times4\times4]$, $[3\times8\times8]$, $[3\times16\times16]$, $[3\times32\times32]$ (epoch = 3000). As shown in the figures 5, 6, 7 and 8, our multi-head attention layer has a faster training speed when the input size is $[3\times4\times4]$, $[3\times8\times8]$ or $[3\times16\times16]$, but has a slower training speed when the input size is $[3\times32\times32]$. Although our implementation has an advantage in training speed in most of the scenarios, the input size, however, has significant adverse effects on the training time of our implementation, while it has only minor influence on the training time of PyTorch and TensorFlow multi-head attention layers. This indicates a limitation of our implementation for training data with a large size. Meanwhile, it is found in our experiments that both PyTorch and TensorFlow implementations outperform our method in terms of training speed as the batch size and the number of batches increase, as shown in table 3.

Input Size	Ours (10^{-4})	PyTorch (10^{-4})	TensorFlow (10^{-4})
$[3\times4\times4]$	2.634	0.467	**0.341**
$[3\times8\times8]$	**0.554**	1.956	0.697
$[3\times16\times16]$	**0.0565**	5.523	3.638
$[3\times32\times32]$	**0.0555**	11.03	3.595

TABLE 2
Quantitative comparison on prediction loss, lower loss being better (↓)

Input Size	Ours (s)	PyTorch (s)	TensorFlow (s)
$[3\times4\times4]$	**448.6**	854.4	845.4
$[3\times8\times8]$	**583.0**	854.5	841.6
$[3\times16\times16]$	937.5	858.2	**850.9**
$[3\times32\times32]$	1550.5	865.5	**862.6**

TABLE 3
Training time for 1000 epochs (#batch = 100, batch size = 8)

As for the training convergence, our training loss is visualized in figure 4. To further compare the performance, we sampled 800 random inputs (batch size = 8) from a uniform distribution $X\sim U[-1.0,1.0]$ and trained all the models to predict all-zero masks for 1000 epochs. We demonstrate the best-epoch prediction losses of the three in Table 2.

5 Conclusion

We conclude our contributions in this paper in two aspects:

(1) We present an implementation of the multi-head attention layer in MagmaDNN framework, making possible the development of transformer architectures for MagmaDNN library.

(2) We compare the performance among our multi-head layer with PyTorch and TensorFlow implementations. Compared with competitors, our layer outperforms them by a clear margin in terms of the best-epoch prediction loss, despite a reasonable extra training time for large-scale data.

Notably, our implementation is not flawless. There are two main obstacles for our method to further improve: Scale issue and gradient explosion.

As mentioned in section 4.2, we found in our experiments that the training time of the multi-head attention layer in PyTorch and TensorFlow do not dramatically increase as the input size becomes larger. However, the input size has obvious negative effects on the training time of our implementations, which explains why PyTorch and TensorFlow implementations outperform ours in training speed for inputs of larger size. We suspect that this is due to overheads caused by other parts of the MagmaDNN library

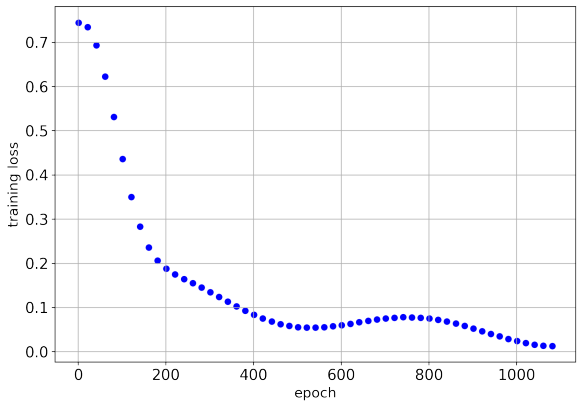

Fig. 4. Training loss in 1000 epochs (learning rate = 10^{-5}).

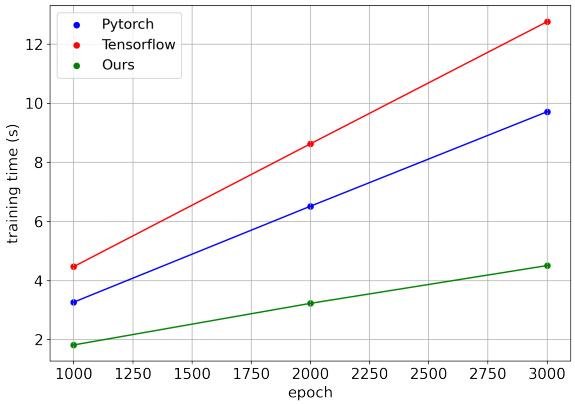

Fig. 5. Training time against epochs when input size = $[3\times4\times4]$

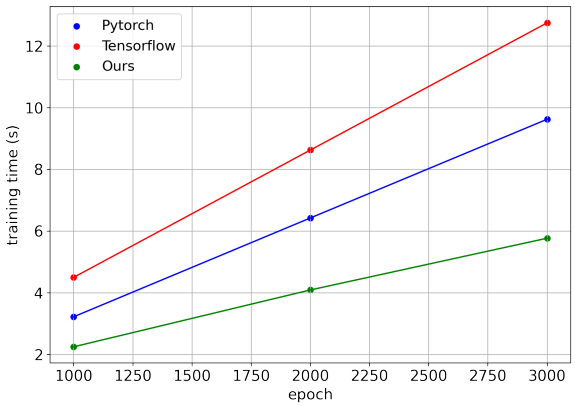

Fig. 6. Training time against epochs when input size = $[3\times8\times8]$

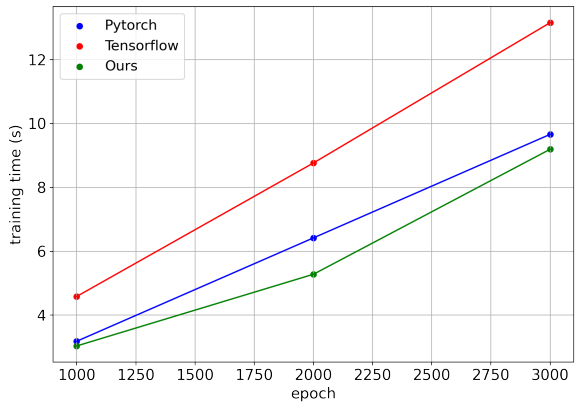

Fig. 7. Training time against epochs when input size = $[3 \times 16 \times 16]$

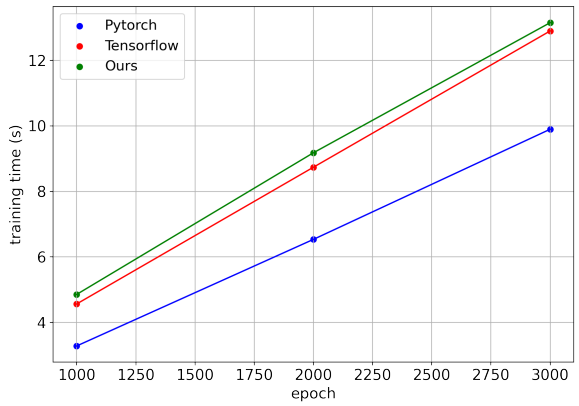

Fig. 8. Training time against epochs when input size = $[3 \times 32 \times 32]$

not directly related to our implementation.

As for gradient explosion, we empirically found that the gradients of weights exploded when we trained our layer for a huge number of epochs, despite having employed Xavier initialization. At an extreme, the values of weights become so large as to overflow and result in NaN values. The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0. Using smaller batch size alleviates this issue to some extent. However, we suggest adding batch normalization layer after multi-head attention layer to alleviate the gradient explosion in the implementation of the transformer model.

Future Direction

In terms of future research directions, main streams of the improvement include **linearized** attention and **sparse** attention. Also, graph attention networks (GATs) [20] are one of the prominent applications of the multi-head attention mechanism. We briefly summarize the idea of linearized attention and GATs to benefit the future development of self-attention mechanism and variants in the MagmaDNN framework.

For linearized attention, related works focus on proposing new kernel functions to linearize softmax function, approximating the attention matrix. For instance, see [21], [22], [23]. More precisely, we re-write the softmax function in (1) as:

$$y_m = \sum_{n=1}^{m} \frac{v_n \phi(k_n, q_m)}{\sum_{l=1}^{m} \phi(k_l, q_m)}$$

for some kernel function ϕ, which, in the classic setting of softmax function, is defined as: $\phi(k, q) = \exp(k^\top q)$. The general idea of linearized function is to replace the kernel function by some other candidates. For example, in [21], the kernel function is chosen as $\phi(k, q) = \widetilde{\phi}(k)^\top \widetilde{\phi}(q)$ for some function $\widetilde{\phi}$, so that

$$y_m = \sum_{n=1}^{m} \frac{(v_n \widetilde{\phi}(k_n)^\top)\widetilde{\phi}(q_m)}{(\sum_{l=1}^{m} \widetilde{\phi}(k_l)^\top)\widetilde{\phi}(q_m)}$$

and therefore

$$\sum_{n=1}^{m} (v_n \widetilde{\phi}(k_n)^\top)\widetilde{\phi}(q_m) = (\sum_{n=1}^{m} v_n \otimes \widetilde{\phi}(k_n)^\top)\widetilde{\phi}(q_m).$$

It has been shown that switching the order of matrix multiplication improves the time efficiency of self-attention for long sequences [24].

For graph attention networks, the core component is the graph attentional layer. The input of the layer is a set of node features $h = [h_1, \ldots, h_N]$, where $h_i \in \mathbb{R}^F$, and its output is a set of new node features $h' = [h'_1, \ldots, h'_N]$, where $h'_i \in \mathbb{R}^{F'}$. In the original paper [20], the node features h_i and h'_i are viewed as column vectors, which we shall follow here. The (single-head) formulation is as follows:

$$h'_i = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} W h_j\right)$$
$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$
$$e_{ij} = a(Wh_i, Wh_j)$$

where σ is a nonlinearity (such as a softmax or logistic sigmoid), $W \in \mathbb{R}^{F' \times F'}$ is a learnable weight matrix, $\mathcal{N}_i$ is some neighborhood of node i, and $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$ is some attention mechanism. The e_{ij} and α_{ij} are called attention coefficients, the latter being the former normalized across $\mathcal{N}_i$. In [20], $\mathcal{N}_i$ is chosen as the set of all first-order neighbors of node i including i itself, and a is chosen as

$$a(x, y) = \text{LeakyReLU}(c^\top [x \parallel y])$$

where $c \in \mathbb{R}^{2F'}$ is a learnable weight vector and $\parallel$ denotes concatenation. The negative input slope in LeakyReLU is set to be $\alpha = 0.2$.

While MagmaDNN has been sufficiently mature for developers and users to implement the graph attentional layer as formulated above, our MHA layer would be of little use in that formulation. One direction is to replace a by our MHA layer (with $Q = K = V$) followed by a nonlinear activation such as ELU, which cuDNN API supports. The multi-head formulation of the layer can be found in [20].

6 Acknowledgement

We express our greatest gratitude towards National Science Foundation, The Chinese University of Hong Kong, and City University of Hong Kong for funding this project. We would like to acknowledge The University of Tennessee, Knoxville for their support. We also thank our mentor Dr. Kwai-Lam Wong for his guidance during the project.

References

[1] Y. Gao, M. Zhou, and D. N. Metaxas, "UTNet: A hybrid transformer architecture for medical image segmentation," in *Medical Image Computing and Computer Assisted Intervention - MICCAI 2021 - 24th International Conference, Strasbourg, France, September 27 - October 1, 2021, Proceedings, Part III* (M. de Bruijne, P. C. Cattin, S. Cotin, N. Padoy, S. Speidel, Y. Zheng, and C. Essert, eds.), vol. 12903 of *Lecture Notes in Computer Science*, pp. 61–71, Springer, 2021.

[2] Y. Gao, M. Zhou, D. Liu, and D. N. Metaxas, "A data-scalable transformer for medical image segmentation: Architectures, model efficiency, and benchmarks," *CoRR*, vol. abs/2203.00131, 2023.

[3] Z. Shen, I. Bello, R. Vemulapalli, X. Jia, and C. Chen, "Global self-attention networks for image recognition," *CoRR*, vol. abs/2010.03019, 2020.

[4] Z. Huang, X. Wang, Y. Wei, L. Huang, H. Shi, W. Liu, and T. S. Huang, "CCNet: Criss-cross attention for semantic segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 6, pp. 6896–6908, 2023.

[5] J. Song, S. Kim, and S. Yoon, "AligNART: Non-autoregressive neural machine translation by jointly learning to estimate alignment and translate," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021* (M. Moens, X. Huang, L. Specia, and S. W. Yih, eds.), pp. 1–14, Association for Computational Linguistics, 2021.

[6] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with GPUs," *Numerical Computations with GPUs*, pp. 1–26, 2014.

[7] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, pp. 232–240, June 2010.

[8] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*, (Atlanta, GA), pp. 1–8, IEEE Computer Society, April 19-23 2010. DOI: 10.1109/IPDPSW.2010.5470941.

[9] R. Nath, S. Tomov, and J. Dongarra, "An improved MAGMA GEMM for Fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, pp. 511–515, Nov. 2010.

[10] R. Nath, S. Tomov, and J. Dongarra, "Accelerating GPU kernels for dense linear algebra," in *Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR'10*, (Berkeley, CA), Springer, June 22-25 2010.

[11] D. Nichols, N.-S. Tomov, F. Betancourt, S. Tomov, K. Wong, and J. Dongarra, "MagmaDNN: Towards high-performance data analytics and machine learning for data-driven scientific computing," in *ISC High Performance*, (Frankfurt, Germany), Springer International Publishing, Springer International Publishing, 2019-06 2019.

[12] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2015 - 18th International Conference Munich, Germany, October 5 - 9, 2015, Proceedings, Part III* (N. Navab, J. Hornegger, W. M. W. III, and A. F. Frangi, eds.), vol. 9351 of *Lecture Notes in Computer Science*, pp. 234–241, Springer, 2015.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, eds.), pp. 5998–6008, 2017.

[15] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)* (J. Burstein, C. Doran, and T. Solorio, eds.), pp. 4171–4186, Association for Computational Linguistics, 2019.

[16] J. F. Kolen and S. C. Kremer, *Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies*, pp. 237–243. 2001.

[17] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, L. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, vol. abs/1609.08144, 2016.

[18] NVIDIA, "API reference." https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnSeqDataDescriptor_t.

[19] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010* (Y. W. Teh and D. M. Titterington, eds.), vol. 9 of *JMLR Proceedings*, pp. 249–256, JMLR.org, 2010.

[20] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, OpenReview.net, 2018.

[21] Z. Qin, W. Sun, H. Deng, D. Li, Y. Wei, B. Lv, J. Yan, L. Kong, and Y. Zhong, "cosFormer: Rethinking softmax in attention," in *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*, OpenReview.net, 2022.

[22] I. Schlag, K. Irie, and J. Schmidhuber, "Linear transformers are secretly fast weight programmers," in *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event* (M. Meila and T. Zhang, eds.), vol. 139 of *Proceedings of Machine Learning Research*, pp. 9355–9366, PMLR, 2021.

[23] K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlós, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, D. B. Belanger, L. J. Colwell, and A. Weller, "Rethinking attention with performers," in *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*, OpenReview.net, 2021.

[24] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are RNNs: Fast autoregressive transformers with linear attention," in *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, vol. 119 of *Proceedings of Machine Learning Research*, pp. 5156–5165, PMLR, 2020.