University of California
Santa Barbara

Addressing Data Explosion Issue in Emerging Deep Learning Applications

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Zheng Qu

Committee in charge:

Professor Yuan Xie, Co-Chair
Professor Yufei Ding, Co-Chair
Professor Timothy Sherwood
Professor Zheng Zhang

June 2023

The Dissertation of Zheng Qu is approved.

Professor Timothy Sherwood

Professor Zheng Zhang

Professor Yufei Ding, Committee Co-Chair

Professor Yuan Xie, Committee Co-Chair

May, 2023

Addressing Data Explosion Issue in Emerging Deep Learning Applications

This dissertation is dedicated to my family, to my parents and grandparents. Thank you for always caring about me, encouraging me, and believing in me.

Acknowledgements

I have always considered myself slow to appreciate the value of things in my life. One day, I was driving down Mesa Rd. in my second-hand Tiguan, enjoying the breeze and sunset of Goleta; the next, I found myself in an office in Beijing, 6,000 miles from where I spent the past five years pursuing my Ph.D. This journey felt both long and short, yet I couldn't have reached this point without help.

I am deeply grateful for the support I received throughout this time. First and foremost, I extend my sincerest gratitude to my advisor, Professor Yuan Xie, whose continuous guidance has been invaluable. He taught me how to be a qualified researcher, a diligent worker, and a committed person. His insights, passion, and generosity will remain with me as I move forward in my career and life.

I also thank Professor Yufei Ding, my co-advisor, for her valuable advice on my research and career choices. I am fortunate to have worked under her guidance. Additionally, I appreciate my thesis committee members, Professor Timothy Sherwood and Professor Zheng Zhang, for their insightful input. Discussions with Tim were always inspiring, while Professor Zhang provided significant help on my Tensor-train projects.

I am very lucky to have had two great internship experiences during my Ph.D. study. In 2020, I worked at Alibaba DAMO Academy under the supervision of Dr. Dimin Niu, Dr. Shuangchen Li, and Dr. Hongzhong Zheng. In 2022, I worked at Nvidia Developer Technology Team, mentored by Dr. William Raveane, Rawn Henry, and Dr. Murat Efe Guney. Working with industry talents is very exciting, and really broadened my horizon.

I feel truly blessed to have pursued my Ph.D. at SEAL Lab, surrounded by a group of kind, intelligent, and hardworking students. I am grateful for the postdoc researchers including Dr. Lei Deng, Dr. Xin Ma, Dr. Xing Hu, Dr. Shuangchen Li, Dr. Fengbin Tu, and Dr. Jiayi Huang, for their hands-on guidance on my research projects. I want to express my special thanks to my close collaborators, Dr. Liu Liu and Zhaodong Chen. Liu is like a brother to me, both in research and in life. He is a very talented researcher, always full of inspiring ideas, and led me to explore my interests in sparse NN acceleration. Zhaodong is one of the most diligent people I have ever met. He is also extremely good at discovering opportunities and delivering encouraging results with his strong execution skills. This dissertation contains equally-contributed work, as well as co-first-authored publications with Liu and Zhaodong. Their contributions are reflected in this dissertation and their own. I also want to thank all the SEALers, including Dr. Maohua Zhu, Dr. Peng Gu, Dr. Xinfeng Xie, Dr. Dylan Stow Randall, Dr. Itir Akgun, Dr. Abanti Basak, Dr. Wenqin Huangfu, Dr. Tianqi Tang, Dr. Gushu Li, Dr. Ling Liang, Dr. Jilan Lin, Bangyan Wang, Nan Wu, Guyue Huang, Yuying Quan, Siqi Li, and Hao Li. It has been a great time working with all of my colleagues.

Lastly, I express my deepest gratitude to my parents, Huifen Xiang and Weidong Qu. They have been a constant source of support, strength, and light throughout my life.

Curriculum Vitæ
Zheng Qu

Education

2023	Ph.D. in Electrical and Computer Engineering (Expected), University of California, Santa Barbara.
2020	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2018	B.S. in Electronic Engineering, Tsinghua University.

Publications

[1] Jilan Lin*, Ling Liang*, **Zheng Qu**, Ishtiyaque Ahmad, Liu Liu, Fengbin Tu, Trinabh Gupta, Yufei Ding, Yuan Xie. “INSPIRE: In-Storage Private Information Retrieval via Protocol and Architecture Co-design,” ISCA’22

[2] **Zheng Qu***, Liu Liu*, Fengbin Tu, Zhaodong Chen, Yufei Ding. “DOTA: Detect and Omit Weak Attentions for Scalable Transformer Acceleration,” ASPLOS’22 (*co-primary)

[3] Zhaodong Chen*, **Zheng Qu***, Liu Liu, Yufei Ding, Yuan Xie. “Efficient Tensor Core-based GPU Kernels for Structured Sparsity Under Reduced Precision,” SC’21 (*co-primary)

[4] Abanti Basak, **Zheng Qu**, Jilan Lin, Alaa R Alameldeen, Zeshan Chishti, Yufei Ding, Yuan Xie. “Improving Streaming Graph Processing Performance using Input Knowledge,” MICRO’21

[5] Liu Liu*, Jilan Lin*, **Zheng Qu**, Yufei Ding, Yuan Xie. “ENMC: Extreme Near-Memory Classification via Approximate Screening,” MICRO’21

[6] Liu Liu, **Zheng Qu**, Lei Deng, Fengbin Tu, Shuangchen Li, Xing Hu, Zhenyu Gu, Yufei Ding, Yuan Xie. “DUET: Boosting Deep Neural Network Efficiency on Dual-Module Architecture,” MICRO’20

[7] Youjie Li, Jongse Park, Mohammad Alian, Yifan Yuan, **Zheng Qu**, Peitian Pan, Ren Wang, Alexander Schwing, Hadi Esmaeilzadeh, Nam Sung Kim. ”A Network-Centric Hardware-Algorithm Co-design to Accelerate Distributed Training of Deep Neural Networks,” MICRO’18

[8] Liu Liu*, **Zheng Qu***, Zhaodong Chen, Fengbin Tu, Yufei Ding, Yuan Xie. “Dynamic Sparse Attention for Scalable Transformer Acceleration,” IEEE Transactions on Computers (*co-primary)

[9] Ling Liang, **Zheng Qu**, Zhaodong Chen, Fengbin Tu, Yujie Wu, Lei Deng, Guoqi Li, Peng Li, Yuan Xie. “H2learn: High-efficiency Learning Accelerator for High-accuracy

Spiking Neural Networks," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[10] **Zheng Qu**, Bangyan Wang, Lei Deng, Hengnu Chen, Ling Liang, Jilan Lin, Guoqi Li, Zheng Zhang, Yuan Xie. "Hardware-Enabled Efficient Data Processing with Tensor-Train Decomposition," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems

[11] Hengnu Chen, Lei Deng, **Zheng Qu**, Ling Liang, Tianyi Yan, Yuan Xie, Guoqi Li. "Tensor Train Decomposition for Solving Large-scale Linear Equations," Neurocomputing

[12] Bangyan Wang, Lei Deng, **Zheng Qu**, Shuangchen Li, Zheng Zhang, Yuan Xie. "Efficient Processing of Sparse Tensor Decomposition via Unified Abstraction and PE-interactive Architecture," IEEE Transactions on Computers

[13] Zichen Fan*, Zheyu Liu*, **Zheng Qu***, Fei Qiao, Qi Wei, Xinjun Liu, Yinan Sun, Shuzheng Xu, Huazhong Yang. "ASP-SIFT: Using Analog Signal Processing Architecture to Accelerate Keypoint Detection of SIFT Algorithm," IEEE Transactions on Very Large Scale Integration Systems (*co-primary)

[14] Zhaodong Chen, **Zheng Qu***, Yuying Quan, Liu Liu, Yufei Ding, Yuan Xie. "Dynamic N: M fine-grained structured sparse attention mechanism," ASPLOS'22 (*co-primary)

Abstract

Addressing Data Explosion Issue in Emerging Deep Learning Applications

by

Zheng Qu

With the continuous booming development of deep learning, many kinds of model variants are being proposed to tackle more difficult machine learning tasks, such as Transformers, Deep Learning Recommendation Models, and Graph Neural Networks. These emerging deep learning models, while being sufficiently better than prior methods, also require much more hardware resources to train and deploy. To systematically tackle this issue, we approach it from a data-centric perspective and argue that the root cause of the software-hardware imbalance is the data explosion in emerging deep learning models.

To continue the scaling of deep learning applications and bridge the gap between hardware performance and application requirements, this dissertation proposes to leverage data redundancy to effectively reduce model cost and benefit hardware design. We first categorize the data in a deep learning model into three types, namely input dataset, model parameters, and computational results. While parameter redundancy has been extensively studied in prior work, data representation and computational redundancy are rarely discussed. On the base of this observation, we introduce four software-hardware co-designs to explore the other two types of data redundancy and thus improve deep learning efficiency. Specifically, in order to reduce the cost of intermediate computational results in Transformer models, the first two designs leverage dynamic runtime approximation with customized GPU kernel and ASIC design. To release the memory and computation burden caused by massive input training data, the third and fourth design focuses on using high-order tensor decomposition with domain-specific knowledge to achieve high-quality and aggressive data compression. Training on the compressed dataset leads to comparable model accuracy with much less hardware consumption.

Overall, this dissertation demonstrates opportunities and approaches to tackle data explosion in emerging deep learning models. Our methods cover both dynamically generated data and offline trainable data, both deep learning training and inference, and both general computing platforms (e.g., GPGPU) and customized accelerators.

Contents

Chapter 1

Introduction

Over the past years, deep learning based approaches have been driving enormous breakthroughs in various scientific applications, and have changed people's lives in all kinds of aspects. Different kinds of model variants are being proposed to incorporate domain-specific knowledge, such as Convolutional Neural Networks (CNNs) for computer vision [1], attention-based Transformer Neural Networks (Transformers) for language modeling [2], Deep Learning Recommendation Models (DLRMs) for personalization [3], and Graph Neural Networks (GNNs) for bioinformatics [4, 5]. While the model structure varies a lot among different applications, one general trend is commonly shared. That is, the size of these models, as well as the associated data during training and inference, has been scaling at a dramatic speed.

Using larger data and bigger models is one of the most reliable ways to improve the application performance [6]. With larger data, we can support the training and inference procedure with more important and valuable information. With more parameters, the capacity and representative power of the model can be improved to deal with complicated tasks. Unfortunately, all of the above techniques do not come as a free lunch. In fact, following the big-data big-model approach almost certainly leads to more and more hardware consumption. Moreover, the scaling of the algorithm complexity is significantly faster than the scaling of hardware computational power and memory capacity. As a result shown in [7], we are witnessing a continuously increasing software-hardware gap in emerging deep learning applications. The time we spent on training a state-of-the-art deep learning model has increased from hours to days and even months.

Such imbalance has led to other critical issues including environmental and economic costs. Summarizing the data presented in [8], the estimated carbon footprint of training a Transformer model (with fine-tuning and experimentation) is 7x of a human being's CO_2 emission per year. The number is from 2019 and it has increased even more since then. Therefore, in order to maintain a fast development of deep learning research and production, it is vital for us to find effective methods to bridge the gap between algorithm consumption and hardware efficiency.

1.1 Problem Formulation and Analysis

While the above introduction gives the overall scope and motivation for this work, the underlying problem to be solved is still unclear. To unveil the pain point, here we provide a systematic formulation of the problem to facilitate a comprehensive analysis. Specifically, we approach the current situation from a data-centric perspective, where each deep learning model is composed of three types of data, namely dataset representation, model parameters, and intermediate computational results. As shown in Figure 1.1, no matter how the application and model structure change, these three types of data always exist and lay at the core of the deep learning models. Algorithms and Hardware platforms are the methods that we leverage to associate the interaction and transition among these data types.

For example, a single pass of forward propagation starts from a batch of input samples,

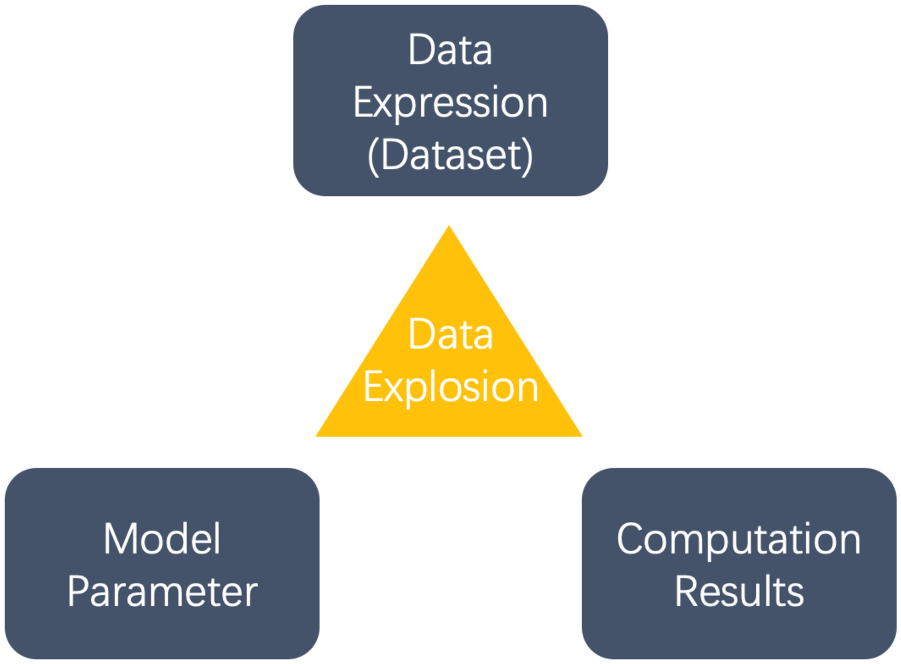


Figure 1.1: A data-centric perspective to understand the data explosion issue in deep learning models.

and uses the current version of the model parameters to generate a batch of computational results. These computational results may only be the final output if the model is used for inference, or contain intermediate hidden feature maps when a backward propagation is required. Memory system is used to store the data, and arithmetic units are implemented to perform the computations. It is true that the final performance of a deep learning application is a joint consequence of software and hardware design. But here we formulate the problem as Figure 1.1 to highlight the impact of data, because we believe that it is the **explosion** of the data that leads to many unseen challenges and results in the increasing algorithm-hardware gap we observed today.

Based on this problem formulation, we can now fit in different deep learning models and analyze the specific type(s) of data explosion it contains. The variance in data type also leads to different challenges and solutions, as we will illustrate throughout this dissertation.

1.2 Opportunities and Challenges

The most important observation of this dissertation is that: *Where there is data explosion, there exists data redundancy.*

1.2.1 Parameter Redundancy

During the past decade, exploring and leveraging parameter redundancy has been one of the most popular topics in deep learning research. The underlying idea is that, instead of having a dense, large, and precise model, we can train a sparse, smaller, and quantized model while maintaining an on-par model accuracy. Various techniques have been proposed to achieve this objective, such as model pruning [9], knowledge distillation [10], model quantization [11], and so on. For emerging deep learning models, this idea can be even more important, because the number of parameters has been exponentially growing, causing significant challenges to model training and deployment.

Despite the convenience to apply previous techniques to larger models, the outcome is unsatisfying. This is because these techniques, due to the limitation of their compression mechanism, only allow for a moderate compression ratio. Usually, the reduction of model size is at most tens of times. As a consequence, for models with extremely large parameter sets, such as Deep Learning Recommendation Models (DLRMs), it is insufficient to address the hardware challenges. Because the weights can still occupy tens of Gigabytes or even Terabytes of space, which exceeds the memory capacity of the computing platforms such as GPUs. Therefore, a more revolutionary approach is needed to address the exponential growth of model parameters and opens up new performance optimization opportunities.

1.2.2 Representation Redundancy

Deep learning applications are essentially statistical methods to understand and analyze real-world instances. Models are built on the representation of these instances. For example, we typically use 3D RGB data to represent images and add another dimension if the object has other information such as time sequence [12]. Also, different images use completely independent representations. While this formulation is straightforward and suitable for certain applications like image and video processing, it may be over-representative for others.

In many emerging deep learning applications such as DLRM and GNN, there exist relationships between different samples. For example, customers in the same category might have similar purchasing behavior, and related papers are usually connected in a citation network. These properties indicate there exists redundancy in the current representation system of the dataset, and reveals opportunities to represent different data samples with partially shared features [13, 14]. However, it is non-trivial to design appropriate new representations for emerging applications. There are several challenges: (1) We need to effectively identify the underlying relationship between the samples. Such methods are usually application- and dataset-specific. (2) We need to find a new mathematical representation for the samples. This new representation not only needs to allow feature sharing, but should also preserve a certain amount of unique information for each sample to be independently identified. (3) After the new mathematical representation is designed, the computation characteristics and memory access behavior are also affected. Therefore, it poses new challenges to the hardware design, and we need to optimize certain software infrastructures and even hardware architecture to efficiently execute the new models.

1.2.3 Computational Redundancy

Finally, during the processing of deep learning models, a massive amount of intermediate data is generated and stored, such as the output activations of each layer. Moreover, due to the firing mechanism of neuron synapses and the use of non-linear activation functions, these outputs usually contain many small and even zero values. Prior work has extensively discussed the opportunities to leverage weight sparsity and input activation sparsity [15, 16, 17, 18]. However, previous work suffers from a major limitation, that is the activation should still be completely generated before it can be pruned and used as input sparsity for the following layer. In emerging deep learning applications such as Transformer models, a majority of the computations and memory access are spent on computing and moving these intermediate data back and forth. Therefore, the conventional pruning method is incapable of handling these scenarios since it does not reduce the cost of the current layer which generates the activations. In order to fully leverage dynamic sparsity, we need to identify the redundancy prior to the execution of the layer and avoid the computation as well as memory access in advance. Obviously, this requires innovations in all different levels of the system, including algorithms, software, and hardware.

1.3 Contributions and Organization

Given the analysis above, we make the following contributions as we organize the thesis accordingly.

In Chapter 2, we introduce the background of different emerging deep learning models that will be discussed in this thesis, including Transformers and Graph Neural Networks. We also give a basic introduction to Tensor-train decomposition, which is the key method that we leverage to explore data representation redundancy.

Chapter 3 proposes DSA, an efficient algorithm to exploit dynamic computational redundancy during the execution of Transformer models. Specifically, we introduce an attention approximation module that can evaluate the attention values in advance. Given the approximated attention score, we can avoid computing and storing the unimportant attentions. We also leverage dimension reduction and quantization to reduce the cost of the attention approximation module, and guarantee the selection accuracy by jointly optimizing the attention approximation module with the rest of the model.

Chapter 4 discusses the challenge and opportunities to implement DSA on GPU platforms. Following the discussion, we propose to identify structural sparse attention instead of fine-grained sparsity for GPU-based solutions. We also implement customized GPU kernels to reduce attention approximation overhead as well as improve sparse attention performance.

Chapter 5 presents an accelerator architecture, DOTA, to fully take advantage of the DSA mechanism. Unlike GPUs, ASIC design is able to benefit from customized hardware unit as well as specialized dataflow to improve performance and energy efficiency. In DOTA, we propose to reorder the computation sequence of the attention matrix, such that different rows of the attention matrix can be generated in parallel with minimized memory bandwidth requirement. In other words, the performance of fine-grained sparse attention is significantly improved compared with GPU solutions, and a better trade-off can be achieved between model accuracy and hardware efficiency.

Chapter 6 introduces TTD Engine, an efficient software-hardware co-design for runtime data compression using Tensor-train decomposition. Specifically, we optimize the decomposition algorithm of Tensor-train by leveraging low-rankness of the original data, and accelerate the method using specialized architecture. We also presents a new algorithm to implement 2D convolution using both tensor-train format input and tensor-train format weights. A full-TT layer leverages both input activation redundancy and parameter redundancy, significantly outperforms prior one-side Tensor-train layer.

Chapter 7 is a demonstration of exploring dataset representation redundancy in Graph Neural Networks. We propose TT-GNN, an algorithm and hardware co-design to accelerate large scale training of GNNs. In TT-GNN, instead of having each node represented as one feature vector, we represent the complete graph embedding matrix as a tensor-train matrix. Therefore, different nodes will partially share some tensor slices depending on their index. We reorder the graph index according to the vertex connection such that similar nodes will share more partial features. As a result, we are able to represent the graph with a much compact data structure. On the base of this algorithm, we discuss the challenge and opportunities of implementing TT-GNN on both GPUs and ASICs.

Finally, we use Chapter 8 to conclude the thesis and discuss future research opportunities.

Chapter 2

Background and Related Work

In this section, we first summarize the preliminary knowledge of Transformer Neural Networks and Graph Convolutional Networks, followed by the introduction of Tensor-train Decomposition. Finally, we discuss the related work on efficient NN compression method and hardware acceleration.

2.1 Transformer Neural Networks

In recent years, Transformer Neural Networks have drawn a surge of interests from the deep learning community. Lots of Transformer models have been proposed and demonstrated superior performance over traditional Deep Neural Networks (DNNs) [19, 2]. The use of Transformers has spanned over a wide range of application domains including language understanding [20, 21, 22], image processing [23, 24, 25], and generative modeling [26, 27, 28].

A typical Transformer model is composed of stacked encoder (decoder) blocks as shown in Figure 2.1. At the beginning, the input sentence with n tokens is first transformed into an embedding matrix $X \in \mathbb{R}^{n \times d}$. Then, the input embedding matrix is

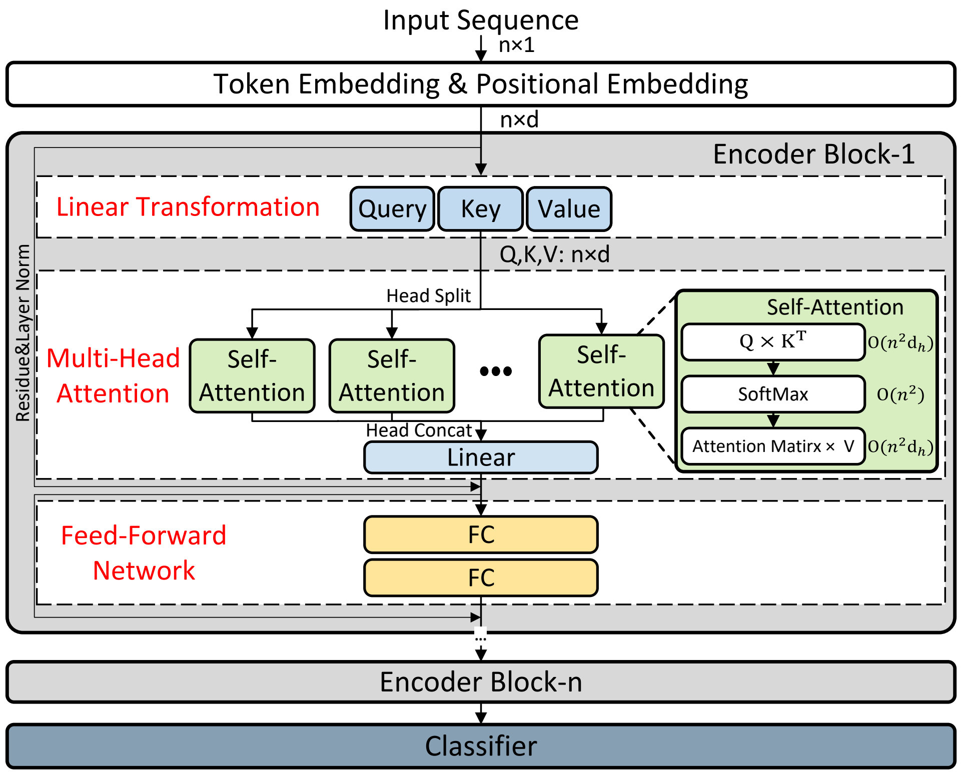


Figure 2.1: Transformer model architecture.

processed by blocks of encoders. We split each encoder into three stages, namely Linear Transformation, Multi-Head Attention, and Feed-Forward Network (FFN). In the transformation stage, we multiply the input with three weight matrices to obtain Query (Q), Key (K), and Value (V) as

$$Q, K, V = XW_Q, XW_K, XW_V \tag{2.1}$$

After linear transformation, the attention weights $A \in \mathbb{R}^{n \times n}$ is defined as

$$A = SoftMax(\frac{QK^T}{\sqrt{d_k}}) \tag{2.2}$$

where SoftMax$(\cdot)$ is computed row-wise. Finally, the output values are generated by multiplying attention weights A with the projected values V as

$$Z = AV. \tag{2.3}$$

The output of the Multi-Head Attention is added with the encoder's input through a residue connection, and a layer normalization is applied afterwards. Finally, a Feed-Forward Network (FFN) containing two fully-connected (FC) layers, followed by another residual connection and layer normalization is applied to generate the output of the encoder. As presented in Figure 2.1, the same encoder structure is repeated and stacked for multiple times in a single Transformer. Usually, a classifier is added at the end to make predictions.

2.2 Graph Convolutional Neural Networks

Originating from spectral graph analysis and fueled by the success of machine learning, graph neural networks (GNNs) have drawn a surge of interest and have been applied to various applications involving non-Euclidean graph-structured data. During the past few years, a wide range of GNN models [29, 30, 31, 32] have been proposed to solve graph-related problems. Exciting progress has been achieved by GNNs in domains such as recommendation systems [33], relation prediction [34], chemistry analysis [35], financial security [36], protein discovery [4, 5], EDA [37, 38, 39] and so on.

As shown by a toy model presented in Figure 7.2. Given an undirected graph, we denote it as $G = (V, E)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in the graph. Each node is described by a feature vector of length F, and all the node features together forms a 2D feature matrix $X \in \mathbb{R}^{|V| \times F}$. In most cases, matrix X is dense and of large-scale due to the massive amount of nodes contained in real-world

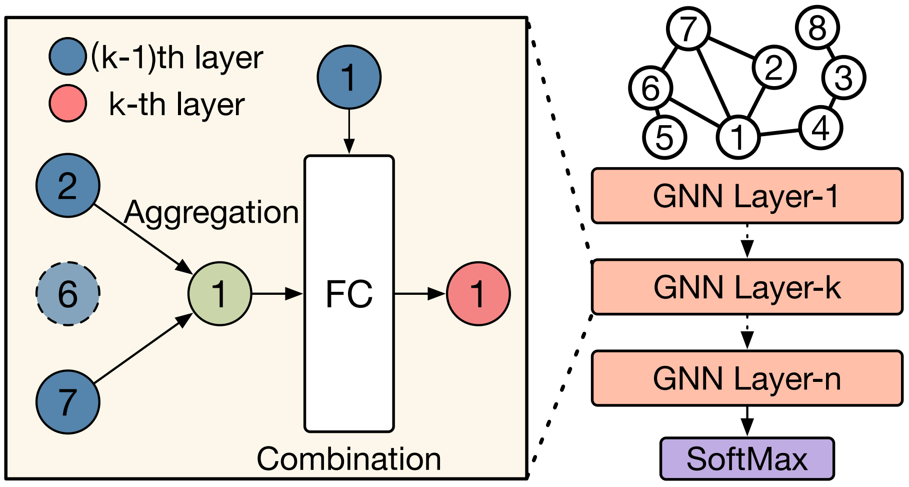


Figure 2.2: Illustration of a sample GNN model.

graphs.

During GNN processing, each GNN layer follows a two-stage procedure, namely *Aggregation* and *Combination*. As shown in Figure 7.2 and equations in below, during aggregation, each node v will collect feature vectors from its sampled neighborhood $N(v)$ to generate an aggregated feature a_v^k. The aggregation operator can be flexibly designed, where common choices include *Mean, Max, MLP* and so on. After this, the aggregated feature is combined with source node v's feature vector $h^{(k-1)_v}$. The combination operator utilizes these two vectors to generate hidden representation $h^k(v)$ of node v.

$$a_v^k = \textbf{Aggregate}(u : u \in N(v) \cup v)$$

$$h_v^k = \textbf{Combine}(a_v^k, h_v^{(k-1)})$$

2.3 Tensor-train Decomposition

Tensor-train Decomposition (TTD) is originally proposed by Oseledets in [40]. The overall procedure of the naive TTD algorithm is given in Alg. 3. In TTD, we try to approximately represent a given tensor $\mathcal{A}$ with tensor $\mathcal{B}$, which can be described as:

$$\mathcal{B}_{i_1,i_2\cdots,i_d} = G_1(i_1)G_2(i_2)\cdots G_d(i_d). \tag{2.4}$$

Each $G_k(i_k)$ is an $r_{k-1} \times r_k$ matrix, where r_k is called the TT-rank that can be either predefined before the decomposition or decided during runtime according to the required decomposition accuracy. G_k is an $r_{k-1} \times I_k \times r_k$ tensor core extracted from the original high-order tensor. In each TTD iteration, we need to perform Singular Value Decomposition (SVD) of an auxiliary matrix to get a tensor core. Therefore, it takes d sequential TTD iterations to finish the decomposition of a given tensor. Besides, at the beginning of each iteration, we need to reshape the given matrix into the required size before we can perform SVD. With the TT-format data, we can simply contracting these tensor cores together to reconstruct the approximated tensor which is close to the original tensor $\mathcal{A}$.

Notice that, the product of these parameter-dependent matrices in Equation (7.2) is a matrix of size $r_0 \times r_d$, this indicates the boundary condition of $r_0 = r_d = 1$. Moreover, since $r_0 = r_d = 1$, TTD can also be visually represented by a graph called linear tensor network, as shown in Figure 6.3. There are two different types of nodes in this graphical representation. The rectangles are the tensor cores with the spatial indices (i_k from the original tensor) and auxiliary indices α_k. The circles are indeed links to connect two adjacent tensor cores with same auxiliary index α_k. This means that these two tensor cores are contracted together, and further being contracted with the following tensor cores to form the final d-dimensional tensor.

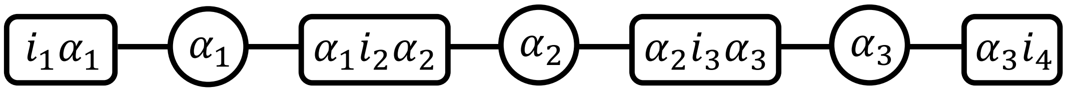


Figure 2.3: A tensor-train network.
Figure of a 3-way tensor

The most important step of TTD is how to extract these tensor cores from the original high order tensor. In this work, we focus on the classical TT-SVD approach, which computes such TTD using d-sequential SVDs of auxiliary matrices.

```
Algorithm 1 TT-SVD
Require: $d$-dimensional tensor $\mathcal{A}$, approximation error $\epsilon$.
Ensure: Tensor cores $G_1, ..., G_d$ of the TT-approximation $\mathcal{B}$ in the TT format with TT-
    ranks $r_k$ equal to the $\delta$-ranks of the unfoldings $A_k$ of $\mathcal{A}$. The approximation error
    satisfies:
                    $||\mathcal{A} - \mathcal{B}||_F \leq \epsilon ||\mathcal{A}||_F$
 1: {Initialization} Compute the truncation parameter:
                    $\delta = \epsilon \sqrt{d-1} ||\mathcal{A}||_F$
 2: Temporary tensor: $C = \mathcal{A},\ r_0 = 1$.
 3: for $k = 1$ to $d-1$ do
 4:   $C$ =reshape($C$, $[r_{k-1}I_k, numel(C)/(r_{k-1}I_k)]$)
 5:   Compute $\delta$-truncated SVD:
          $C = USV^T + E,\ ||E||_F \leq \delta,\ r_k = rank_\delta(C)$
 6:   New tensor core: $G_k$ =reshape($U$, $[r_{k-1}, I_k, r_k]$)
 7:   $C = SV^T$
 8: end for
 9: $G_d = C$
10: Return tensor $\mathcal{B}$ in the TT format represented by tensor cores $G_1, ..., G_d$.
```

2.4 Related Work on Data Redundancy Elimination in Neural Networks

In this section we present the literature survey related to data redundancy of Neural Networks, as well as hardware acceleration for emerging Neural Network models.

2.4.1 Data Redundancy in Neural Networks

As previously mentioned, static weight pruning is the most commonly applied technique to explore data redundancy, as it helps to decrease memory footprint and data access directly from the model prespective. Fine-grained weight sparsity can be applied to DNN and utlized by NN accelerators through compressed storage and computation skipping of zero weights [15, 16, 41, 42, 43, 44]. Due to the hardware inefficiency, coarse-grained weight sparsity is further proposed to mitigate the indexing overhead and irregular access [45, 46, 47, 48].

As compared to static parameter redundancy, other studies leverage dynamic redundancy and focus on improving runtime efficiency with intelligent dataflow mapping and compute orchestration strategy. One typical solution is to use $ReLU$-induced activation sparsity as either input sparsity detection [15, 17, 18, 49, 50, 51, 47, 44, 52] or output sparsity prediction [53, 54, 55, 56]. Besides, some work proposes channel-wise compression and gating methodology to reduce computations of certain NN layers [45, 57, 58].

2.4.2 Related Work on Transformer Acceleration

There have been a few recently proposed works targeting the acceleration of attention and Transformer. MnnFast [59] skips the computation of specific value vectors if its attention weights is lower than the threshold. This method can only benefit the attention output computation rather than attention weights computation. A^3 [60] is the first work to apply approximation to the attention weights for computation reduction. A^3 involves a sorting-based preprocessing phase that needs to be done outside the accelerator. ELSA [61] improves the approximation method by directly using sign random projection to estimate the angle between query and key vectors. SpAtten [62] proposes cascade token pruning and head pruning to reduce the cost of both self-attention block and subsequent layers in the Transformer model. The proposed method can be regarded as adding structured sparsity constraints to the attention matrix, as it directly removes several rows and columns. As for hardware design, SpAtten supports both decoder and encoder processing. Finally, OPTIMUS [63] proposes a GEMM architecture to accelerate Transformer inference. It focuses on accelerating sequential decoding process and proposes technique to maintain resource utilization.

2.4.3 Related Work on Tensor-train-based Neural Network

Tensor-train Decomposition (TTD) [40] is originally used to decompose high order tensor data and break the curse of dimensionality through efficient implementation of basic operations. [64] sees the opportunity of adopting TTD to reduce the modes size and computation complexity of Convolution Neural Networks (CNNs). The key idea is to change the 2D weight matrix into a parameterized tensor-train weight and train the entire model from scratch. In other words, this approach can be regarded as replacing the FC and Conv layer in CNNs with a Tensor-train (TT) layer. TT layer has fewer parameters and less computation complexity for inference. Since then, TT layer has also been used in other DNN models such as RNNs [65] and Transformers [66]. The unique computation pattern of Tensor-train also inspires research effort on customized accelerator design [67] for these Tensorized Neural Networks (TNNs). Tensor-train is also leveraged to replace the large embedding layer used in Deep Learning Recommendation Models [13]. The central idea remains the same as before. The difference is that none of the previous DNN models have such large weight matrix that consumes more than 99% of the model capacity with up to TBs of memory consumption. Therefore, this work demonstrates the important potential of tensor-train method in such extreme-scale models, which could potentially help control the explosive demands on computational infrastructure.

Chapter 3

Dynamic Approximation for Computational Redundancy in Transformer Models

This chapter presents our work on exploring dynamic computational redundancy using efficient runtime approximation. Specifically, we present dynamic sparse attention, an algorithm to reduce self-attention complexity used in Transformer Neural Networks

3.1 Motivation

Before we describe our method in detail, we first introduce the preliminaries of the standard attention mechanism used in vanilla Transformers. Then, we discuss the challenge of serving long sequences under the quadratic complexity of attention. Finally, we demonstrate that redundancy exists in attentions and dynamic sparse patterns are naturally expressed in attention.

3.1.1 Preliminaries of Attention

The attention mechanism is the essential component of Transformers [19]. Self-attention operates on input representations of length l, $X \in \mathbb{R}^{l \times d}$, with three linear projections namely, query, key, and value as

$$Q, K, V = XW_Q, XW_K, XW_V \tag{3.1}$$

, where $Q \in \mathbb{R}^{l \times d_k}$ denotes the queries, $K \in \mathbb{R}^{l \times d_k}$ denotes the keys, and $V \in \mathbb{R}^{l \times d_v}$ denotes the values. After linear projections, the attention weights $A \in \mathbb{R}^{l \times l}$ is defined as

$$A = \phi(\frac{QK^\top}{\sqrt{d_k}}) \tag{3.2}$$

where ϕ is the row-wise $softmax(\cdot)$ function. Finally, the output values are computed by multiplying the attention weights A with the projected values V as

$$Z = AV. \tag{3.3}$$

Serving Transformer-based models is challenging when the input sequence length l is large. When using long sequences, computing Eq. (3.2) and Eq. (3.3) consumes the majority of operations and becomes the bottleneck of model evaluation. The asymptotic complexity of attention $O(l^2 d_k + l^2 d_v)$ is quadratic to sequence length l.

3.1.2 Intrinsic Sparsity in Attention Weights

A number of efficient Transformer variants have been proposed to mitigate the quadratic complexity of self-attention [25, 68, 69, 70]. One straightforward way to exploit the intrinsic redundancy in attention is forming sparse patterns as in

$$A = \phi(QK^\top - c(1 - M)), \tag{3.4}$$

where $M \in \{0, 1\}^{l \times l}$ represents the sparse attention pattern, c is a large constant ($1e^4$) such that where $M_{ij} = 0$, indicating unimportant attention, $A_{ij} = 0$ after $softmax$ normalization. Here, we omit $\sqrt{d_k}$ for simplicity. The sparse patterns can be pre-determined into global, block, random, or a combination of different patterns. Another way to determine sparse patterns is through trainable masks. However, all these methods explore static or fixed sparse patterns, restricting viable attention connections.

3.1.3 Dynamic Sparse Patterns in Attention

A common motivation of sparse attention methods is that not all attention weights, i.e., probabilities, are equally important in Eq. (3.3). A large portion of attention weights do not contribute to attention output and are redundant. In other words, only a small portion of attention weights are useful. However, we find that sparse patterns in attention are inherently dynamic and data-dependent.

Here, we further support our hypothesis by showing the original attention weights matrix (after $softmax$ normalization) in Figure 3.1. The model used here is a vanilla Transformer and the benchmark is Text Classification from Google Long-Range Arena[71]. Figure 3.1 indicates that only a small amount of attention weights are with large magnitude and a significant portion is near zero. We want to emphasize that this shows the raw attention weights without forcing any sparsity constraints or fine-tuning, which indicates that redundancy naturally exists in attention. In short, attention mechanism exhibits the focused positions on a set of important tokens.

More importantly, the attention weights have dynamic sparse patterns. As shown in

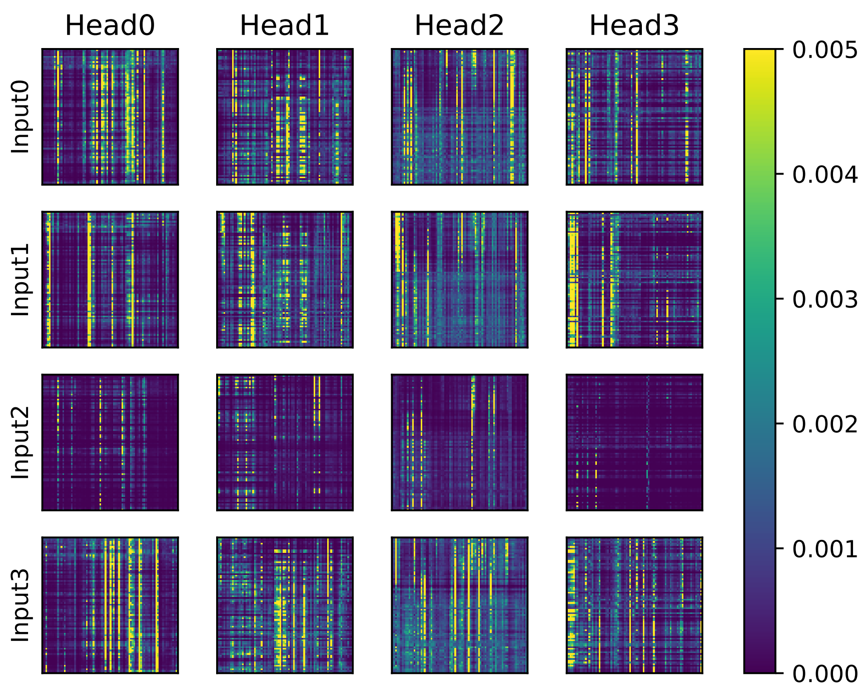


Figure 3.1: Visualization of attention weights from different inputs and attention heads. Only a small amount of attention weights are important. Note values > 0.005 are clamped to show as 0.005.

Figure 3.1, the sparse patterns in attention weights are dynamically changing depending on the input sequence. Different heads in multi-head attention also have different sparse patterns. The characteristic of dynamic sparsity in attention weights motivates us to explore effective methods to eliminate the redundancy and save computations. Prior work on static or fixed sparse patterns cannot capture the dynamically changing attention weights. Recent studies reveal similar dynamic sparse patterns in attention using modified attention [72, 73]. Instead, we promote the intrinsic dynamic sparse patterns in attention of standard transformers, and our focus is on practical acceleration of long sequences. A recent study shows that pruning near-zero attention values during inference has limited effect on accuracy [74]. The problem that our work is targeting is not only pruning unimportant attention values but also predicting which attention values to prune and saving more computations of attention scores as in Eq. 3.2

3.2 Dynamic Sparse Patterns in Attention

From Section 3.1, we show that attention weights have intrinsic sparse patterns, and the positions of important attention weights are dynamically changing as different input sequences. While attention exhibits dynamic sparse patterns, how to efficiently and effectively obtain the dynamic sparse patterns remains challenging. We formulate the process of identifying sparse attention patterns as a prediction problem. The key challenge is how to obtain an approximate attention predictor that can accurately find the sparse patterns while keeping the prediction overhead small.

Here, we present *Dynamic Sparse Attention* (DSA) that exploits sparse patterns in attention weights to reduce computations. The principle is to effectively search for dynamic sparse patterns without enforcing strict and static constraints on attention while keeping the searching cost small. Our approach leverages trainable approximation to predict sparse patterns. As shown in Figure 3.2, we use a prediction path based on low-rank transformation and low-precision computation. The prediction path processes input sequences functionally similar to query and key transformations but at much lower computational costs. Given the prediction results that approximate $QK^\top$ well, we can search sparse patterns based on the magnitude of prediction results.

3.2.1 Design of Prediction Path

We denote attention scores as $S = QK^\top$ and omit the scaling factor for simplicity. As shown in Figure 3.2(a), two general matrix-matrix multiplication kernels (GEMM) and one $softmax$ kernel consume the majority of computations in self-attention. We construct a pair of approximate query and key transformations in the prediction path to

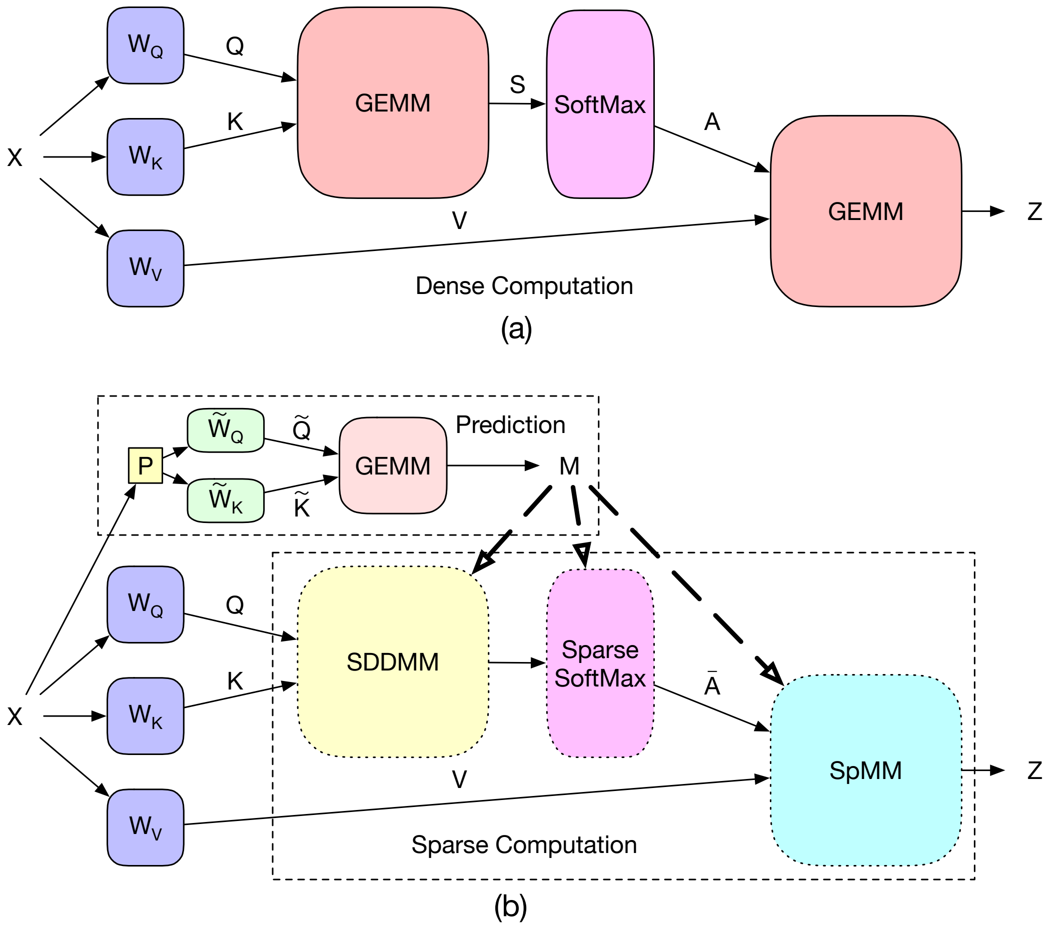


Figure 3.2: (a) Standard full attention; (b) Dynamic sparse attention with approximation-based prediction and sparse computation.

compute for approximate score $\tilde{S}$, as in

$$\tilde{Q}, \tilde{K} = XP\tilde{W_Q}, XP\tilde{W_K}. \tag{3.5}$$

Here $P \in \sqrt{\frac{3}{k}} \cdot \{-1, 0, 1\}^{d \times k}$ is a sparse random projection matrix shared by both paths, and $\tilde{W_Q} \in \mathbb{R}^{k \times k}, \tilde{W_K} \in \mathbb{R}^{k \times k}$ are parameters in approximating query and key.

Then, we have approximate attention scores $\tilde{S} = \tilde{Q}\tilde{K}^\top$. From $\tilde{S}$, we can predict sparse attention masks M using thresholds, where the threshold values are either fixed by tuning from the validation set or determined by $top-k$ searching. When $\tilde{S}$ is well approximated with accurate attention scores S, the large scores in $\tilde{S}$ are also large in S

with high probability. The resulting sparse attention weights $\bar{A}$ is used to multiply the value matrix V similar to Eq. 3.3.

Optimization of Approximation. The random projection matrix P is constant after initialization and shared by two approximate transformations. We obtain the trainable parameters, $\tilde{W_Q}$ and $\tilde{W_K}$, through minimizing the mean squared error (MSE) as the criterion to optimize for approximation:

$$L_{MSE} = \frac{1}{B}||S - \tilde{S}||_2^2 = \frac{1}{B}||QK^\top - \tilde{Q}\tilde{K}^\top||_2^2 \tag{3.6}$$

where B is the mini-batch size.

Given the motivation of finding dynamic sparse patterns, the hypothesis of our method is that there exist *oracle* sparse patterns that perform well. Such that the optimization target is to approximate full attention scores S well enough to predict sparse patterns. We further give the results of applying oracle sparse patterns by directly dropping small-magnitude attention weights during inference without fine-tuning the model. As listed in Table 3.1, around 90% (up to 97%) of small attention weights can be dropped with negligible accuracy loss.

Table 3.1: Sparsity in attention weights, where values $< \theta$ are set to zero. A significant portion of attention weights that have small magnitude are redundant. The accuracy metrics are Exact Match (EM) and F1 Score.

Case	Sparsity	EM	F1
Base	0%	81.49	88.70
$\theta = 0.001$	75% - 95%	81.50	88.70
$\theta = 0.01$	94% - 97%	80.51	87.85

3.2.2 Model Adaptation

When sparse attention scores are masked out to generate sparsity in attention, the remaining attention weights, i.e., the important weights, are scaled up as the denominator becomes small. Leaving the disturbed attention weights intact will degrade model quality. As a countermeasure, we propose to fine-tune model parameters with dynamic sparse constraints, referred to as model adaptation. With adaptation, the model evaluation accuracy can recover to be on par with full attention baselines, while the computational costs are significantly reduced.

We do not change the computational graph and the loss function of the original model, except adding dynamic sparse constraints in attention as mask M. As a result, the new attention $\bar{A}$ are sparse and only have important weights from prediction. Given a pre-trained model, our method jointly fine-tunes the model parameters and parameters of the prediction path as in

$$L = L_{Model} + \lambda L_{MSE} \tag{3.7}$$

where λ is the regularization factor of MSE. Our method can also train from scratch with initialized model parameters.

Our method approximates the original attention score with a low-rank matrix $\tilde{S}$. When training the model with loss function in Eq. 3.6, the gradient from L_{MSE} will be passed to both the low-rank approximation $\tilde{S}$ and the original attention score S. Intuitively, this loss function not only makes $\tilde{S}$ a better approximation of S, but also makes S easier to be approximated by a low-rank matrix, i.e., by reducing the rank of S. On the other hand, the loss L_{Model} guarantees the rank of S to be high enough to preserve the model accuracy. In other words, the joint optimization of L_{Model} and L_{MSE} implicitly learns a low-rank S with a learnable rank depending on the difficulty of the task. Our design brings two advantages. First, the rank of S will be automatically adjusted to tasks with different difficulty levels. Hence, our method can potentially achieve higher accuracy on difficult tasks and higher speedup on simple tasks compared with low-rank approximation methods using fixed rank. Second, as the rank of $\tilde{S}$ only implicitly influences the rank of S, the final result is less sensitive to the hyper-parameter k.

3.2.3 Computation Saving Analysis

DSA introduces additional computations in the prediction step, but the overall computation saving from sparse attention kernels is fruitful and can have practical speedup. The original full attention takes $O(l^2 d_k + l^2 d_v)$ MACs (multiply-and-accumulate operations) asymptotically. However, the asymptotic analysis does not consider practical concerns such as sparsity, quantization, and data reuse. Here, we augment the traditional asymptotic analysis with a sparsity factor α and a quantization factor β. In this way, DST prediction takes $O(\beta l d_k k + \beta l^2 k)$ MACs; DST attention takes $O(\alpha l^2 d_k + \alpha l^2 d_v)$ MACs. Both α and β are determined depending on tasks and underlying hardware platforms. In our settings, we choose α between 90% and 98% and our GPU kernels can achieve practical speedups. We assume the baseline model uses FP32 as the compute precision and set prediction precision to be INT4. The execution time on $softmax$ is not revealed in asymptotic analysis but is one of the major time-consuming components. Our method can also save the time of $softmax$ kernel with the same sparse attention patterns.

3.2.4 Implications for Efficient Deployment

Compared with standard attention, DSA exhibits two new features that can potentially affect model deployment. Firstly, a light-weight prediction path is attached to the attention layer to search for dynamic sparse patterns. The prediction involves approximation of attention scores, which is essentially a low-precision matrix-matrix multiplication (GEMM). While NVIDIA GPUs with Tensor Cores support data precision as low as

INT8 and INT4, DSA prediction can tolerate INT2 computation on certain benchmarks. Therefore, specialized hardware is preferable when seeking ultra-efficient attention estimation.

Secondly, the predicted sparse patterns can be used to reduce unnecessary attention computations. In other words, instead of computing $QK^\top$ and AV as two dense GEMM operations, we can reformulate $QK^\top$ as a sampled dense dense matrix multiplication (SDDMM) and AV as a sparse matrix-matrix multiplication (SpMM). When processing SDDMM and SpMM kernels on GPU, data reuse is the key disadvantage that limits its performance compared with GEMM. Therefore, we extend DSA to support structural sparsity that can improve the data reuse of both SDDMM and SpMM kernels. We implement customized kernels that take advantage of the sparsity locality to improve kernel performance, achieving practical runtime speedup on NVIDIA V100 GPU. Also, we demonstrate our choice of structural sparsity pattern and that DSA is able to maintain the model expressive power with the extra constraints.

As for specialized hardware, the advantage of DSA can be fully exploited as the specialized architecture and dataflow is able to deal with fine-grained sparsity, therefore achieving optimal sparsity ratio and computation reduction. However, the challenge also arises as irregular sparsity causes load imbalance and under-utilization of processing elements. Moreover, instead of independently executing SDDMM and then SpMM, we point out that more optimization opportunities can be explored when considering the whole process as a two-step SDDMM-SpMM chain.

3.3 Algorithmic Evaluation

In this section, we evaluate the performance of DSA over representative benchmarks from Long-Range Arena [71]. We first compare the model accuracy of DSA with dense vanilla transformers and other efficient transformer models. Then, we present a sensitivity study over different configurations of the prediction path. By choosing different number of prediction parameters, DSA is able to achieve flexible trade-offs between computational cost and model accuracy. Finally, we study the model efficiency of DSA by analyzing the computational cost (MACs) and relative energy consumption.

3.3.1 Experiment Settings

The datasets used are from Long-Range Arena (LRA), which is a benchmark suite for evaluating model quality under long-sequence scenarios. In LRA, different transformer models are implemented using Jax [1] API and optimized with just-in-time ($jax.jit$) compilation. We implement DSA on top of the vanilla transformer provided by LRA and compare it with other models included in LRA. Specifically, the self-attention layer in the vanilla transformer is augmented by the DSA method as described in Section 3.2. All the other model configurations are kept the same for a fair comparison.

We incorporate three tasks from the LRA benchmark in our experiment, including Text Classification, Document Retrieval, and Image Classification. The Long ListOps and Pathfinder tasks are excluded. We provide benchmark descriptions and experiment configurations in the supplemental material.

3.3.2 Accuracy

Figure 3.3 presents the overall model accuracy of DSA on different LRA tasks. In this experiment, the DSA model is fine-tuned from a pre-trained vanilla transformer by jointly updating the model parameters and prediction parameters using the combined loss of L_{MSE} and L_{Model}. Different percentage numbers indicate the sparsity ratio that

[1] https://github.com/google/jax

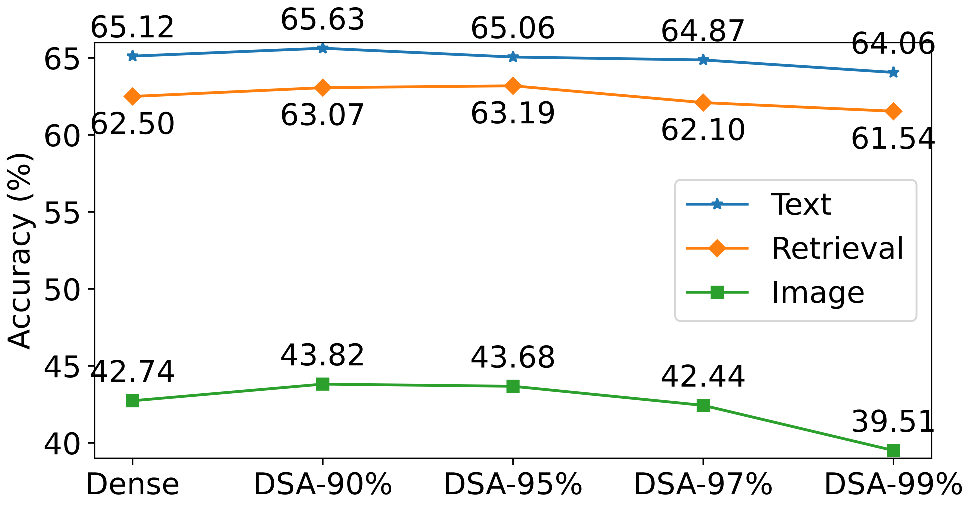


Figure 3.3: Overall model accuracy of DSA (**fine-tuned** from a pre-trained checkpoint) compared with vanilla dense transformer.

we applied to the DSA models. For instance, **DSA-90%** means that we only keep 10% of the attention weights in each row of the attention matrix, while masking out all the other 90% of the weights. The sparsity ratio constraint is uniform for all the heads and attention layers in the DSA model.

As shown in Figure 3.3, for all the evaluated tasks, dense transformer possesses a considerable amount of redundancy in the attention matrix under the long-sequence condition, which supports our previous claim in Section 3.1. Specifically, we can safely mask out up to 95% of the attention weights without suffering from any accuracy degradation. In fact, by jointly optimizing the model parameters to adapt dynamic sparse attention, DSA delivers slightly higher performance with 90% and 95% sparsity ratio. Even with up to 99% of sparsity, DSA still demonstrates promising performance with negligible accuracy drop compared with the dense baseline. We also compare DSA with other efficient Transformer methods on the LRA benchmark and show that DSA maintains a strong performance compared with related work.

The encouraging performance of DSA mainly comes from two aspects. Firstly, joint

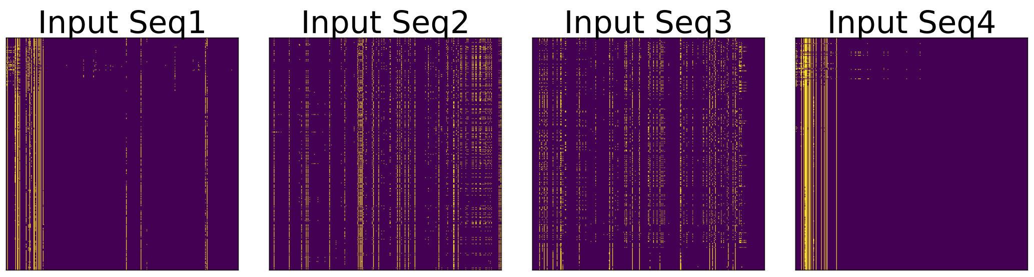


Figure 3.4: *Oracle* attention mask generated by *top-k* selection.

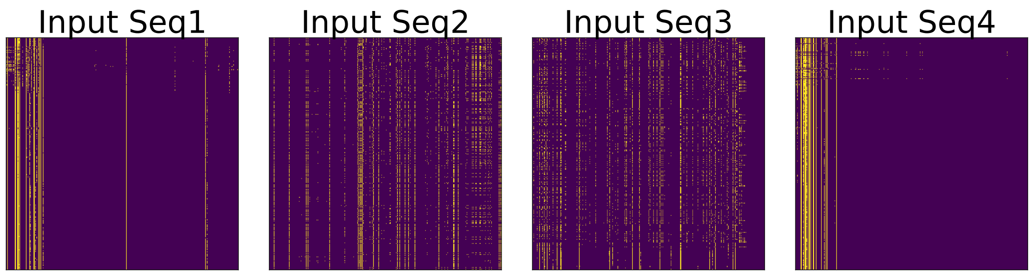


Figure 3.5: Sparse attention mask generated by DSA prediction.

optimization ensures that the DSA model can well adapt to the sparse attention patterns for computing the attention output. Secondly, the trainable prediction path is able to accurately capture the input-dependent patterns. Figure 3.4 shows the *oracle* sparse patterns of four different input sequences obtained from *top-k* selection over the original full attention matrix. The yellow dots indicate that the important positions in the attention matrix, while the purple region is masked out. Figure 3.5 shows the sparsity patterns generated by DSA prediction. As we can see from the two figures, horizontally, the sparse attention pattern changes with different input sequences. Vertically, the predicted patterns are very close to the *oracle* patterns. In our experiments, the prediction accuracy is around $85 \sim 95\%$.

To make sure the high performance of DSA comes from our proposed approach rather than the pre-trained model itself, we further test two cases on the Text Classification

Table 3.2: Change of **DSA-90%** model accuracy when sweeping random projection scale σ and quantization precision.

σ	**0.1**	**0.16**	**0.2**	**0.25**	**0.33**	**Base**
DSA-90%	65.32	65.25	65.17	65.46	65.63	65.12
Precision	**Random**	**INT2**	**INT4**	**INT8**	**FP32**	**Base**
DSA-90%	60.42	64.23	65.56	65.69	65.63	65.12

dataset. Firstly, we apply a 99% sparsity constraint on the vanilla transformer, but with a static local attention pattern. Secondly, we use a short sequence with dense attention, and let the total number of tokens in the short sequence matches with the number of important tokens in the long-sequence scenario. The results show that these two cases perform very poorly on the task, delivering a model accuracy of only 53.24% and 54.16% compared with 64.04% accuracy achieved by **DSA-99%**. This further supports our previous discussion.

3.3.3 Design Space Exploration of Prediction Path

One of the most important design choices of DSA is the configuration of the Prediction Path. Overall, we want the predictor to accurately capture dynamic sparse patterns. However, we also want to minimize the cost of prediction while maintaining DSA model accuracy. Thus, while we involve trainable parameters for prediction, we also introduce random projection matrix $P \in \{-1, 0, 1\}^{d \times k}$ to control the prediction parameters ($\tilde{W_Q} \in \mathbb{R}^{k \times k}, \tilde{W_K} \in \mathbb{R}^{k \times k}$), and use low-precision to reduce the computation overhead. Here, we present the sensitivity results regarding different choices of the reduced dimension size and quantization precision.

We first sweep over different sizes of k and evaluate the accuracy of **DSA-90%** on the LRA Text Classification task. Here, we use $\sigma = k/d \in (0, 1]$ to represent the size of the predictor. A Larger σ value indicates more prediction parameters and better representation power, but also larger computation overhead. As we can see from Table 3.2, DSA demonstrates relatively stable performance with different σ values. Even with $\sigma = 0.1$, **DSA-90%** still achieves a slightly higher accuracy compared with vanilla transformer. We believe this is because we use predictor to indicate the positions of the important attention weights, while passing the accurate attention weights to the output. Therefore, our predictor module can tolerate highly approximate computation as long as it can capture the relative importance in the attention matrix.

To further study the performance and the impact of the predictor, we conduct another experiment to sweep over different quantization precision, while fixing σ to be 0.25. As shown in Table 3.2, **DSA-90%** achieves good accuracy with quantized precision as low as 4-bit. Accuracy degradation occurs when the precision further scales down to 2-bit. As we go deeper into the predictor module, we collect and show the prediction accuracy in each attention block of this 4-layer DSA model. The prediction accuracy is defined by the percentage of the correct guesses among the total number of predictions. For example, for a **DSA-90%** model working on a sequence length of 2000, for each row of the attention matrix, the predictor will output 200 positions to be important. If 100 of these 200 locations actually matches with the *top-k* results, the prediction accuracy is 50%. As shown in Figure 3.6, the predictor is able to maintain its prediction accuracy even with 4-bit quantization. When the precision is 2-bit, the prediction accuracy suffers a significant degradation, dropping from $60 \sim 90\%$ to $25 \sim 55\%$. Despite this, the overall model accuracy is acceptable, with only 0.89% degradation compared with the baseline transformer. We believe this is because, for the binary Text Classification task, it is more crucial to capture the very few most important attentions. Although the prediction accuracy becomes lower, the most important positions are preserved and therefore maintaining the overall model accuracy. Finally, in Figure 3.6 and Table 3.2 we include a special case of randomly selecting 10% important positions. With this random

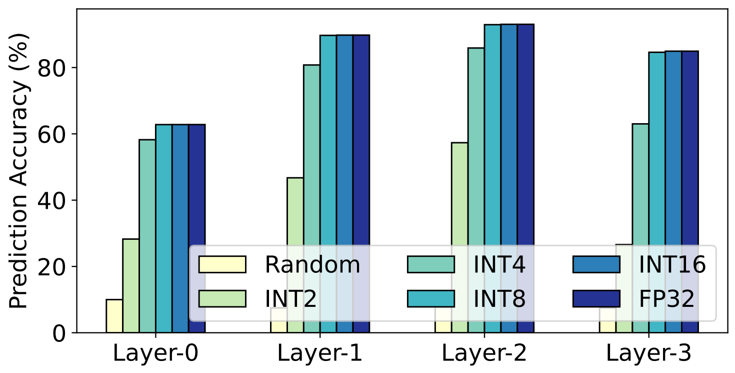


Figure 3.6: The prediction accuracy of DSA in a 4-layer **DSA-90%** model with different quantization precision.

mask applied to the model, the prediction accuracy is less than 10%, and overall model accuracy directly drops to 60.42%. This result supports our previous analysis.

3.3.4 Model Efficiency

Before diving into the implementation of DSA on different hardware platforms, we first provide a theoretical analysis to illustrate the model efficiency. We start with presenting the number of required MAC operations for each attention layer. We use MAC number as the computational cost metric because the majority of the operations in the self-attention layer are matrix multiplications. We break down the total MAC operations into three parts: (1) Linear: General Matrix-matrix Multiplication(GEMM) for computing Query, Key, and Value. (2) Attention: GEMM for computing attention weight matrix and output Value. (3) Other: Other GEMMs inside the attention block like Feed-Forward layers. As we introduced earlier, the two GEMM operations in the part (2) scale quadratically with the sequence length, and we transform them to be SDDMM and SpMM in our DSA model to reduce both computation and memory consumption. Based on this setting, the computational cost breakdown of different models used in our LRA experiment is shown in Figure 3.7. Comparing different tasks, the tasks with longer sequence length

(Text and Retrieval) are more bounded by the Attention part. The benefit of using DSA is also more significant on the 4K tasks. Comparing within each task, it is obvious that DSA model with higher sparsity ratio delivers higher computation savings. Overall, DSA achieves $2.79 \sim 4.35\times$ computation reduction without any accuracy degradation.

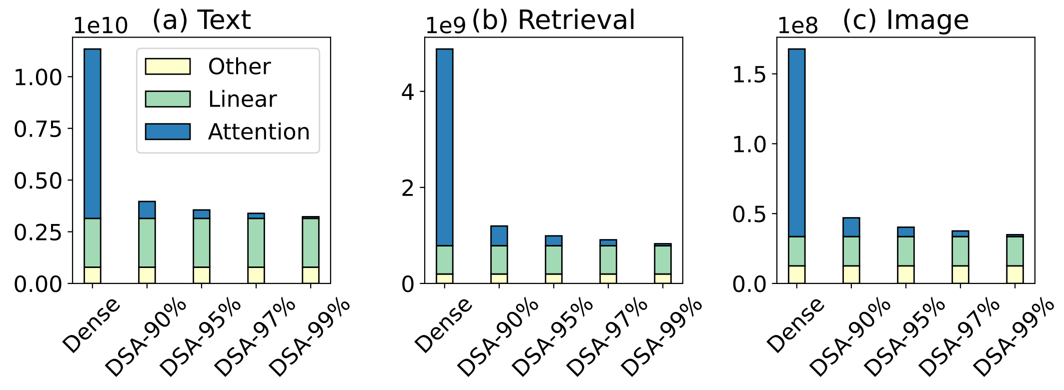


Figure 3.7: Computational cost measured in the number of MACs.

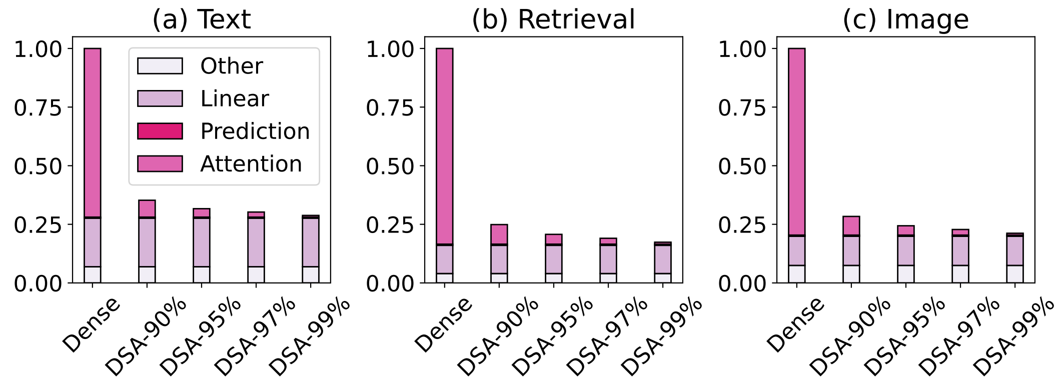


Figure 3.8: Relative energy consumption projected to vanilla transformer.

Note that we do not include the computation overhead of the prediction path for generating the sparsity mask. This is because the computations conducted in prediction are in reduced precision rather than full-precision. Besides, it is inappropriate to directly project the number of low-precision MACs to the number of FP32 MACs.

Furthermore, we show the the energy consumption of DSA relative to the dense attention. We use **DSA-95%** as an example and choose $\sigma = 0.25$ and INT4 quantization. Each INT4 MAC's energy cost is projected to the relative factor of FP32 MAC, where the factor number is referenced from industry-level simulator [75] with 45nm technology. From the figure we can see that, even with the predictor overhead considered, the overall benefit is still compelling by virtue of the high dynamic sparsity ratio and low-cost prediction.

3.4 Conclusion

In this chapter we introduce an approximation-based algorithm to explore dynamic computational redundancy in Transformer Neural Networks. By jointly optimizing the proposed attention detector and Transformer model, we are able to achieve more than 90% of attention sparsity with zero accuracy degradation. We also provide a theoretical analysis of the proposed algorithm regarding MAC reduction and energy consumption reduction.

Chapter 4

Software Hardware Co-design for GPU-based Efficient Transformer Acceleration

In this Chapter, we discuss the challenge of implementing DSA on GPUs and provide our solution to achieve practical speedup using dynamic sparse attention.

4.1 GPU Deployment of DSA

In Section 3.3.4, we analyze the potential of DSA in terms of reducing the total cost of Transformers. While the estimated number of MAC operations and relative energy consumption present very promising results, it remains challenging to achieve practical speedup and energy reduction on real hardware systems. In this section, we dive deeper into this problem as we discuss the implementation of DSA on GPUs. Specifically, we evaluate the challenge of mapping DSA onto GPU architectures, and demonstrate the flexibility of DSA to enable efficient algorithm-hardware co-designs.

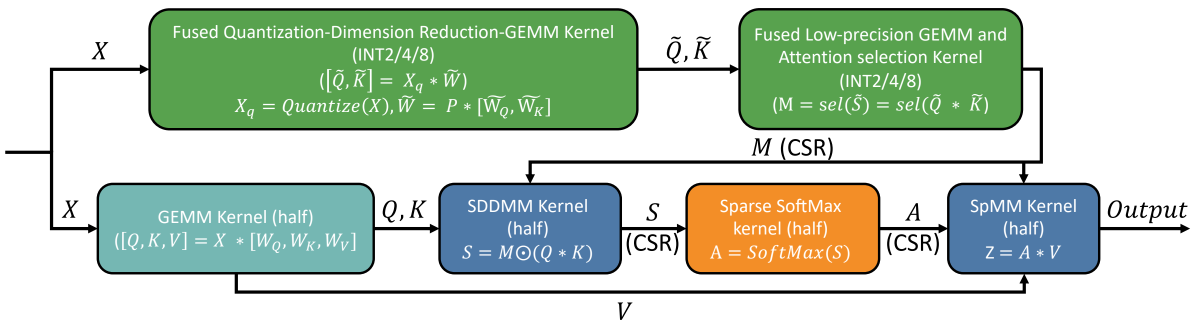

Fused Quantization-Dimension Reduction-GEMM Kernel (INT2/4/8)
$([\tilde{Q}, \tilde{K}] = X_q * \tilde{W})$
$X_q = Quantize(X), \tilde{W} = P * [W_Q, W_K]$
Fused Low-precision GEMM and Attention selection Kernel (INT2/4/8)
$(M = sel(\tilde{S}) = sel(\tilde{Q} * \tilde{K})$
$\tilde{Q}, \tilde{K}$
X
M (CSR)
GEMM Kernel (half)
$([Q, K, V] = X * [W_Q, W_K, W_V]$
Q, K
SDDMM Kernel (half)
$S = M \odot (Q * K)$
S (CSR)
Sparse SoftMax kernel (half)
$A = SoftMax(S)$
A (CSR)
SpMM Kernel (half)
$Z = A * V$
$Output$
V

Figure 4.1: Overall GPU operator pipeline of DSA. $\odot$ indicates element-wise multiplication.

4.2 DSA Operator Pipeline

The GPU execution pipeline of DSA is shown in Figure 4.1. We first perform quantization over the input feature map X. The quantized feature map X_q is multiplied with low-precision weight matrices $\tilde{W_Q}$ and $\tilde{W_K}$ to generate approximated Query and Key matrices $\tilde{Q}$, $\tilde{K}$. We implement $\tilde{Q} * \tilde{K}$ with a low-precision GEMM kernel to generate the approximated attention weights $\tilde{S}$. Important attention connections are selected based on the values of $\tilde{S}$. The selected attention locations are represented as a CSR matrix M. Given this attention sparsity, we can reformulate $QK^\top$ as the sampled dense dense matrix multiplication (SDDMM) and AV as the sparse matrix-matrix multiplication (SpMM). Various open source sparse matrix multiplication kernels [76, 77, 78] can be leveraged to perform these two operations.

There are mainly two challenges when implementing DSA on GPU. Firstly, the latency of attention approximation and selection needs to be carefully handled, such that it can be fully covered by the savings from sparse matrix computation. Secondly, due to irregular memory access and low data reuse, sparse matrix operations are difficult to significantly outperform dense matrix operations on state-of-the-art GPU systems [76]. Thus, we need to adapt algorithms to help improve kernel efficiency, while satisfying model accuracy requirements. Based on this analysis, we present the implementation details of each kernel within the execution pipeline.

4.3 Quantization and Random Projection

Normally, a quantization operator can be implemented as an epilogue of the previous kernel, such that the previous kernel directly outputs quantized results. However, since we also need the un-quantized feature map X to compute sparse attention, in DSA we fuse quantization with the subsequent kernel instead of the preceding kernel. Specifically, as shown in Figure 4.1, the quantization is performed together with random projection and $\tilde{Q}, \tilde{K}$ computations. Each time we load a tile of X from global memory to GPU shared memory. After quantizing this tile, we keep it inside the shared memory to be directly used by the following operators. Therefore, we can save a round trip to the global memory for the quantized feature map.

Furthermore, as shown in equation 3.5, X_q is back to back multiplied with random projection matrix P and low-precision weight matrices $\tilde{W_Q}$, $\tilde{W_K}$. Since both P and $[\tilde{W_Q}, \tilde{W_K}]$ are fixed during inference, we can pre-compute the result of $P * [W_Q, W_K]$ and store it as a equivalent weight matrix $\tilde{W}$. Therefore, the consecutive linear transformations can be represented by a single low-precision weight matrix.

To sum up, as shown in Figure 4.1, by applying pre-computation and quantization fusion, we are able to generate $[\tilde{Q}, \tilde{K}]$ using input X with just one single kernel.

4.4 Attention Approximation and Selection

As shown in Figure 4.1, we compute the approximated attention weights using a low-precision dense GEMM kernel. The precision is together decided by model tolerance and hardware support. While DSA achieves no accuracy degradation with INT4 computation, Nvidia V100 GPU only offers INT8 arithmetic. On the contrary, the latter GPU architectures such as Ampere and Turing offer INT4 support, and are able to further exploit algorithm potential. After generating the approximated attention values, we perform attention selection by either comparing the results with pre-trained thresholds or by perform local top-K selection. Kernel fusion is leveraged again, as we directly do the pruning right after the results are computed and cached in shared memory. Therefore, the kernel only outputs the sparsity mask without having to write the approximate attention values.

4.5 Sparse Attention Computation

The key enabler for DSA's performance improvements is the use of sparse GPU kernels for attention computation. Specifically, we use SDDMM kernel to compute attention score, and use SpMM kernel to compute the final attention output. There are multiple open source implementations can be utilized. For example, under fine-grained sparsity and single precision, the SDDMM and SpMM kernel proposed in [79] can outperform dense GEMM kernel under $> 71\%$ and $> 90\%$ sparsity, respectively. Besides, *cusparse* [78] also achieves practical speedup at $> 80\%$ sparsity for single precision data.

However, it is far from enough to solely beat single precision GEMM operation, because inference engine is usually deployed under half (FP16) precision using advanced Tensor Core architectures. When half precision (FP16) is used for computation, above fine-grained kernels can hardly compete with GEMM kernel [76]. Consequently, the performance gain on sparse matrix multiplication can hardly mitigate the overhead of computing the prediction path in DSA. To tackle this problem, structural dynamic sparsity can be introduced to the attention selection. Specifically, instead of selecting independent attention weights, we can enforce block-wise and vector-wise constraints. Also, trade-off

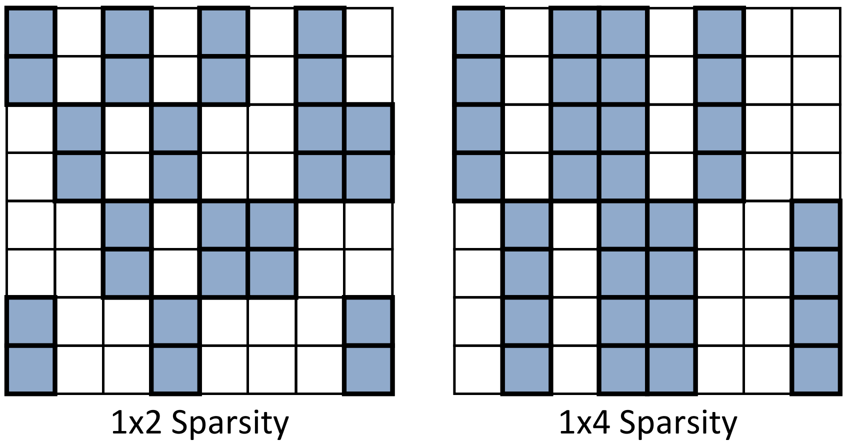


Figure 4.2: Column-vector sparse encoding [76].

can be made by adjusting the block size, as larger blocks deliver higher speedup but can potentially cause accuracy loss.

In our work, we experiment on vector sparsity using the Text Classification benchmark. The performance statistics can be generalized to other benchmarks as long as a similar model configuration and comparable sparsity ratio can be guaranteed. As shown in Figure 4.2, we choose column-vector sparse encoding, where the attention elements are pruned in a column-vector granularity. Column-vector sparsity provides the same data reuse as block sparsity, but its smaller granularity makes it more friendly to model training [76]. Table 4.1 gives the corresponding kernel speedup and model accuracy under 90% sparsity ratio. The data type is FP16 for 1×4 and 1×8 sparsity and FP32 for fine-grained sparsity. As we can see, DSA can be flexibly combined with different sparsity patterns, achieving practical runtime speedup on GPU while maintaining on-par model accuracy with full attention.

To shed some light on the results, we can trace back to the visualizations of the attention matrix in Figure 3.1. As shown by the figure, despite the sparse and dynamic characteristics of the attention matrix, the distribution of important attention connections exhibits a certain degree of locality. For example, there exist some global tokens

Table 4.1: Model accuracy and kernel speedup over *cuBLASHgemm*. We implement customized SDDMM/SpMM kernel for $1 \times 4/1 \times 8$ sparsity and reuse the kernel in [79] for fine-grained sparsity. Experiments are done on NVIDIA V100 GPU. Baseline accuracy is 65.63.

Sparsity Pattern	vec 1×4	vec 1×8	Fine-grained
SpMM Speedup	1.57×	1.94×	1.85×
SDDMM Speedup	0.94×	1.15×	1.09×
Accuracy(%)	-0.02	-0.1	+0.5

that attend to most of the tokens within a sequence. Therefore, some columns of the attention matrix will contain many important positions. Besides, local attention also indicates row-wise locality, as a token is likely to be influenced by its neighbors. Therefore, row-vector sparsity can be added to DSA for performance/accuracy exploration as well. While these fixed locality patterns have been well discussed in prior work [69, 68], DSA illustrates the dynamic distribution which motivates us to propose the prediction path to efficiently locate these important connections.

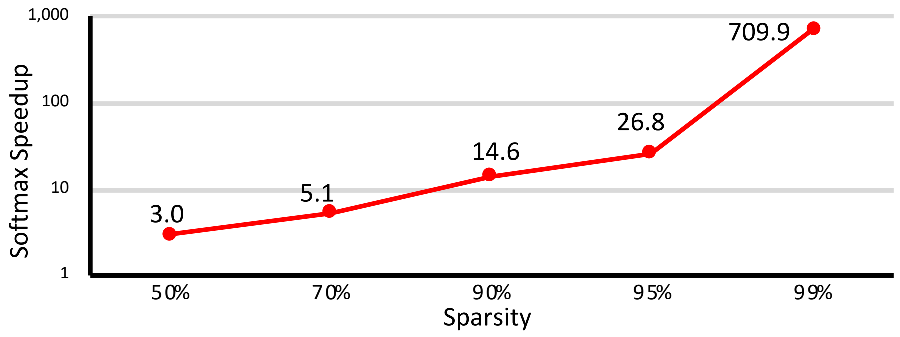


Figure 4.3: Speedup of softmax with different sparsity ratios.

4.6 Sparse Softmax Computation

Under the long-sequence scenario, the softmax function could be a bottleneck. Let h, l, and d be the number of head, sequence length, and feature dimension of each head, respectively. Our profiling result shows that with $h = 8$, $l = 4096$, $d = 64$, softmax contributes 47% of the total execution time of the multi-head self-attention layer. By sparsifying the attention matrix, DSA directly saves both memory access and computation consumption of the softmax function to reduce execution time. We evaluate the latency of the pytorch-implemented softmax function on NVIDIA V100 GPU. Following the configuration in Text Classification Benchmark, we set batch size=16, $h = 4$, $l = 2000$ and enforce different sparsity ratios. Figure 4.3 shows that the reduced softmax achieves $3.0 \sim 709.9\times$ speedup compared with dense softmax function.

4.7 Evaluation and Comparison

With the above implementation strategy, we demonstrate the overall performance of DSA on an Nvidia V100 GPU under half precision inference scheme. Specifically, we evaluate the latency and memory footprint of the attention mechanism of DSA, and compare it with other methods, including the dense vanilla Transformer and three representative efficient transformers (Performer [80], Reformer [81], and Linformer [82]). We also experiment on different sparsity ratios of DSA. Finally, we present the end-to-end speedup of DSA over dense vanilla Transformers. We do not compare the end-to-end performance of DSA with other efficient Transformers, because the result is also affected by the design of non-attention modules. For DSA we are able to keep the other Transformer layers the same as dense Vanilla Transformer and therefore providing a fair end-to-end comparison.

As shown in Figure 4.5, for all the evaluated benchmarks, DSA with vector sparsity is able to achieve practical speedup over dense Vanilla Transformer under a 95% sparsity ratio. The result comes from all the software-hardware co-design techniques that we proposed above, including hardware-efficient attention approximation, vector-sparsity encoding, and customized kernel design. For DSA-90, the benefit of sparse computation

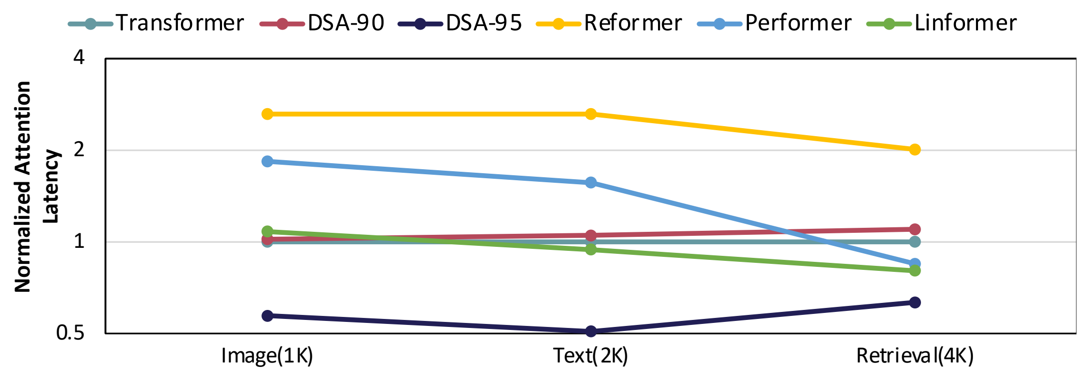


Figure 4.4: Normalized GPU computation latency of different attention methods relative to the dense Vanilla Transformer. For DSA we select 1×8 attention sparsity with sparsity ratio set to 90% and 95%.

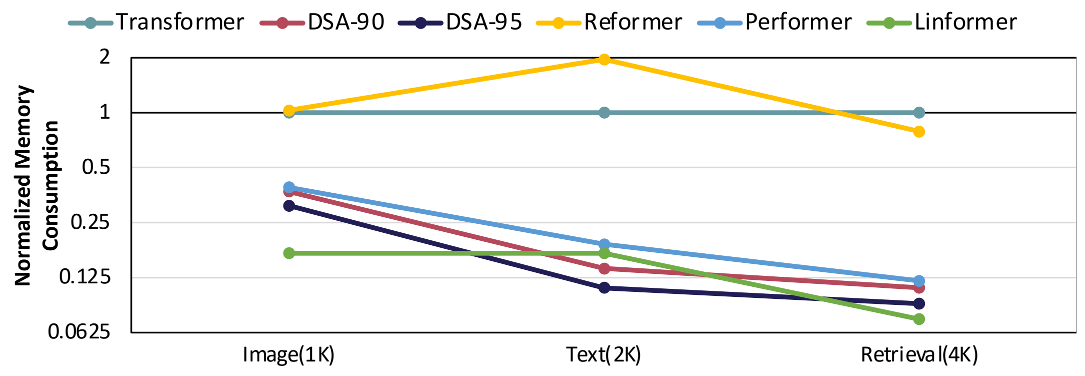


Figure 4.5: Normalized memory consumption of different attention methods relative to the dense Vanilla Transformer. For DSA we select 1×8 attention sparsity with sparsity ratio set to 90% and 95%.

failed to cover the INT8 approximation overhead, but DSA still has comparable performance as dense Transformers. On the other hand, many existing efficient Transformers, despite having a linear computation complexity, cannot deliver actual wall-clock speedup due to additional overhead caused by their customized attention mechanisms. We do observe those efficient Transformers demonstrating a good scalability, indicating their potential to be applied to ultra-long sequences, such as 8K and 16K.

As for memory consumption, we collect the peak memory allocated during the attention computation, and compare the result across different models. As shown in Figure 4.4, both DSA-90 and DSA-95 can achieve significant memory reduction compared with dense attention. Our kernel design plays a very important roll here. Although DSA computes a dense approximated attention, which is supposed to have a quadratic memory footprint. However, we perform attention selection right after the results is computed on-chip, without being written back to GPU global memory. Therefore, we avoid the most expensive data movement of DSA. For the original attention computation path, DSA reduces the memory consumption by using the CSR format for the involved sparse matrices.

Finally, we compare the end-to-end Transformer inference performance of DSA with dense vanilla Transformer on the Text Classification benchmark. We evaluate both 1×4 and 1×8 vector sparsity with 95% sparsity ratio. As shown in Table 4.2, DSA is able to achieve a $1.19 \sim 1.71\times$ speedup with less than 0.12% of accuracy degradation.

Table 4.2: End-to-End Performance Speedup. Baseline accuracy is 65.06.

Sparsity (95%)	Self-Attention	End-to-end	Accuracy
1×4	1.23×	1.19×	-0.06
1×8	1.97×	1.71×	-0.12

4.8 Conclusion

While DSA is able to sparsify a significant part of the attention map, naively mapping the algorithm to GPU platforms is undesirable due to the overhead of attention approximation and the limited performance of sparse kernels. In this chapter, we present a complete implementation pipeline of DSA to address these challenges. By sacrificing a small portion of sparsity flexibility and ratio, we are able to improve the kernel performance to a point where an overall speedup is achievable. Experiments show that with 1×4 and 1×8 vector sparsity and 95% sparsity ratio, we can deliver a $1.19 \sim 1.71\times$ speedup with less than 0.12% of accuracy degradation.

Chapter 5

DOTA: Detect and Omit Weak Attentions for Scalable Transformer Acceleration

In this Chapter, we move one step further to discuss the opportunities of using ASIC design to fully leverage the algorithmic potential of DSA. We present DOTA, a customized architecture for efficient Transformer inference acceleration.

5.1 DOTA System Design

We present DOTA's hardware system, which is capable of performing scalable Transformer inference by efficiently utilizing the detected attention graph. We specifically address three system-level challenges. First, long-sequence Transformer models involve large GEMM/GEMV computations with configurable hidden dimensions. Therefore, to effectively execute different Transformer models, we need to disassemble the algorithm and identify the essential components. We provide abstraction of the model that helps

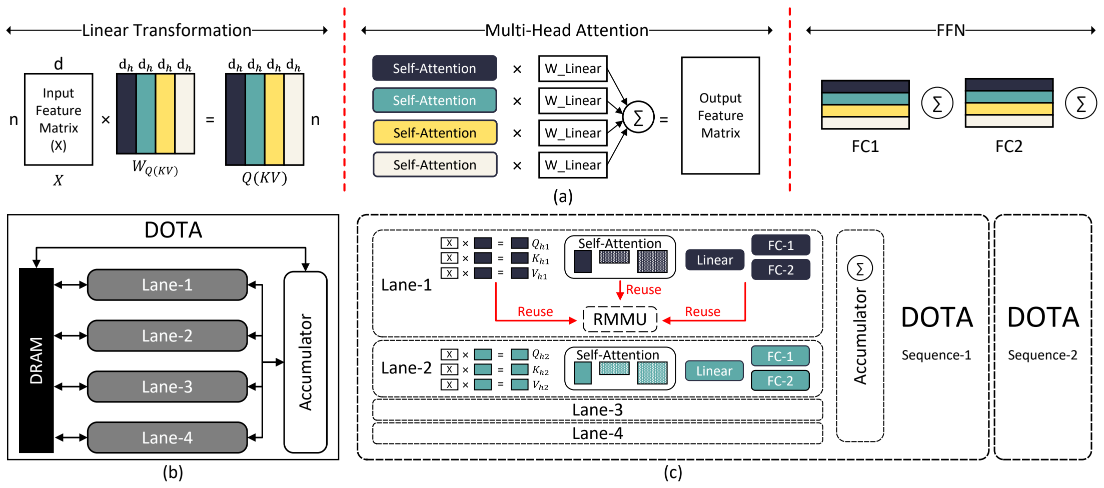


Figure 5.1: DOTA system design. (a) The abstraction of a single encoder block. We divide each encoder into three sequential stages. Each stage contains multiple GEMM operations that can be further cut into chunks (represented by different colors) and mapped to different compute Lanes. (b) Overall system design of DOTA. Each compute Lane communicates with off-chip DRAM for input feature. The intermediate results are summed up in the Accumulator. (c) Computation mapping between the algorithm and hardware. Each DOTA accelerator processes one input sequence, and each Lane computes for one chunk (color).

us to design a scalable and unified architecture for different Transformer layers, achieving good area- and power-efficiency. (Section 5.1.1). Second, apart from implementing normal precision arithmetics, DOTA also needs to support low-precision computations required by the attention detection. Instead of separately implementing all the arithmetics, a reconfigurable design would be preferred as it can dynamically balance the computation throughput of multi-precision computations. (Section 5.1.2). Finally, to efficiently compute over the detected attention graph, we should tackle the workload imbalance and irregular memory access caused by attention sparsity (Section 5.1.3).

5.1.1 Overall System Architecture

We use Figure 7.10 to illustrate the overall system architecture of DOTA, and explain how it execute a single encoder block. Running decoders can be considered as a special case of encoder with strict token dependency. As depicted by the figure, DOTA processes one input sequence at a time. Different input sequences share the same weights while requiring duplicated hardware resources to be processed in parallel. Therefore, we can scale-out multiple DOTA accelerators to improve sequence-level parallelism.

For each encoder, we split it into three GEMM stages namely Linear Transformation, Multi-Head attention, and FFN. The GEMM operations in different stages need to be computed sequentially due to data dependency, while each GEMM can be cut into multiple chunks and processed in parallel. Therefore, as shown by Figure 7.10, we locate 4 compute Lanes in the DOTA accelerator and dedicate each Lane to the computation of one chunk. For example, during Transformation stage, each Lane contains a fraction of weight W_Q, W_K, W_V and generates a chunk of QKV. We make the chunk's size equal to the attention head size h_d. Thus, for Multi-Head Attention, each Lane can directly use the chunks previously generated by itself to compute for self-attention, keeping the data local during execution. Finally, the FC layers in the FFN stage can be orchestrated in a similar way.

As we can see, different compute Lanes share the same input at the beginning of a encoder, whereas the weights and intermediate results are unique to each Lane. Therefore, we avoid data exchanging as well as intermediate matrix split and concatenation among the Lanes. An exception of the above discussion is that, at the end of Multi-Head attention and each FC layers in FFN, we need to accumulate the results generated by each Lane. In DOTA, this is handled by a standalone Accumulator. We locate four Lanes in one DOTA accelerator because 4 is the least common multiple of the attention head numbers across all the benchmarks we evaluated. More Lanes can be implemented for higher chunk-level parallelism.

Inside each Lane, as shown in Figure 5.2, there is an SRAM buffer, a Reconfigurable Matrix Multiplication Unit (RMMU), a Detector for attention selection, and a Multi-

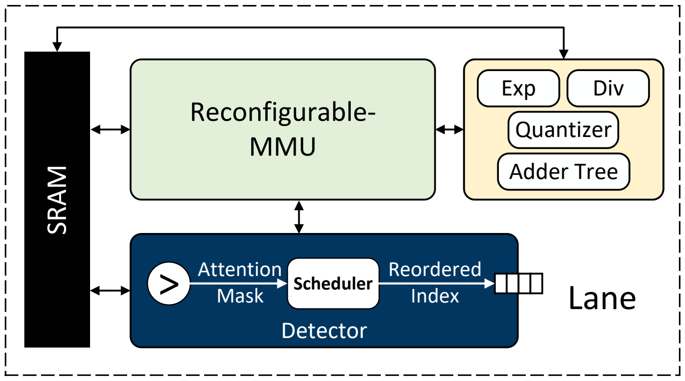


Figure 5.2: Architecture of each compute Lane.

Function Unit for special operations such as Softmax and (De)Quantization. As discussed above, one large RMMU is utilized to execute all different-precision GEMM operations in each stage. Specifically, RMMU first computes low-precision (IN2/4) estimated attention score. The low-precision results are sent to the Detector to be compared with preset threshold values for attention selection. Besides selecting important attentions to be calculated later, the Detector also contains a Scheduler to rearrange the computation order of these important attention values. We incorporate this reordering scheme to achieve balanced computation and efficient memory access (Section 5.1.3).

After obtaining the reordered attention selection results, RMMU starts to compute the attention output under FX16 precision (equation 2.2, 2.3). In order to avoid overflow during the computation, we need to dequantize the FX16 computation results of $Q * K$ into floating-point numbers before applying the softmax function. This is done in the Multi-Function Unit, and scaling factors are stored in the global SRAM buffer, which is accessible to the MFU. Thus, the exponent and division are done using floating-point arithmetic. The softmax results are quantized again to keep the consecutive computation $(A * V)$ still in fixed-point format.

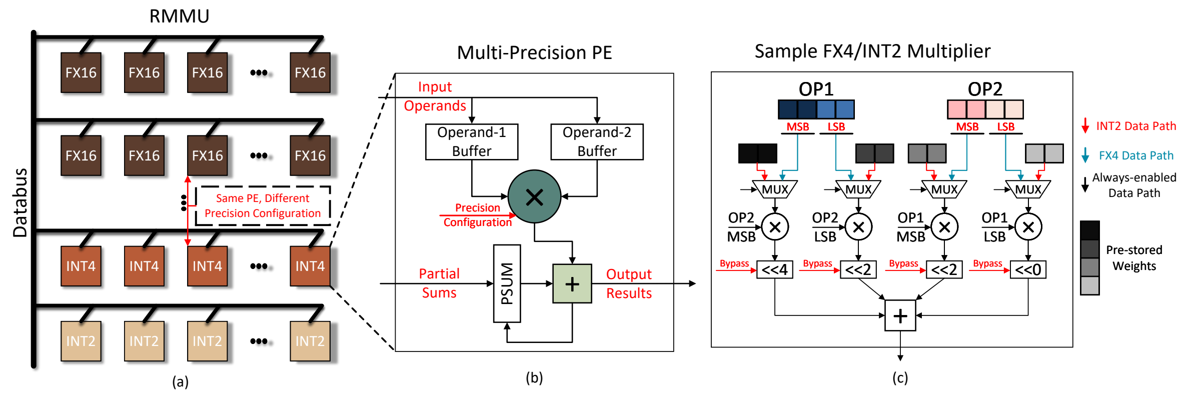


Figure 5.3: Design of the Reconfigurable Matrix Multiplication Unit. (a) RMMU is composed of a 2D PE array, where each row can be configured to a specific computation precision. (b) Each PE is a multi-precision MAC unit. (c) A sample FX4/IN2 multi-precision multiplier. The key is to build up high precision multiplication data path with low precision multipliers. In low precision mode, we split and multiply the input operands with pre-stored weights and perform in-multiplier accumulation. Therefore, the computation throughput is quadratically improved while input/output bit-width are kept the same as high precision mode.

5.1.2 Reconfigurable Matrix Multiplication Unit

As presented in Figure 5.2, each compute Lane contains a Reconfigurable Matrix Multiplication Unit (RMMU) which supports MAC operation in different precision. Low-precision computation occurs during the attention detection. Naively, we can support this feature with separate low-precision arithmetic units, but with the cost of extra resources to implement all supported precision levels. Besides, the decoupled design can only provide constant computation throughput for each precision, but the ratio of attention detection with respect to the other parts of the model varies from benchmark to benchmark. Thus, we need to dynamically control the computation throughput of attention detection and computation to achieve better resource utilization and energy-efficiency.

To tackle this problem, we present RMMU as shown in Figure 5.3. The key idea is to design computation engine with configurable precision. As we can see from Figure 5.3, RMMU is composed of a 32×16 2-D PE array, where each PE is a fixed-point (FX) MAC unit. The PE supports FX16, INT8, INT4, and INT2 computations. FX16 is used for important attention computation and the rest are for attention detection. The RMMU can be configured to different precision at a row-wise granularity. Therefore, we can flexibly control how many rows of PE use FX16 for computation and how many rows adopt low precision to balance the computation throughput.

We design the multi-precision multiplier based on two common knowledge of computing arithmetic. Firstly, a fixed-point multiplier is essentially an integer multiplier, only with a different logical explanation of the data. Secondly, we can use low-precision multipliers as building blocks to construct high-precision multipliers [83]. Without loss of generality, we present the implementation of an FX4/INT2 multiplier in Figure 5.3 (c). As we can see, each operand is divided into MSBs and LSBs and then sent to an INT2 multiplier. A INT2 multiplier takes one fraction from each operands and generates a 4-bit partial sum. Therefore, we need four INT2 multipliers to generate all the required partial sums. The four partial sums are shifted and accumulated to give the final 8-bit result. On the other hand, if the multiplier is in INT2 computation mode, the four INT2 multipliers is able to provide four times higher computation throughput. Note that, we need 16-bit input and 16-bit output each cycle to facilitate all the INT2 multipliers. However, an FX4 multiplication only requires half the bit-width (8-bit for input/output). We address this problem by keeping half the input stationary in the multiplier, and accumulate the INT2 multiplication results before sending them out. Therefore, the input bit-width is the same as FX4 computation while the output consumes 6-bit instead of 16-bit. In other words, when working on INT2 data, we utilize the multiplier as a tiny input-stationary MAC unit which can perform 4 INT2 multiplications and accumulations each cycle.

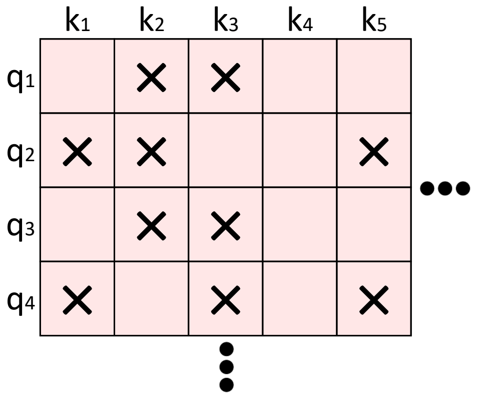
To summarize, we implement multi-precision PEs in the RMMU and ensure a scalable computation throughput when using the low-precision data. Our final design implements

FX-16 multiplier built up from low-precision INT multipliers as discussed above.

5.1.3 Token-Parallel Dataflow for Sparse Attention Computation

After RMMU generates estimated attention scores, we use the Detector unit to select important attention connections. Specifically, as depicted in Figure 5.2, the Detector loads estimated attention scores from SRAM and compare them with preset thresholds. A binary mask is generated after the comparison, with 1s representing the selected connections. The Scheduler further processes the binary mask to rearrange the computation order for each token, and stores the reordered connection IDs in the Queue. Later, RMMU will load Key and Value vectors according to these IDs to compute the attention output. Multiple tokens are processed in parallel, each corresponding to one row of the attention matrix. We name this Token-parallel dataflow, which can improve Key/Value data reuse and reduce total memory access. In this subsection, we use three different examples to demonstrate the benefits, challenges, and our solutions to compute the attention output with the detected attention graph and Token parallelism.

Token-Parallel Dataflow. As shown by the example in Figure 5.4, the 4×5 matrix is the sparse attention graph with important connections marked with crosses. Prior work process each Query (Token) one by one, meaning that the attention weights and output are computed row by row. As a result, we need to load ten keys from the memory, even though only four different keys are required. On the contrary, processing all four queries in parallel, as shown in Figure 5.4, significantly reduces the total memory accesses because some key vectors can be loaded once and shared by multiple rows. This example shows that exploring token-level parallelism benefits memory accessing when attention weight matrix has such row-wise localities. We observe similar locality in real



Dataflow	Procedure	Total Mem Access
Row-by-Row	Load k2, k3, k1, k2, k5, k2, k3, k1, k3, k5	10 Key Vectors
Token-Parallel (w/o reorder)	Load (k1, k2), Load(k2, k3), Load(k5)	5 Key Vectors

Figure 5.4: Token-level parallelism reduces key/value vector memory access.

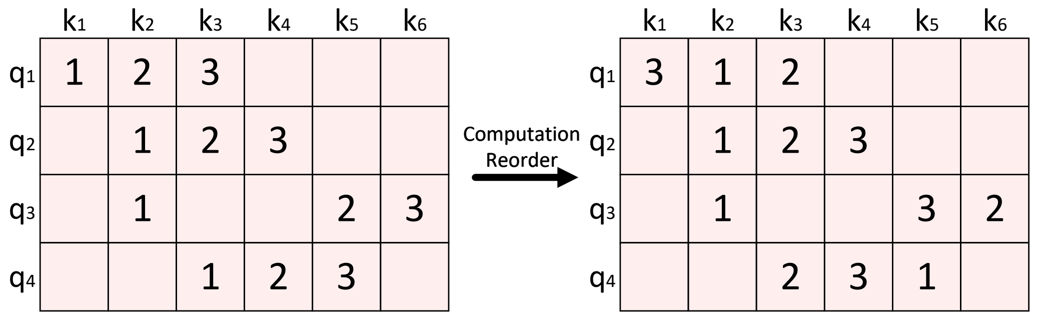
attention graphs. On one hand, there are usually some important tokens in one sentence that attend to multiple tokens. On the other hand, a token is likely to attend to its neighbor tokens within a certain window size. We perform design space exploration (see Section 5.2.5) and find that processing four queries in parallel is a good trade-off point for hardware resources consumption and memory access savings. Thus, in DOTA, each Header processes four query vectors in parallel.

Workload Balancing. One challenge of parallel token processing is the workload imbalance issue among different rows. Figure 5.4 shows that different queries may have various numbers of important key vector pairs, which may further cause resource underutilization and performance degradation. One solution is to let early-finished PEs switch to the processing of other queries. However, this will generate extra inter-PE communications as well as query reloading. Therefore, we tackle this problem directly from algorithm perspective without affecting the underlying hardware. Specifically, we add a constraint to force all the rows in the attention matrix to have the same number of selected attention connections. This constraint ensures that each vertex in the selected sparse attention graph have same number of incoming edges. We will further prove in Section 5.2.2 that the added constraint has negligible influence on model accuracy.

Out-of-Order Execution. Finally, we propose hardware-enabled out-of-order execution to further improve key/value reuse and reduce total memory access. As shown in Figure 5.5, suppose all four queries have balanced workload and are processed in parallel. With left-to-right computation order, we first compute (q_1, k_1), (q_2, k_2), (q_3, k_3), (q_4, k_3), and then (q_1, k_2), (q_2, k_3), (q_3, k_5), (q_4, k_4), and finally (q_1, k_3), (q_2, k_4), (q_3, k_6), (q_4, k_5). Consequently, some originally shared keys will have to be reloaded and the locality is broken. In this example, the required total memory access is 11 vectors, which is only one vector less compared with no parallelism.

To address this problem, we design a locality-aware scheduling algorithm to reorder the computation of each query. As shown in Figure 5.5 and 5.6, we start with issuing the keys that are shared by most queries. When scheduling partially shared keys like k_2, we also need to schedule computations for the unassigned query, which is q_4. To do so, we first look for keys that belong to q_4 alone. If not found, we move on to keys shared by q_4 and another query, and so on. In this example, there are no key vectors that are owned by q_4. Therefore, we go to the second best choice, which is k_5. Thus, in the first round, we schedule k_2 for $q_{1,2,3}$ and k_5 for q_4. Although this breaks the locality of q_5, the greedy search ensures overall minimal memory access. Besides, since each query is scheduled for exactly one connection at each round, and they have same total connections, this ensures the synchronization of each rows and maximizes resource utilization and performance. The complete scheduling algorithm is presented in Algorithm 2. Note that, the scheduling only needs to be performed once, and the generated computation order is reused for computing attention output using attention weights A and Value matrix V.

We design a Scheduler to implement the scheduling algorithm. As shown in Figure 5.6, the Scheduler first stores each connection ID in the corresponding buffer according to the 4-bit binary mask generated after threshold comparison. For example, according to



Dataflow	Procedure	Total Mem Access
Token-Parallel w/o Out-of-order Execution	Load (k1,k2,k3), Load(k2,k3,k5,k4), Load(k3,k4,k6,k5),	11 Key Vectors
Token-Parallel w/ Out-of-order Execution	Load (k2,k5), Load(k3,k6), Load(k1,k4,k5)	7 Key Vectors

Figure 5.5: Even with token parallelism, the computation order of each row still matters and affects total memory access.

Algorithm 2 Locality-Aware Scheduling Algorithm.

```
Require: A set of buffers $B$ that store the selected connection IDs for query $q_1, q_2, q_3, q_4$.
    e.g., $B_{0110}$ stores IDs that are required by $q_2$ and $q_3$.
Ensure: A computation order that achieves optimal Key and Value data reuse.
 1: Issue all the IDs in $B_{1111}$ (required by all 4 queries)
 2: while $B_{1110}$ is not empty do
 3:   Issue an ID in $B_{1110}$
 4:   if $B_{0001}$ is not empty then
 5:     Issue an ID in $B_{0001}$
 6:   else
 7:     Search and Issue an ID in $B_{xxx1}$
 8:     Move the issued ID from $B_{xxx1}$ to $B_{xxx0}$
 9:   end if
10: end while
11: Repeat 2-10 for all the other buffers.
```

Figure 5.5, '1' is stored in *buff-1000*, '2' is stored in *buff-1110.* Then, the Scheduler starts issuing computations from *buff-1111.* Besides, when k_5 is scheduled for q_4 during the step-1, '5' will be moved to *buff-0010*, meaning that now it only belongs to q_3. We use a Finite-State Machine to implement the condition statements and control logic.

In summary, we explore token-level parallelism with software-enabled workload-balancing and hardware-enabled out-of-order execution to efficiently compute the attention output. The proposed strategy can be generalized and used in other applications with the same two-step matrix multiplication chain as shown in equation 5.1. (SoftMax is optional.)

$$O = (Q * K) * V = A * V \tag{5.1}$$

More importantly, even with out-of-order execution, the final result is automatically generated in a regular order. Because the irregular computation only affects the intermediate matrix A, which is completely consumed during the computation. In contrast, exploring same reordering in CNN would require a crossbar-like design to correctly store the output result [84].

5.1.4 System Design Completeness

Decoder Processing For decoders, since the input tokens have to be processed sequentially, the core operation would be GEMV and the performance is memory-bounded. DOTA reduces total memory access by efficiently filtering out majority of the attention connections.

Memory Modules The on-chip memory is implemented as banked SRAM module that can be configured to store different types of data. We implement a custom simulator to obtain the capacity and bandwidth requirement of the SRAM module. We facilitate each Lane with a 640KB SRAM (10 64KB banks). Therefore, DOTA has a total on-chip SRAM capacity of 2.5MB. The bandwidth requirements of embedding layer and decoders are significantly higher than other layers. Therefore, we make sure the SRAM bandwidth meets the need of the computation-bounded layers, while leaving embedding and decoder to be memory-bounded.

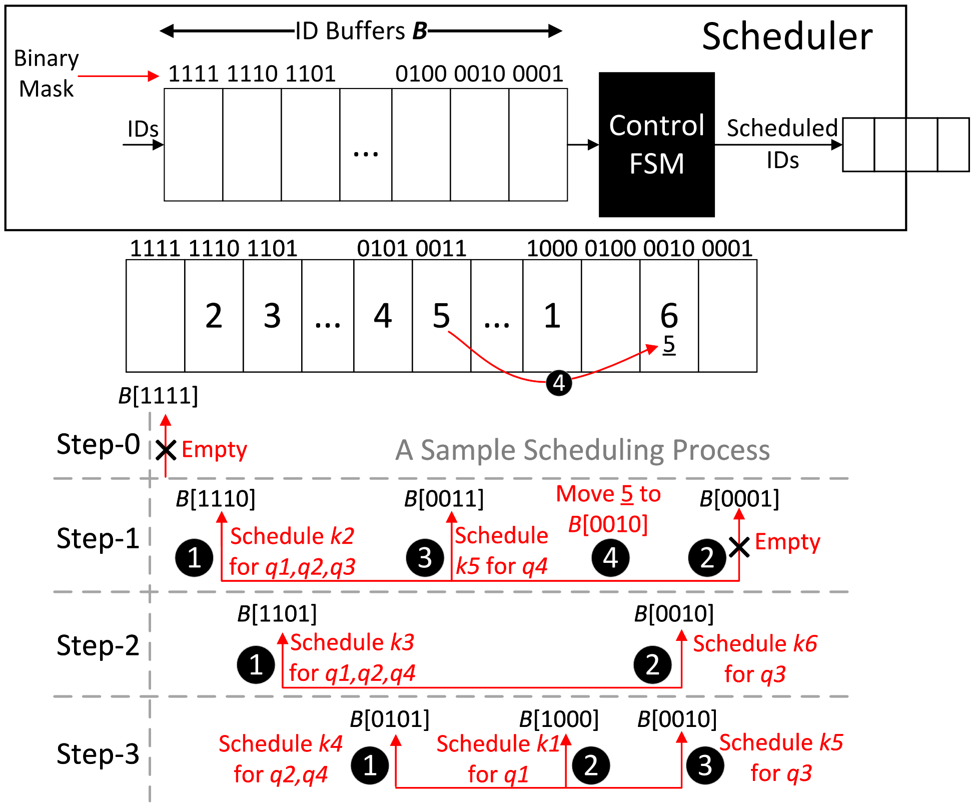


Figure 5.6: Design of the Scheduler and the scheduling process of Figure 5.5.

5.2 Evaluation

In this section we present the evaluation results of DOTA.

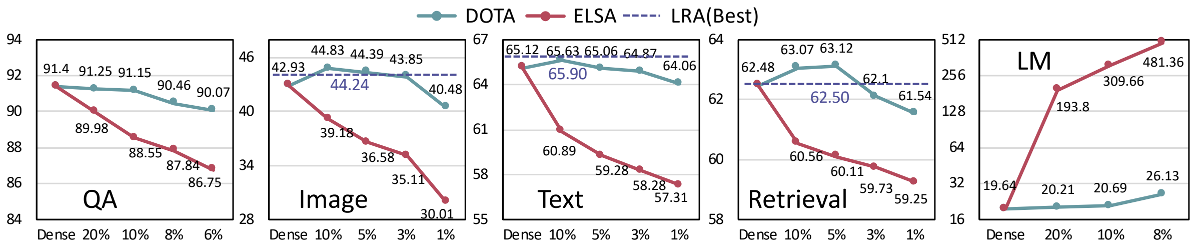


Figure 5.7: Model accuracy of DOTA comparing with dense baseline and ELSA under different retention ratios across the benchmarks. The performance metric of GPT-2 is perplexity score, the lower the better. The other dataset uses accuracy, the higher the better. The purple line indicates the best results provided by the LRA benchmark.

5.2.1 Evaluation Methodology

Benchmarks. Our experiments include series of representative Transformer benchmarks with challenging long-sequence tasks. We first run BERT (large) [2] on question answering task (QA) using the Stanford Question Answering Dataset (SQuAD) [85] v1.1 with a sequence length of 384. To scale our evaluation to longer sequences, we further select three tasks from Long-Range-Arena [86] (LRA), which is a benchmark suite tailored for long-sequence modeling workloads using Transformer-based models. Specifically, the first benchmark performs image classification on CIFAR10 [87], where each image is processed as a sequence length of 1K. The second task is a text classification problem built on the IMDb reviews dataset [88] with a sequence length of 2k. The third task aims to identify if two papers in the ACL Anthology Network [89] contain a citation link. The papers are modeled as 4k input sequences to the Transformer model. Finally, we use GPT-2 [26] to evaluate causal language modeling (LM) on Wikitext-103 [90] using sequences of 4K length.

Software Experiment Methodology. We implement our attention detection mechanism on top of each baseline Transformer, and jointly optimize the model with attention selection enabled. We study the effectiveness of our method by evaluating the model performance in terms of accuracy or perplexity with respect to the retention ratio of the sparse attention graph. Besides, we further compare DOTA's accuracy with state-of-the-art algorithm-hardware co-design (ELSA [61]) and pure software Transformer models presented in LRA [86].

Hardware Experiment Methodology. The system configuration and consumption of DOTA is shown in Table 5.1. We implement DOTA in RTL, and synthesize it with Synopsys Design Compiler using TSMC 22nm standard cell library to obtain power and area statistics. The power and area of SRAM module are simulated by CACTI [91]. We

Table 5.1: Configurations, Power, and Area of DOTA under 22nm Technology and 1GHz Frequency.

Hardware Module		**Configuration**	**Power**(mW)	**Area**(mm^2)
Lane		4 Lanes per accelerator	2878.33	2.701
Lane	RMMU	32*16 FX-16	645.98	0.609
	Filter	Token Paral. = 4	9.13	0.003
	MFU	16 Exp, 16 Div 16*16 Adder Tree	60.73	0.060
Accumulator		512 accu/cycle	139.21	0.045
DOTA (w/o SRAM)		2TOPS	3017.54	2.746
SRAM		2.5MB	0.51(Leakage)	1.690

implement a custom simulator for performance and energy-efficiency evaluation. The simulator is integrated with the software implementations of the Transformer models. We further conduct design space exploration to search for optimal system design choices.

Hardware Baselines We quantitatively compare DOTA with NVIDIA V100 GPU and ELSA [61], while qualitatively discuss the difference between DOTA and other customized hardware (See Section 5.3). When comparing with GPU, we scale up DOTA's hardware resource to have a comparable peak throughput (12 TOPS) as V100 GPU (14 TFLOPS). The energy consumption of DOTA is also re-simulated for fair comparison. When comparing with ELSA's performance, we extend and validate our simulator to support ELSA's dataflow. Then, we re-synthesize DOTA with the same data representation, computation resources and technology node as ELSA to compare the energy-efficiency.

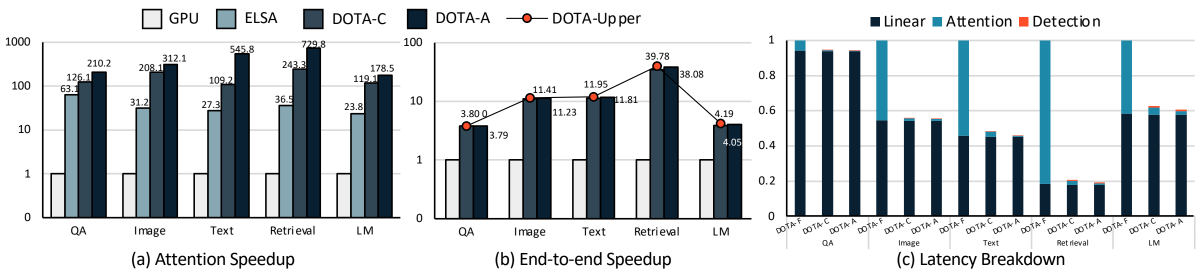


Figure 5.8: (a) Speedup of DOTA over GPU and ELSA on attention block. (b) End-to-end speedup over GPU. Red dots indicate the theoretical performance upper-bound of an accelerator. (c) Normalized latency breakdown of DOTA. DOTA-F means to compute the *Full* attention graph with DOTA without detection and omission. DOTA-C (Conservative) and DOTA-A (Aggressive) both adopt attention detection, while DOTA-C allows for an accuracy degradation less than 0.5% and DOTA-A allows for 1.5%.

5.2.2 Algorithm Performance

We present the model accuracy of DOTA in Figure 5.7, and compare it with dense Transformer model as well as other software baselines. For DOTA, we first add the row-wise attention connection constraint and then select optimal quantization precision and dimension reduction factor (σ) based on design space exploration (Section 5.2.5). For ELSA, our implementation delivers aligned results on QA compared with the original paper, and we extend it to other datasets.

As we can see, across all the tested benchmarks, DOTA is able to achieve comparable or slightly higher model accuracy compared with the dense baseline, while selecting only $3 \sim 10\%$ of the attention connections. Furthermore, DOTA significantly outperforms ELSA in accuracy-retention trade-offs. For example, on QA task with 1.5% of accuracy degradation interval, DOTA delivers 3.3× higher reduction ratio by keeping 6% of the connections, while ELSA needs to keep 20%. The gap becomes even larger on long-sequence benchmarks, which indicates that our detection method is more scalable with long sequence. Furthermore, we also provide leading results given by the LRA [86] benchmarks on image classification, text classification, and document retrieval tasks. As shown in the figure, DOTA achieves on-par or better accuracy than LRA's leading results with 5% to 10% of retention ratio.

5.2.3 Speedup

Figure 5.8 presents the speedup of DOTA over the baselines. We evaluate both stand along attention block as well as the end-to-end performance improvements. We provide two versions of DOTA by setting the accuracy degradation of DOTA-C (Conservative) to be less than 0.5%, and limiting the degradation of DOTA-A (Aggressive) within 1.5%. As for ELSA, although it fails to reach the above accuracy requirement, we follow the original setting [61] and set the retention ratio to be 20% for performance evaluation.

As we can see, comparing with GPU, DOTA-C achieves 152.6× and 9.2× average speedup on attention computation and Transformer inference, respectively. On the other hand, DOTA-A achieves on average 341.8× and 9.5× speedups at the cost of a slightly higher accuracy degradation. The speedup mainly comes from three aspects. Firstly, DOTA benefits from highly specialized and pipelined datapath. Secondly, the attention detection mechanism significantly reduces the total computations. Finally, the Token-parallel dataflow with workload balancing and out-of-order execution further improves resource utilization.

The end-to-end speedup is lower than that of attention computation, since the proposed detection method is tailored to the cost reduction of self-attention blocks. We add another baseline by assuming the accelerator always works at its peak throughput, and the attention computation has a ignorable cost. Combining this peak throughput assumption and Amdahl's law [92], we can derive the theoretical speedup upper bound for DOTA. As we can see, the real performance of DOTA is relatively close to the upper bound by virtue of the extremely small retention ratio and hardware specialization. We only compare DOTA and ELSA on attention computation performance, because ELSA does not support end-to-end Transformer execution. As we can see from Figure 5.8 (b), on average, DOTA-C is 4.5× faster than ELSA and DOTA-A is 10.6× faster. This improvements mainly come from lower retention ratio and Token-parallel dataflow.

The latency breakdown in Figure 5.8 (c) delivers two key messages. Firstly, the latency of attention estimation is negligible compared with the overall consumption. Therefore, the Detector is both accurate and hardware efficient as we expected. Secondly, with the proposed detection method and system architecture, the cost of attention has been significantly reduced. The new performance bottleneck is Linear computation, which can be optimized with weight pruning and quantization. These classic NN optimization techniques can be fluently transplanted on DOTA, because our system is designed on top a GEMM accelerator with multi-precision arithmetic support and sparse computation dataflow. Overall, DOTA delivers scalable Transformer inference acceleration.

5.2.4 Energy-Efficiency

As shown in Table 5.1, each DOTA accelerator consumes a total power of 3.02W. RMMU and Accumulator are the two major contributing factors to the dynamic power consumption, whereas SRAM and RMMU together occupies the most chip area. We compare DOTA's energy-efficiency with GPU and ELSA. The results are shown in Figure 5.9. As we can see, DOTA-C achieves 618∼5185× and 1.97∼5.14×energy-efficiency improvements over GPU and ELSA, while DOTA-A achieves 1236∼8642× and 3.29∼12.20× improvements over these two baselines. The energy saving mainly comes from two parts. Firstly, despite the attention estimation overhead, the proposed attention detection largely reduces overall cost of attention computation and memory access. Secondly,

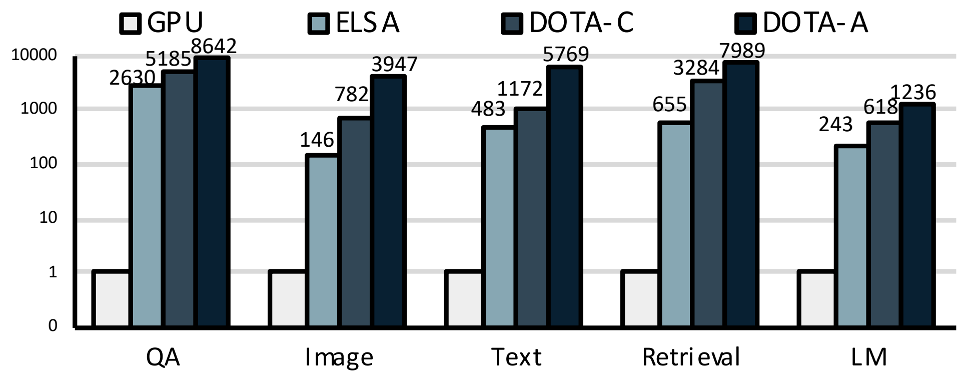


Figure 5.9: Energy-efficiency comparisons.

both external memory access and on-chip SRAM access are saved to a large extent. On one hand, the hardware specialization helps improve intermediate data reuse between the pipeline stages. On the other hand, Token-parallel dataflow effectively utilizes attention connection locality to improve Key/Value data reuse. The energy breakdown of DOTA exhibits similar pattern as the latency breakdown. That is, with effective attention reduction, FC-layer consumes around 84.9∼99.3% of the total energy cost, while attention detection only consumes 0.11∼0.34%. This further illustrates the efficiency of the proposed algorithm-hardware co-design.

5.2.5 Design Space Exploration

We search and select optimal architectural settings for DOTA through design space exploration.

Dimension Reduction Scale As discussed above, the dimension reduction scale σ directly affects the size of the input and weight matrices involved in attention detection. Therefore, a small σ can effectively control the overhead of attention estimation, but the Detector's performance will also be limited. We experiment on the Text classification benchmark, fixing the retention ratio and quantization precision while only adjusting the scale values. The results are shown in Figure 5.10 (a).

As we can see, for Text classification, the scale factor can be as small as 0.2 without affecting the overall model accuracy. Therefore, the hidden dimension in approximation is *floor*(64*0.2)=12, compared with the original dimension 64. Besides, σ is a hyper-parameter which does not influence the underlying hardware. Therefore, each benchmark can use its own optimal σ value.

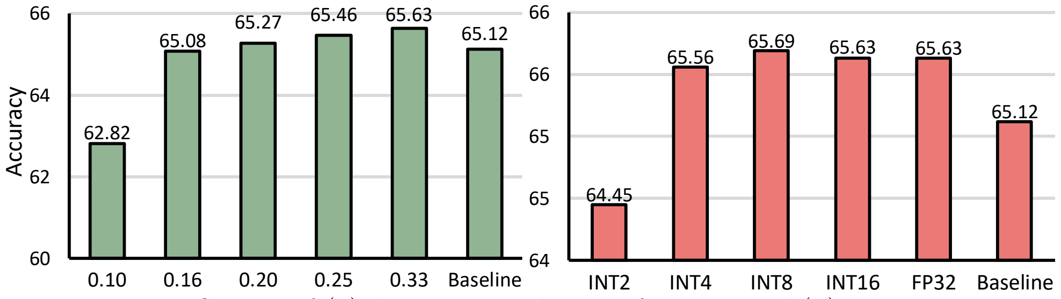


Figure 5.10: Influence of (a) dimension reduction factor σ and (b) quantization precision on overall model accuracy using Text classification benchmark. Retention ratio = 10%.

Precision of Attention Detection Another factor that affects the attention detection cost is the choice of quantization precision. Furthermore, the precision also influences the design complexity of RMMU. For each benchmark, we fix σ and retention ratio and sweep over different quantization precision. Figure 5.10 (b) presents the experiment results on Text classification benchmark. As we can see, the quantization precision could be as low as 2-bit with negligible accuracy degradation. After our experiments, we found that INT4 is a safe precision for all the benchmarks, while some can tolerate INT2 computations. Therefore, our final RMMU design supports INT2, INT4, and INT8 apart from FX16. INT8 computation is required when X, $\tilde{W_Q}$, and $\tilde{W_K}$ are INT4 data. As the estimated $\tilde{Q}$ and $\tilde{K}$ will be in INT8 precision.

Token Parallelism Our token-parallel dataflow leverages locality among important attention distribution to improve memory access. Higher parallelism increases data reuse and reduces total memory access, but also results in growing size of the Scheduler unit. Therefore, we aim to find an optimal trade-off point that achieves lowest overall energy consumption. Figure 5.11 shows the case on Text classification benchmark with Retention ratio to be 10%. The left axis indicates the normalized memory access cost of Key and Value, while the right axis is the required number of buffers in the Scheduler. Figure 5.11 mainly delivers two key messages. Firstly, as shown by the solid blue bar, leveraging row-parallelism does help reduce memory accesses, but increasing the parallelism has diminishing returns. This is because attention distribution exhibits certain but only a limited degree of locality. Secondly, increasing row-parallelism causes exponential growth in scheduling overhead. In Figure 5.11, this is shown by the red line (buffer requirement) and the dotted blue bar (scheduling energy consumption). After summing up the memory cost (solid blue bar) and scheduling cost (dotted blue bar) together, we choose the shortest one because it represents the sweetest spot with the lowest total energy consumption. As we can see, parallelism 4 has the lowest total height, which means 4 is the best setting for Text classification. We also evaluate on other benchmarks and most benchmarks have an optimal parallelism to be or around 4. Therefore, we choose 4 as the final setting in DOTA.

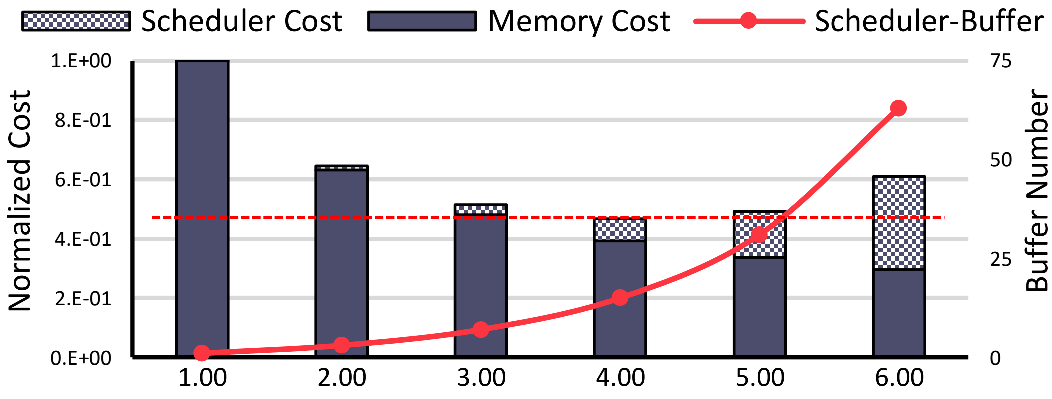


Figure 5.11: Key/Value memory access (left axis) and Scheduler buffer requirement (right axis) with different Token parallelism. The hatched area is the projected cost of Scheduler.

5.3 Related Work

In this Section, we mainly discuss related work on efficient Transformer models from the algorithm perspective and hardware accelerators for Transformers and Self-Attention. For general DNNs, quantization and low-precision support have been proposed [93, 94, 95, 96, 97]. While sharing the high-level similarity, our method focuses on attention operations that are not parameterized. Hence, those methods applied on model parameters are not applicable to our scenario. Our work is in the scope of dynamic pruning on attentions as we discussed and compared with other related work. Approximation for DNNs is also a line of related work [98, 99, 100, 101]. Finally, hardware accelerators for DNNs are related to executing the non-attention components of Transformers [102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114].

5.3.1 Efficient Transformer Models

Recent studies propose efficient variants of Transformer models to mitigate the quadratic memory complexity of long sequence modeling [81, 115, 116]. However, these methods are impractical for efficient inference as they focus on training memory footprint reduction while trading off more computations for clustering or grouping.

Another line of work exploit static or fixed sparse patterns in attention, such as local windows, block-wise, dilated, or a combination of static patterns [69, 25, 117]. However, as discussed in Section **??**, the sparse attention graphs are inherently dynamic depending on input sequences. Hence, these approaches lack the capability of capturing dynamic sparse attentions.

5.3.2 Attention and Transformer Accelerators

There have been a few recently proposed work targeting the acceleration of attention and Transformer. MnnFast [59] skips the computation of specific value vectors if its attention weights is lower than the threshold. This method can only benefit the attention output computation rather than attention weights computation. A^3 [60] is the first work to apply approximation to the attention weights for computation reduction. However, A^3 involves a sorting-based preprocessing phase that needs to be done outside the accelerator, causing inevitable performance and energy overhead. ELSA [61] improves the approximation method by directly using sign random projection to estimate the angle between query and key vectors. Although the approximation becomes much more hardware friendly, the detection accuracy and model quality is hurt. DOTA addresses all of the above limitations by simultaneously concerning detection accuracy and efficiency. In terms of hardware design, prior work only implements attention block with no token parallelism, while DOTA supports end-to-end inference acceleration with Token-parallel dataflow to improve system performance.

SpAtten [62] proposes cascade token pruning and head pruning to reduce the cost of both self-attention block and subsequent layers in the Transformer model. The proposed method can be regarded as adding structured sparsity constraints to the attention matrix, as it directly removes several rows and columns. Based on our visualization and experiments, we believe that despite a certain degree of locality, such constraint is not flexible enough to capture the irregularly distributed attention connections. As for hardware design, SpAtten supports both decoder and encoder processing, but it is also mostly tailored to attention acceleration with very few discussions on end-to-end execution.

Finally, OPTIMUS [63] proposes a GEMM architecture to accelerate Transformer inference. It focuses on accelerating sequential decoding process and proposes technique to maintain resource utilization. Although OPTIMUS avoids computing redundant attention weights, such redundancy is due to naturally existed token dependency, rather than the weak connections we discussed in this work. Thus, the self-attention still has quadratic cost and OPTIMUS does not scale on longer sequences.

5.4 Conclusion

In this work, we address the challenge of scalable Transformer inference. Specifically, we first propose algorithm optimization to reduce the quadratic cost of self-attention mechanism. Our method efficiently detects and omits weak connections in attention graphs to skip the corresponding computations and memory accesses. Furthermore, we provide system-level support for end-to-end large Transformer model inference. We first effectively abstract the Transformer model to design a scalable and unified architecture. Then, we implement the proposed attention detection method with efficient hardware specialization techniques. Our final evaluation results sufficiently demonstrate the effectiveness of the proposed algorithm and system design.

Chapter 6

Efficient Runtime Data Compression with Tensor-train Decomposition

In the previous three chapters, we introduce a complete top-down design flow to explore dynamic computational redundancy in the acceleration of Transformer models. In the following two chapters, we discuss the opportunities on the data structure side. Specifically, we apply tensor-train decompositions to the representation of the data to achieve efficient compression. Using tensor-train significantly changes the memory consumption of the model, while also imposing multiple new hardware challenges, as we will further discuss in the below content.

6.1 Introduction

Tensor is a high-dimensional generalization of vector and matrix, and is a natural choice for efficiently solving high-dimensional big data analysis problems. Compared with matrix analysis, multiway data processing is more versatile and has the potential to capture multiple interactions and couplings [118]. Previous studies have demonstrated its use in diverse branches of data analysis, such as EDA, signal and image processing, biometrics, quantum computing, and so forth [119, 120, 121, 118, 122, 123]. Nevertheless, processing big data with tensor-based approaches is challenging due to the high dimensionality and large data size. Therefore, more and more attentions are drawn to tensor decomposition to compress tensors in terms of both dimension and size, which has been playing an important role in data mining, pattern recognition, object detection and classification [124, 125, 126, 127, 128, 129, 130].

Tensor-train decomposition (TTD)[40] is one of the most popular tensor decomposition methods because of its ability in providing highly compressed tensor data while keeping significant computation accuracy with customizable constraints. More importantly, it also enables efficient data processing on the base of TT-format data. However, there are still challenges existed in TT-format data processing. The reasons are of two folds. First, obtaining the TT representation, the initial step for TT-based data processing, is time consuming because of the iterative decomposition procedure over large-scale tensor data. In each of the TTD iteration, a truncated singular value decomposition (SVD) is used to decompose a large intermediate matrix, which is both memory- and compute-intensive. Second, there is a big gap to adapt a typical algorithm to the TT-based method. On one hand, normal operations like addition and multiplication cause the TT-rank to grow significantly [40], which require us to approximate the TT result afterwards. On the other hand, some simple element-wise operations like ReLU in neural networks, can be very complicated for TT-format data, because each of the original element is now represented as a sequence of matrix multiplication. Therefore, additional efforts are needed if we want to effectively take advantage of the TT-format data analysis. Previous work have mainly focused on directly using TT-format data to perform simple computations like matrix multiplications [120, 130]. However, efficient execution of TTD itself and implementing more complicated operations in TT format are rarely touched.

In this work, we aim at addressing the mentioned problems with the following approaches. (1) To reduce the TTD overhead, we propose TTD Engine, the first customized architecture for efficient execution of the TTD algorithm. Instead of naively implementing the original TTD algorithm, we adapt it by virtue of the special high-order tensor data structure as well as data sparsity and symmetricity. (2) To bridge the gap between TT-format data and application algorithms, we move forward by proposing a decomposed computation pattern for element-wise operations and resolving the rank-growth issue with the help of TTD Engine. We conduct a case study on the base of TTD Engine to implement convolutional operations over TT-format data, which are considered to be difficult and inefficient for TT-based data processing. We show that with specialized hardware support and algorithm design, it is possible and beneficial to reformulate the existing operations in various applications using the TT format to achieve better efficiency.

Our contributions in this work are summarized as follows:

- We develop a hardware friendly computing scheme for TTD by adjusting the computation pattern of SVD within each TTD iteration. The modified SVD explores data sparsity and symmetricity during the computation process to reduce the overall compute cost.

- Based on the proposed scheme, we present the first TTD accelerator with decoupled PE array design and optimized dataflow. Experimental results show that TTD Engine achieves up to 36.9$\times$ and 9.9$\times$ speedup over state-of-the-art CPU and GPU implementations respectively, while providing significant improvements on energy efficiency. We further use a real-world MRI image dataset to perform image compression as a demo application on the proposed TTD accelerator.

- We demonstrate the benefit of hardware-enabled TT-format data analysis by addressing the rank-growth issue with TTD Engine and proposing a decomposed computation pattern for element-wise operations. A case study is presented to use TTD Engine to accelerate data convolution which shows considerable speedup over CPU when dealing with large-scale vectors.

6.2 Background

6.2.1 Tensor Knowledge and Notations

Tensors are multidimensional data arrays, which can be viewed as natural generalizations of vectors and matrices. Each dimension has its own coordinates and length. An N-way tensor, also called an Nth-order tensor, is a tensor with N dimensions or modes. For example, a third-order tensor has three indices, and can be visually described by Figure 6.1. Under this setting, vectors can be viewed as first-order tensors and denoted as $\mathbf{a}$, while matrices are second-order tensors that we denote as A. Finally, high-dimensional tensors are represented with $\mathcal{A}$ in the further content. A real-valued tensor of order N can be denoted as $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and its entry is $a_{i_1,i_2,\ldots,i_N}$.

By using only a subset of the indices in the original tensor and fixing the rest, we can get a subtensor. Particularly, a vector-valued subtensor, also termed as a fiber, is generated by using only one index from the original tensor. A matrix-valued subtensor uses two indices, and is therefore called a slice as shown in Figure 6.2.

The unfolding matrix of a tensor is generated by reordering the elements of the original N-way tensor into a matrix. In our paper, we focus on the special case of the unfolding matrix, which is called the mode-n unfolding matrix. Its definition is concise. Specifically, for a tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, its mode-$n$ unfolding matrix is generated by arranging the mode-n fibers to be the column of the target matrix, and is denoted as

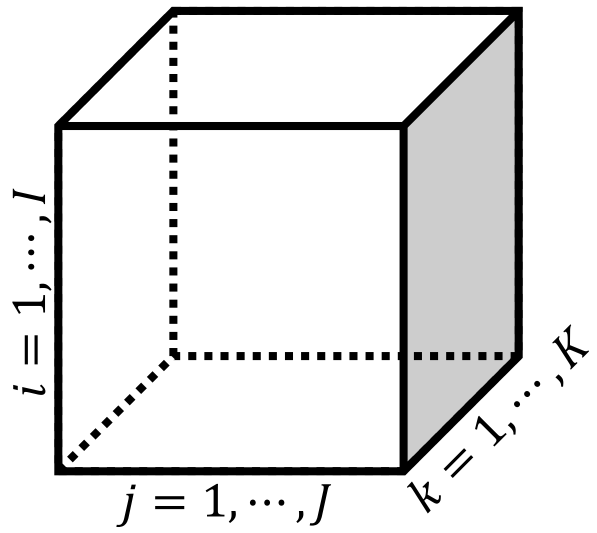


Figure 6.1: A 3rd-order tensor.

$A_{(n)}$. The notation of the unfolding matrix will be further used when we describe the TTD algorithm.

Generalized from matrix multiplication, two tensors can also be multiplied together to form up a new tensor, such process is called tensor contraction. The full definition and procedure of tensor multiplication is much more complicated than those in the matrix case, which are detailed for example in [131]. Here in this work, we only consider the mode-n contraction, i.e., multiplying a tensor by a matrix (or vector). Given tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and matrix $M \in \mathbb{R}^{J \times I_n}$, then the mode-$n$ product $\mathcal{A} \times_n M \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N}$ is obtained by the contraction over the n^{th} dimension, i.e. each element of $\mathcal{A} \times_n M$ equals $\sum_{i_n=1}^{I_n} x_{i_1} x_{i_2} \cdots i_N \times m_{ji_n}$.

6.2.2 Tensor Train Decomposition (TTD)

TTD is originally proposed by Oseledets in [40]. The overall procedure of the naive TTD algorithm is given in Alg. 3. In TTD, we try to approximately represent a given

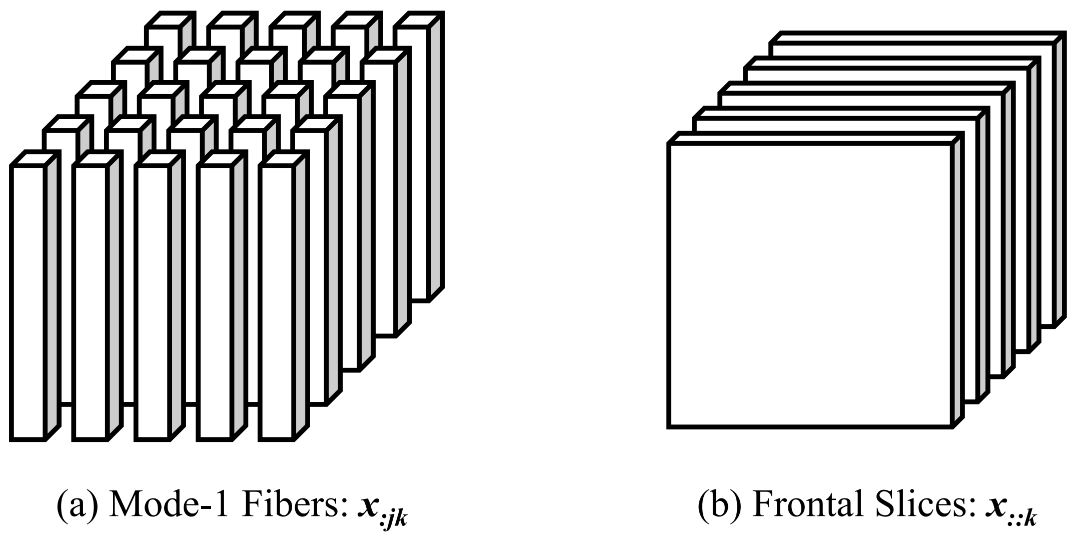

(a) Mode-1 Fibers: $\boldsymbol{x}_{:jk}$
(b) Frontal Slices: $\boldsymbol{x}_{::k}$

Figure 6.2: Fibers and slices of a 3rd-order tensor.

tensor $\mathcal{A}$ with tensor $\mathcal{B}$, which can be described as:

$$\mathcal{B}_{i_1,i_2\cdots,i_d} = G_1(i_1)G_2(i_2)\cdots G_d(i_d). \tag{6.1}$$

Each $G_k(i_k)$ is an $r_{k-1} \times r_k$ matrix, where r_k is called the TT-rank that can be either predefined before the decomposition or decided during runtime according to the required decomposition accuracy. G_k is an $r_{k-1} \times I_k \times r_k$ tensor core extracted from the original high-order tensor. In each TTD iteration, we need to perform Singular Value Decomposition (SVD) of an auxiliary matrix to get a tensor core. Therefore, it takes d sequential TTD iterations to finish the decomposition of a given tensor. Besides, at the beginning of each iteration, we need to reshape the given matrix into the required size before we can perform SVD. With the TT-format data, we can simply contracting these tensor cores together to reconstruct the approximated tensor which is close to the original tensor $\mathcal{A}$.

Notice that, the product of these parameter-dependent matrices in Equation (7.2) is a matrix of size $r_0 \times r_d$, this indicates the boundary condition of $r_0 = r_d = 1$. Moreover,

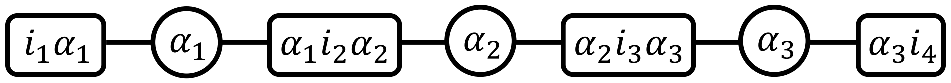


Figure 6.3: A tensor-train network.

Algorithm 3 TT-SVD

Require: d-dimensional tensor $\mathcal{A}$, approximation error ϵ.

Ensure: Tensor cores $G_1, ..., G_d$ of the TT-approximation $\mathcal{B}$ in the TT format with TT-ranks r_k equal to the δ-ranks of the unfoldings A_k of $\mathcal{A}$. The approximation error satisfies:

$$||\mathcal{A} - \mathcal{B}||_F \leq \epsilon ||\mathcal{A}||_F$$

```
1:  {Initialization} Compute the truncation parameter:
        $\delta = \epsilon\sqrt{d-1}||\mathcal{A}||_F$
2:  Temporary tensor: $C = \mathcal{A},\ r_0 = 1$.
3:  for $k = 1$ to $d-1$ do
4:      $C$ =reshape($C$, $[r_{k-1}I_k, numel(C)/(r_{k-1}I_k)]$)
5:      Compute $\delta$-truncated SVD:
            $C = USV^T + E,\ ||E||_F \leq \delta,\ r_k = rank_\delta(C)$
6:      New tensor core: $G_k$ =reshape($U$, $[r_{k-1}, I_k, r_k]$)
7:      $C = SV^T$
8:  end for
9:  $G_d = C$
10: Return tensor $\mathcal{B}$ in the TT format represented by tensor cores $G_1, ..., G_d$.
```

since $r_0 = r_d = 1$, TTD can also be visually represented by a graph called linear tensor network, as shown in Figure 6.3. There are two different types of nodes in this graphical representation. The rectangles are the tensor cores with the spatial indices (i_k from the original tensor) and auxiliary indices α_k. The circles are indeed links to connect two adjacent tensor cores with same auxiliary index α_k. This means that these two tensor cores are contracted together, and further being contracted with the following tensor cores to form the final d-dimensional tensor.

The most important step of TTD is how to extract these tensor cores from the original high order tensor. In this work, we focus on the classical TT-SVD approach, which computes such TTD using d-sequential SVDs of auxiliary matrices.

6.3 SVD Algorithm Adaptation

As introduced above, the computation of TTD is dominated by sequential SVD decompositions over the temporary auxiliary matrices. Therefore, reducing the SVD latency is vital for accelerating TTD algorithm. In this section, we present our observations of these auxiliary matrices that motivate us to design an adapted SVD decomposition that directly reduces the overall latency from algorithmic level. The proposed SVD also enables more efficient hardware implementation which will be demonstrated in Section 6.4&6.5.

First, consider a given matrix $A \in \mathbb{R}^{m\times n}$, the singular value decomposition of A is defined by:

$$A = USV^T \tag{6.2}$$

where U and V are orthogonal matrices of $m \times r$ and $n \times r$, as $UU^T = I_m$, $VV^T = I_n$ (I_m is the identity matrix of size $m \times m$, same for I_n). S is a diagonal matrix such that $S = diag(\sigma_1, \sigma_2, \cdots, \sigma_r)$, σ_k are the singular values of A. Among the existing numerical methods to compute SVD decomposition, one-sided Jacobi [132] is considered to be the most hardware friendly because of its fast convergence rate and good algorithm parallelism. The basic idea is to zero out off-diagonal elements using a series of orthogonal transformations between each pair of the matrix columns, and repeat this procedure for multiple iterations until convergence. Prior works[133, 134, 135] have proposed several customized accelerators for SVD using Jacobi method.

However, directly using the Jacobi method for TT-SVD is inefficient. To be more specific, in TTD, the auxiliary matrices to be decomposed are both large and unbalanced (i.e., one dimension is significantly longer than the other). For example, the first matrix to be decomposed is the first-mode unfolding matrix whose size is $I_1 \times I_2 I_3 \cdots I_N$. While

Algorithm 4 Adapted SVD Algorithm

```
Require: 2-dimensional matrix $A_{m\times n}$ where $n \gg m$, total iteration number $N$.
Ensure: Approximate decomposition of $A = U \times SV^T$ with an orthonormal matrix $U$ and orthgonal matrix $SV^T$.
1: {Initialization} $i = 0$, $B = AA^T$, $B \in \mathbb{R}^{m\times m}$, $Q_H = I_n$
2: Compute Arnoldi Iteration: $Q_k^T H Q_k = B$, where $H$ is a symmetric and tridiagonal matrix since $B$ is symmetric.
3: while $i \leq N$ do
4:     $d = H[n-1, n-1]$
5:     $H = H - dI_n$
6:     $Q_i R_i = qr(H)$
7:     $H = R_i Q_i + dI_n$, $H$ stays symmetric and tridiagonal.
8:     $Q_H = Q_H Q_i$
9: end while
10: $U = Q_k Q_H$
11: $SV^T = U^T A$
12: Return $U$, $SV^T$
```

Jacobi method requires multiple iterations to converge, within each iteration, we need to load and update the whole matrix. For tensor data, the size of such matrix can easily exceeds the capacity of caches (in CPU and GPU) and on-chip buffers (in customized accelerators). As a consequence, significant latency and energy consumption will be caused by excessive data access from the main memory module (e.g., DRAM). Moreover, designing multiple levels of memory hierarchy is also ineffective since there are no data reuse between different iterations of the Jacobi method.

Therefore, on the base of our observations and analysis, we propose an adapted SVD algorithm targeted for the large-scale unbalanced matrix. As shown in Alg. 4, the modified approach can be divided into three phases. Given an auxiliary matrix A, we first compute a matrix transpose multiplication $B = A \times A^T$. As a result, B is an $m \times m$ symmetric matrix whose size is much smaller compared with A. Then, to obtain $A = USV^T$, we can instead compute the eigenvalue decomposition (EVD) of B. In our approach, we use the Arnoldi method [136] followed by the shifted QR algorithm to compute the EVD result. Applying the Arnoldi method on matrix B gives us an orthonormal basis Q_k of B's Krylov subspace, and a symmetric tridiagonal matrix H where $H = Q_k B Q_k^T$. We then apply the shifted QR algorithm to obtain the eigenvectors of matrix H, which we denote as Q_H. Note that, Q_H is called the Ritz vectors of B that can be used to compute the eigenvectors of B. Finally, after we obtain the eigenvectors, which are indeed the left singular vectors of matrix A, we can compute SV^T with $SV^T = U^T A$.

Mathematically, the proposed SVD provides same results as typical Jacobi-based SVD. However, when dealing with large unbalanced matrix, it has following advantages: (1) We avoid constantly loading and updating (writing) matrix A. Such operations are inefficient as matrix A is often stored in high-cost memory, e.g., off-chip DRAM. (2) Both the Arnoldi method and the QR algorithm can be implemented based on modified Gram-Schmidt orthogonalization (MGS), which can be efficiently mapped onto our proposed architecture in Section IV. (3) The symmetric property of B greatly simplifies the process of Arnoldi method and QR algorithm. For Arnoldi method, when the input matrix is symmetric, the output matrix H will automatically become symmetric and tridiagonal. Therefore, we can directly skip the computations regarding the zero elements in H (output sparsity). For QR algorithm, since the input matrix H is symmetric and tridiagonal, the complexity of each iteration is significantly reduced. More importantly, matrix H will stay symmetric and tridiagonal after each iteration, which means such characteristic will benefit every QR iteration through out the whole process.

6.3.1 SVD Algorithm Evaluation

Table 6.1: Computation complexity & external memory access

Method	**Computation Complexity**	**Memory Access**
Jacobi	$iter1 \times O(m^2 n)$	$iter1 \times O(m^2 n)$
Ours	$O(m^2 n) + iter2 \times O(m^2)$	$O(mn)$

Table I lists the algorithm complexity and memory consumption between the proposed SVD and standard one-sided Jacobi SVD, where m, n are the matrix dimensions and $iter1, iter2$ denote the number of iterations performed in each approach. As we can see from Table I, our approach is more computational efficient when $iter2$ is comparable or smaller than $iter1$. We will demonstrate in Section 6.3.B and Section 6.7 that, when we seek for a low-rank output tensor-train(high compression ratio), which is normally the case of using TTD, then we only need approximate SVD results. Therefore, $iter2$ would be close to the number of iterations in Jacobi method, which makes the above analysis reasonable.

Moreover, as for memory footprint, the Jacobi method updates the whole matrix(A) within each iteration. Since A is of large-scale, it needs to be stored in DRAM rather than on-chip SRAM. Thus, constant access to matrix A will suffer from lower off-chip memory bandwidth and cost higher energy consumption. On the contrary, the proposed approach mainly operates on matrix B, which is much smaller and can be stored on-chip. Although the on-chip data movement will be more frequent, we prove in Section 6.6 with our experiments that this benefits the overall performance while lowering the energy consumption.

The final advantage of using the adapted SVD algorithm is its impacts on the hardware design. By enabling symmetricity and sparsity in matrix B, we open more hardware possibilities to reduce the decomposition latency with a dedicated accelerator. These properties are hard to be adopted by the conventional computing platforms like CPU and GPU.

6.3.2 Influence on TTD Accuracy

With the less computation complexity and memory footprints for processing the targeted large-scale unbalanced matrix, we further demonstrate the decomposition accuracy when applying the proposed SVD in TTD decomposition. To do so, we implement a customized TTD based on our adapted SVD algorithm, and compare it with the standard TTD function integrated in *tntorch* [137]. The accuracy of our proposed SVD algorithm can be controlled by the iteration number N, which further reflects on the end to end accuracy of the Tensor-Train decomposition. In our experiments, we set N to be 5, 10 and 15. In contrast, the Jacobi-based-SVD typically requires more rounds of iterations (around 30 or even higher) with longer per iteration latency. We use randomly generated tensor data as the input and decompose it using different TTD implementations. Then, we contract the tensor cores together to reconstruct the tensor data. The error between the reconstructed tensor and the original tensor is defined by the following equation, where $\mathcal{A}'$ is the reconstructed tensor, $\mathcal{A}$ is the original tensor and *norm* is the Euclidean norm:

$$error = \frac{norm(\mathcal{A}' - \mathcal{A})}{norm(\mathcal{A})}. \tag{6.3}$$

We compare the accuracy of the proposed TTD and the standard TTD by dividing their error values. Therefore, the higher the number is, the larger error it has compared with the standard TTD. We show the comparison in Figure 6.4, where y-axis is the relative error and x-axis is the compression ratio. A higher compression ratio means that the TT ranks are set to be lower to get smaller tensor cores. As we can see from the results, the proposed approach can achieve comparable accuracy with the standard TTD when the TT ranks are low (i.e., high compression ratio). When the compression ratio is up to 42.6, we can achieve nearly the same accuracy compared with the standard

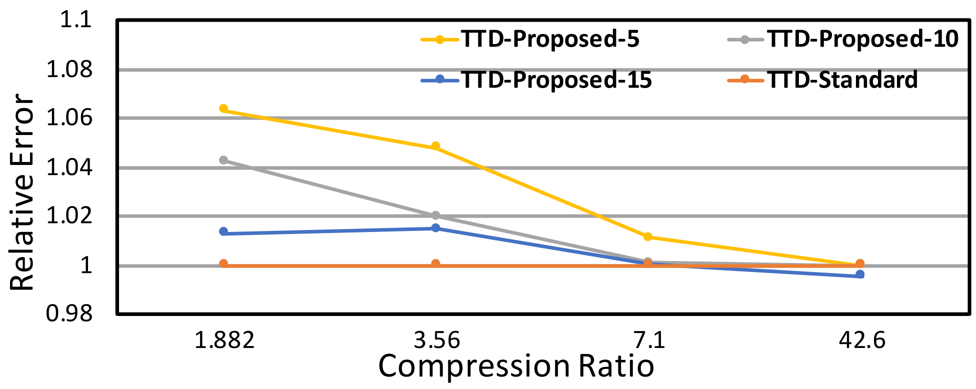


Figure 6.4: Accuracy comparison between the proposed TTD and the standard TTD.

TTD under all the three settings of N. For $N = 15$, the relative error is even smaller than 1, indicating that it even has less error than the standard TTD. As the compression ratio decreases, the relative error will increase, which implies that the proposed TTD is less accurate than the standard TTD when the TT ranks are high. Fortunately, such case rarely happens in practice because TTD is designed to be used to compress high order tensor with low TT ranks for good compression ratios. For example, when being used to compress weight matrices in deep neural networks(DNNs), prior work [138, 139] achieve acceptable accuracy loss with the compression ratio to be 82.87$\times$ for CNNs on the CIFAR-10 [140] dataset.

6.3.3 Discussion

Numerical Stability

In order to increase the efficacy of computing the QR factorization, we use the Gram-Schmidt orthogonalization. However, the classical GS method can be numerically unstable due to the rounding error when processing with finite precision. We solve this problem by using the stabilized modified Gram-Schmidt method (MGS). Specifically, traditional GS method computes a new vector by subtracting it with all of its projection vectors based on the existing unit vectors. In the modified GS method, for a new given vector, we start with eliminating the projection vector of the first unit vector to get a new candidate vector. The second projection vector to be eliminated is computed based on the new candidate vector instead of directly using the original vector. It is proved that this approach gives the same result as the original formula in exact arithmetic and introduces significantly smaller errors in finite-precision arithmetic.

Novelty

The problem of numerically computing singular value decomposition (SVD) has already been well studied. However, in terms of implementing TTD, prior work have not proposed the idea of transferring the large unbalanced SVD problem to a symmetric small eigenvalue decomposition (EVD) problem. Moreover, using the Arnoldi method and the shifted QR algorithm to approximate the EVD result is normally not suggested, as the number of iterations grows rapidly when requiring a particularly high decomposition accuracy. In our work, we take the advantage of the low-rank property of TTD to explore more efficient implementations while maintaining overall decomposition accuracy. Such low-rank property comes from our observations across different practical applications where TTD is adopted, like CNN/RNN, image compression, and quantum analysis. In these applications, TTD is used to achieve very high compression ratio without influencing much on the overall application accuracy. Thus, this high-compression condition ensures the low-rank settings for our previous analysis. Finally, as we mentioned above, the objective of using our proposed SVD is to eventually benefit the hardware implementation and reduce the execution complexity, which will be further demonstrated in Section 6.4-6.6.

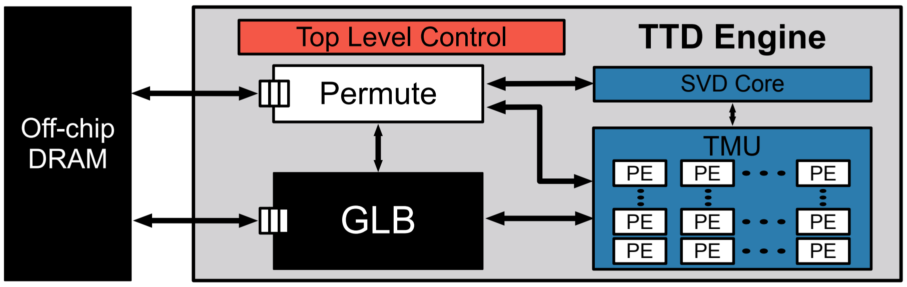


Figure 6.5: TTD Engine architecture overview.

6.4 TTD Engine Overview

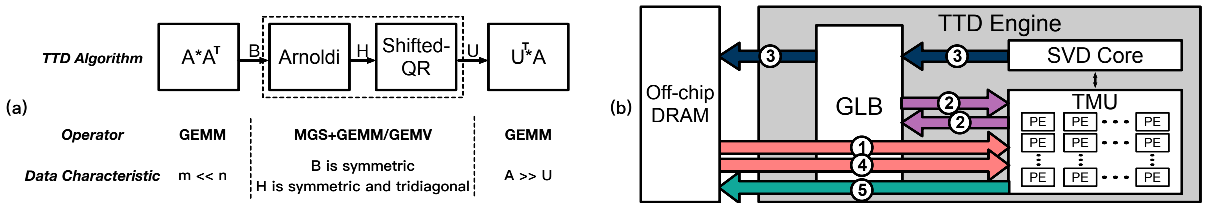


Figure 6.6: TTD algorithm abstraction and TTD Engine execution dataflow. (a) shows the key operators and data characteristics; (b) illustrates the data movement with arrow indicating the direction, color indicating specific data, and the number indicating the order.

Based on the analysis above, we present the overview of our TTD Engine in this section. We focus on addressing two key challenges during the hardware design. On one hand, the hardware should efficiently implement the proposed TTD algorithm, providing acceptable performance speedup and efficiency improvement. We call this, the **Specialization** of the hardware. On the other hand, it should also have the flexibility to support general matrix/tensor and even tensor-train operations, so that it can be further adopted to accelerate different applications using the tensor-train processing scheme. We call this, the **Generality** of the hardware.

With these two design objectives, we show the top-level architecture of TTD Engine in Figure 6.5. The off-chip DRAM stores the original tensor data that are unable to be fitted on-chip. Therefore, the accelerator communicates with the external DRAM through a bidirectional data bus and stores intermediate data in the Global Buffer (GLB) for on-chip data reuse. The computing resources are mainly organized into two modules, the Tensor Multiplication Unit (TMU) and the SVD Core. Both of the two modules adopt a spatial 2D processing element (PE) array architecture. TMU efficiently handles regular matrix/tensor operations, including matrix-matrix, matrix-tensor multiplications and so forth. TMU and SVD core can work together to execute the modified TTD algorithm proposed in Section 6.3. The Permute Unit reshapes the auxiliary matrices to be decomposed between each TTD iteration.

As illustrated in Figure 6.5, TTD Engine applies the Spatial Architecture (SA) design of domain specific accelerators (DSAs) [141]. The SA-style DSAs exploit high compute parallelism by direct communication between the PE array. Besides, the hierarchical memory organization from GLB to the PE's local memory further improves data reuse, achieving higher bandwidth utilization and energy efficiency. Therefore, SAs are widely used to accelerate deep learning algorithms like Convolutional Neural Networks (CNNs)[142, 143], Recurrent Neural Networks (RNNs) [144, 145] and Personalized recommendations [146]. In TTD Engine, while such generality is well preserved, we further add specialized Permute Unit and SVD Core to achieve the efficient execution of Tensor-train Decomposition.

In the rest of this section and Secion 6.5, we focus on illustrating the specialized architecture and dataflow design of TTD Engine when processing Tensor-train Decomposition algorithm. We will dive more into the generality with real world application demos presented in Section 6.7.

6.4.1 Overall Dataflow

We use Figure 6.6 to illustrate how we implement the proposed TTD algorithm. To do so, we first provide an abstraction of the algorithm to extract the key operators as shown in Figure 6.6(a). We also mark the special characteristics of these operators' input/output data, which can further simplify the hardware design. Corresponding to the key operators, Figure 6.6(b) demonstrates the data movement in TTD Engine.

In each decomposition iteration, we first compute $B_i = A_{(i)}A_{(i)}^T$, where $1A_{(i)}$ is the current auxiliary matrix. This step is essentially matrix-matrix multiplication, but the input matrix has unbalanced size where its width n is much longer than its height m. To execute this step on TTD Engine, we load $A_{(i)}$ patch by patch from off-chip memory to GLB. Tensor Multiplication Unit (TMU) and SVD Core work together as a larger PE array to compute matrix $2B_i$.

During the EVD decomposition of matrix B_i, both the Arnoldi iteration and the Shifted QR algorithm can be represented by a two-step process: column orthogonalization & data update. While the former step is realized using modified Gram-Schmidt (MGS), the latter step is nothing but matrix-vector/matrix-matrix multiplication. In TTD Engine, we use SVD core to perform MGS, the resulting matrix will be generated column by column, and will be sent to TMU immediately to perform data update. We use such decoupled design of TMU and SVD core to pipeline the two-step EVD while increasing local data reuse.

After we obtain the left singular matrix of B_i, which we denote as $3U_i$, we can directly permute and output U_i as the extracted tensor core. We then load matrix $4A_{(i)}$ again to compute $5S_iV_i^T = U_i^TA_{(i)}$. The result will be reshaped by the permute unit and sent out as the auxiliary matrix to be decomposed in the next TTD iteration.

As the iteration continues, the matrix to be decomposed would become smaller and more balanced. TTD Engine can also support these cases by storing the matrix completely in GLB and using the Jacobi method for SVD decomposition.

6.5 TTD Engine Architecture

In this section, we present the detailed architectures of different modules in TTD Engine. We also discuss how we take advantages of the data's special characteristics to efficiently map the algorithm onto hardware.

6.5.1 Tensor Multiplication Unit (TMU)

Figure 6.7 presents the 2D PE architecture of TMU. Each PE could communicate with its neighbors and also the GLB through an NoC. FIFOs are used at the I/O interface of each PE to balance the data movement between the NoC and the computation. The PE consists of a MAC unit for Multiply-and-Accumulate (MAC) operation, local buffers for matrix and partial sum data, and the PE's local control logic. For normal matrix multiplication, TMU can work as a systolic array to provide high computation throughput with simplified control flow. However, apart from the general matrix multiplication, we still need to consider the following special cases during the computation of TTD.

Large-scale matrix × small-scale matrix

Each of the TTD iteration ends up with a matrix-matrix multiplication between the current auxiliary matrix $A_{(i)}$ and the left singular matrix U_i. In most cases, $A_{(i)}$ is much larger than U_i and is stored off-chip. Therefore, in order to reduce the high-cost memory access, we keep a patch of $A_{(i)}$ stationary in TMU and load the corresponded blocks of U_i from the GLB. After finishing all the computations associated with the current patch, we load another patch of $A_{(i)}$ and repeat the process. In this way, although we need

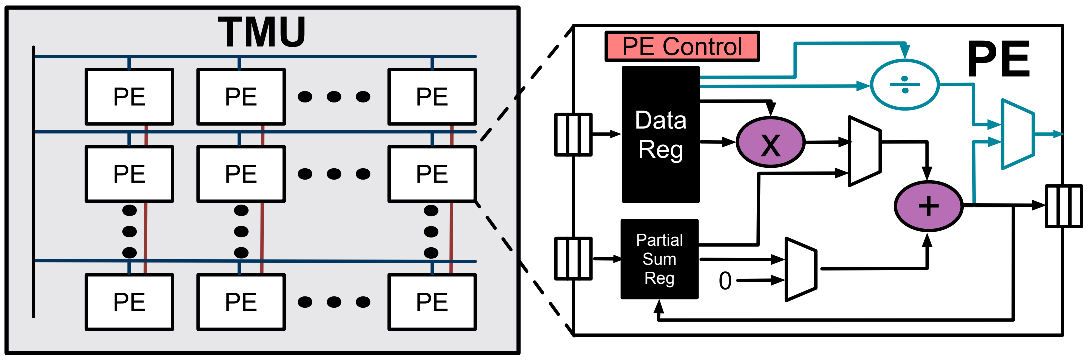


Figure 6.7: TMU architecture. The blue-colored logic only exists in PEs inside SVD Core.

to traverse matrix U_i several times in GLB, the large-scale matrix $A_{(i)}$ is loaded only one time from the off-chip DRAM. Therefore, the high-cost off-chip memory access is replaced with low-cost local memory access.

Large-scale matrix transpose multiplication

The first step of the proposed SVD algorithm is to perform a matrix transpose multiplication using the unbalanced matrix $A_{(i)}$. Since we already know that the result will be a symmetric matrix, we can save almost half of the redundant computations by only calculating the upper triangular part of the output matrix. For illustrative purpose, we use Figure 6.8 to demonstrate the matrix transpose multiplication for a matrix $A = \{a_1, a_2, ...a_{16}\} \in \mathbb{R}^{8\times16}$ with a 4×4 TMU PE array. The resultant matrix B can be considered as the sum of 16 submatrices where each submatrix $B_i = a_i \times a_i^T$. Therefore, we can group up 4 PEs as a PE set to compute a specific submatrix. The reasons for us to choose outer product to compute B are of two folds. First, using outer product only requires a single traversal through the original matrix to finish the computation. This is especially beneficial as matrix A is stored in high-cost off-chip DRAM. Second, in normal cases, buffering the output submatrices for accumulation can be expensive, but since the columns of matrix A are short, the submatrices computed by these columns

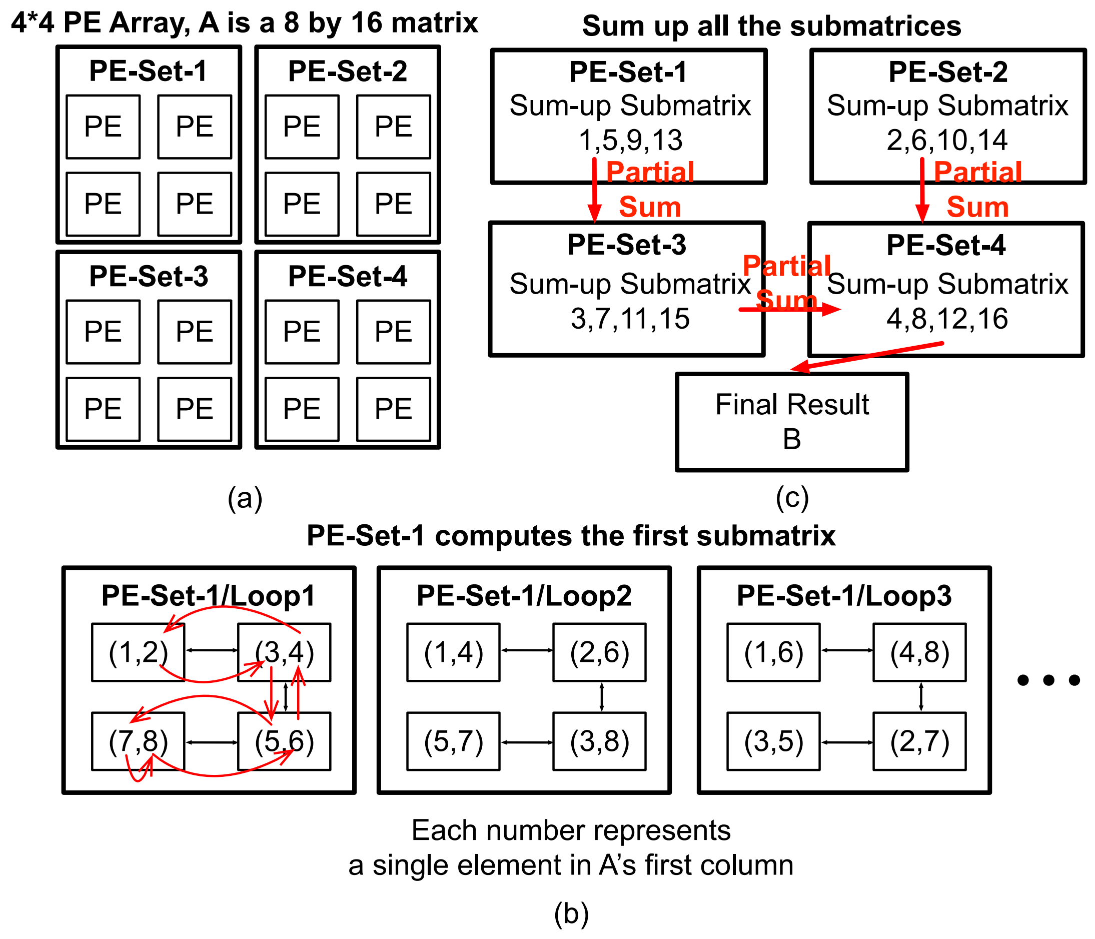

4*4 PE Array, A is a 8 by 16 matrix
PE-Set-1
PE-Set-2
PE-Set-3
PE-Set-4
PE
(a)
Sum up all the submatrices
PE-Set-1 Sum-up Submatrix 1,5,9,13
PE-Set-2 Sum-up Submatrix 2,6,10,14
PE-Set-3 Sum-up Submatrix 3,7,11,15
PE-Set-4 Sum-up Submatrix 4,8,12,16
Partial Sum
Final Result B
(c)
PE-Set-1 computes the first submatrix
PE-Set-1/Loop1
(1,2) (3,4) (7,8) (5,6)
PE-Set-1/Loop2
(1,4) (2,6) (5,7) (3,8)
PE-Set-1/Loop3
(1,6) (4,8) (3,5) (2,7)
Each number represents a single element in A's first column
(b)

Figure 6.8: Using TMU to compute large-scale matrix multiplication: (a) Several PEs are grouped together to compute a specific submatrix; (b) The data movement between different PEs after each computation loop; (c) Summing up all the submatrices across the PE array.

become much smaller and easier to buffer on chip.

As shown in Figure 6.8(b), inside the PE set, each PE is distributed with two elements of a specific column a_i. During each loop, the PE multiplies these two elements together to generate a single element in the submatrix. After each loop, different PEs from the same PE set will exchange data between each other. The data exchanging order is predetermined according to the column length. In this example, the red arrow indicates the data movement direction after each multiplication. Such order avoids all the redundant computations. Finally, all the submatrices are accumulated together to get the result. Note that, if the column is too long, each PE may contain multiple elements of the column. In such case, the PE will generate a small block of the output submatrix after each computation loop.

Tensor core contraction

TTD Engine is also designed to be able to perform tensor core contraction to recover the original tensor data. For tensor core contraction, each time we contract the last mode of the current tensor with the first mode of the next tensor core. Therefore, we only need to permute the tensor core and load the existing tensor in its original order. We assume the tensors are always stored by incrementing the mode-1 index, then the second mode index, and so on. To be more specific, suppose we have finished contracting the first m tensor cores which gives us tensor $\hat{G} \in \mathbb{R}^{I_1 \times I_2 \cdots \times I_m \times r_m}$. The next step is to contract $G_{m+1} \in \mathbb{R}^{r_m \times I_{m+1} \times r_{m+1}}$ with $\hat{G}$. Thus, we first permute G_{m+1} using TTD Engine's permute unit and store it in the GLB as $\hat{G}_{m+1} = r_m \times I_{m+1} r_{m+1}$. Then, we can treat both of them as matrices and perform matrix-matrix multiplication.

6.5.2 SVD Core

As introduced previously, the process of the adapted SVD algorithm can be represented by MGS and data update. Data update is essentially matrix-vector/matrix-matrix multiplication that can be efficiently mapped onto TMU. As for MGS, there are two problems need to be addressed. First, the orthogonalization between two columns requires the division operation. Thus, as shown in Figure 6.7, PEs inside the SVD core are further facilitated with dividers for the operation. Second, the MGS algorithm consists of multiple column orthogonalizations between different pairs of columns that have inter data dependency. Thus, it is important to design a mapping strategy that can maximize the computation resource utilization without breaking the data dependency.

Here we use Figure 6.9 to demonstrate the data dependency and mapping strategy. As shown in Figure 6.9, each parenthesis indicates an orthogonalization operation between two columns. For instance, $(2, 1)$ means to orthogonalize column 2 over the reference column 1. Therefore, only column 2 will be updated after this operation. According to MGS, there are two types of data dependency during the computation.

The first type is that, we cannot use a column as a reference column until it is finalized. For example, column 3 needs to be orthogonalized with column 2 and column 1. Therefore, we need to perform $(3, 1)$ and $(3, 2)$ before we can perform operations that use column 3 as the reference column, e.g., $(4, 3), (5, 3) \cdots$. The first type is marked by red arrows in Figure 6.9. The second type of data dependency is that, we cannot simultaneously orthogonalize the same column with two different reference columns. For example, $(3, 1)$ and $(3, 2)$ cannot be executed at the same time. The second type is represented by blue lines in Figure 6.9. It also shows that each time we cannot choose more than one operations from the same line.

Based on the analyses above, we propose the mapping of the MGS algorithm as shown in Figure 6.9. The idea is to choose as many operations as possible from the same vertical line. When we reach the end of one line and have to move across another line, we move to its adjacent line and start from the top. Both two types of data dependencies are avoided to the utmost extent using this mapping. In this example, suppose the SVD core can at most orthogonalize 3 pairs of columns in the same cycle. Then, we first choose $(2, 1), (3, 1), (4, 1)$, and then $(5, 1), (3, 2), (4, 2)$. Due to the second type of data dependency, we can only execute $(5, 2), (4, 3)$ at the third cycle, $(5, 3)$ in the fourth cycle, and $(5, 4)$ in the final cycle. Thus, it takes 5 cycles to finish the MGS process. If we increase the computation resources to support 4 pairs of columns. We

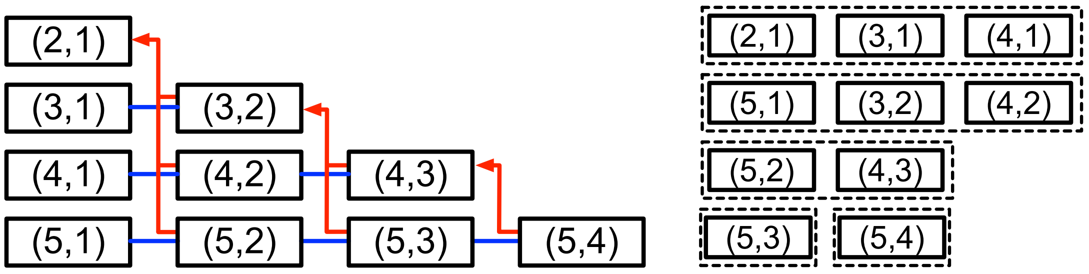


Figure 6.9: Mapping the GS orthogonalization onto SVD Core. Operations in the same dotted rectangle are executed simultaneously.

can reach a maximum throughput of 4 cycles to finish the MGS process. However, the resource utilization will be lower. Therefore, we design a lightweight SVD core that can achieve near-optimal performance with better resource utilization. Also, choosing most operations from the same vertical line increases the data reuse of the reference column that can further improve energy efficiency.

6.5.3 Permute Unit

The tensor permute unit is located between the GLB and external DRAM to reshape the input/output tensor/matrix data. It is used in the below cases: (1) During tensor core contraction, we use the permute unit to reshape the small tensor core; (2) After we compute the left singular matrix U, we permute it into a 3-mode tensor core. (3) After we multiply the current auxiliary matrix A with U^T, we need to permute the result to be the input matrix for the next TTD iteration.

6.6 Evaluation

6.6.1 Evaluation Methodology

Evaluation Platform. The proposed TTD Engine is implemented in RTL and synthesized in Synopsys Design Compiler with TSMC 45nm standard cell library to obtain the area and power estimation. The timing and energy of on-chip memory are simulated with CACTI [91]. We also develop a cycle-accurate simulator based on RTL implementations to evaluate the performance of TTD Engine.

Baselines. We compare our TTD Engine with the state-of-the-art CPU and GPU. The CPU baseline is an Intel Core i7 8700 processor (14nm), which has 12 SMT cores running at 3.2GHz and 12MB LLC. For GPU comparison, we use NVIDIA Titan V GPU (12nm) that is equipped with 5120 tensor cores and 12GB HBM2. We choose the Tensor Toolbox [147] as the software implementation on CPU and TnTorch [137] on GPU.

Benchmarks. We use synthetic data for the performance evaluation, with tensor sizes of 64KB, 4MB, 256MB, 1GB, and 8GB. The synthetic data are generated with built-in functions in each open-source library. For example, in TnTorch, we use torch.rand()/torch.randn() to generate the tensor data. For the decomposition speed comparison, we don't care about the actual value and distribution of the synthetic data. But for accuracy comparison, we keep the data identical across different implementations. We also set different decomposition parameters to examine how the performance is sensitive to the ranks of tensors. It is worth mentioning that, using synthetic data does not affect the generality of the experiments at all. Instead, it is because of the flexibility of synthetic benchmarks that enables us to evaluate TTD Engine's performance over various input patterns, including the cases that are frequently or rarely encountered in practical applications.

6.6.2 TTD Engine Summary

Table 6.2 presents the summary of the TTD Engine specifications. We use 16-bit fixed point arithmetics to implement our design. As listed in the table, with 16×16 PEs in TMU and 8×8 PEs in SVD Core running at 400MHz, our accelerator yields a peak performance of 128GMAC/s. Each PE has a 128B register, therefore, TMU and SVD core together have a 40KB of SRAM capacity. The global SRAM buffer is 1MB. Therefore, the total on-chip memory capacity is 1064KB. We show the area and power breakdown in Figure 6.10, from which we can see that the power is dominated by fixed-point operators as a fraction of 76%, while the total area is dominated by on-chip SRAM with a ratio of 62%.

Table 6.2: TTD configuration summary.

Item	Specification
Technology	TSMC 45nm GP standard VT
Total Area	$6.94mm^2$
Total Power	$2.89W$
Number of PEs	256 (TMU) + 64 (SVD Core)
Global Buffer	1MB (SRAM)
Register per PE	128B
Arithmetic Precision	16-bit fiexed-point
Frequency	400MHz
Peak Performance	128GMAC/s

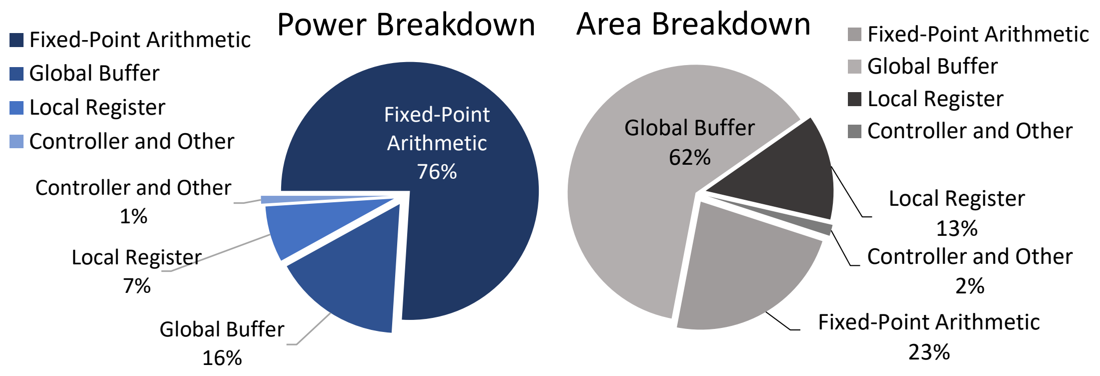


Figure 6.10: Power and area breakdown.

6.6.3 Overall Performance and Energy Efficiency

We compare the performance of our TTD Engine with CPU and GPU over synthetic data that have different sizes ranging from 64KB to 8GB. For each fixed input tensor size, we manually set the targeting output tensor ranks to 3 different levels to adjust the compression ratio. For example, for a single 4MB tensor, a low-rank decomposition means we generate a low-rank tensor-train from the original tensor data, which indicates a higher compression ratio with larger decomposition error compared with a high-rank result. Usually in real world applications, the rank matches with the low-rank and medium-rank cases in our experiments.

We first evaluate the decomposition accuracy among different implementations. The accuracy is measured with the absolute reconstruction error as expressed by equation (6.3), Section 6.3. Figure 6.11 shows the reconstruction error under different rank-levels averaging over all sizes of the input tensors. As we can see from the figure, in all three different rank levels referring to different compression ratios, the reconstruction accuracy scores among the implementations are comparable. Also, the proposed approach performs closer to (or even better than) the standard TTD when we are expecting a low-rank output tensor-train.

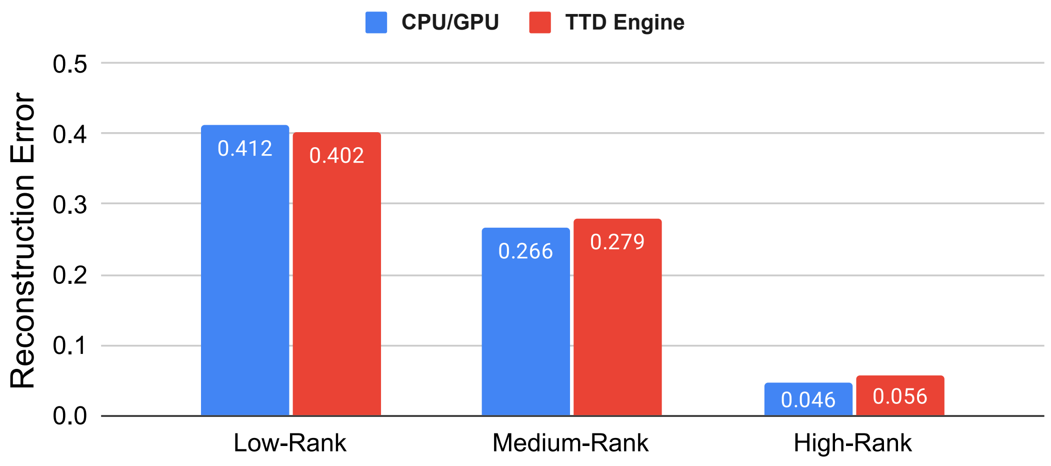


Figure 6.11: Average reconstruction error of CPU/GPU and TTD Engine under different rank-levels(compression ratio).

Then, we compare the decomposition speed of TTD Engine with CPU and GPU. As we can see from the results in Figure 6.12(a), TTD Engine significantly outperforms

CPU's performance. On average, it is $14.9\times \sim 36.9\times$ faster than the CPU implementation. Compared with GPU, TTD Engine can achieve speedup on benchmarks that are smaller than 1GB. If the input tensor size exceeds this limit, the speedup over GPU decreases. The reason is because when the tensor size keeps growing, the whole computation process tends to be dominated by the matrix transpose multiplication of the first few TTD iterations. For extremely large-scale matrix multiplication, TTD Engine is limited by computation resource and memory bandwidth, which dilutes the benefit of the proposed algorithm and dataflow optimization. However, in real cases, datasets are usually large for its number of samples rather than the size of each sample. Therefore, typically we do not need to consider a single tensor with a size of 8GB or even larger. Besides, the good scalability of TTD Engine makes it efficient to improve the performance by increasing the on-chip resources.

Also, for a given tensor, the lower the needed TT-ranks are, the higher speedup TTD Engine can achieve. This is because TTD Engine computes the singular vectors in the order of the singular values, and stops the computation as soon as the first r vectors are obtained. Whereas a typical truncated-SVD computes the complete SVD first, and then choose r vectors to output. This makes TTD Engine particularly suited for low-rank decompositions of a tensor.

Finally, we compare the energy efficiency of TTD Engine with CPU and GPU implementations over the same benchmarks. As shown in Figure 6.13, TTD Engine consumes, on average, 47.2x and 231.6x less energy than CPU and GPU, respectively. Such improvement is mainly gained from two aspects. First, we exploit low-cost data movement through the algorithm-hardware co-design while reducing high-cost external memory access. Second, we exploit the data sparsity and symmetricity during the computation process that helps to reduce both compute and memory consumption. Besides, when dealing with larger tensor, the energy efficiency improvements over GPU tends to be lower and less separable between different bars. Similar with the above analysis, when the decomposition process is more dominated by the matrix-transpose multiplication, the savings that come from adopting symmetricity and sparsity during the computation contributes less to the overall improvements.

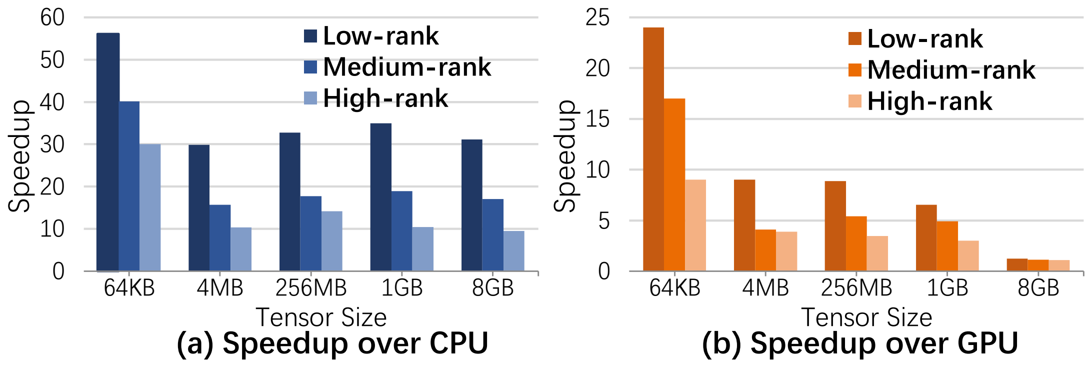


Figure 6.12: Speedup of TTD Engine over CPU and GPU.

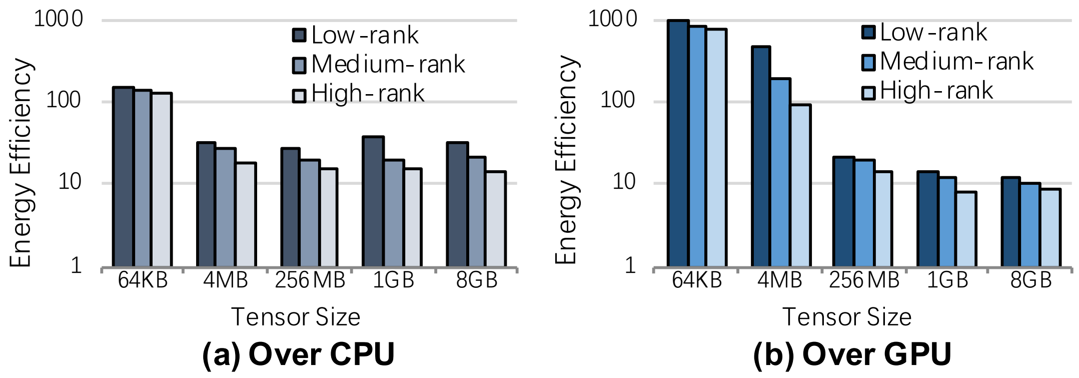


Figure 6.13: Energy reduction of TTD Engine over CPU and GPU.

6.6.4 Benefits Breakdown

In Section 6.6.C, we compare the overall performance between TTD Engine and CPU/GPU implementations. Here, we further decouple the contribution of the proposed

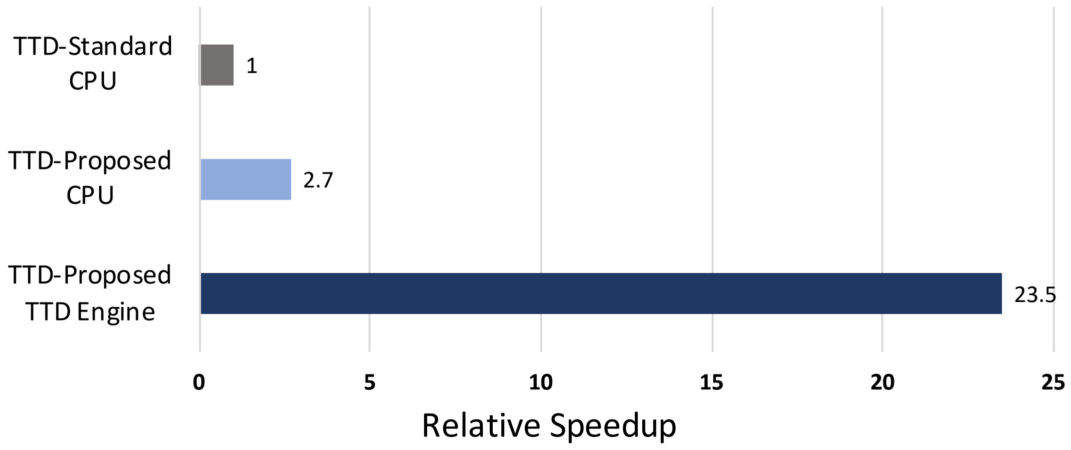


Figure 6.14: Performance contribution breakdown

SVD algorithm and accelerator design to separately demonstrate their benefits. We first run the proposed TTD and standard TTD algorithm on the same CPU to collect the decomposition time. Then, the proposed TTD algorithm is executed on TTD Engine to be compared with the other two cases. As we can see from Figure 6.14, by using the adapted SVD algorithm alone, we are able to achieve, on average, 2.7× speedup over the original CPU baseline. This speedup mainly comes from the computation reduction brought by the algorithm modification with a small number of iterations($iter2$). However, without dedicated architecture design and specialized dataflow, the sparsity and symmetricity of the matrices are hard to be utilized to benefit the overall performance. This introduces unnecessary computations which dilutes the final speedup. Thus, when TTD Engine is finally used, it further brings another 8.7× times speedup over the TTD-proposed-CPU and provides a final 23.5× speedup over the TTD-standard-CPU baseline.

6.7 Application Demo

Through Section 6.3 to Section 6.6, we demonstrated the effectiveness of TTD Engine when processing TTD with the proposed algorithm-hardware co-design approach. In this section, we will further illustrate how TTD Engine can be extended to accelerate different applications.

The first application we choose is medical image compression. TTD is now being used in a wide range of disciplines, including EDA design and simulation, machine learning, medical imaging, etc. One of the direct benefits of TTD is that, it saves considerable amount of memory space for storing the big data required by these applications. Moreover, with the decomposed results, many previous complex computations could be executed much faster and easier. To demonstrate such effectiveness, we first choose medical image compression as an example demo. We use the proposed accelerator to generate TTD results for a real-world magnetic resonance imaging (MRI) image benchmark, which significantly reduces the size of the benchmark while preserving good image quality.

In addition, we further illustrate the benefits of using TT-format data by proposing a TT-based convolution scheme. The proposed TT-convolution algorithm directly uses the TT-format data as input and performs convolution operations based on the convolution kernel. With the decomposed TT-format data, we can greatly reduce the overall computational complexity as well as memory consumption.

6.7.1 Medical Image Compression

We take medical imaging application as our first example. MRI is a safe and painless technique and is therefore widely used to generate detailed images of the brain and the brain stem. For general research purpose, numerous brain images are required and the data can easily reach to several gigabytes and even terabytes. Thus, it is very memory consuming to store the dataset. In our experiment, we choose a typical brain image dataset that contains 766 brain images of size 512×512 and is in total about 420MB large. The image dataset is compressed with TTD Engine and other open-source libraries running on CPU and GPU. We compare the decomposition performance between different architectures in terms of compression time and reconstruction error under a specific compression ratio of 7.1×.

The experimental results are shown in Table III. As for decomposition time, TTD Engine achieves 20.4× and 13.9× speedup over CPU and GPU, respectively. The speedup is close to the 4MB bar in Figure 6.12 even though the dataset is 420MB. This is because we compress each 512×512 image (1MB) separately, which gives us a higher performance speedup compared with compressing the dataset as a large single tensor. As for the reconstruction error presented in Table III, the proposed TTD achieves slightly better decomposition accuracy compared with standard TTD library given the same targeting compression ratio. This matches the conclusion we presented in Section III.A, that the proposed approach is able to generate comparable result when the compression ratio is not very low. Here, we also use equation (6.3) to measure the absolute reconstruction error.

Finally, to provide an intuitive comparison, four pairs of original images and reconstructed images are randomly selected from the dataset and presented in Figure 6.15.

Table 6.3: Decomposition Performance&Accuracy

Hardware	**Comp. Ratio**	**Speedup over CPU**	**Error**
CPU	7.1×	1x	0.158
GPU	7.1×	1.47x	0.158
TTD Engine	7.1×	20.40x	0.157

Finally, TTD is compared with other compression techniques like PCA and SVD under this specific application scenario. We sweep through different compression factors and compare the reconstruction error between different methods. Figure 6.16 delivers

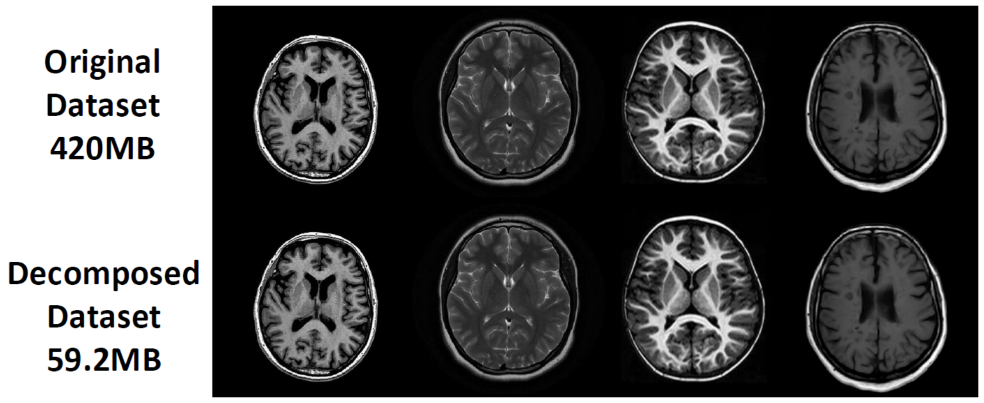


Figure 6.15: Comparison between original and decomposed MRI images.

the results and implies several conclusions. Firstly, TTD achieves lower absolute error compared with SVD in all the test cases. Secondly, when being compared with PCA, TTD tends to perform better in the cases with larger compression ratios, while doing worse in the low-compression cases. This indicates its advantage for providing highly compressed data with rather low error, which matches our previous analysis. Finally, PCA and SVD cannot deliver highly compressed images. As shown in the figure, no matter how we reduce the SVD/PCA parameters, SVD cannot deliver the two highest compressed cases and PCA cannot reach the compression ratio as high as 1638×. This is because SVD and PCA are pure 2D data processing techniques and are limited by the dimensions of the original image. On the contrary, TTD is able to first consider the 2D image (matrix) as a high-dimension tensor and then perform decomposition on all of its dimensions to achieve aggressive compression. In real applications, we can flexibly choose the desired compression technique based on different requirements for accuracy and compression ratio.

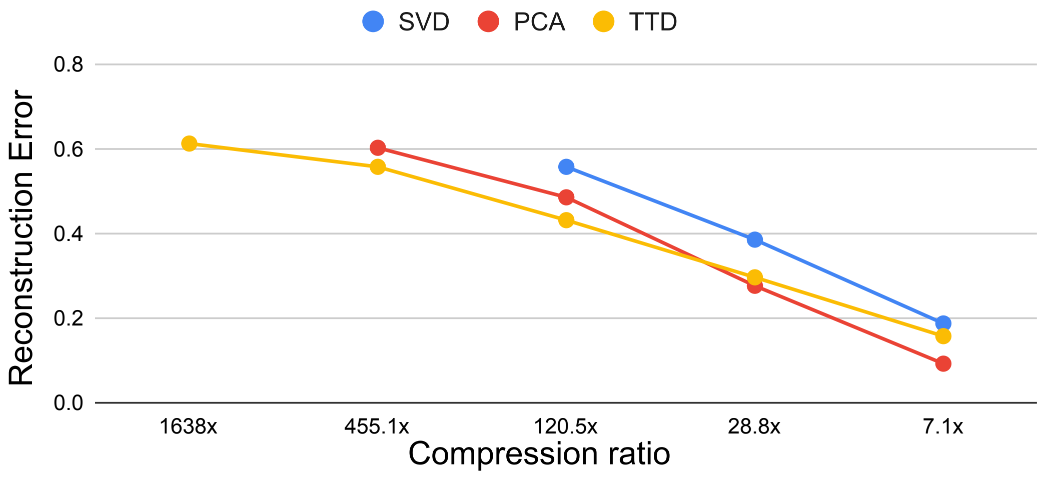


Figure 6.16: Comparing TTD with PCA and SVD for MRI image compression.

6.7.2 One-Dimensional Convolution

As aforementioned, there is a gap between decomposing a tensor and using the decomposed data to develop TT-based algorithms for practical applications. Previous work [40] has proved that basic TT operations like TT-Addition, TT-Multiplication, TT-GEMV, and scalar product have less complexity than directly operating on original large-scale tensor data. Moving forward, for the first time, we introduce how to use TTD Engine as the base architecture to perform TT-format data convolution on the decomposed data. We believe data convolution is a promising example to demonstrate the potential and benefit of using TTD for more complicated operations and applications. On one hand, element-wise operations are commonly used but rarely studied for TT-format data. On the other hand, multidimensional convolution stands at the core of many important applications including image processing, machine learning, and EDA.

For illustrative purposes, we first consider an 1D data convolution with an 1×3 convolution window sliding over an 8-element vector $\mathbf{v}$. The vector is reshaped into a $2 \times 2 \times 2$ tensor $\mathcal{V}$ and then represented with 3 TT-cores, G_1, G_2, G_3. As shown in

Figure 6.17, if we reverse core G_3's second dimension by exchanging the purple column with the pink column, the order of tensor $\mathcal{V}$'s third dimension is also reversed. This is further equivalent to switching every two consecutive elements in $\mathbf{v}$. Similarly, if we reverse core G_2's second dimension, it is equivalent to switching every two consecutive **pairs** of elements in $\mathbf{v}$.

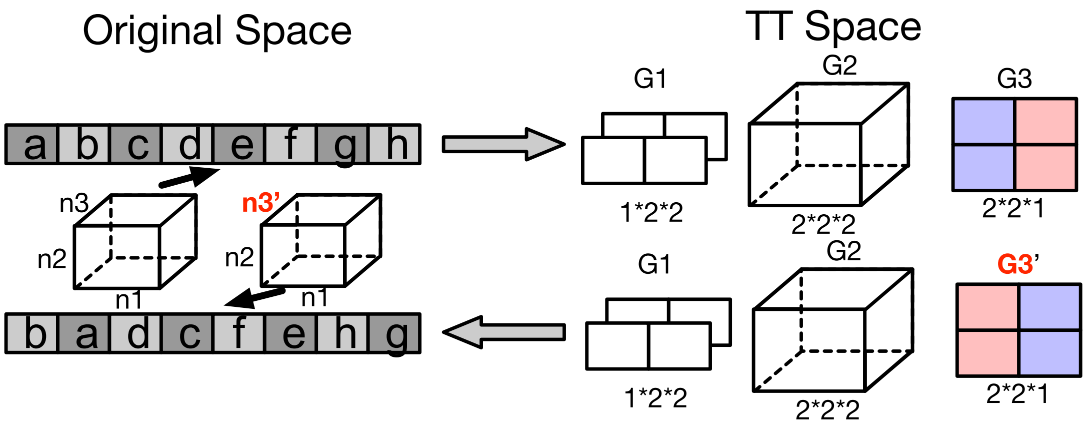


Figure 6.17: Demonstration of TT data and its characteristics.

Therefore, we demonstrate the process of TT-based 1D convolution in Figure 6.18. Without loss of generality, we assume all the weights to be equal to one. We already know that, we can operate on a specific dimension by modifying its corresponding tensor core. Thus, we can represent the final convolution result by the sum of several sub-vectors. The principle is to ensure that the TT-format of each sub-vector can be efficiently obtained from the original tensor-train format data.

As shown in Figure 6.18, for this specific 1D convolution example, the final result is represented with the sum of 3 sub-vectors. Taking the first one as an example, in every consecutive pair, the first element is the sum of the original two elements, and we keep the second element unchanged. Therefore, to get the TT-format of this sub-vector, we can simply sum up G_3's second dimension to form up a new column and replace the first

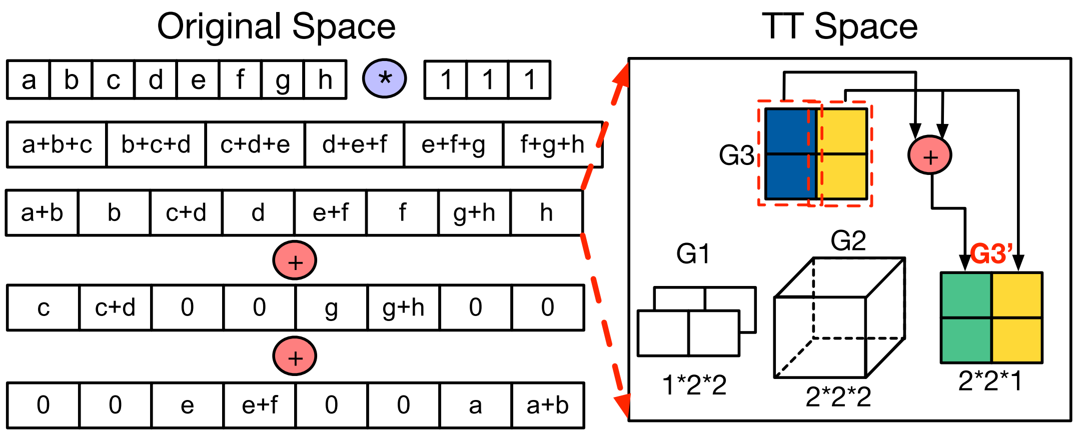


Figure 6.18: 1D Convolution in TT-format.

one, while leaving the second column the same as before. The other sub-vectors require similar operations. After this, we add the 3 tensor-train format sub-vectors to obtain the final convolution result. Note that, TT-Addition requires no computations but to merge the corresponding tensor cores together.

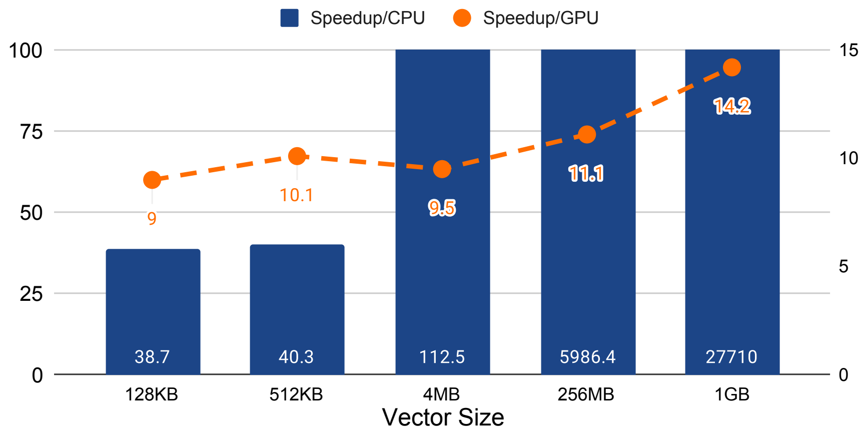


Figure 6.19: Speedup over CPU and GPU when performing 1D convolution.

In TTD Engine, we first load or compute the original tensor-train format data. Then, the remaining operations are simple vector/matrix additions and multiplications over the tensor cores. TMU and SVD core can work together as an efficient 2D PE array to handle these operations. This idea can be further generalized to vectors with longer length and with different convolutional kernel sizes. Moreover, even if the vector's length increases dramatically, the tensor cores stay small, which makes the TT-format processing much more efficient. We implement the TT-format 1D convolution based on our TTD Engine, and compare it with baseline 1D convolution kernel running on CPU and GPU. As shown by the results in Figure 6.19, when comparing with CPU, TTD Engine achieves significant speedup ranging from 38.7× to 21725×. Also, the speedup almost scales linearly after the 4MB bar. This is because as soon as the vector's size reaches a certain limit, the processing time of CPU grows proportional to the size of the tensor. However, with TT-format convolution scheme, the total computation is greatly reduced and much less influenced by the size of the vector. Therefore, the speedup of TTD Engine over CPU will grow rapidly for larger tensor. On the contrary, the speedup over GPU is more stable, ranging from 9.0× to 14.2×. This is because when the vector's size is small, the GPU execution time is not dominated by the computation, but other non-computation cost like kernel launching time. Only when the size is large enough, like from 256MB to 1GB, the computation time increases and the advantage of TTD Engine will be more obvious. For even larger input vector, we believe TTD Engine can achieve higher speedup over GPU as long as the TT representation of the vector is available.

6.7.3 Generalization to high-order convolution

Using the same idea, the TT-based 1D convolution scheme can be further generalized to high-order convolutions so that it can support various applications such as image processing and machine learning. Here we show how to apply the TT-based data processing to the 2D convolution problem. Firstly, the TT-format representation of a matrix $W \in \mathbb{R}^{M \times N}$ is given as follow:

$$W = G_1 * G_2 * \cdots * G_d \tag{6.4}$$

Where $G_i \in \mathbb{R}^{r_{i-1} \times m_i \times n_i \times r_i}$, $\prod m_i = M, \prod n_i = N$.

Different from TT-format vectors, each tensor core now has four dimensions, including two rank-dimensions, an m-dimension and an n-dimension. Using an example shown in Figure 6.20, suppose we have an $M \times N$ image where $M = N = 8$, and we want to perform a 2D convolution with a kernel of size 3×3. First, we can decompose this matrix into 3 tensor cores with a shape of $1 \times 2 \times 2 \times r_1$, $r_1 \times 2 \times 2 \times r_2$ and $r_2 \times 2 \times 2 \times 1$, respectively. We know that, a 2D convolution can be considered as two 1D convolutions along each of the dimension. This can be directly applied to TT-based convolution. Thus, we first perform a 1D convolution along the m dimension. In the 1D case, reversing the third core's second dimension is equivalent to switching every two consecutive pairs in the original vector. Here, as illustrated in Figure 6.20, if we reverse the m-dimension of the third core, every two consecutive rows in the original matrix will be exchanged. In other words, modifying the m-dimension of the tensor core is equivalent to operating on the whole rows of matrix W.

After changing the m-dimension, we indeed get several sub-matrices with modified rows. Similarly, we can further modify the columns of these sub-matrices by operating on the n-dimensions of the tensor cores. As illustrated in the example in Figure 6.20, if we reverse the n-dimension of core G_1, the left half of the matrix will be exchanged with the right half. Finally, we add these sub-matrices together, to get the 2D Convolution result. Similar to the 1D convolution, the high order convolution can also be efficiently executed on TTD Engine once the original tensor-train is obtained. After this, the proposed TT-convolution ensures the remaining computations to be executed only on certain slides of the few tensor cores. More importantly, if operations like TT-Addition cause the result

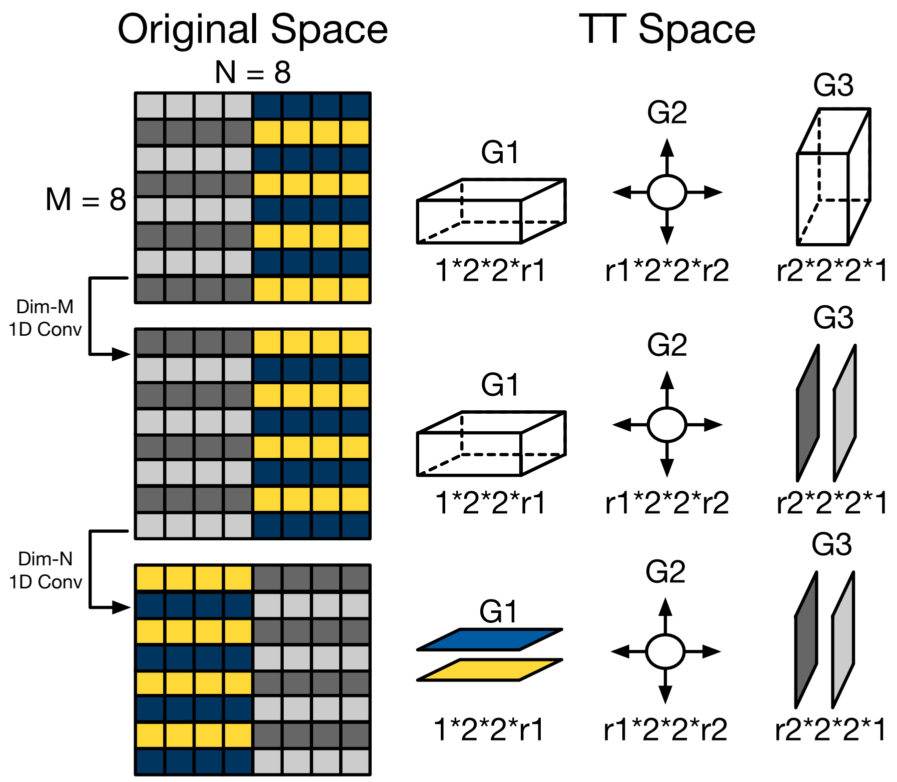


Figure 6.20: TT-based 2D Convolution.

tensor-train to have high TT-ranks, we can directly re-decompose the tensor-train with TTD Engine to get a new approximation with much lower TT-ranks.

In this section, we present three case studies using TTD Engine for different applications. Medical image compression shows the straightforward benefits to decompose high order data using TTD to reduce memory consumption. Furthermore, we propose tensor-train data convolution to show the effectiveness of the TT-based data processing in terms of reducing computational complexity.

6.8 Related Work

Tensor Decomposition Algorithms. Tensor decomposition attempts to compress and represent a high-dimensional tensor with a smaller number of factor tensors. In this work, we aims at accelerating the TTD algorithm. In fact, there are also other efficient tensor decomposition methods apart from TTD. Polyadic Decomposition **(PD)** expresses an n-way tensor as the sum of r rank-1 terms. Particularly, when r is the minimal rank, the decomposition is called Canonical Polyadic Decomposition **(CPD)** . It is also called Canonical Decomposition (CANDECOMP) or Parallel Factor (PARAFAC) in the tensor community [148, 149]. Tucker decomposition[150, 151, 152] treats a tensor as a multilinear transformation of a core tensor $\mathcal{G}$ by the factor matrix B. It can be considered as an expansion in rank-1 terms that is not necessarily canonical. Among all the decomposition methods, TTD is preferred for high-order tensors since its resulting tensor factors have a low storage requirement linearly dependent on the number of orders and the dimension depth. Moreover, TT has a unique feature, that is it can be implemented with cross approximation [153] without knowing the whole tensor.

SVD Hardware Accelerators. To the best of our knowledge, TTD Engine is the first work to accelerate TTD. In fact, the whole tensor hardware community is still lacking exploration. Previous work have more focused on the acceleration of matrix decomposition. Accelerator design for SVD is a huge fraction [133, 134, 135]. However, these works have some restrictions that motivate us to conduct the algorithmic adaption together with our TTD hardware design. First, many of previous SVD accelerators target only matrices with a certain shape or size. For instance, some can only support the square-shape matrices, while others cannot work when the input matrix's dimension exceeds the predefined dimension length. Second, directly applying Hestenes-Jacobi method to large-scale unbalanced matrices is extremely memory-inefficient, as it requires constant reads and writes for the original matrix that cannot be stored in local memory.

Tensor Hardware Accelerator. As aforementioned, the study of the tensor hardware, especially tensor-decomposition hardware is still at the early stage. [154] proposes the first FPGA-based accelerator for tucker decomposition. It focuses purely on the acceleration of Tucker decomposition, while the data processing techniques using Tucker-format tensor data are not covered. Other work [155, 156] mainly address the problem of designing general computation kernels for dense/sparse tensor data. While efficient hardware implementations for general Tensor-Tensor multiplications, Tensor-matrix multiplications are proposed, these work still suffer from the curse of dimensionality essentially due to the lack of decomposed tensor data.

6.9 Conclusion

This paper presented the first customized architecture to accelerate TTD, a promising tensor technique that is increasingly used in EDA optimization, big data analysis, and machine learning. Experimental results show the proposed TTD Engine is at least 14.9$\times$ and 4.1$\times$ faster than its CPU and GPU counterparts, respectively. We scale a demo of our TTD Engine on an FPGA board and perform medical imaging compression tasks to demonstrate the application potential. Moreover, we have conducted a case study to use TT-method to implement convolutional operations. The TT-based convolution has shown significant advantages when dealing with large-scale data. With customized algorithm design and specialized hardware support, TTD has the potential to break the curse of dimensionality of big data processing, and this work may stimulate more efforts on this topic.

In the future, we plan to extend TTD Engine following two directions. 1) Since advanced TT Decomposition employs cross-approximation for low-rank matrix factorization, we plan to add the corresponding support in TTD Engine. Therefore, the users can choose the specific matrix factorization method they want to adopt when decomposing the tensor. 2) We plan to further demonstrate the effectiveness of TTD Engine when performing end-to-end applications using the introduced TT-format data processing pattern.

Chapter 7

TT-GNN: Efficient On-Chip Graph Neural Network Training via Embedding Reformation and Hardware Optimization

7.1 Introduction

Originating from spectral graph analysis and fueled by the success of machine learning, graph neural networks (GNNs) have drawn a surge of interest and have been applied to various applications involving non-Euclidean graph-structured data. During the past few years, a wide range of GNN models [29, 30, 31, 32] have been proposed to solve graph-related problems. Exciting progress has been achieved by GNNs in domains such as recommendation systems [33], relation prediction [34], chemistry analysis [35], financial security [36], protein discovery [4, 5], EDA [37, 38, 39] and so on.

Despite the great application potential, training GNNs on large graphs is challeng-

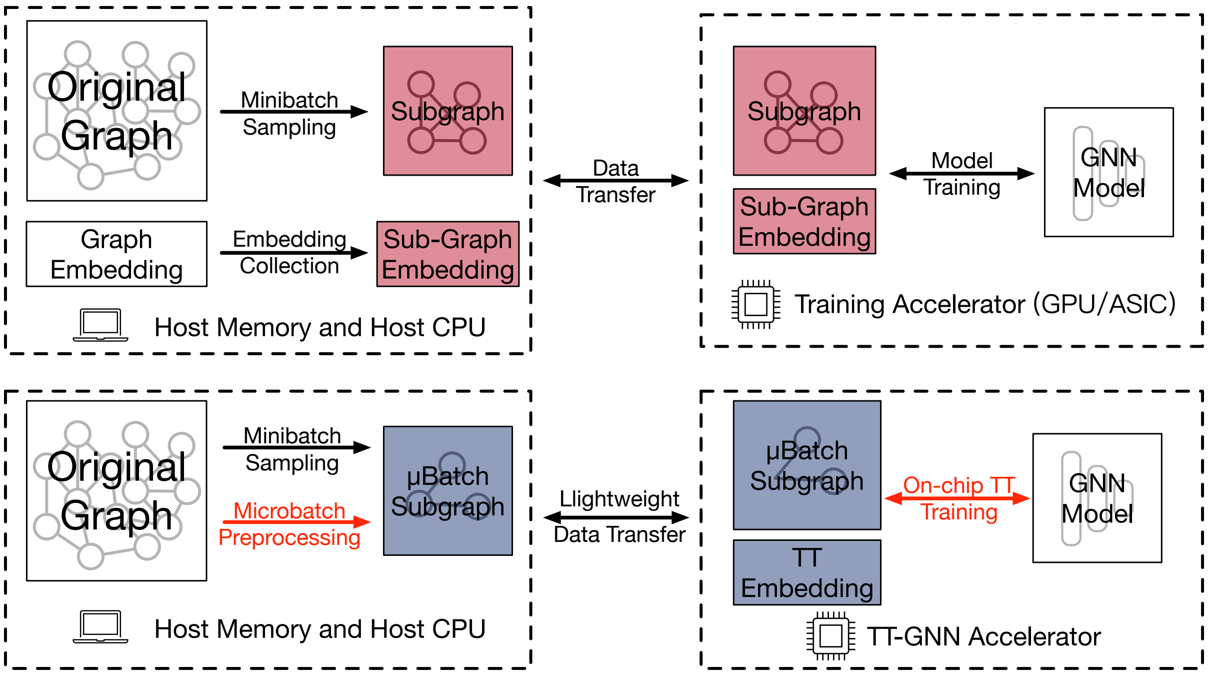


Figure 7.1: Illustration of typical minibatch training pipeline and TT-GNN training pipeline.

ing due to the need to store graph data and move them along the memory hierarchy. Given the increasingly large problem size, minibatch training is currently the most widely adopted approach to train a GNN model[29]. As shown in Figure 7.1, each minibatch takes two steps. The first step is to sample a subgraph from the original graph. The structure of the subgraph and its corresponding node embeddings together form a minibatch of training data. In this paper, we consider the case where the graph is very large, such that the graph data are stored in a host system memory. Consequently, the subgraph preparation is handled by the host processor, such as a host CPU. After obtaining the minibatch training data, it is sent to training hardware such as GPU to execute the model training. In this second step, we perform forward and backward propagation on the subgraph to update model parameters.

To speed up minibatch GNN training, prior works have proposed diverse software and hardware techniques targeting different stages of the training pipeline. Some work

[157, 158, ?] aims at improving GNN computation efficiency with algorithmic and software optimizations. Others focus on reducing neighbor sampling latency [159] and data loading cost [160] to hide the subgraph preparation overhead. However, they all assume an unchangeable setting, that is each node of the graph should be independently represented by a feature vector. This assumption further leads to the explosion of the graph representation when the number of nodes scales to millions and billions. Eventually, memory capacity is saturated and training performance is compromised. According to our profiling experiments, collecting node features from the host memory can take $27.9 \sim 61.1\%$ of the training time on a typical CPU-GPU system.

In this work, we tackle this problem by effectively compressing the graph feature matrix and storing it closer to computation resources for faster memory access. Specifically, we observe that different graph node features contain inter-relationships that can be well preserved even after applying low-rank approximation. Therefore, we consider using Tensor-train (TT) to represent the graph feature instead of using a 2D embedding matrix. In this way, we can represent the graph using a much more compact TT data structure while maximally preserving the representation capability. As shown in Figure 7.1, the resultant TT graph embedding can be stored in the accelerator's on-chip buffer, and the embedding is jointly trained with the Graph Neural Network with much less memory consumption.

Although the algorithmic modification greatly reduces the memory cost of training GNNs, it imposes several new hardware challenges. (1) During the forward pass, TT-format embeddings need to be decompressed into the original vector format before being processed by the GNN model. Reversely, we also need to generate the TT-format gradient during the backward pass. Naively handling these TT-related computations is expensive, yet, exploring effective intermediate data reuse is non-trivial. (2) Although we can store the TT-format embedding in the on-chip buffer, the decompressed features used in each minibatch might still exceed on-chip memory capacity. Therefore, we need a more fine-grained dataflow to further split each minibatch into smaller compute graphs.

To tackle the aforementioned challenges, we propose TT-GNN, a training system that incorporates software and hardware co-optimizations for efficient GNN learning at scale. Firstly, to mitigate TT computation overhead, we propose a unified algorithm to jointly handle TT decompression and TT gradient derivation. The proposed algorithm can be flexibly configured to be more compute-efficient by caching more reusable results, or more memory-efficient by tolerating some recomputation overhead. Secondly, by evaluating on-chip memory capacity and training configuration, TT-GNN dynamically breaks down a minibatch into smaller microbatches that can be fitted on-chip. To reduce redundant computations caused by neighbor sharing across different microbatches, we cache the last few layers of the GNN model on-chip, and only fan out from an intermediate layer if necessary. The microbatch composition and scheduling order is designed to maximize data reuse both across and within microbatches. Finally, we explore the reuse opportunities of aggregated partial sums which benefit both neighbor aggregation in forward propagation and gradient scattering in backward propagation.

Combining the algorithm and architecture co-design, TT-GNN achieves 1.55$\sim$4210$\times$ training speedup and 2.83$\sim$2254$\times$ energy efficiency improvements compared with the baseline CPU-GPU system on a series of GNN benchmarks. The key contribution of this work is summarized as follows:

- We perform in-depth characterization of GNN training on a standard CPU-GPU system, locating the training pipeline bottleneck being the feature collection and uncovering the underlying causes.
- Motivated by the profiling results, we propose to compress the feature matrix such that it can be held in faster memory. We also conduct preliminary experiments to

demonstrate that performing on-chip decompression is more efficient than retrieving the feature from off-chip memory.

- We propose a training system with software hardware co-optimizations tailored for efficient GNN training. In our design, only the graph sampling is executed in the host system, while the graph embedding collection, as well as GNN training, are fully handled on-chip.

- We evaluate TT-GNN on a series of GNN datasets, demonstrating the effectiveness of the proposed design and the possibility to train large GNNs with limited resources.

7.2 Background and Motivation

In this section, we first present the basics of Graph Neural Networks. We then introduce our in-depth GNN training characterization on a GPU system, which motivates us to propose TT-GNN.

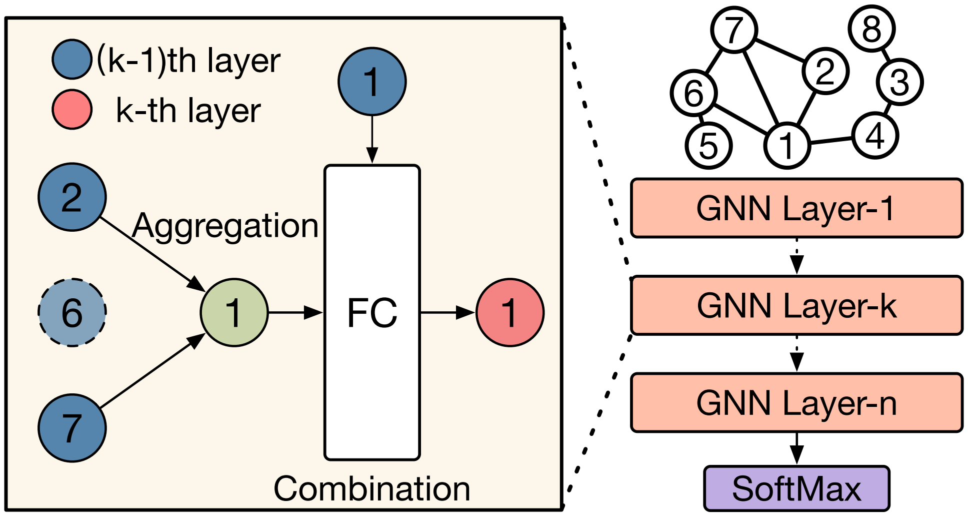


Figure 7.2: Illustration of a sample GNN model.

7.2.1 GNN Basis and Minibatch Training

We start with introducing some basic notations in GNNs. Given an undirected graph, we denote it as $G = (V, E)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in the graph. Each node is described by a feature vector of length F, and all the node features together forms a 2D feature matrix $X \in \mathbb{R}^{|V| \times F}$. In most cases, matrix X is dense and of large-scale due to the massive amount of nodes contained in real-world graphs.

During GNN processing, each GNN layer follows a two-stage procedure, namely *Aggregation* and *Combination*. As shown in Figure 7.2 and equations in below, each node v will collect feature vectors from its sampled neighborhood $N(v)$ to generate an aggregated feature a_v^k. The aggregation operator can be flexibly designed, where common choices include *Mean, Max, MLP* and so on. After this, the aggregated feature is combined with source node v's feature vector $h^{(k-1)_v}$. The combination operator utilizes these two vectors to generate hidden representation $h^k(v)$ of node v.

$$a_v^k = \textbf{Aggregate}(u : u \in N(v) \cup v)$$

$$h_v^k = \textbf{Combine}(a_v^k, h_v^{(k-1)})$$

To train a GNN model, we typically adopt the minibatch strategy. As illustrated in Figure 7.3, for each minibatch, we fan out from a group of target nodes. When considering the receptive field, we sample a fixed-size set of neighbors instead of using the full neighborhood for each node. This results in a funnel-shaped network, where the cost of each layer follows a decreasing order. To perform the GNN computation, we start from the input nodes of the first layer, use their feature vectors and follow the graph structure to perform aggregation and combination. The generated hidden node features will be further used as the input to the next layer.

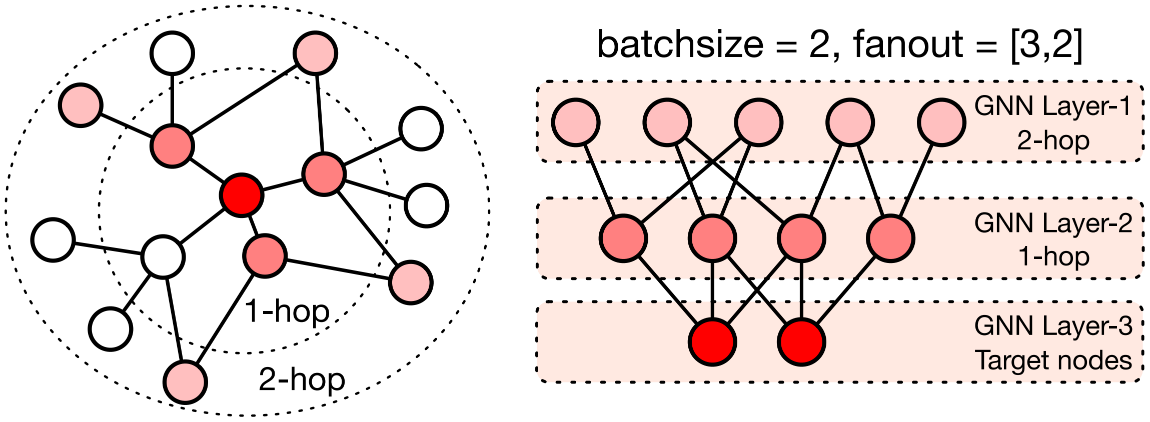


Figure 7.3: Sampling-based minibatch GNN training.

7.2.2 GNN Training Characterization

As mentioned above, there are mainly two types of data structures used in minibatch GNN training, the graph structure represented in CSR format, and corresponding feature embedding stored in a 2D matrix. Since a real-world graph may contain a massive amount of nodes and edges, both graphs CSR and embedding matrix can consume large memory space.

We observe that the location of the graph data significantly affects the overall training performance. When both graph structure and embedding matrix can be fit into GPU device memory, we can directly perform sampling and feature collection on GPU [161], therefore avoiding transferring data between host memory and device memory. However, if the data exceeds GPU's memory capacity, the sampled data will have to be sent via the system interconnect (e.g., PCIe). To illustrate the performance gap, we conduct a profiling experiment using a popular GNN model (GraphSAGE [29]) and a real-world benchmark (ogbn-products [162]). The model is implemented in DGL [161], and experiments are done on an Nvidia 3090 GPU using Nsight System.

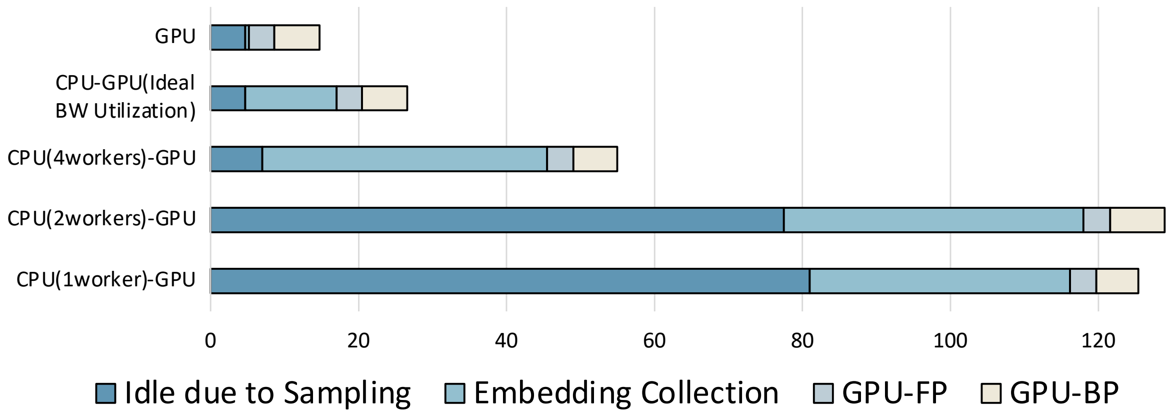


Figure 7.4: Average Latency(ms) Breakdown of Training One Minibatch on 3090. The batchsize is set to 500, with a 3-hop neighbor fan-out of $[5, 10, 15]$

Figure 7.4 shows the training latency comparison when the graph is stored in GPU HBM or in the host DRAM. The end-to-end latency is broken down into different steps. As we can see from the figure, under the same batchsize, for each epoch, training on HBM is $3.74 \sim 8.77\times$ faster than training on host DRAM. The performance difference purely comes from the sub-graph preparation stage. When the graph is completely stored in HBM, GPU performs parallel graph sampling and directly fetches node features from HBM. Therefore, the combined latency of sampling and feature collection is shorter than the latency of forward and backward propagation. This further indicates opportunities to fully hide the subgraph preparation overhead with pipelined execution.

On the contrary, CPU-based graph sampling and feature collection are much slower, uncovering the subgraph preparation cost. To improve graph sampling efficiency, we can issue multiple threads ($\#worker$) to simultaneously perform sampling for different minibatches. The generated subgraphs will be stored in a task queue to be fetched later. As a result, when $\#worker$ is set to 4, the per-minibatch sampling latency only consumes 15.5% of the total training time, as opposed to 61.8% in single thread implementation. However, compared with graph sampling, it is non-trivial to address the embedding collection overhead. The datapath is inevitably longer because we need to first copy the required features from host memory to device memory through PCIe. This additional step is long enough to be a deal breaker of a perfect execution pipeline.

In our experiments, we also notice that the feature collection kernel does not fully saturate PCIe bandwidth due to insufficient memory requests to be issued. As shown in the Table below, the average PCIe bandwidth utilization for different batchsizes is $32.1 \sim 35.2\%$. Therefore, we projected a theoretical lower bound of feature collection latency as shown in the second line of Figure 7.4. The result indicates that improving PCIe utilization with locality-enhancing techniques such as graph partitioning is beneficial, but insufficient to address the problem, as the total latency of sub-graph preparation is still longer than the combined latency of GPU forward and backward propagation.

Table 7.1: Avg. PCIe utilization under different batchsizes.

Batchsize	500	1000	2000	4000
Utilization(%)	33.18	32.10	34.10	35.20

In summary, to fully address the subgraph preparation problem, a more effective way is to shorten the datapath by storing the embedding matrix closer to computation resources. In this work, we achieve this by utilizing a much more compact embedding representation structure. We also customize the system dataflow and hardware accelerator which enables a more efficient on-chip GNN training scheme.

7.2.3 TT Decomposition and TT Representation

Before going into the details of TT-GNN, we introduce the fundamental idea of using Tensor-train Decomposition (TTD) to compress a matrix. TTD has been originally proposed as a generalization of Singular Value Decomposition for high order tensors [40]. Given a d-dimension tensor $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_d}$, TTD decomposes it into a sequence of

3-dimension tensors. Therefore, each scalar in $\mathcal{A}$ can be derived as follows:

$$\mathcal{A}(i_1, i_2 \cdots, i_d) \approx \mathcal{G}_1(:, i_1, :)\mathcal{G}_2(:, i_2, :) \cdots \mathcal{G}_d(:, i_d, :). \tag{7.1}$$

$\mathcal{G}_k$ is a tensor of size $r_{k-1} \times I_k \times r_k$, where r_k is called the TT-rank. r_0 and r_d are set to 1 such that the product of the above matrix sequence is a scalar. Other TT-ranks can be either predefined before the decomposition or decided during runtime according to the required decomposition accuracy. Higher TT-ranks increase the decomposition accuracy but also increase the size of the TT-format representation.

Apart from decomposing tensors, TTD can also be utilized to deal with large vectors and matrices. Specifically, in order to apply TTD on a matrix X of size $M \times N$, we need to factorize M into $\prod_{k=1}^{d} m_k$ and factorize N into $\prod_{k=1}^{d} n_k$. This allows us to reformat matrix X as a $2d$-dimension tensor $\mathcal{X} \in \mathbb{R}^{(m_1 \times m_2 \times) \times (m_2 \times n_2) \cdots \times (m_d \times n_d)}$. Thus, the matrix can now be decomposed with TTD and represented as follows:

$$\mathcal{X}((i_1, j_1), (i_2, j_2) \cdots, (i_d, j_d)) \approx \mathcal{G}_1(:, i_1, j_1, :) \cdots \mathcal{G}_d(:, i_d, j_d, :). \tag{7.2}$$

Prior works have leverage TTD to compress weight matrices in Neural Network models, such that the number of model parameters is significantly reduced [163, 164, 165, 13].

7.3 TT-format GNN Training

In this section, we introduce the workflow of applying Tensor-train decomposition on Graph Neural Networks, which is originally proposed in [14]. Essentially, we need to add a one-time preprocessing step prior to the model training to define a trainable TT-format embedding. The key idea is to align graph topological information with the Tensor-train data structure. Specifically, as shown in Figure 7.5, we first perform a

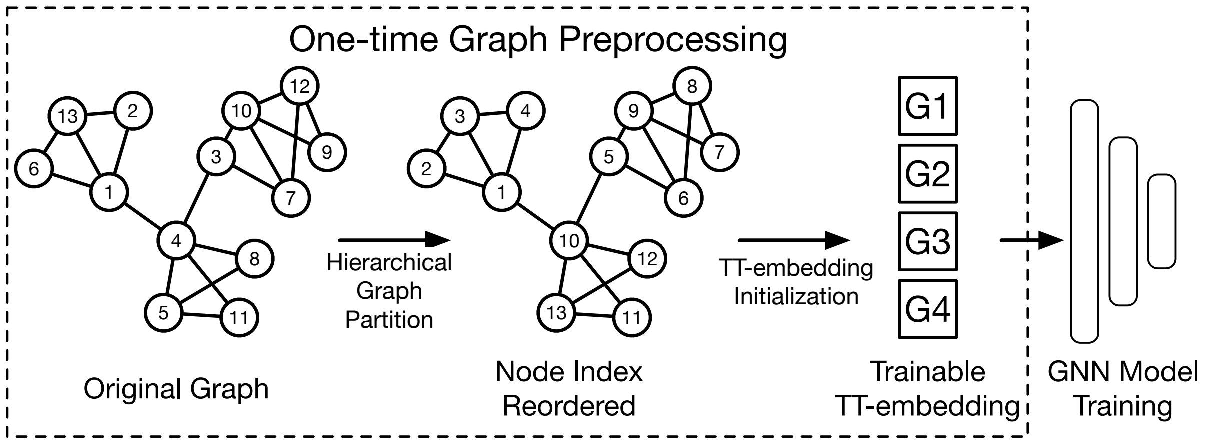

One-time Graph Preprocessing
Original Graph
Hierarchical Graph Partition
Node Index Reordered
TT-embedding Initialization
G1
G2
G3
G4
Trainable TT-embedding
GNN Model Training

Figure 7.5: Illustration of TT-GNN workflow.

hierarchical graph partition (e.g., METIS [166]) to group the nodes into multiple levels of clusters. Then, we reorder the graph nodes based on the partition results, such that nodes in the same partition will have continuous indices. In this way, we can directly reflect graph homophily in the embedding representation. For example, suppose we apply a three-level METIS partition over the graph, which results in a [10,10,10] index system. In this setting, node 101 will be mapped to [1,0,1], and its embedding will be represented by $\mathcal{G}_1(:,1,:,:) \cdot \mathcal{G}_2(:,0,:,:) \cdot \mathcal{G}_1(:,1,:,:)$. Similarly, node 102 will be mapped to [1,0,2], and node 312 will be mapped to [3,1,2]. As a result, node 101 and 102 will share the first two tensor core representations, while being more different from node 312. In this way, we are able to adjust the degree of feature sharing across different nodes by reordering the node indices according to the neighborhood similarity.

Originally, each node is represented with a feature vector of length F, and all the node features together form a 2D feature matrix $X \in \mathbb{R}^{N \times F}$ $(N = |V|)$. By applying TTD to X, the feature matrix is now represented as:

$$\mathcal{X} = \mathcal{G}_1 * \mathcal{G}_2 * \cdots * \mathcal{G}_d \tag{7.3}$$

where $\mathcal{G}_i \in \mathbb{R}^{r_{i-1} \times n_i \times f_i \times r_i}$, $N = \prod_{i=1}^{d} n_i$ and $F = \prod_{i=1}^{d} f_i$.

To extract the k^{th} row from the feature matrix, it is equivalent to first finding the projection index $(n_0^k, n_1^k, \cdots, n_d^k)$, fixing each corresponding n-index in $\mathcal{G}_k$, and finally calculating the product of the tensor sequence.

$$X(k,:) = \mathcal{G}_1(:, n_0^k, :, :) * \mathcal{G}_2(:, n_1^k, :, :) * \cdots * \mathcal{G}_d(:, n_d^k, :, :) \tag{7.4}$$

Finally, as shown in Figure 7.5, after defining the TT embedding structure and node indices with graph partitioning results, we further need to initialize the TT-format embedding parameters. Prior work [14] has demonstrated the superiority of orthogonal initialization regarding convergence effectiveness. In TT-GNN we adopt the same strategy to initialize the parameters, and the TT-format embedding will be jointly trained with the GNN model.

7.3.1 Compression Ratio and Model Accuracy

Since Tensor-train allows partial feature sharing across graph nodes, it is naturally a much more compact embedding representation. Before we need $O(NF)$ space to store the uncompressed features, with TT-GNN, we only need $O(dNf_ir^2)$ elements to represent all the node features in the graph. To provide an intuition, Reddit[29] contains 232965 nodes and the length of each feature vector is 602. In our experiments, we have $d = 7$, $r = 5$, n_i and f_i within $[3, 5]$. Therefore, the compression ratio is 60976×, reducing the size of the embedding matrix from 534.99 MB to 8.98 KB.

In the table below we list the accuracy and compression ratio (CR) of TT-GNN on different benchmarks. We compare TT-GNN with two baselines, **ORIG EMB** means training the GNN model on the original embedding matrix, and **TRAINABLE** means training a 2D embedding together with the GNN model. As we can see from the results, TT-GNN achieves orders of magnitude compression ratio and better accuracy compared with 2D trainable embeddings. On the other hand, applying TT causes accuracy degradation on certain benchmarks. Overall, TT-GNN is more suitable under the scenario where we lack node features, thereby requiring learning the embeddings during training [14].

Table 7.2: TT-GNN Accuracy and Compression Ratio.

Dataset	Orig EMB	Trainable	TT	CR
Cora	81.4%	60.7%	78.1%	6189×
Reddit	95.6%	91.1%	93.3%	60976×
ogbn-arxiv	72.3%	72.1%	72.2%	32546×
ogbn-products	78.9%	73.4%	74.2%	132268×

7.4 Challenge and Opportunity

In this section, we describe the opportunities and challenges when adopting TT-GNN for efficient training of Graph Neural Network models. We also present the experiments and preliminary analysis that we conducted, which leads to the dedicated architecture and dataflow in the following section.

The straightforward benefit of using a compressed format embedding is that we can store it closer to the compute unit, thus reducing the time required for fetching these embeddings for training. As mentioned earlier in section 7.2, moving the embedding to GPU's HBM is efficient enough to hide the embedding fetching latency. While this seems to be a free lunch for TT-GNN, it also leads to new hardware challenges.

Decompression Overhead: The new TT-format embedding brings us a significant compression ratio but also introduces computation overhead when we decompress the TT-feature back to the original feature vector. As shown by equation 7.4, fetching one feature vector now becomes a sequence of matrix multiplication, as we need to gradually contract out all the rank dimensions when recovering the embedding. To provide some intuition over the cost, we compare the theoretical decompression complexity to the computation cost of forward propagation of the GraphSAGE [29] model on the Reddit dataset. The GraphSAGE model has two graph convolution layers, with a neighbor fan-out to be $\{10, 25\}$. The forward function can be expressed as equation 7.5. Since TT-rank affects the computation complexity of the decompression, we sweep over multiple possible rank values. We also select different batchsizes as it will influence the portion of shared neighbors, and eventually the decompression complexity as well.

$$h_v^k = \sigma(\mathbf{W} \cdot MEAN(\{h_v^{k-1}\} \cup \{h_u^{k-1}, \forall u \in \mathcal{N}(v)\})) \tag{7.5}$$

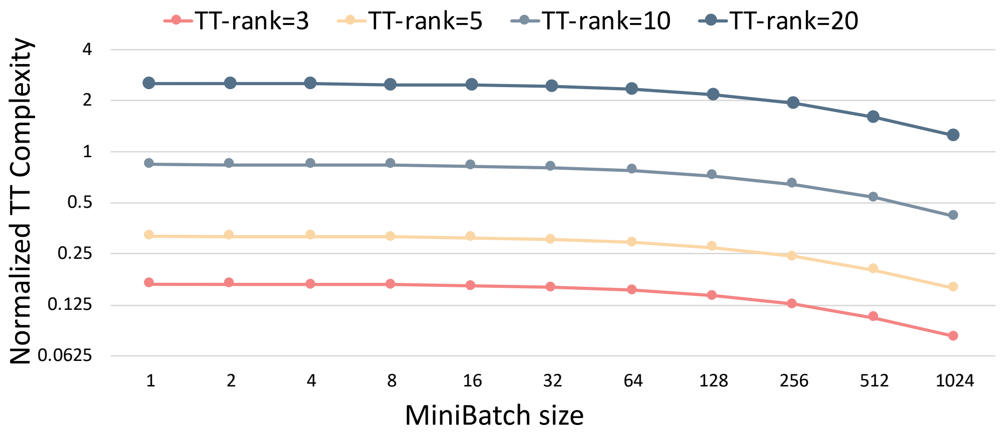


Figure 7.6: Per-minibatch computation complexity of TT decompression relative to the forward propagation complexity of a two-layer GraphSAGE model.

The results are shown in Figure 7.6. For each minibatch size and each rank value, we normalize the computation cost of TT-decompression to the cost of forward propagation. The first thing to be noticed is that, TT computation overhead increases exponentially with the rank values. The cost of decompressing one minibatch is almost the same as running the whole network when TT-rank is equal to 10, not to mention an even larger rank value. Secondly, TT-GNN is in favor of larger minibatch sizes. This is because when more target nodes are considered in one minibatch, they will share more common neighbors at the input layer, resulting in sublinear increase of the input nodes. On the other hand, the forward propagation cost is mainly affected by the number of sampled edges, which is decided by the preset fan-out as long as the nodes have enough neighbors to be sampled. In conclusion, the decompression of TT-GNN can have a comparable cost as running the GNN model, thus should be efficiently handled.

Trading Computation for Memory Efficiency In the problem above we argue that TT-GNN prefers a larger minibatch size, as more shared neighbors help avoid redundantly decompressing the same input nodes. However, as we further show with Table 7.3, this strategy only holds true when prior decompressed features can be cached on-chip. In Table 7.3 we compare the energy consumption of accessing one original feature vector from HBM, with the energy consumption of accessing the corresponding TT-format embedding in an SRAM buffer and decompressing it on-chip. For HBM estimation, we borrow the data from prior work [167] and assume a 3.97 pJ/bit of energy consumption. We use CACTI [91] to get the simulated result of the SRAM buffer and borrow data from prior work [168] to estimate the energy consumption of floating point operations. From the comparison, we find that when using a relatively small rank value, directly performing TT-decompression on-chip consumes less energy compared with fetching the feature vector from off-chip memory. This indicates a potential design choice to eliminate off-chip feature access by performing TT decompression whenever needed. The challenges, however, are of two folds. On one hand, using small rank values will introduce larger compression errors, which may have a negative impact on model accuracy. On the other hand, replacing memory access with TT decompression will cause a massive amount of features to be recomputed. We want to reduce such repetitive computation as much as possible by efficiently utilizing limited on-chip memory.

Table 7.3: Energy consumption comparison between fetching original feature from off-chip HBM and decompressing corresponding TT-feature from on-chip SRAM.

Dataset	TT(r=3)	TT(r=5)	TT(r=10)	TT(r=20)
SRAM(pJ)	633	1339	4099	13882
TT-decomp.(pJ)	13082	41860	222640	1332160
HBM(pJ)	64075	64075	64075	64075

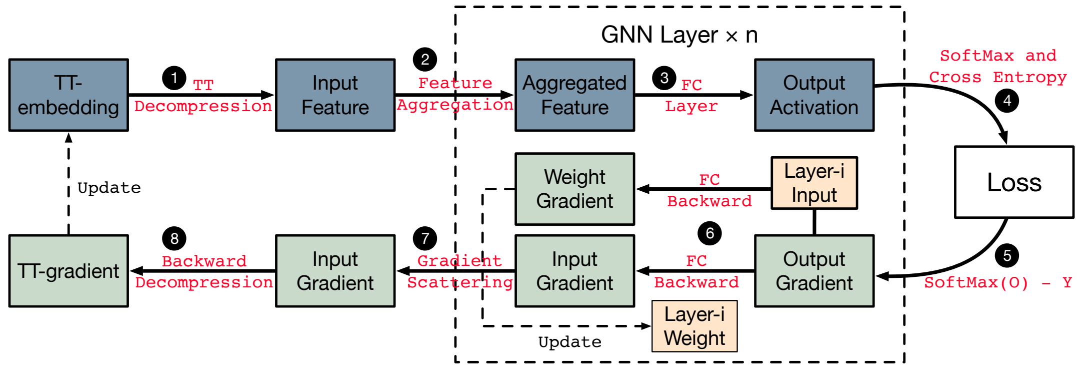


Figure 7.7: TT-GNN Training dataflow

7.5 TT-GNN Training Dataflow

To exploit the algorithmic potential of tensor-train, we present the TT-GNN dataflow in this section. Overall, we address the training problem with a top-down design, as we gradually decompose the problem to be fitted on-chip. Specifically, the proposed dataflow mainly consists of three main parts. (1) To completely eliminate off-chip memory access under dynamic training configurations (e.g., minibatch size, GNN configuration), we introduce the Hybrid Minibatch-Microbatch tiling strategy to adaptively control the size of the subgraph being trained on the accelerator. To reduce the redundant computations caused by neighbor sharing across microbatches, as well as maximize data reuse within each microbatch, we customize the microbatch composition and scheduling order. (2) We propose a unified algorithm to handle TT decompression during forward pass and TT-gradient computation during backward pass. The proposed algorithm exploits data reuse among these two operators and provides a flexible mechanism to trade-off between compute efficiency and memory consumption. (3) Finally, we improve the aggregation and gradient scatter efficiency by offline reorganizing the microbatch subgraph as soon as it is generated. In this section, we provide a detailed walkthrough of our TT-GNN dataflow assuming a two-layer (two levels of neighbor fan-out) GraphSAGE model with a $MEAN$ function as the aggregating operator.

7.5.1 Highlevel Training Dataflow

Figure 7.7 presents the computation graph of a TT-GraphSAGE model. We use squares to indicate the data at each layer and use arrows to illustrate the operations that transform these data between each other. As shown in the figure, 182 the forward propagation starts with a TT-layer, where the TT-format embeddings will be decompressed into a minibatch of input vectors to be sent to the model. The decompression operation, as we show in section 7.2, is essentially a sequence of small tensor contractions which can be implemented as matrix multiplications. 183 After we obtain these input feature vectors, each node in the hidden layer will fetch its neighbor features and perform the aggregation function. In this case, the aggregation is simply a $MEAN$ function. 184 The aggregation is followed by an Apply function, where typically the hidden node feature and the aggregated neighbor feature are combined together using a Fully Connected layer to generate the hidden node features. This two-step message passing is repeated n times depending on the number of hidden layers in the GNN model. 185 Finally, we apply the SoftMAX operation to obtain the final classification result.

186 Reversely, the backward propagation starts from the classification loss and ends at the TT-layer. 187 At each GNN layer, the output gradient is first propagated through the NN layer with matrix multiplication. 188 Then, the hidden feature gradient needs

to be scattered back to the input nodes. In other words, the gradient of each hidden node will be scattered and accumulated to all the used input nodes during the forward aggregation. 189 Finally, after the gradient of the model input features is obtained, we use equation 7.6 to compute the gradient of TT-embeddings.

$$\frac{\partial L}{\partial G_i(:, n_i^k, :, :)} = \prod_{j=1}^{i-1} G_j(:, n_j^k, :, :) * \frac{\partial L}{\partial X(k, :)} * \prod_{j=i+1}^{d} G_j(:, n_j^k, :, :) \tag{7.6}$$

7.5.2 From Minibatch to Microbatch

As illustrated in Figure 7.8 (a), the biggest difference between minibatch GNN training and conventional full-batch GCN training is the inconsistent cost of each layer caused by neighbor fan-out. Due to the neighbor sampling mechanism, there will be more and more nodes and edges as we approach the input layer. This also indicates an increasing memory and computation cost. The selection of minibatch size, which is essentially the number of destination nodes (2 in this sample), will also affect the sampled graph size and the corresponding minibatch training cost. Generally, as shown by Figure 7.8 (b), when we process the who minibatch layer by layer, if any of the layers exceeds on-chip memory capacity, we will have to use off-chip memory for temporary storage. The white circles indicate node features stored off-chip, and red dashed lines represent associated off-chip memory access. with Tensor-train format embedding and on-chip decompression, we naturally eliminate inefficient off-chip embedding loading, as shown in Figure 7.8 (c). However, the intermediate node features can still cause off-chip storage. Therefore, we propose to further break the minibatch into smaller groups which we called **microbatch**, which can be completely fitted on-chip.

Intuitively thinking, a microbatch can be obtained by simply selecting a portion of the destination nodes from the original minibatch. As shown in Figure 7.8 (d), a smaller

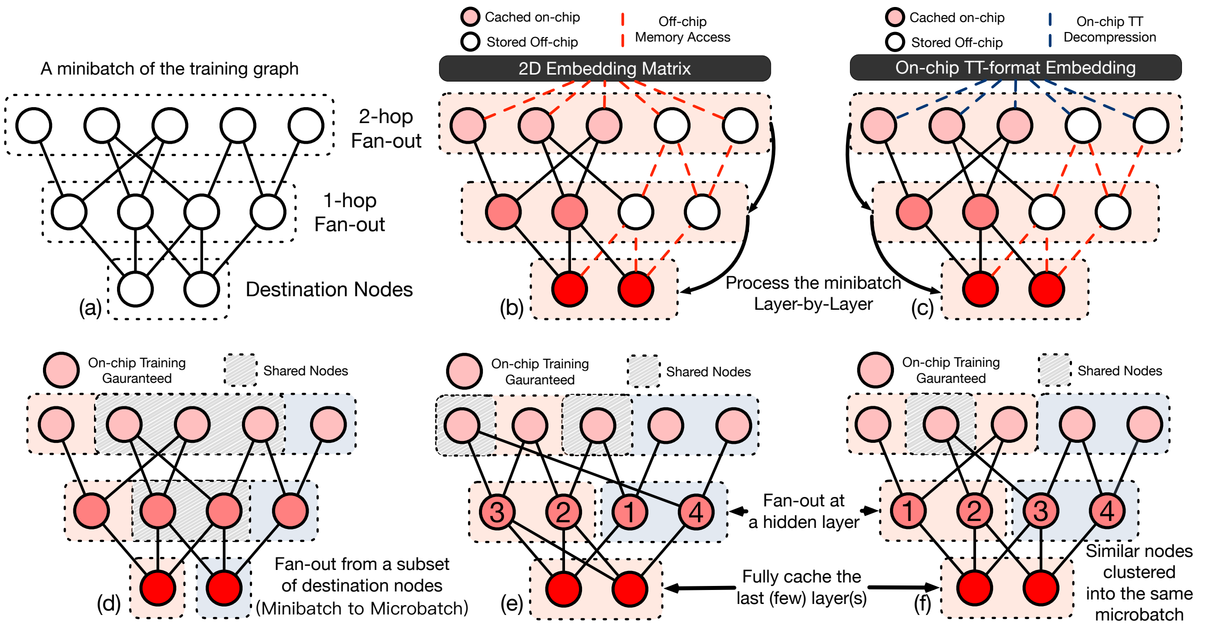

Cached on-chip
Stored Off-chip
Off-chip Memory Access
On-chip TT Decompression
A minibatch of the training graph
2D Embedding Matrix
On-chip TT-format Embedding
2-hop Fan-out
1-hop Fan-out
Destination Nodes
Process the minibatch Layer-by-Layer
(a)
(b)
(c)
On-chip Training Gauranteed
Shared Nodes
Fan-out from a subset of destination nodes (Minibatch to Microbatch)
Fan-out at a hidden layer
Fully cache the last (few) layer(s)
Similar nodes clustered into the same microbatch
(d)
(e)
(f)
1
2
3
4

Figure 7.8: Step-by-step walk-through of TT-GNN's Hybrid Minibatch-Microbatch on-chip training dataflow.

subgraph can be sampled from the selected nodes and their neighborhoods. This is equivalent to setting the minibatch size to be a smaller value in the first place, except that we do not update the model parameter after the backward pass of the microbatch. However, this naive strategy will incur redundant computations and memory access across different microbatches. In this example, suppose we are breaking this minibatch with 2 destination nodes into two microbatches, each with 1 destination node. Due to neighborhood sharing, although the destination nodes of the two microbatches are completely different, they could share common nodes in the hidden layer, and even more in the input layer. Consequently, all the computations related to these shared nodes will be redundantly computed unless we can cache the previously computed node features. However, the limited on-chip memory capacity only provides us with a tight reuse distance budget. Even if we can cache the shared nodes, the memory access over the shared nodes is still inevitably repeated across different microbatches. The situation gets worse with larger batchsize, deeper network architecture, and with the added TT-layer at the beginning.

To tackle the above-mentioned challenge and enable efficient on-chip training with as little overhead as possible, we propose our Hybrid Minibatch-Microbatch tiling strategy.

Hybrid Minibatch-Microbatch Tiling: As presented in Figure 7.8 (e), the first insight is that the last few layers in a GNN model are much smaller compared with the beginning layers. In other words, the cost of caching all the destination nodes and their close neighbors is relatively low. Therefore, instead of breaking the minibatch directly from the output layer, we keep the last few layers the same as the original minibatch and start tiling at an intermediate layer. In this example, we reserve the space for all two destination nodes and break the minibatch into microbatches at the hidden layer. The benefit is very straightforward. As we can see from figure (e), for each microbatch, after the target hidden nodes are generated, it can be directly used to compute the last layer. The hidden node feature will be added to the partial sum of the destination nodes which are always on-chip. In this way, there will be no shared hidden nodes across the microbatches, and all the hidden node features only need to be computed and used one single time. We call this Hybrid Minibatch-Microbatch Tiling as it works in a microbatch fashion at first but eventually merges into the minibatch output. Another benefit of using this strategy is that it reduces the number of shared neighbors at the first (few) layers. As shown by the example in Figure 7.8 (e), since each microbatch contains less nodes compared with (d), the shared neighbors in the input layer are also reduced, which leads to fewer redundant TT-decompression.

The method works similarly in backward propagation. First, the gradient of the hidden nodes only needs to be computed and stored for one time as there is no neighbor sharing across microbatches. On the other hand, for shared neighbors in the first layer, the gradient derived from one microbatch is only a partial sum. We seek to avoid

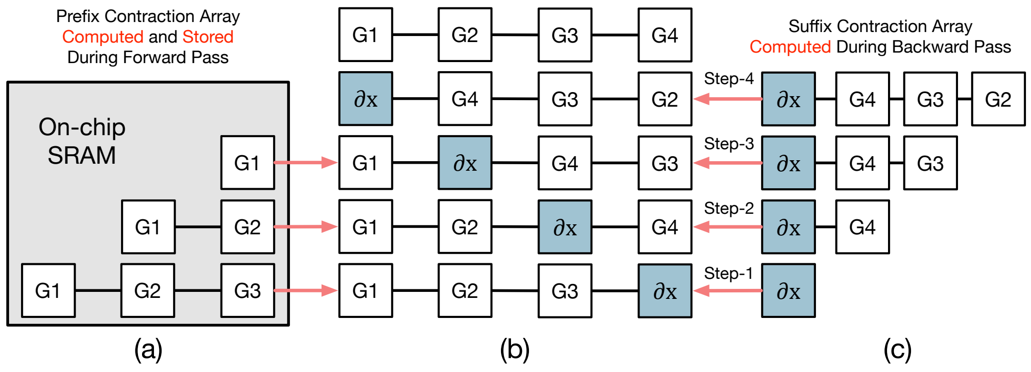


Figure 7.9: The contraction flow of TT decompression as well as the gradient computation during the backward pass.

caching these partial sums to be accumulated because the first layer is the most memory-consuming layer. Therefore, we can directly use the gradient in each microbatch to derive the TT-format gradient of the TT embeddings. The TT-format gradient consumes much less space and is always stored on-chip. An exception is that we will delay the computation of TT gradient only if we know the gradient of a specific node will be accumulated in the next consecutive microbatch (only consider one-step reuse). This information is available to us as we decide the composition and scheduling order of the microbatches when we perform minibatch sampling. In either way, we avoid caching the vector format gradient of the first layer, so that to control memory consumption.

Microbatch Selection and Scheduling Order As mentioned above, the shared neighbors in the first few layers can still cause redundant TT decompression and TT-gradient computation. To address the problem, we further propose to customize the microbatch composition and scheduling order to maximize intra- and inter-microbatch data reuse. Figure 7.8 (d) and (e) provide an illustration. Originally in Figure 7.8 (d), we group node 3 and node 2 into one microbatch, and group node 1 and 4 in another microbatch. This results in two shared neighbors at the first layer. One solution is to schedule these two microbatches next to each other, so that the shared neighbors can be cached on-chip and reused. For another solution, as shown in Figure 7.8 (f), we can group nodes with similar neighborhoods into the same microbatch. In this case, if we select node 1 and 2 to be the first microbatch, and node 3 and 4 to be the second, then there would be only one shared neighbor across these two microbatches. Reducing the overhead of redundant computation even if these two microbatches are not processed consecutively.

As we can see, these two strategies tackle the problem at different levels. Thus, in TT-GNN, we combine them into a unified strategy. Recall that at the beginning of the TT-GNN training, we first reorder the graph nodes according to the METIS partition results. Therefore, the reordered node index naturally indicates neighborhood similarity. In other words, nodes with close index values should be grouped into the same minibatch. Therefore, given a set of hidden nodes to be scheduled, we first sort these nodes according to their indices. After this, we can simply traverse the index list and group consecutive nodes into one microbatch. Besides, the consecutive microbatches will also be scheduled sequentially. In this way, we can efficiently obtain the microbatch composition as well as scheduling order together with one single pass.

7.5.3 Microbatch Dataflow Walk-through

In the above subsections, we have managed to break the minibatch into microbatches with minimized overhead, such that the microbatch can be completely processed on-chip. We further argue that there still exists performance improvement opportunities within each microbatch. Therefore in this subsection, we walk through the forward and backward pass of each microbatch to illustrate our intra-minibatch optimizations.

TT Decompression and Update As shown in Figure 7.7, TT decompression is required during forward propagation, and during the backward pass we need to compute the TT-gradient to update TT-embeddings. The corresponding equations used for these two operations are presented earlier in equation 7.4 and 7.6.

We observe that both TT decompression and TT-gradient can be considered as contracting a tensor-train network. We use Figure 7.9 as an illustration. In this example, we have four TT cores. To obtain an input feature vector from the TT-embedding, we need to extract a small tensor from each TT-core that together forms a tensor-train network. This is shown as the top tensor-train ($G1 - G2 - G3 - G4$) in Figure 7.9 (b). On the other hand, in the backward pass, we need to separately compute the gradient of the four tensors, which is represented as the bottom four tensor-trains in Figure 7.9 (b). As we can see, although the operation is still tensor-train contraction, one of the tensors should be replaced by the gradient of the feature vector.

To effectively explore data reuse in this problem, we propose to compute the required tensor-trains with the combination of prefix and suffix array. As shown in Figure 7.9 (a), during forward pass, we use an array to store the intermediate prefix contraction results. On the contrary, we only need to maintain a single suffix contraction result to generate the output gradient of each tensor. For example, as shown in Step-1 of Figure 7.9 (c), we first use the vector gradient and the cached $G1-G2-G3$ to generate the last tensor-train. Then, we update the suffix contraction result by multiplying ∂x with $G4$, and use another cached prefix result to generate the next tensor-train. Eventually, we can obtain all the TT-gradients with the stored prefix array and a suffix contraction result.

Note that, we are able to flexibly trade-off between compute efficiency and memory consumption with this algorithm. For example, we can choose to skip storing the prefix array during the forward pass and recompute it in the backward pass. This can significantly reduce the memory cost. On the other hand, we can simultaneously compute the prefix and suffix array in the forward pass, thereby reducing the sequential computation flow in the backward pass at the cost of higher memory consumption.

Neighbor Aggregation and Gradient Scatter Neighbor aggregation and gradient scatter are two important operations in a GNN model. During forward pass, we collect the neighbor information of each target node and generate the aggregated feature vector. In the back pass, we need to scatter the gradient of the target node back to all its neighbors. From a message flow perspective, these two operations are reversed from each other. However, computationally both of them can be formulated into a Sparse-Dense Matrix Multiplication (SpMM) operation, where the sparse matrix operator is the adjacency matrix of the subgraph. Moreover, the sparse matrix of the scatter SpMM is simply the transpose of the aggregation SpMM.

To improve the compute efficiency of such SpMM operation, prior works have proposed searching algorithms [169, 170] to exploit intermediate data reuse. The key idea is to introduce a new set of aggregation nodes, where these nodes are essentially the partial sums of the input nodes. By identifying the popular partial sums as the aggregation nodes, we can avoid redundantly aggregating the associated input features, with very little memory overhead. In TT-GNN, we use a similar method but apply it to both forward pass and backward pass to save computations.

7.6 System and Accelerator Architecture

In this section, we introduce the complete design of the TT-GNN training system. The overall system-level architecture is presented in Figure 7.10. The proposed training dataflow is implemented in a dedicated accelerator, which is further attached to a host processor. Since TT-GNN does not compress the graph structure, the adjacency list is stored in the host memory. During training, the host processor is responsible for sampling minibatches from the graph adjacency list. To facilitate an efficient on-chip learning procedure, the host processor will further execute two tasks. (1) It will analyze the memory

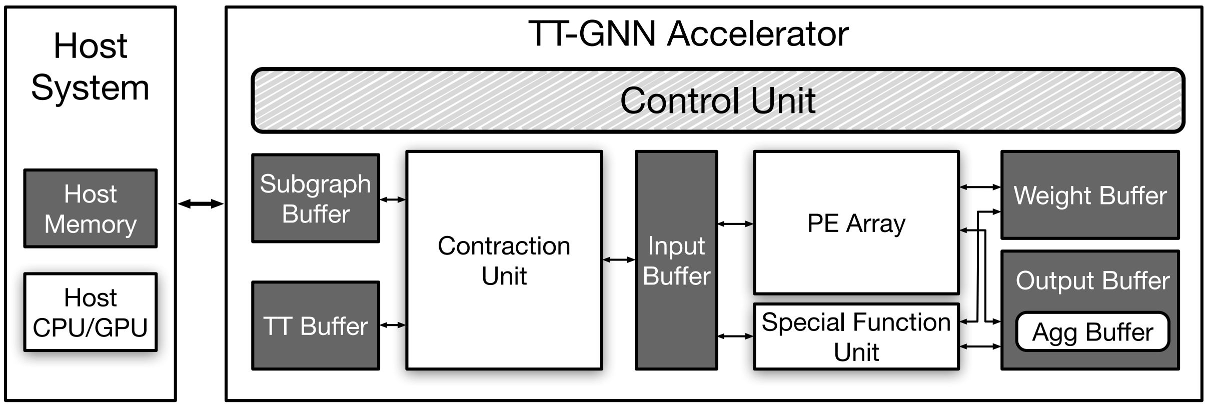


Figure 7.10: Overview of the TT-GNN Training System.

consumption of the minibatch and decompose the minibatch into microbatches if necessary. The procedure used for microbatch selection and scheduling is already discussed above in Section 7.5.2. (2) After the microbatches are decided, the host processor will further preprocess the compute graph to identify the intermediate aggregation set. As we mentioned in Section 7.5.3, this helps improve the SpMM efficiency. As soon as one microbatch is generated, it will be pushed to a task queue together with the dataflow configuration. The accelerator will execute the microbatch training based on the scheduled tasks. At the same time, the host processor can simultaneously prepare multiple minibatches and generate the associated microbatches.

As shown in Figure 7.10, TT-GNN accelerator mainly consists of the following modules: (1) A Contraction Unit that handles TT-decompression and TT-gradient computation. (2) A PE Array that is responsible for GNN related operations, including FC forward and backward computation, neighbor aggregations, as well as gradient scattering. (3) On-chip SRAM modules that store different types of data, including TT embeddings, microbatch subgraph structure, dataflow configuration, node features, model parameters, and all the computed gradients. (4) An overall Control Unit that orchestrates the mem-

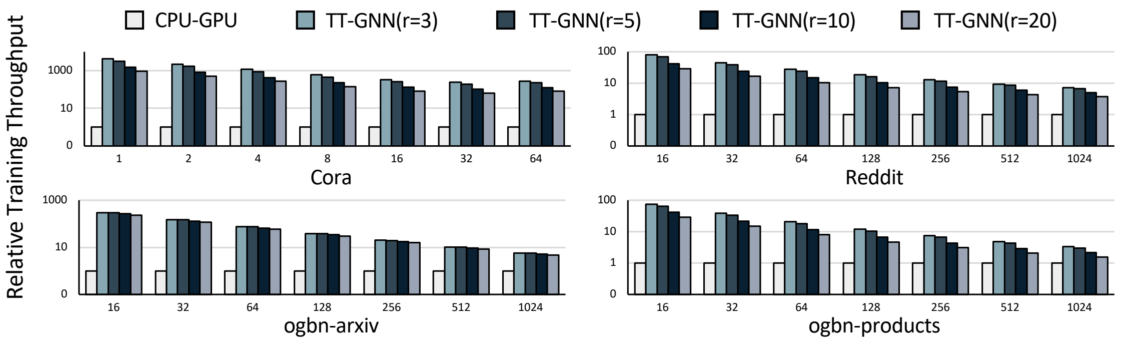


Figure 7.11: Relative training throughput compared with baseline CPU-GPU system.

ory and computation resources using the dataflow configuration file provided by the host processor.

Contraction Unit and PE Array Although the TT Contraction Unit and the PE Array handle different stages of the GNN training, the underlying computation pattern is common. For TT-decompression and TT-gradient computation, the operation is tensor-train contraction, which can be further decomposed into sequences of matrix multiplications. For GNN-related computation, the PE array takes care of matrix multiplication in the FC layer and the vector-wise addition used during aggregation and gradient scattering. Therefore, both TT contraction Unit and PE Array adopt a classic 2D Mac array architecture so that we can efficiently map the parallel vector operations to the modules. We decouple the design of the Contraction Unit as well as the PE array so that they can operate in a pipelined manner. Since we do not need to update the TT-embeddings across different microbatches, we can decompress the input node features for the next microbatch while processing the forward and backward pass of the current microbatch.

Special Function Unit The Special Function Unit incorporates floating point arithmetics that can handle functions including division, exponential operations, modular operations, and so on. These basic operators are composed together to implement SoftMax

Table 7.4: Summary of dataset statistics

Dataset	#Node	#Edge	#Label	Feat Len
Cora	2,708	10,556	7	1,433
Reddit	232,965	114,615,892	41	602
ogbn-arxiv	169,343	1,166,243	40	128
ogbn-products	2,449,029	61,859,140	47	100

function, index projection between node IDs and TT-index, Optimizer-related computations (e.g., parameter update in Adam [171]), batch normalization, and so on.

On-chip Memory TT-GNN has multiple on-chip SRAM buffers for storing different types of data used during training. The TT-embeddings and TT-gradients are stored in *TT*-Buffer. The microbatch graph structure, as well as the dataflow configuration file generated by the host processor, are stored in the *Subgraph*-Buffer. The *Input*-Buffer caches the decompressed input node features before being processed by the GNN model. It also stores the vector-format feature gradient. *Weight*-Buffer stores GNN model parameters and parameters gradients. *Output*-Buffer caches all the activation maps as well as the gradients of the hidden nodes. Finally, we specifically allocate a fraction from the *Output*-Buffer as the *Aggregation*-Buffer to store intermediate aggregated partial sums. As we discussed in Section 7.5.3, this helps improve the computation efficiency of Neighbor Aggregation and Gradient Scattering. The size of the *Aggregation*-Buffer is configurable and will be adjusted depending on the benchmark characteristics. This information is obtained from microbatch generation and is included in the microbatch configuration file.

7.7 Evaluation Methodology

In this section, we present the designed experimental methodology to evaluate TT-GNN.

7.7.1 Benchmark and Implementation

We implement both TT-GNN training and baseline GNN model training with Deep Graph Library [161]. We also implement the proposed Microbatch generation and preprocessing strategy in software and integrate it into the training pipeline. After a minibatch is sampled by CPU, the subgraph will go through the microbatch generation stage. We take the model architecture, training configuration, as well as TT-GNN hardware parameter as input and generate a sequence of microbatches tailored for on-chip TT-GNN training. Finally, we use GraphSAGE [29] as the model architecture and select a series of GNN benchmarks to evaluate TT-GNN, including Cora, Reddit, and two node property prediction datasets from Open Graph Benchmark [162]. The basic attributes of each graph benchmark are listed in Table 7.4.

7.7.2 Hardware Performance

Hardware Implementation and Modeling. The system configuration and hardware consumption of TT-GNN are shown in Table 7.5. Power and area statistics of customized modules are obtained from synthesizing RTL implementation using Synopsys Design Compiler under TSMC 22nm standard cell library. The latency, power, as well as area of SRAM modules, are simulated with CACTI [91]. For performance and energy-efficiency evaluation, we implement a custom simulator that is integrated with the software framework to capture real training traces. The simulator is also used to perform design space exploration.

Hardware Baseline We compare TT-GNN with a standard CPU-GPU training system containing a single Nvidia 3090 GPU and an AMD Ryzen Threadripper 3970X 32-Core CPU. In the baseline system, graphs are originally stored in host DRAM and loaded to device memory during training. Sub-graph sampling is offloaded to CPU, and

Table 7.5: Configurations, Power, and Area of TT-GNN under 22nm Technology and 1GHz Frequency. The on-chip SRAM is divided into a 64KB Subgraph Buffer, a 64KB TT Buffer, a 32MB Input Buffer, a 2MB Weight Buffer, and a 4MB Output Buffer.

Hardware Module	**Configuration**	**Power**(mW)	**Area**(mm^2)
Contraction Unit	16×16 FP16 MAC	441.94	0.41
PE Array	32×16 FP16 MAC	968.97	0.93
SFU	16 Exp, 16 Div	78.21	0.071
SRAM Buffer	38.125MB	1048.44	27.15

we issue multiple threads to achieve the shortest sampling latency. For TT-GNN, the TT-format embedding can be stored on-chip, while the graph edge list and sub-graph sampling are executed on the host system. For performance comparison, we scale up TT-GNN's configuration to have the same peak computation throughput as the 3090 GPU.

7.8 Evaluation Results

With the above experimental methodology, we present the evaluation of TT-GNN in this section.

7.8.1 Performance Evaluation

Training Throughput

We first compare the training performance between TT-GNN and the baseline CPU-GPU training system across different benchmarks and minibatch sizes. As shown in Figure 7.11, overall, TT-GNN achieves 1.55∼4210× throughput improvement over the baseline. The speedup mainly comes from three aspects. First, TT-GNN avoids fetching off-chip embedding through effective compression and on-chip decompression. This significantly reduces the latency of minibatch collection. In our design, the minibatch sampling and TT decompression are pipelined with the on-chip GNN training. This improves compute resource utilization while also hides the subgraph preparation latency. Second, the proposed Hybrid Minibatch-Microbatch dataflow improves both inter- and intra-microbatch data reuse to reduce computation and memory access. Finally, we leverage aggregation redundancy within each microbatch subgraph by caching the partial sums during neighbor aggregation and gradient scattering. Besides the overall trend, we also observe that TT-GNN achieves higher speedup under smaller batchsizes. This is because GPU suffers from severe resource under-utilization when the batchsize is small. The fixed latency such as kernel launching overhead and idleness caused by subgraph sampling also accounts for a larger fraction with small batchsizes.

Latency Breakdown

Figure 7.12 presents the average latency breakdown of executing one minibatch. Overall, the minibatch sampling and TT computation have a comparable latency with forward and backward propagation. This supports our pipelined design to fully hide the subgraph preparation overhead. Besides, on benchmarks such as ogbn-arxiv, the number of input nodes per destination node is much less. As a result, the computation is more dominated by FC layers, leading to a larger portion of forward and backward propagation. Note that, the TT-rank value will significantly change the complexity of Tensor-train contraction, and thus affecting the latency of TT decompression and TT-gradient computation. In TT-GNN, we reduce this impact by caching the prefix contraction result during the forward propagation, and reuse it for TT-gradient computation. Overall, as we discussed in Section 7.5.3, we are able to generate all the required Tensor-trains with a complexity equal to contracting only two tensor-trains.

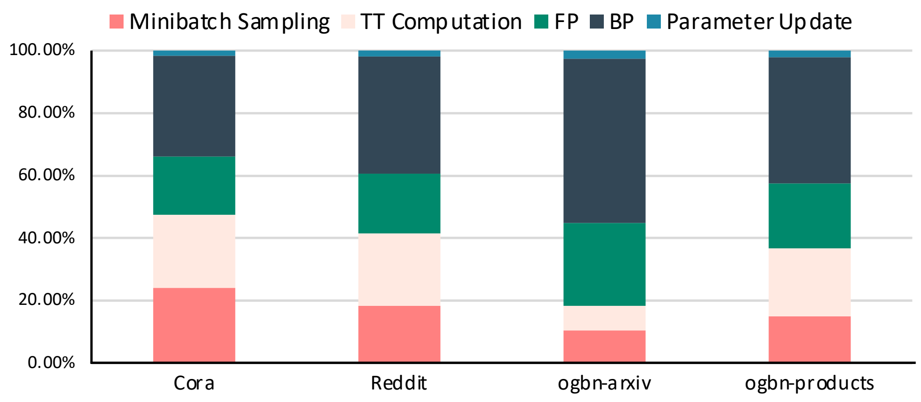


Figure 7.12: Training latency breakdown of one minibatch.

Impact of Different Techniques

We use the Reddit dataset to demonstrate the effect of the proposed techniques. As shown in Figure 7.13, we break down the speedup of TT-GNN to the specific techniques we discussed above. Overall, the specialized accelerator design and on-chip learning mechanism bring 11.2× of performance improvements. This benefit is further amplified by 1.25× with Hybrid Microbatch-Minibatch dataflow, and by 1.11× with the aggregation partial sum reuse. In our experiment, we observe that the benefit of reusing intermediate partial sum is less than the reported number in literature [170, 169]. This is because we can only operate on the microbatch-level compute graph, where neighbor sharing is less effective.

7.8.2 Energy-efficiency

Finally, we show the energy-efficiency improvements of TT-GNN with Figure 7.14. As we can see, TT-GNN has 2.83× to orders of magnitude better energy efficiency than the baseline system. Apart from the natural benefit of using specialized dataflow and ASIC design, the most important advantage is that we completely avoid off-chip memory access

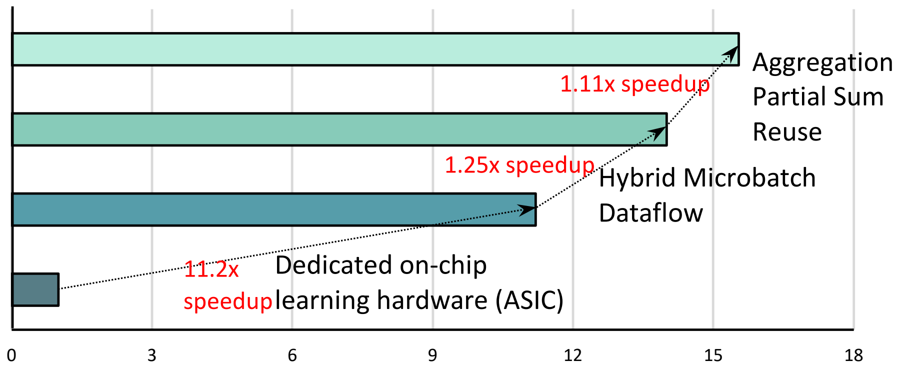


Figure 7.13: Speedup breakdown of TT-GNN on Reddit.

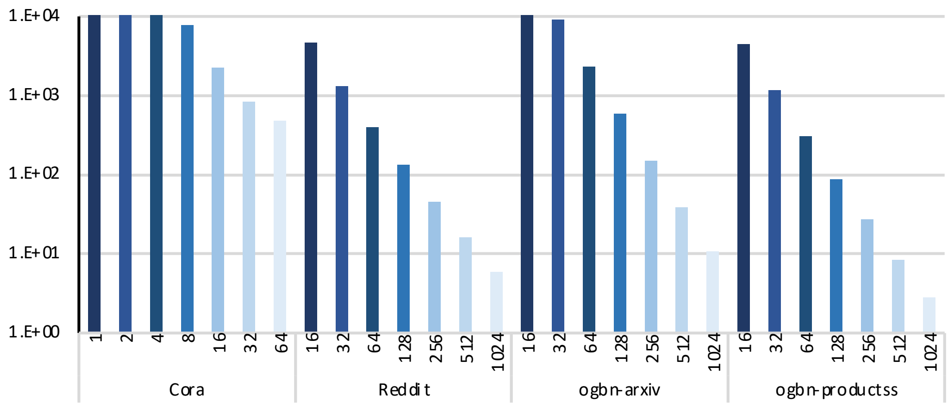


Figure 7.14: Relative energy-efficiency improvements of TT-GNN over baseline CPU-GPU training system.

during the microbatch execution. This is a significant portion of the energy consumption in the original training setting. Similar to the speedup analysis, the advantage of a dedicated on-chip training accelerator over GPU is larger on smaller batchsizes, as GPU suffers from resource under-utilization and fixed energy consumption.

7.9 Related Work

Tensor-train Decomposition (TTD) [40] is originally used to decompose high order tensor data and break the curse of dimensionality through efficient implementation of basic operations. [64] sees the opportunity of adopting TTD to reduce the modes size and computation complexity of Convolution Neural Networks (CNNs). The key idea is to change the 2D weight matrix into a parameterized tensor-train weight and train the entire model from scratch. In other words, this approach can be regarded as replacing the FC and Conv layer in CNNs with a Tensor-train (TT) layer. TT layer has fewer parameters and less computation complexity for inference. Since then, TT layer has also been used in other DNN models such as RNNs [65] and Transformers [66]. The unique computation pattern of Tensor-train also inspires research effort on customized accelerator design [67] for these Tensorized Neural Networks (TNNs).

Recently, [13] proposes to use TT-layer to replace the large embedding layer used in Deep Learning Recommendation Models. The central idea remains the same as before. The difference is that none of the previous DNN models have such large weight matrix that consumes more than 99% of the model capacity with up to TBs of memory consumption. Therefore, this work demonstrates the important potential of tensor-train method in such extreme-scale models, which could potentially help control the explosive demands on computational infrastructure.

7.10 Conclusion

In this paper, we propose TT-GNN, a training system that adopts Tensor-train Decomposition to compress the memory-consuming feature embedding matrix, which leads to an on-chip learning implementation. TT-GNN adaptively breaks down a minibatch into smaller microbatches that can be fitted on-chip. The microbatch composition and scheduling order are designed to maximize data reuse and reduce redundant computations both across and within microbatches. We also propose a unified algorithm to jointly handle TT decompression during forward propagation and TT gradient derivation during backward propagation. Combining the software and hardware optimizations, the proposed software-hardware solution is able to outperform existing CPU-GPU training systems on both training performance ($1.55 \sim 4210\times$) and energy efficiency ($2.83 \sim 2254\times$).

Chapter 8

Conclusion and Future Work

In this thesis, we discuss the systematic and architectural challenges brought by the emerging development of machine learning models. We identify the root cause being the explosion of data used by these applications, including explosive training dataset, ever-scaling model parameters, and exponentially increasing intermediate results. We envision a future where such a situation will continue to be more and more severe, making the gap between algorithm and hardware systems even larger.

Therefore, to tackle the problem and maintain a healthy development of the AI industry, we argue that it is important to intelligently identify and leverage different types of data redundancy in these emerging models. To achieve this, we need to understand both the machine learning models as well as the underlying hardware platform. Most time an algorithm and hardware co-design is required to achieve final performance and energy-efficiency improvements. Specifically in this thesis, we propose two different methods that correspond to two types of data redundancy in existing machine learning models that are rarely discussed in prior work.

In large models like Transformers, the cost of computing self-attention scales quadratically with respect to the input sequence. However, a lot of connections evaluated by the dense attention are unnecessary. Therefore, we propose to use approximated attention as a means of estimating the importance score and thus avoid computing the unimportant attention. Such approximation-based redundancy elimination is very useful when the intermediate activation contains many unimportant values which have a very limited impact on the final output. Similar use cases include extreme classification [172], CNNs [173], and RNNs [173]. Adopting runtime attention approximation introduces low-precision operators as well as sparse operators to the model. Therefore, depending on the underlying platforms, we customize our hardware implementation strategies as well. For GPU acceleration, we introduce small structure sparsity to balance kernel efficiency and model accuracy. On the other hand, we fully support irregular sparse attention with specialized accelerator architecture design.

There are cases where large datasets or embedding tables are utilized in a neural network model, such as DLRM [3] and GNN [29]. Conventional model compression techniques fail to address such data explosion issues due to insignificant reduction ratios. Furthermore, these applications usually contain inter-node or inter-sample similarity which indicates opportunities to adopt a more compact data representation. Based on this analysis, we propose two software-hardware co-designs utilizing Tensor-train decomposition to reduce the cost of data representation in emerging neural network models. Adopting Tensor-train to the data representation completely changes the computational pattern, which leads to a differentiated hardware system design.

With the contribution of this dissertation, we would like to emphasize the following statements:

- The size of the neural network models will continue to scale at an extremely fast speed, causing more challenges to the underlying hardware systems.
- A possible angle to approach this problem is by analyzing it from a data-centric

perspective. This is because any type of neural network model can be regarded as an interacting system between the dataset, model parameter, and computation results (activations). The data explosion always happens among one or multiple types of data in this problem formulation. Therefore, we can identify different types of data redundancy based on this formulation and address the issue directly from the root cause.

- While application-level hints can lead to innovative algorithmic designs to exploit data redundancy, achieving practical hardware performance improvements requires more effort. Therefore, algorithm, software, and hardware co-design is an effective way to trade-off between model performance and hardware efficiency.

Finally, with the booming development of large foundation models, future neural network models are more likely to adopt a Transformer-based architecture. Although this may reduce the diversity of neural network architecture, the scaling problem still exists, and our data-centric problem formulation also stands. As a result, efficient and innovative techniques for leveraging data redundancy will always be an important direction. Furthermore, when solving domain-specific problems, the ability of current LLMs is still insufficient. Therefore, it would be promising to apply domain-specific models like CNNs and GNNs to the design of such domain-specific large foundation models. The resulting large models will exhibit features from both Transformers and prior domain-specific models. Thus, experience from accelerating prior backbone models will guide future hardware solutions as well. We wish the idea and practice from this dissertation can serve as a basis to help push forward the research of future AI software hardware co-optimization.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *Imagenet classification with deep convolutional neural networks*, *Commun. ACM* **60** (may, 2017) 84–90.

[2] J. Devlin, M.-W. 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, Volume 1 (Long and Short Papers)*, pp. 4171–4186, 2019.

[3] M. Naumov, D. Mudigere, H. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy, *Deep learning recommendation model for personalization and recommendation systems*, *CoRR* **abs/1906.00091** (2019) [arXiv:1906.0009].

[4] M. Réau, N. Renaud, L. C. Xue, and A. M. J. J. Bonvin, *DeepRank-GNN: a graph neural network framework to learn patterns in protein–protein interfaces*, *Bioinformatics* **39** (11, 2022) [https://academic.oup.com/bioinformatics/article-pdf/39/1/btac759/48448995/btac759_supplementary_data.pdf]. btac759.

[5] K. Guo and M. J. Buehler, *Rapid prediction of protein natural frequencies using graph neural networks*, *Digital Discovery* **1** (2022) 277–285.

[6] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, *Scaling laws for neural language models*, *CoRR* **abs/2001.08361** (2020) [arXiv:2001.0836].

[7] OpenAI, *Ai and compute*, .

[8] MIR, *Training a single ai model can emit as much carbon as five cars in their lifetimes*, .

[9] S. Han, H. Mao, and W. J. Dally, *Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding*, in *4th International Conference on Learning Representations, ICLR 2016, San Juan,*

Puerto Rico, May 2-4, 2016, Conference Track Proceedings (Y. Bengio and Y. LeCun, eds.), 2016.

[10] G. Hinton, O. Vinyals, and J. Dean, *Distilling the knowledge in a neural network*, 2015.

[11] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, *A survey of quantization methods for efficient neural network inference*, *CoRR* **abs/2103.13630** (2021) [arXiv:2103.1363].

[12] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, *Imagenet: A large-scale hierarchical image database*, in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[13] C. Yin, B. Acun, X. Liu, and C.-J. Wu, *Tt-rec: Tensor train compression for deep learning recommendation models*, 2021.

[14] C. Yin, D. Zheng, I. Nisa, C. Faloutos, G. Karypis, and R. Vuduc, *Nimble gnn embedding with tensor-train decomposition*, 2022.

[15] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, *Eie: efficient inference engine on compressed deep neural network*, in *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pp. 243–254, IEEE, 2016.

[16] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, *Ese: Efficient speech recognition engine with sparse lstm on fpga*, in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75–84, ACM, 2017.

[17] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, *Cnvlutin: Ineffectual-neuron-free deep neural network computing*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–13, IEEE, 2016.

[18] D. Kim, J. Ahn, and S. Yoo, *A novel zero weight/activation-aware hardware architecture of convolutional neural network*, in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1462–1467, March, 2017.

[19] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, *Attention is all you need*, in *Advances in Neural Information Processing Systems*, pp. 6000–6010, 2017.

[20] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, *Improving language understanding by generative pre-training*, .

[21] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, *Language models are few-shot learners*, 2020.

[22] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, *Exploring the limits of transfer learning with a unified text-to-text transformer*, *arXiv preprint arXiv:1910.10683* (2019).

[23] N. Carion, F. Massa, G. Synnaeve, N. Usunier, A. Kirillov, and S. Zagoruyko, *End-to-end object detection with transformers*, in *European Conference on Computer Vision*, pp. 213–229, Springer, 2020.

[24] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, *Image transformer*, in *International Conference on Machine Learning*, pp. 4055–4064, PMLR, 2018.

[25] R. Child, S. Gray, A. Radford, and I. Sutskever, *Generating long sequences with sparse transformers*, 2019.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, *Language models are unsupervised multitask learners*, *OpenAI blog* **1** (2019), no. 8 9.

[27] M. Chen, A. Radford, R. Child, J. Wu, H. Jun, D. Luan, and I. Sutskever, *Generative pretraining from pixels*, in *Proceedings of the 37th International Conference on Machine Learning* (H. D. III and A. Singh, eds.), vol. 119 of *Proceedings of Machine Learning Research*, pp. 1691–1703, PMLR, 13–18 Jul, 2020.

[28] A. Ramesh, M. Pavlov, G. Goh, S. Gray, C. Voss, A. Radford, M. Chen, and I. Sutskever, *Zero-shot text-to-image generation*, 2021.

[29] W. L. Hamilton, R. Ying, and J. Leskovec, *Inductive representation learning on large graphs*, *CoRR* **abs/1706.02216** (2017) [arXiv:1706.0221].

[30] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, *Graph attention networks*, 2018.

[31] Q. Huang, H. He, A. Singh, S.-N. Lim, and A. Benson, *Combining label propagation and simple models out-performs graph neural networks*, in *International Conference on Learning Representations*, 2021.

[32] W. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C. Hsieh, *Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks*, *CoRR* **abs/1905.07953** (2019) [arXiv:1905.0795].

[33] S. Wu, W. Zhang, F. Sun, and B. Cui, *Graph neural networks in recommender systems: A survey*, 2020.

[34] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, *Convolutional networks on graphs for learning molecular fingerprints*, *CoRR* **abs/1509.09292** (2015) [arXiv:1509.0929].

[35] Z. Yang, M. Chakraborty, and A. D. White, *Predicting chemical shifts with graph neural networks*, *bioRxiv* (2020) [https://www.biorxiv.org/content/early/2020/08/27/2020.08.26.267971.full.pdf].

[36] H. Zhao, Y. Wang, J. Duan, C. Huang, D. Cao, Y. Tong, B. Xu, J. Bai, J. Tong, and Q. Zhang, *Multivariate time-series anomaly detection via graph attention network*, 2020.

[37] D. S. Lopera, L. Servadei, G. N. Kiprit, S. Hazra, R. Wille, and W. Ecker, *A survey of graph neural networks for electronic design automation*, in *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pp. 1–6, 2021.

[38] Y. Ma, Z. He, W. Li, L. Zhang, and B. Yu, *Understanding graphs in eda: From shallow to deep learning*, *Proceedings of the 2020 International Symposium on Physical Design* (2020).

[39] B. Khailany, H. Ren, S. Dai, S. Godil, B. Keller, R. Kirby, A. Klinefelter, R. Venkatesan, Y. Zhang, B. Catanzaro, and W. J. Dally, *Accelerating chip design with machine learning*, *IEEE Micro* **40** (2020), no. 6 23–32.

[40] I. V. Oseledets, *Tensor-train decomposition*, *SIAM Journal on Scientific Computing* **33** (2011), no. 5 2295–2317, [https://doi.org/10.1137/090752286].

[41] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, *Cambricon-x: An accelerator for sparse neural networks*, in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 20, IEEE Press, 2016.

[42] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, *Scnn: An accelerator for compressed-sparse convolutional neural networks*, in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 27–40, ACM, 2017.

[43] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher, *Ucnn: Exploiting computational reuse in deep neural networks via weight repetition*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 674–687, IEEE, 2018.

[44] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, *Sparten: A sparse tensor accelerator for convolutional neural networks*, in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), pp. 151–165, ACM, 2019.

[45] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, *Scalpel: Customizing dnn pruning to the underlying hardware parallelism*, in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 548–560, ACM, 2017.

[46] H. Kung, B. McDanel, and S. Q. Zhang, *Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization*, *arXiv preprint arXiv:1811.04770* (2018).

[47] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, *Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 15–28, IEEE, 2018.

[48] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, *Permdnn: Efficient compressed dnn architecture with permuted diagonal matrices*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 189–202, IEEE, 2018.

[49] J. Zhu, J. Jiang, X. Chen, and C.-Y. Tsui, *Sparsenn: An energy-efficient neural network accelerator exploiting input and output sparsity*, in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 241–244, IEEE, 2018.

[50] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, and T. Delbruck, *Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps*, *IEEE transactions on neural networks and learning systems* (2018), no. 99 1–13.

[51] M. Mahmoud, K. Siu, and A. Moshovos, *Diffy: a déjà vu-free differential deep neural network accelerator*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 134–147, IEEE, 2018.

[52] J. Zhang, C. Lee, C. Liu, Y. S. Shao, S. W. Keckler, and Z. Zhang, *Snap: A 1.67 — 21.55tops/w sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm cmos*, in *2019 Symposium on VLSI Circuits*, pp. C306–C307, 2019.

[53] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, *Predictivenet: an energy-efficient convolutional neural network via zero prediction*, in *Circuits and Systems (ISCAS), 2017 IEEE International Symposium on*, pp. 1–4, IEEE, 2017.

[54] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, *Prediction based execution on deep neural networks*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 752–763, IEEE, 2018.

[55] V. Aklaghi, A. Yazdanbakhsh, K. Samadi, H. Esmaeilzadeh, and R. Gupta, *Snapea: Predictive early activation for reducing computation in deep convolutional neural networks*, ISCA, 2018.

[56] S. Cao, L. Ma, W. Xiao, C. Zhang, Y. Liu, L. Zhang, L. Nie, and Z. Yang, *Seernet: Predicting convolutional neural network feature-map sparsity through low-bit quantization*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 11216–11225, 2019.

[57] X. Gao, Y. Zhao, Łukasz Dudziak, R. Mullins, and C. zhong Xu, *Dynamic channel pruning: Feature boosting and suppression*, in *International Conference on Learning Representations*, 2019.

[58] W. Hua, Y. Zhou, C. De Sa, Z. Zhang, and G. E. Suh, *Boosting the performance of cnn accelerators with dynamic fine-grained channel gating*, in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), pp. 139–150, ACM, 2019.

[59] H. Jang, J. Kim, J.-E. Jo, J. Lee, and J. Kim, *Mnnfast: A fast and scalable system architecture for memory-augmented neural networks*, in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 250–263, 2019.

[60] T. J. Ham, S. J. Jung, S. Kim, Y. H. Oh, Y. Park, Y. Song, J.-H. Park, S. Lee, K. Park, J. W. Lee, and D.-K. Jeong, *A3: Accelerating attention mechanisms in neural networks with approximation*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 328–341, 2020.

[61] T. J. Ham, Y. Lee, S. H. Seo, S. Kim, H. Choi, S. J. Jung, and J. W. Lee, *Elsa: Hardware-software co-design for efficient, lightweight self-attention mechanism in neural networks*, in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 692–705, IEEE, 2021.

[62] H. Wang, Z. Zhang, and S. Han, *Spatten: Efficient sparse attention architecture with cascade token and head pruning*, in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 97–110, 2021.

[63] J. Park, H. Yoon, D. Ahn, J. Choi, and J.-J. Kim, *Optimus: Optimized matrix multiplication structure for transformer neural network accelerator*, *Proceedings of Machine Learning and Systems* **2** (2020) 363–378.

[64] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, *Tensorizing neural networks*, *CoRR* **abs/1509.06569** (2015) [arXiv:1509.0656].

[65] Y. Yang, D. Krompass, and V. Tresp, *Tensor-train recurrent neural networks for video classification*, *CoRR* **abs/1707.01786** (2017) [arXiv:1707.0178].

[66] V. Khrulkov, O. Hrinchuk, L. Mirvakhabova, and I. Oseledets, *Tensorized embedding layers for efficient model compression*, *ArXiv* **abs/1901.10787** (2019).

[67] C. Deng, F. Sun, X. Qian, J. Lin, Z. Wang, and B. Yuan, *Tie: Energy-efficient tensor train-based inference engine for deep neural network*, in *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (New York, NY, USA), p. 264–278, Association for Computing Machinery, 2019.

[68] I. Beltagy, M. E. Peters, and A. Cohan, *Longformer: The long-document transformer*, *arXiv preprint arXiv:2004.05150* (2020).

[69] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontañón, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, *Big bird: Transformers for longer sequences*, in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), 2020.

[70] H. Shi, J. Gao, X. Ren, H. Xu, X. Liang, Z. Li, and J. T. Kwok, *Sparsebert: Rethinking the importance analysis in self-attention*, *arXiv preprint arXiv:2102.12871* (2021).

[71] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, *Long range arena: A benchmark for efficient transformers*, *CoRR* **abs/2011.04006** (2020) [arXiv:2011.0400].

[72] G. M. Correia, V. Niculae, and A. F. Martins, *Adaptively sparse transformers*, *arXiv preprint arXiv:1909.00015* (2019).

[73] H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, T. Adler, L. Gruber, M. Holzleitner, M. Pavlović, G. K. Sandve, *et. al.*, *Hopfield networks is all you need*, *arXiv preprint arXiv:2008.02217* (2020).

[74] T. Ji, S. Jain, M. Ferdman, P. Milder, H. A. Schwartz, and N. Balasubramanian, *On the distribution, sparsity, and inference-time quantization of attention values in transformers*, *arXiv preprint arXiv:2106.01335* (2021).

[75] T. Tang, S. Li, L. Nai, N. Jouppi, and Y. Xie, *Neurometer: An integrated power, area, and timing modeling framework for machine learning accelerators industry track paper*, in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 841–853, 2021.

[76] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, *Efficient tensor core-based gpu kernels for structured sparsity under reduced precision*, .

[77] T. Gale, M. Zaharia, C. Young, and E. Elsen, *Sparse GPU kernels for deep learning*, *CoRR* **abs/2006.10901** (2020) [arXiv:2006.1090].

[78] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, *Cusparse library*, in *GPU Technology Conference*, 2010.

[79] T. Gale, M. Zaharia, C. Young, and E. Elsen, *Sparse gpu kernels for deep learning*, *arXiv preprint arXiv:2006.10901* (2020).

[80] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. Colwell, and A. Weller, *Rethinking attention with performers*, 2021.

[81] N. Kitaev, L. Kaiser, and A. Levskaya, *Reformer: The efficient transformer*, in *International Conference on Learning Representations*, 2020.

[82] S. Wang, B. Li, M. Khabsa, H. Fang, and H. Ma, *Linformer: Self-attention with linear complexity*, *arXiv preprint arXiv:2006.04768* (2020).

[83] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, *Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 764–775, 2018.

[84] L. Liu, Z. Qu, L. Deng, F. Tu, S. Li, X. Hu, Z. Gu, Y. Ding, and Y. Xie, *Duet: Boosting deep neural network efficiency on dual-module architecture*, in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 738–750, 2020.

[85] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, *Squad: 100,000+ questions for machine comprehension of text*, 2016.

[86] Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler, *Long range arena : A benchmark for efficient transformers*, in *International Conference on Learning Representations*, 2021.

[87] A. Krizhevsky, *Learning multiple layers of features from tiny images*, tech. rep., 2009.

[88] A. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, *Learning word vectors for sentiment analysis*, in *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pp. 142–150, 2011.

[89] D. R. Radev, P. Muthukrishnan, V. Qazvinian, and A. Abu-Jbara, *The acl anthology network corpus*, *Language Resources and Evaluation* **47** (2013), no. 4 919–944.

[90] S. Merity, C. Xiong, J. Bradbury, and R. Socher, *Pointer sentinel mixture models*, *arXiv preprint arXiv:1609.07843* (2016).

[91] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, *Cacti 6.0: A tool to model large caches*, *HP Laboratories* (01, 2009).

[92] G. M. Amdahl, *Computer architecture and amdahl's law*, *Computer* **46** (2013), no. 12 38–46.

[93] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, *Deep learning with limited numerical precision*, in *International conference on machine learning*, pp. 1737–1746, PMLR, 2015.

[94] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, *Quantized neural networks: Training neural networks with low precision weights and activations*, *The Journal of Machine Learning Research* **18** (2017), no. 1 6869–6898.

[95] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, *Quantization and training of neural networks for efficient integer-arithmetic-only inference*, in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2018.

[96] S. Jain, S. Venkataramani, V. Srinivasan, J. Choi, P. Chuang, and L. Chang, *Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors*, in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2018.

[97] E. Park, D. Kim, and S. Yoo, *Energy-efficient neural network accelerator based on outlier-aware low-precision computation*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 688–698, IEEE, 2018.

[98] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan, *Energy-efficient reduce-and-rank using input-adaptive approximations*, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **25** (2016), no. 2 462–475.

[99] D. Lee, S. Kang, and K. Choi, *Compend: Computation pruning through early negative detection for relu in a deep neural network accelerator*, in *Proceedings of the 2018 International Conference on Supercomputing*, pp. 139–148, 2018.

[100] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, *Snapea: Predictive early activation for reducing computation in deep convolutional neural networks*, in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 662–673, IEEE, 2018.

[101] Z. Liu, A. Yazdanbakhsh, T. Park, H. Esmaeilzadeh, and N. S. Kim, *Simul: An algorithm-driven approximate multiplier design for machine learning*, *IEEE Micro* **38** (2018), no. 4 50–59.

[102] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, *A dynamically configurable coprocessor for convolutional neural networks*, *ACM SIGARCH Computer Architecture News* **38** (2010), no. 3 247–257.

[103] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, *Neuflow: A runtime reconfigurable dataflow processor for vision*, in *2011 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPR Workshops 2011)*, pp. 109–116, IEEE, 2011.

[104] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, *Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning*, *ACM Sigplan Notices* **49** (2014), no. 4 269–284.

[105] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, *Dadiannao: A machine-learning supercomputer*, in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 609–622, IEEE Computer Society, 2014.

[106] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen, *Pudiannao: A polyvalent machine learning accelerator*, in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 369–381, ACM, 2015.

[107] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, *Shidiannao: Shifting vision processing closer to the sensor*, in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[108] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, *Optimizing fpga-based accelerator design for deep convolutional neural networks*, in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 161–170, ACM, 2015.

[109] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, *Neurocube: A programmable digital neuromorphic architecture with high-density 3d memory*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 380–392, IEEE, 2016.

[110] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos, *Stripes: Bit-serial deep neural network computing*, in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[111] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, *Minerva: Enabling low-power, highly-accurate deep neural network accelerators*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, IEEE, 2016.

[112] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et. al.*, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[113] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, *Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks*, *IEEE Journal of Solid-State Circuits* **52** (2017), no. 1 127–138.

[114] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, *Tetris: Scalable and efficient neural network acceleration with 3d memory*, *ACM SIGOPS Operating Systems Review* **51** (2017), no. 2 751–764.

[115] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, *Efficient content-based sparse attention with routing transformers*, *Transactions of the Association for Computational Linguistics* **9** (2021) 53–68.

[116] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D.-C. Juan, *Sparse sinkhorn attention*, in *International Conference on Machine Learning*, pp. 9438–9447, PMLR, 2020.

[117] J. Qiu, H. Ma, O. Levy, S. W. tau Yih, S. Wang, and J. Tang, *Blockwise self-attention for long document understanding*, 2020.

[118] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and H. A. PHAN, *Tensor decompositions for signal processing applications: From two-way to multiway component analysis*, *IEEE Signal Processing Magazine* **32** (March, 2015) 145–163.

[119] Z. Zhang, K. Batselier, H. Liu, L. Daniel, and N. Wong, *Tensor computation: A new framework for high-dimensional problems in eda*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **36** (April, 2017) 521–536.

[120] Z. Zhang, X. Yang, I. V. Oseledets, G. E. Karniadakis, and L. Daniel, *Enabling high-dimensional hierarchical uncertainty quantification by ANOVA and tensor-train decomposition*, *CoRR* **abs/1407.3023** (2014) [arXiv:1407.3023].

[121] T. Kolda and B. Bader, *Tensor decompositions and applications*, *SIAM Review* **51** (2009), no. 3 455–500, [https://doi.org/10.1137/07070111X].

[122] A. Cichocki, *Era of big data processing: A new approach via tensor networks and tensor decompositions*, *CoRR* **abs/1403.2048** (2014) [arXiv:1403.2048].

[123] A. Cichocki, R. Zdunek, A.-H. Phan, and S.-i. Amari, *Nonnegative Matrix and Tensor Factorizations: Applications to Exploratory Multi-Way Data Analysis and Blind Source Separation.* 10, 2009.

[124] S. Klus, P. Gelß, S. Peitz, and C. Schütte, *Tensor-based dynamic mode decomposition*, *Nonlinearity* **31** (jun, 2018) 3359–3380.

[125] L. Sorber, M. V. Barel, and L. D. Lathauwer, *Optimization-based algorithms for tensor decompositions: Canonical polyadic decomposition, decomposition in rank-(lr, lr, 1) terms, and a new generalization*, *SIAM Journal on Optimization* **23** (2013) 695–720.

[126] P. Comon and C. Jutten, *Handbook of Blind Source Separation.* 2010.

[127] L. De Lathauwer, P. Comon, and N. Mastronardi, *Special issue on tensor decompositions and applications*, *SIAM J. Matrix Analysis Applications* **30** (01, 2008).

[128] A. Cichocki, *Tensor networks for big data analytics and large-scale optimization problems*, *CoRR* **abs/1407.3124** (2014) [arXiv:1407.3124].

[129] I. Oseledets and S. Dolgov, *Solution of linear systems and matrix inversion in the tt-format*, *SIAM Journal on Scientific Computing* **34** (2012), no. 5 A2718–A2739, [https://doi.org/10.1137/110833142].

[130] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, *Tensorizing neural networks*, *CoRR* **abs/1509.06569** (2015) [arXiv:1509.0656].

[131] B. W. Bader and T. G. Kolda, *Algorithm 862: Matlab tensor classes for fast algorithm prototyping*, *ACM Trans. Math. Softw.* **32** (Dec., 2006) 635–653.

[132] P. de Rijk, *A one-sided jacobi algorithm for computing the singular value decomposition on a vector computer*, *SIAM Journal on Scientific and Statistical Computing* **10** (1989), no. 2 359–371, [https://doi.org/10.1137/0910023].

[133] X. Wang and J. Zambreno, *An fpga implementation of the hestenes-jacobi algorithm for singular value decomposition*, in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pp. 220–227, May, 2014.

[134] R. Brent, F. T. Luk, and C. VAN LOAN, *Computation of the singular value decomposition using mesh-connected processors*, *Journal of VLSI and Computer Systems* **1** (01, 1985).

[135] L. M. Ledesma-Carrillo, E. Cabal-Yepez, R. d. J. Romero-Troncoso, A. Garcia-Perez, R. A. Osornio-Rios, and T. D. Carozzi, *Reconfigurable fpga-based unit for singular value decomposition of large m x n matrices*, in *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 345–350, Nov, 2011.

[136] W. E. ARNOLDI, *The principle of minimized iterations in the solution of the matrix eigenvalue problem*, *Quarterly of Applied Mathematics* **9** (1951), no. 1 17–29.

[137] R. Ballester, *tntorch: Tensor network learning with pytorch*, *https://github.com/rballester/tntorch*.

[138] A. Novikov, D. Podoprikhin, A. Osokin, and D. Vetrov, *Tensorizing neural networks*, in *Advances in Neural Information Processing Systems 28 (NIPS)*. 2015.

[139] T. Garipov, D. Podoprikhin, A. Novikov, and D. Vetrov, *Ultimate tensorization: compressing convolutional and FC layers alike*, *arXiv preprint arXiv:1611.03214* (2016).

[140] A. Krizhevsky, *Learning multiple layers of features from tiny images*, *University of Toronto* (05, 2012).

[141] Y. Chen, J. Emer, and V. Sze, *Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.

[142] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, *Shidiannao: Shifting vision processing closer to the sensor*, in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 92–104, 2015.

[143] Y. Chen, J. Emer, and V. Sze, *Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks*, in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.

[144] T. Zhao, Y. Zhang, and K. Olukotun, *Serving recurrent neural networks efficiently with a spatial accelerator*, 09, 2019.

[145] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. Rush, G.-Y. Wei, and D. Brooks, *Masr: A modular accelerator for sparse rnns*, 08, 2019.

[146] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, *Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations*, *ArXiv* **abs/2005.05968** (2020).

[147] I. V. Oseledets, S. Dolgov, V. Kazeev, D. Savostyanov, O. Lebedeva, P. Zhlobich, T. Mach, and L. Song, *Tt-toolbox*, .

[148] J. D. Carroll and J.-J. Chang, *Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition*, *Psychometrika* **35** (Sep, 1970) 283–319.

[149] R. A. Harshman, P. Ladefoged, H. G. von Reichenbach, R. I. Jennrich, D. Terbeek, L. Cooper, A. L. Comrey, P. M. Bentler, J. Yamane, and D. Vaughan, *Foundations of the parafac procedure: Models and conditions for an "explanatory" multimodal factor analysis*, 1970.

[150] L. Tucker and C. Harris, *Implications of factor analysis of three way matrices for measurements of change*, in *Problems in measuring change.* University of Wisconsin Press, Madison, 1963.

[151] L. R. Tucker, *The extension of factor analysis to three-dimensional matrices*, in *Contributions to mathematical psychology.* (H. Gulliksen and N. Frederiksen, eds.), pp. 110–127. Holt, Rinehart and Winston, New York, 1964.

[152] L. R. Tucker, *Some mathematical notes on three-mode factor analysis*, *Psychometrika* **31** (Sep, 1966) 279–311.

[153] I. Oseledets and E. Tyrtyshnikov, *Tt-cross approximation for multidimensional arrays*, *Linear Algebra and its Applications* **432** (2010), no. 1 70 – 88.

[154] K. Zhang, X. Zhang, and Z. Zhang, *Tucker tensor decomposition on FPGA*, *CoRR* **abs/1907.01522** (2019) [arXiv:1907.0152].

[155] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. Herr, C. Hughes, T. Mattson, and P. Dubey, *T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations*, in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 181–189, 2019.

[156] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang, *Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations*, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 689–702, 2020.

[157] L. Zhang, Z. Lai, S. Li, Y. Tang, F. Liu, and D. Li, *2pgraph: Accelerating gnn training over large graphs on gpu clusters*, in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 103–113, 2021.

[158] T. Liu, Y. Chen, D. Li, C. Wu, Y. Zhu, J. He, Y. Peng, H. Chen, H. Chen, and C. Guo, *BGL: gpu-efficient GNN training by optimizing graph data I/O and preprocessing*, *CoRR* **abs/2112.08541** (2021) [arXiv:2112.0854].

[159] T. Kaler, N. Stathas, A. Ouyang, A. Iliopoulos, T. B. Schardl, C. E. Leiserson, and J. Chen, *Accelerating training and inference of graph neural networks with fast sampling and pipelining*, *CoRR* **abs/2110.08450** (2021) [arXiv:2110.0845].

[160] Y. Bai, C. Li, Z. Lin, Y. Wu, Y. Miao, Y. Liu, and Y. Xu, *Efficient data loader for fast sampling-based gnn training on large graphs*, *IEEE Transactions on Parallel and Distributed Systems* **32** (2021), no. 10 2541–2556.

[161] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, *Deep graph library: Towards efficient and scalable deep learning on graphs*, *CoRR* **abs/1909.01315** (2019) [arXiv:1909.0131].

[162] M. Z. Y. D. H. R. B. L. M. C. J. L. Weihua Hu, Matthias Fey, *Open graph benchmark: Datasets for machine learning on graphs*, *arXiv preprint arXiv:2005.00687* (2020).

[163] C. C. Onu, J. E. Miller, and D. Precup, *A fully tensorized recurrent neural network*, *CoRR* **abs/2010.04196** (2020) [arXiv:2010.0419].

[164] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, *Tensorizing neural networks*, *CoRR* **abs/1509.06569** (2015) [arXiv:1509.0656].

[165] M. Gabor and R. Zdunek, *Convolutional neural network compression viaÂ tensor-train decomposition onÂ permuted weight tensor withÂ automatic rank determination*, in *Computational Science – ICCS 2022* (D. Groen, C. de Mulatier, M. Paszynski, V. V. Krzhizhanovskaya, J. J. Dongarra, and P. M. A. Sloot, eds.), (Cham), pp. 654–667, Springer International Publishing, 2022.

[166] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, *SIAM J. Sci. Comput.* **20** (Dec., 1998) 359–392.

[167] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, *Fine-grained dram: Energy-efficient dram for extreme bandwidth systems*, in *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 41–54, 2017.

[168] M. Courbariaux and Y. Bengio, *Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1*, *CoRR* **abs/1602.02830** (2016) [arXiv:1602.0283].

[169] C. Chen, K. Li, Y. Li, and X. Zou, *Regnn: A redundancy-eliminated graph neural networks accelerator*, in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, (Los Alamitos, CA, USA), pp. 429–443, IEEE Computer Society, apr, 2022.

[170] Z. Jia, S. Lin, R. Ying, J. You, J. Leskovec, and A. Aiken, *Redundancy-free computation graphs for graph neural networks*, 2019.

[171] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[172] L. Liu, J. Lin, Z. Qu, Y. Ding, and Y. Xie, *Enmc: Extreme near-memory classification via approximate screening*, in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, (New York, NY, USA), p. 1309–1322, Association for Computing Machinery, 2021.

[173] L. Liu, Z. Qu, L. Deng, F. Tu, S. Li, X. Hu, Z. Gu, Y. Ding, and Y. Xie, *Duet: Boosting deep neural network efficiency on dual-module architecture*, in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 738–750, 2020.