University of California
Santa Barbara

Elastic Processing and Hardware Architectures for Machine Learning

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

Doctor of Philosophy
in
Computer Science

by

Liu Liu

Committee in charge:

Professor Yuan Xie, Co-Chair
Professor Yufei Ding, Co-Chair
Professor Timothy Sherwood
Professor William Yang Wang

June 2022

The Dissertation of Liu Liu is approved.

Professor Timothy Sherwood

Professor William Yang Wang

Professor Yufei Ding, Committee Co-Chair

Professor Yuan Xie, Committee Co-Chair

March 2022

Elastic Processing and Hardware Architectures for Machine Learning

by

Liu Liu

To my wife Yushan Huang,
for her love, encouragement, and support.
To our sons Ethan and Nathan,
for the love and joy they have been bringing to our family.

Acknowledgements

Pursuing a Ph.D. cannot be done without the great help offered by so many people in my journey. Firstly, I am extremely grateful to Professor Yuan Xie for many years of guidance and support through my Ph.D. program. I have been very fortunate to work under his supervision. He has been a role model for me in both work and life. I appreciate the autonomy that he entrusted in me, allowing me to pursue my research interests. More importantly, his positive criticism and visionary advice keep me staying out of my comfort zone and advancing in my academic career.

I would also like to extend my deepest gratitude to my Ph.D. co-advisor, Professor Yufei Ding, for her advice on my thesis research. She encouraged me to pursue a career in academia and helped me in many aspects to secure a position. Her encouragement and support motivate me to keep being proactive as a non-traditional researcher without a strong CS background. I am very fortunate to have Yufei as my co-advisor.

I would like to express my deepest appreciation to my thesis committee members, Professor Timothy Sherwood and Professor William Wang, for their invaluable feedback on my research. Tim's passion as a computer scientist working in hardware architecture motivates my career. William is my first mentor in Natural Language Processing (NLP) and Deep Learning. His foresight inspired my research projects on large-scale NLP model acceleration.

I would like to thank my industry mentors for providing thoughtful guidance into my research. Dr. Zhenyu Gu, Dr. Edward Yang, Dr. Jingwei Zhang, and Dr. Fei Sun all gave me insightful suggestions and hands-on guidance during my summer internships at Alibaba DAMO Academy. Dr. Shaoshan Liu helped me a lot on the collaborated project during my early Ph.D. time. I am thankful for many academic collaborators on their help, particularly, Prof. Guoqi Li, Prof. Chao Wang, and Prof. Yuanqing Cheng.

I have been fortunate to collaborate with many kind, intelligent, and diligent students and postdoc researchers at UCSB Seal-Lab, especially Dr. Ping Chi, Dr. Shuangchen Li, Dr. Maohua Zhu, Dr. Peng Gu, Dr. Dylan Stow Randall, Dr. Xing Hu, Dr. Lei Deng, Ling Liang, Bangyan Wang, Zheng Qu, Jilan Lin, Zhaodong Chen, Yuke Wang, and Dr. Fengbin Tu. Many thanks to my group comrades for those fun and insightful discussions, including Jia Zhan, Itir Akgun, Abanti Basak, Wenqin Huangfu, Nan Wu, Gushu Li, Xinfeng Xie, Tianqi Tang, and Guyue Huang. In this dissertation, Chapter 6 and Chapter 7 include equally-contributed work and co-authored publications with Jilan Lin and Zheng Qu, respectively; some contents will also appear in their dissertations.

Finally, I would like to thank my family for their continuous support; especially my dad, who has been and will always be my mentor and life coach. Countless and everlasting thanks to my wife, Yushan Huang, for her love and support.

Curriculum Vitæ

Liu Liu

Education

2022	Ph.D. in Computer Science (Expected), University of California, Santa Barbara.
2015	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2013	B.E. in Information Display and Optoelectronics, University of Electronic Science and Technology of China

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." In *2022 49th International Symposium on Computer Architecture*. (*co-primary)

[2] Zheng Qu*, **Liu Liu***, Fengbin Tu, Zhaodong Chen, Yufei Ding, Yuan Xie. "DOTA: Detect and Omit Weak Attentions for Scalable Transformer Acceleration." In *2022 27th International Conference on Architectural Support for Programming Languages and Operating Systems*. (*co-primary)

[3] Bangyan Wang, Lei Deng, Fei Sun, Guohao Dai, **Liu Liu**, Yu Wang, Yuan Xie. "A One-for-All and $O(V \log(V))$-cost Solution for Parallel Merge Style Operations on Sorted Key-Value Arrays." In *2022 27th International Conference on Architectural Support for Programming Languages and Operating Systems*.

[4] Fengbin Tu, Zihan Wu, Yiqi Wang, Ling Liang, **Liu Liu**, Yufei Ding, Leibo Liu, Shaojun Wei, Yuan Xie, Shouyi Yin. "A 28nm 15.59μJ/Token Full-Digital Bitline-Transpose CIM-based Sparse Transformer Accelerator with Pipeline/Parallel Reconfigurable Modes." In *2022 International Solid-State Circuits Conference*.

[5] Zhaodong Chen*, Zheng Qu*, **Liu Liu**, Yufei Ding, Yuan Xie. "Efficient Tensor Core-based GPU Kernels for Structured Sparsity under Reduced Precision." In *2021 International Conference for High Performance Computing, Networking, Storage, and Analysis*. (*co-primary)

[6] **Liu Liu***, Jilan Lin*, Zheng Qu, Yufei Ding, Yuan Xie. "ENMC: Extreme Near-Memory Classification via Approximate Screening." In *2021 54th IEEE/ACM International Symposium on Microarchitecture*. (*co-primary)

[7] **Liu Liu**, Jie Tang, Shaoshan Liu, Bo Yu, Yuan Xie, Jean-Luc Gaudiot. "Π-RT: A Runtime Framework to Enable Energy-Efficient Real-Time Robotic Vision Applications on Heterogeneous Architectures." *IEEE Computer 54, no. 4 (2021): 14-25.*

[8] **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." In *2020 53rd IEEE/ACM International Symposium on Microarchitecture.*

[9] **Liu Liu**, Lei Deng, Zhaodong Chen, Yuke Wang, Shuangchen Li, Jingwei Zhang, Yihua Yang, Zhenyu Gu, Yufei Ding, Yuan Xie. "Boosting Deep Neural Network Efficiency with Dual-Module Inference." In *2020 International Conference on Machine Learning.*

[10] Fei Sun, Minghai Qin, Tianyun Zhang, **Liu Liu**, Yen-Kuang Chen, Yuan Xie. "Computation on Sparse Neural Networks and its Implications for Future Hardware." In *2020 57th ACM/IEEE Design Automation Conference.*

[11] **Liu Liu***, Lei Deng*, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, Yuan Xie. "Dynamic Sparse Graph for Efficient Deep Learning." In *2019 Seventh International Conference on Learning Representations.* (*co-primary)

[12] Lei Deng, Ling Liang, Guanrui Wang, Liang Chang, Xing Hu, Xin Ma, **Liu Liu**, Jing Pei, Guoqi Li, Yuan Xie. "Semimap: A Semi-Folded Convolution Mapping for Speed-Overhead Balance on Crossbars." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, no. 1 (2018): 117-130.

[13] Lei Deng, Zhe Zou, Xin Ma, Ling Liang, Guanrui Wang, Xing Hu, **Liu Liu**, Jing Pei, Guoqi Li, Yuan Xie. "Fast Object Tracking on a Many-Core Neural Network Chip." *Frontiers in neuroscience* 12 (2018): 841.

[14] Shuang Wu, Guoqi Li, Lei Deng, **Liu Liu**, Dong Wu, Yuan Xie, Luping Shi. "$L1$-Norm Batch Normalization for Efficient Training of Deep Neural Networks." *IEEE transactions on neural networks and learning systems* 30, no. 7 (2018): 2043-2051.

[15] **Liu Liu**, Ping Chi, Shuangchen Li, Yuanqing Cheng, Yuan Xie. "Building Energy-Efficient Multi-Level Cell STT-RAM Caches with Data Compression." In *2017 22nd Asia and South Pacific Design Automation Conference.*

[16] Shuangchen Li, **Liu Liu**, Peng Gu, Cong Xu, Yuan Xie. "NVSim-CAM: A Circuit-Level Simulator for Emerging Nonvolatile Memory based Content-Addressable Memory." In *2016 IEEE/ACM International Conference on Computer-Aided Design.*

[17] Peng Gu, Shuangchen Li, Dylan Stow, Russell Barnes, **Liu Liu**, Eren Kursun, Yuan Xie. "Leveraging 3D Technologies for Hardware Security: Opportunities and Challenges." In *2016 International Great Lakes Symposium on VLSI.*

Abstract

Elastic Processing and Hardware Architectures for Machine Learning

by

Liu Liu

Machine Learning (ML) techniques, especially Deep Neural Networks (DNNs), have been driving innovations in many application domains. These breakthroughs are powered by the computational improvements in processor technology driven by Moore's Law. However, the need for computational resources is insatiable when applying ML to large-scale real-world problems. Energy efficiency is another major concern of large-scale ML. The enormous energy consumption of ML models not only increases costs in data-centers and decreases battery life of mobile devices but also has a severe environmental impact. Entering the post-Moore's Law era, how to keep up performance and energy-efficiency with the scaling of ML remains challenging.

This dissertation addresses the performance and energy-efficiency challenges of ML. The thesis can be encapsulated in a few questions. Do we need all the computations and data movements involved in conventional ML processing? Does redundancy exist at the hardware level? How can we better approach large-scale ML problems with new computing paradigms? This dissertation presents how to explore the elasticity in ML processing and hardware architectures: from the algorithm perspective, redundancy-aware processing methods are proposed for DNN training and inference, as well as large-scale classification problems and long-range Transformers; from the architecture perspective, balanced, specialized, and flexible designs are presented to improve efficiency.

Contents

Chapter 1

Introduction

Machine Learning (ML) algorithms, especially Deep Neural Networks (DNNs), have been driving innovations in many application domains, including computer vision, speech recognition, and natural language processing (NLP); potentially, more intelligent applications in autonomous and medical. These breakthroughs are powered by the computational improvements in processor technology driven by Moore's Law. However, the need for computational resources is insatiable when applying ML to large-scale real-world problems. Energy efficiency is another major concern of large-scale ML. The enormous energy consumption of ML models not only increases costs in data-centers and decreases battery life of mobile devices but also has a severe environmental impact. For example, the training process of large-scale NLP models can emit carbon dioxide equivalent to nearly five times the lifetime carbon footprint of an average American car. Entering the post-Moore's Law era, how to keep up performance and energy-efficiency with the scaling of ML remains challenging.

This dissertation presents an software-hardware co-design approach to the performance and energy-efficiency challenges. The thesis is how to explore elasticity in ML methods and hardware to enhance efficiency, which can be encapsulated in a few ques-

tions: Do we need all the computations and data movements involved in conventional ML processing? Does redundancy exist at the hardware architecture level? How can we better approach large-scale ML problems with new computing paradigms?

1.1 Elastic Processing for Machine Learning

Machine Learning (ML) workloads demand high computational and representational costs, from Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), to large-scale classification workloads and Transformer-based models. On one hand, ML training poses challenges on system-level memory capacity, compute capability, and network fabrics. On the other hand, serving ML models in resource-limited platforms is challenging when having strict latency, energy, and reliability requirements. However, conventional ML methods follow static and dense computational graphs and ignore the elasticity in ML processing. Specifically, this dissertation shows that redundant computations and representations exist in neural network activations. Depending on the performance and quality requirements, elastic ML processing exhibits dynamic and sparse computational graphs.

1.1.1 Boosting Efficiency with Dual-Module Inference

Serving DNN models in low latency and energy is critical not only for real-time interaction but also for reducing costs in data-centers and increasing the endurance of edge devices. However, in prior studies using model compression, *all* activations need uniformly accurate computation. The Dual-Module Inference (DMI) work in Chapter 3 observes that noise-resilience commonly exists in the nonlinear activation functions of DNNs. Leveraging the noise-resilience, we can use a lightweight *little* module that approximates the original DNN layer, referred as the *big* module, to compute activa-

tions that are more noise-resilient. Hence, we can save the expensive memory accesses and computations of the original module when only a small portion of activations need accurate results.

1.1.2 ML Training with Dynamic Sparse Graph

Neuron activations reflect the selectivity of the current stimulus and propagate layer-by-layer, forming different representation levels. The Dynamic Sparse Graph (DSG) work in Chapter 4 finds that redundancy exists in activations. For example, lots of neuron activations for each stimulus sample are small and can be removed. Therefore, the proposed method is to search for critical neurons for constructing a sparse graph dynamically at every iteration. By activating only a small number of neurons with a high selectivity, we can significantly save memory and computations with tolerable quality degradation.

1.2 Elastic Hardware Architectures

Domain-Specific Architectures (DSAs) are pervasive with the end of Moore's Law and Dennard Scaling, addressing the inefficiencies in general-purpose processors such as complex control logic and hardware-managed memory hierarchy. For example, in the DNN domain, DSAs such as Google's TPUs, NVIDIA's Tensor Core, and many academic proposals have been designed to improve the efficiency of DNN processing. However, prior DSAs using a homogeneous processing design lack support for elastic processing, and all DNN activations are computed as the same type. In other words, a lot of computations and data movements are wasteful.

1.2.1 Dual-Module Architecture Design

Motivated by elastic processing discussed in Chapter 3, Chapter 5 introduces the design of dual-module architecture (DUET) with a dedicated Speculator running approximate modules and an Executor running accurate modules. On one hand, the Speculator could become the new bottleneck or increase critical path latency and degrade overall performance. On the other hand, imbalanced workloads caused by neuron-wise dynamic switching could lead to computing resources underutilized in the Executor. The proposed design features fine-grained parallel processing to hide Speculator latency and balanced execution in the Executor to improve utilization. Additionally, DUET can save expensive memory accesses of accurate modules computed by the Executor.

1.2.2 Near-Memory Processing of Extreme Classification

Extreme classification is the essential component of large-scale Machine Learning for a wide range of application domains, including image recognition, language modeling, and product recommendation. As classification categories keep scaling in real-world applications, the classifier's parameters could reach several thousands of Gigabytes, far exceeding the on-chip memory capacity. With near-memory processing (NMP) architectures, offloading the classification component onto NMP units could mitigate the memory-intensive problem. However, naive NMP designs with limited area and power budget cannot afford the computational complexity of full classification.

Chapter 6 approaches the problem by exploring the intrinsic elasticity in extreme classification. The key motivation is that we can afford only the top probabilities from classifiers to be accurate, while keeping the rest approximate. Therefore, Chapter 6 presents a novel screening method to reduce the computation and memory consumption by efficiently approximating the classification output. With approximate results, we can

screen out a small number of key candidates that require accurate results. Therefore, the proposed design features an extreme-classification-tailored NMP architecture (ENMC), to support both screening and candidates-only classification.

1.2.3 Dynamic Sparse Attention in Transformers

Transformers are the mainstream of NLP applications and are becoming increasingly popular in other domains such as Computer Vision. Despite the improvements in model quality, the enormous computation costs make Transformers difficult at deployment, especially when the sequence length is large in emerging applications. Processing attention mechanism as the essential compute pattern of Transformers is the bottleneck of execution due to the quadratic complexity. Prior art explores sparse patterns in attention to support long sequence modeling, but those pieces of work are on static or fixed patterns.

Chapter 7 demonstrates that the sparse patterns are dynamic, depending on input sequences. Thus, the proposed Dynamic Sparse Attention (DSA) approach can efficiently exploit the dynamic sparsity in the attention of Transformers. Compared with other methods, DSA can achieve better trade-offs between accuracy and model complexity. Moving forward, Chapter 7 identifies challenges and provides solutions to implement DSA on existing hardware (GPUs) and specialized hardware, i.e., the DOTA project, to achieve practical speedup and efficiency improvements for Transformer execution.

1.3 Organization

Chapter 2 introduces some preliminaries and related work. As shown in Figure 1.1, this dissertation explores the *elasticity* in ML processing and hardware architectures. From the algorithm perspective, Chapter 3 & 4 propose redundancy-aware processing for DNN inference (DMI) and training (DSG); Chapter 6 tackles large-scale classifica-

tion problems (ENMC); Chapter 7 aims at scalable acceleration for Transformer models (DSA). From the architecture perspective, Chapter 5, Chapter 6, and Chapter 7 explore balanced, specialized, and flexible designs (DUET, ENMC, and DOTA) to improve computational efficiency. Chapter 8 concludes the thesis and discusses future research.

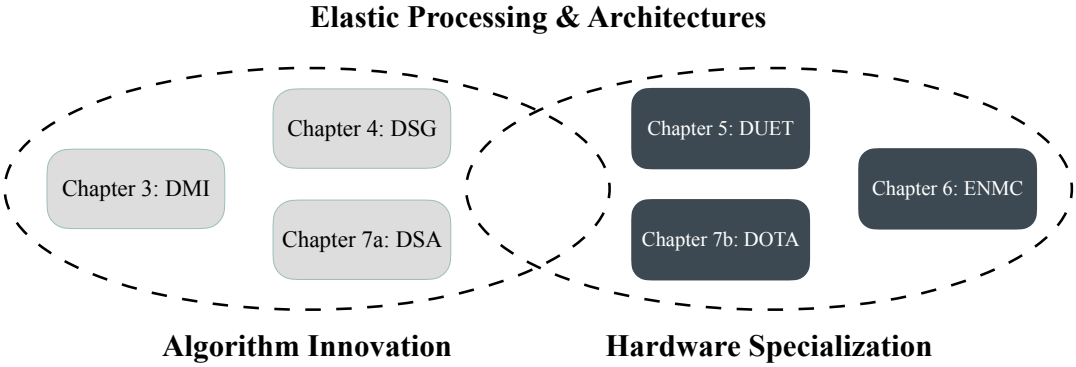


Figure 1.1: The organization of this dissertation

This dissertation comprises work published elsewhere in conference papers:

- Chapter 3: Liu Liu, Lei Deng, Zhaodong Chen, Yuke Wang, Shuangchen Li, Jingwei Zhang, Yihua Yang, Zhenyu Gu, Yufei Ding, and Yuan Xie. "Boosting deep neural network efficiency with dual-module inference." In International Conference on Machine Learning, pp. 6205-6215. PMLR, 2020. [1]

- Chapter 4: Liu Liu, Lei Deng, Xing Hu, Maohua Zhu, Guoqi Li, Yufei Ding, and Yuan Xie. "Dynamic Sparse Graph for Efficient Deep Learning." In International Conference on Learning Representations. 2019. [2]

- Chapter 5: ©2020 IEEE. Reprinted, with permission, from Liu Liu, Zheng Qu, Lei Deng, Fengbin Tu, Shuangchen Li, Xing Hu, Zhenyu Gu, Yufei Ding, and Yuan Xie. "DUET: Boosting deep neural network efficiency on dual-module architecture." In

[1] http://proceedings.mlr.press/v119/liu20c.html
[2] https://arxiv.org/abs/1810.00859

2020 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 738-750, doi: 10.1109/MICRO50266.2020.00066.

- Chapter 6: Liu Liu, Jilan Lin, Zheng Qu, Yufei Ding, and Yuan Xie. 2021. ENMC: Extreme Near-Memory Classification via Approximate Screening. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (*MICRO '21*). Association for Computing Machinery, New York, NY, USA, 1309–1322. DOI:https://doi.org/10.1145/3466752.3480090

- Chapter 7: Zheng Qu, Liu Liu, Fengbin Tu, Zhaodong Chen, Yufei Ding, and Yuan Xie. 2022. DOTA: detect and omit weak attentions for scalable transformer acceleration. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (*ASPLOS 2022*). Association for Computing Machinery, New York, NY, USA, 14–26. DOI:https://doi.org/10.1145/3503222.3507738

Chapter 2

Background and Related Work

Firstly, this chapter summarizes the background on Transformer neural networks and large-scale ML classification. Secondly, the chapter introduces emerging computing paradigms. Lastly, the chapter discusses related work on both efficient processing methods and hardware acceleration.

2.1 Transformer Neural Networks

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 processed by blocks of encoders. Each encoder can be separated into three stages, namely Linear Transformation, Multi-Head Attention, and Feed-Forward Network (FFN). In the transformation stage, three matrix multiplications transform the input into Query (Q), Key (K), and Value (V) as:

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

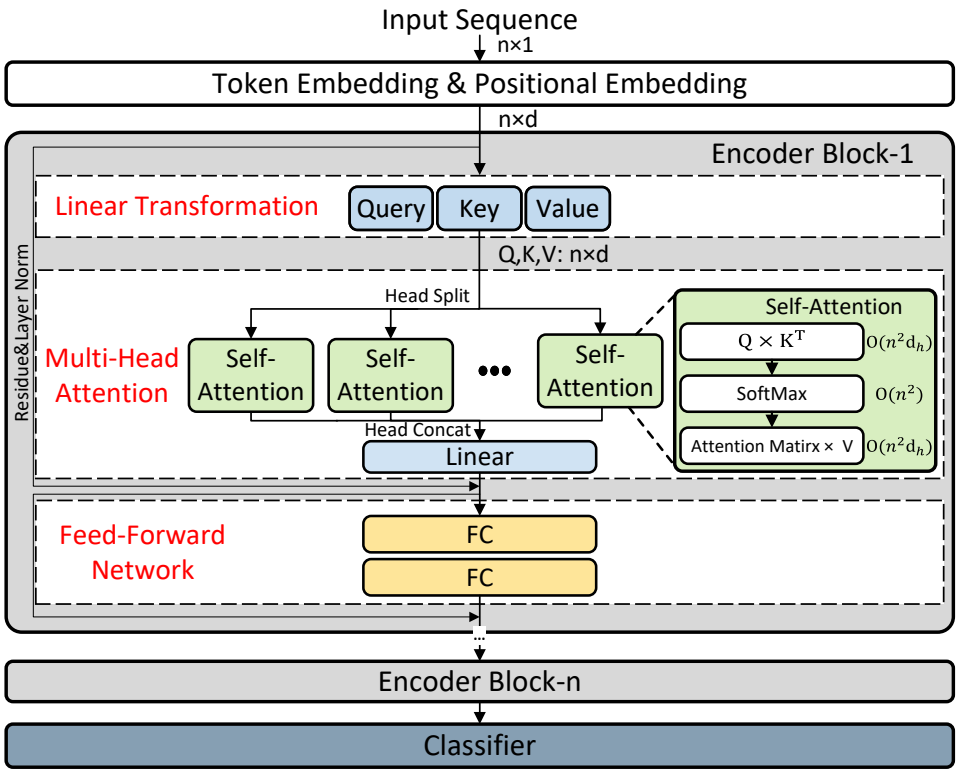


Figure 2.1: Transformer model architecture.

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 Extreme Classification

The Extreme Classification problem refers to multi-class or multi-label classification with extremely large category volume. Many large-scale NLP and recommendation applications can be modeled as a feature extraction part with an extreme classifier. For example, in NLP applications, the typical sequence-to-sequence modeling consists of a stack of encoders, a stack of decoders, and a final classification layer [1, 2, 3].

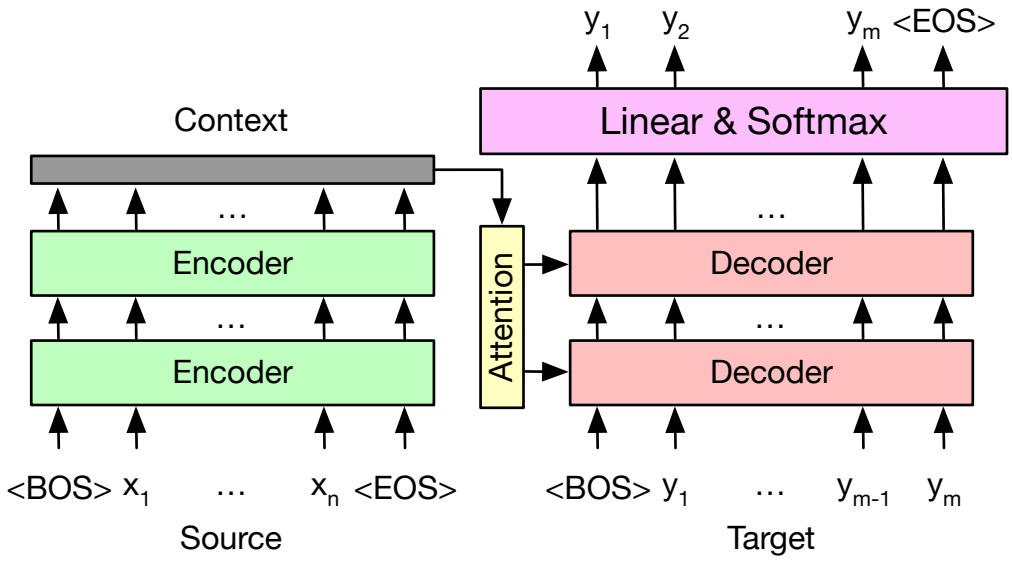


Figure 2.2: Basic components of sequence-to-sequence modeling.

Figure 2.2 illustrates the basic components of sequence-to-sequence modeling for neural machine translation. Each encoder and decoder is a type of DNN layer, such as Transformer layers [4] and recurrent neural networks [2]. The encoders process input

embeddings into hidden representations repeatedly. The decoders that attend over all hidden states from the encoder stack process queries from the previous decoder layer and output decoded hidden vectors. The final classification layer turns the hidden vector from the last decoder layer into a translated word as in translation tasks or probabilities as in language modeling tasks. The classification layer consists of a large linear layer followed by a softmax layer. One way to interpret the linear layer is performing the inner-products of the hidden vector from the decoder stack and a number of weight vectors, which correspond to the target vocabulary size. The softmax function then normalizes the inner-products into probabilities.

Also, in large-scale recommendation systems such as commodity product recommendation and web-page recommendation, extreme classification refers to the problem of multi-class prediction [5, 6, 7]. First, the hidden layers, e.g., DNNs, take dense features and sparse features from users as input. Then, the classification layer maps the output of the last hidden layer, usually through $SoftMax$ normalization, to a probability distribution. For real-world scenarios and next-generation applications, the final classification layer is becoming even more challenging as the computational complexity and the memory usage grows linearly with the category size.

2.3 Emerging Computing Paradigms

Domain-Specific Architectures (DSAs) are pervasive at the end of Moore's Law and Dennard Scaling, addressing the inefficiencies in general-purpose processors such as complex control logic and hardware-managed memory hierarchy [8]. For example, in the DNN domain, DSAs such as Google's TPUs [9], NVIDIA's Tensor Core, and many academic proposals have been designed to improve the efficiency of DNN processing. However, prior DSAs using a homogeneous processing design lack support for elastic processing, and all

DNN activations are computed as the same type. In other words, a lot of computations and data movements are wasteful.

2.3.1 Emerging System Architecture

GPUs and domain-specific accelerators are widely used to process compute-intensive models in the front-end, such as CNNs, RNNs, and Transformers [4]. In contrast, for memory-intensive front-ends like recommendation models and embedding look-ups, CPUs are more favored because of the larger memory capacity. In these scenarios, the processing units (CPU/GPU/accelerator) typically allocate the classification parameters in the local memory, as shown in the Figure 2.3(a).

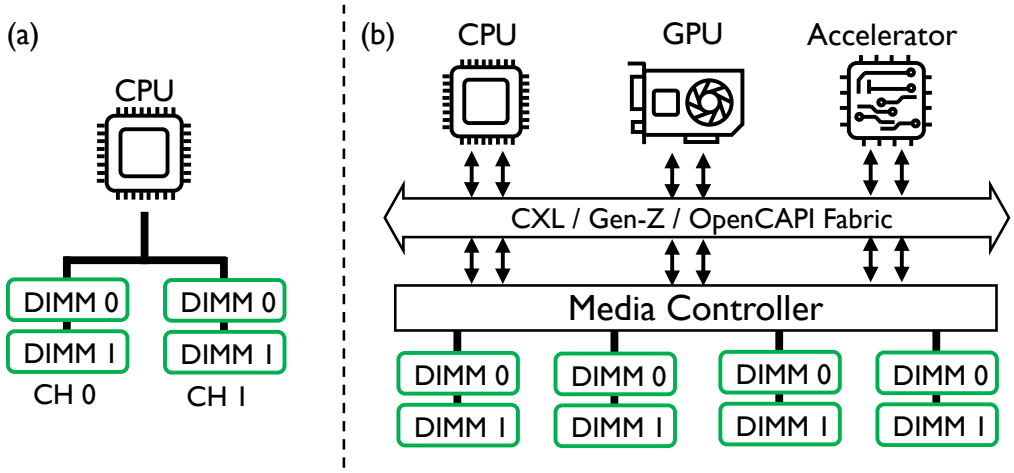


Figure 2.3: The system architecture of classification workloads: (a) Host-only system; (b) A pooled memory architecture to extend the memory capacity.

As discussed in the Section 2.2, the tremendous classification categories essentially need enormous memory capacity. For example, the largest dataset in an academic extreme classification repository [10] consists of 3 million categories, while industries have reported 50 million to 100 million categories used in classification [7, 5]. With the hidden size of 512, the memory usage of classification alone is reaching 190GB. The need for memory is increasing with the scaling of problem size in applications, easily exceeds

the device memory capacity and even system memory capacity. Therefore, employing a memory pool in the system architecture to store classification parameters can be useful, as shown in Figure 2.3(b). Facilitated by emerging memory protocols such as Gen-z [11], GPUDirect [12], CXL [13], etc., the pooled memory could easily stack from 1TB to 10TB DRAMs to tackle the application requirement.

2.3.2 Near-Memory Processing

As DNNs now appear to overwhelm almost every domain in our daily life and such applications are increasingly bandwidth-hungry, near-memory processing (NMP) technique is getting growing attention to accelerate these workloads. As shown in Figure 2.4, leveraging the large internal bandwidth provided by rank parallelism or inside the memory chips, conventional NMPs put customized computation logic beside the data and saves the system bandwidth and memory access latency.

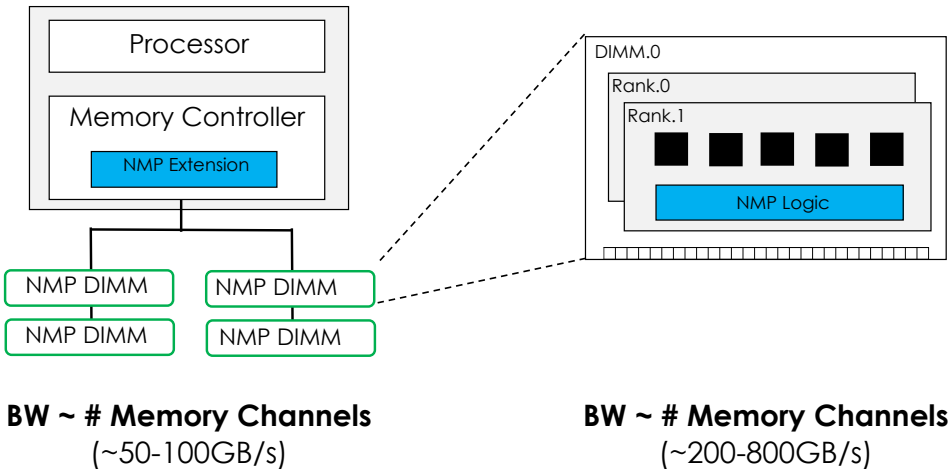


Figure 2.4: Near-Memory Processing offers large bandwidth (BW).

Different NMP techniques can be categorized by the distance between the computation logic and DRAM cell array. Here we generally refer two types of NMP techniques for a DRAM-based memory subsystem: intrusive and non-intrusive NMP. The intrusive

NMP hacks the architecture inside the DRAM device, and the computation logic could be placed at the logic die for a 3D-stacking DRAM module [14, 15, 16, 17, 18], or directly beside the DRAM banks to gain higher bandwidth [19, 20, 21, 22, 23]. The non-intrusive NMP makes use of the rank-level parallelism in the current memory hierarchy. It tries to leverage commodity DRAM chips and places the processing unit at each rank on the DIMM, and thus higher bandwidth can be achieved with multiple ranks in a memory channel [24, 25, 26]. The *ENMC* design in Chapter 6 takes the non-intrusive NMP approach since it requires minimized hardware changes in existing DRAM technology and does not need the support from the DRAM vendors.

2.4 Related Work on Efficient Methods

This section surveys the literature for related methods on redundancy elimination, classified into static redundancy and dynamic redundancy. These two types of redundancy elimination approaches are compatible and can be combined together to further reduce redundant computations and data movements.

2.4.1 Methods on Static Redundancy Elimination

Model compression techniques are commonly used to eliminate static redundancy in Deep Neural Networks (DNN). Compressing DNN models via data quantization, weight sparsity, and knowledge distillation is promising to deliver efficient deployment for inference. Quantization methods on weights and activations have been proposed to reduce model size and operation precision [27, 28, 29, 30, 31]. Weight pruning has been proposed to reduce the parameters of a pre-trained model [32, 33]. While fine-grained pruning could reduce the number of parameters [34, 27, 35], indexing irregular non-zero weights causes extra memory cost and would offset the benefits from reducing parameter size;

it is hard to gain practical acceleration on general-purpose hardware or need hardware specialization [36]. Although structural pruning [37] and knowledge distillation [38] could achieve speedup, the applicability on more complicated tasks such as NMT on large-scale datasets is unstudied; besides, those methods require extensive and iterative retraining via regularization that would increase the training cost and difficult to find a solution.

Many work consider different granularity levels of sparse patterns. In contrast to the fine-grain compression, coarse-grain sparsity was further proposed to optimize the execution speed. Channel-level sparsity was gained by removing unimportant weight filters [39, 40], training penalty coefficients [41, 42, 43], or solving optimization problem [44, 45, 46, 47]. Other medium-grain, i.e., row-wise and column-wise, and coarse-grain, i.e., filter-wise and layer-wise, sparse patterns can be obtained via a L2-norm group-lasso optimization method [48] or Taylor expansion for neuron-wise pruning [49]. However, they just benefit the inference acceleration, and the extra solving of the optimization problem usually makes the training more complicated.

2.4.2 Methods on Dynamic Redundancy Elimination

Other than model compression techniques, many studies propose to skip computations dynamically based on certain criterion such as layer-wise early exit [50] and *ReLU*-induced sparsity prediction in CNNs. The prediction can be achieved through low-precision computation [51, 52] or auxiliary computation [53, 54, 55]. One limitation of these methods is the restricted use case in *ReLU*-based CNNs.

The special cell structure and the temporal input similarity have enabled computation and update skipping in RNNs [56, 57, 58]. However, those methods depend on certain applications and lack of evaluation on NLP tasks such as language modeling and machine translation.

Leveraging dynamic redundancy in training to reduce computations is more challenging. The randomized hashing method is proposed to predict the important neurons and save computations of unimportant neurons [59]. However, the hashing-based method aims at finding neurons whose weight bases are similar to the input vector, which cannot estimate the inner product accurately thus will probably cause significant accuracy loss on large models. A straightforward top-k pruning is proposed to apply on the back propagated errors for training acceleration [60]. But only the backward pass of small fully-connected layers is applied and presented results. Furthermore, the BN compatibility problem that is very important for large-model training still remains untouched. Pruning gradients to accelerate distributed training focuses on multi-node communication but misses the single-node scenario [61].

2.4.3 Efficient Transformers

Transformers with the use of self-attention mechanism are difficult to scale with sequence length because of the quadratic time and memory complexity. Chapter 7 focuses on the exploration of sparse attention patterns in Transformers. Other orthogonal approaches such as parameters sharing [62] can mitigate the issue. Readers can find a survey paper for a more comprehensive view of efficient Transformers [63].

Static Sparse Patterns. A straightforward way to exploit attention sparsity is to set static or fixed sparse patterns, such as local windows, block-wise, dilated patterns, or a combination of static patterns [64, 65, 66]. However, as the sparse attention patterns are inherently dynamic depending on input sequences, those work lack the capability of capturing dynamic sparse patterns. As shown in our evaluation, the sparsity-saving trade-offs of representative methods using static sparse patterns are worse than our dynamic sparse attention approach.

Clustering-based methods. Building upon static block-sparse patterns, another line of research is to group similar tokens into chunks and perform local attention within chunks [67, 68, 69]. The similarity function used to group tokens can be hashing, clustering, or learned sorting. However, those methods are designed for training memory reduction and impractical at inference time when operating on each sequence. The quality of grouping, e.g., convergence of clustering, is not guaranteed at long sequences, and the overhead of on-the-fly clustering is not acceptable.

Approximation methods. Recent work proposes to replace standard attention with forms of approximation of the attention weights [70, 71, 72, 73]. While Chpater 7 provides a comparison in evaluation, those work are out the scope of our discussion for exploring sparsity in (standard) attention. Whether using a form of approximation to replace standard attention or as we suggest to predict sparse patterns explicitly is a design choice leaving up to practitioners.

2.5 Related Work on Hardware Acceleration

Academia and industry have proposed various architectures for the acceleration of DNNs [74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 9, 84, 85]. The focus of this dissertation is on algorithm-architecture co-design to reduce computations and data movements.

Many hardware acceleration work have been designed to support sparsity from static redundancy elimination. Fine-grained weight sparsity was integrated into DNN accelerators through compressed storage and computation skipping of zero weights [33, 86, 87, 88, 89, 90]. Coarse-grained weight sparsity was further proposed to mitigate the indexing overhead and irregular access [91, 92, 93, 94].

Other studies propose dynamic redundancy elimination based on certain criteria. One scenario is leveraging $ReLU$-induced activation sparsity as either input sparsity detection

[33, 95, 17, 96, 97, 98, 90, 99, 94] or output sparsity prediction [100, 101]. Exploiting *ReLU*-induced activation sparsity is only a special case of the dual-module processing discussed in Chapter 3. The approximate results are useful in the insensitive regions instead of only for prediction and then discarded. Besides neuron-wise computation skipping, channel-wise feature map suppressing and gating can reduce computations leveraging the multi-channel feature of CONV layers [91, 54, 102]. However, those studies are limited to saving computations of CONV layers while our design can also save memory access of FC and RNN layers.

Attention and Transformer Accelerators Recent work adopt algorithm and hardware co-design to reduce the cost of attention mechanism. MnnFast [103] proposes to skip the computations of $A \times V$ based on the magnitude of the calculated attention scores. This method can only benefit the second GEMM of attention layer. A^3 [104] introduces attention approximation to prune the unimportant attentions. However, A^3 involves expensive online sorting, which causes significant performance and energy overhead. ELSA [105] uses sign random projection to estimate the attention weights, making the approximation much more hardware efficient, but the model quality is hurt due to inaccurate approximation. In Chapter 7, DSA addresses these limitations by simultaneously considering approximation accuracy and efficiency. Finally, SpAtten [106] proposes cascade token pruning and head pruning to reduce the cost of both self-attention block and subsequent layers. While removing several rows and columns of the attention matrix makes the operation regular and hardware-friendly, this constraint can be too aggressive as the locality of attention weights usually exists in small granularity.

Chapter 3

Dual-Module Inference

This chapter discusses the applicability of elastic processing on Deep Neural Networks inference. Elastic processing in this scenario is essentially leveraging dynamic sparsity obtained from identifying critical vs. trivial neurons.

3.1 Introduction

Deep neural networks (DNNs) play a critical role in many areas like image classification [107, 108, 109], natural language processing (NLP) [110, 2, 111, 112, 113], and graph processing [114, 115, 116, 117]. While the DNN models are usually trained in data-centers, the pre-trained models can be deployed in both data-centers for cloud-based service and edge devices. During inference, there are primary concerns including latency and energy consumption: low latency is critical for real-time interaction, and low energy can help companies reduce cost in data-centers and increase the endurance of edge devices.

In recent years, extensive studies have shown that quantization and pruning are two effective ways to reduce latency and energy consumption of DNNs [32, 33, 34, 27, 35, 118].

However, most existing studies apply pruned and quantized models to compute all the output activations; the overall pruning or quantization ratio is limited by the accuracy loss.

This Chapter presents the observation that most popular activation functions like *ReLU* in convolutional neural networks (CNNs) and *sigmoid*, *tanh* in recurrent neural networks (RNNs) demonstrate noise resilience in particular regions, i.e., the negative region of *ReLU* and the saturation regions of *sigmoid*, *tanh*. With theoretical and experimental evidences, we can see that the noise caused by pruning and quantization in these regions is less influential. This observation motivates the application of more aggressive pruning and quantization to these insensitive regions.

Leveraging the noise resilience of DNNs, the proposed *big-little* dual-module inference (DMI) algorithm, regarding the original pre-trained module as the *big* module, uses a *little* module that has fewer parameters and lower bit-width to approximate the results of the *big* module in the insensitive regions. Executing one DNN layer at inference time takes the following steps: (1) quantize the input activations and forward them through the *little* module; (2) predict which output neurons belong to the sensitive region based on results from the *little* module; (3) compute the output activations of the *big* module in the predicted sensitive region; (4) combine the outputs of the *little* module in the insensitive region and the outputs of the *big* module with in the sensitive region.

The weight matrix of the *little* module is constructed as follows. Assuming the weight matrix of *big* model W^{HH} is an $n \times d$ dense matrix, we first randomly initialize a smaller $n \times k$ quantized dense matrix W^{LL} where $k \ll d$. Then, we multiply it with a $k \times d$ random projection matrix P so that $W^{LL}P$ and W^{HH} have the same size. Because P is very sparse and its entries are either ± 1 or 0, the projection doesn't influence the bit-width of W^{LL}. Therefore, $W^{LL}P$ is a sparse and quantized matrix that has the shape of W^{HH}. The weights of the *little* module are trained in a knowledge-distilling way by mimicking

the behaviors of the *big* module. Specifically, as discussed in prior work [119], we minimize the Frobenius norm of the difference between the flows of the solution procedure (FSP) matrix of the *big* and *little* modules. To accurately predict which output neurons belong to the insensitive region, we minimize the Kullback-Leibler (KL) divergence between the output distributions of the *big* and *little* modules. Our theoretical analysis reveals that both targets can be achieved by minimizing the reconstruction error, i.e., mean-squared error, between the output feature maps of the *big* and *little* modules.

The DMI algorithm can be applied to various types of neural networks, as being evaluated on CNNs, LSTM, and GRU. For the memory-bound RNNs, with overall memory accesses reduced by 40% on a commodity CPU-based server platform, the DMI method can achieve 1.54x to 1.75x wall-clock time speedup with negligible impact on model quality. In addition, the DMI method can reduce the operations of the compute-bound CNNs by 3.02x, with only a 0.5% accuracy drop.

3.2 Motivation

This section discusses the noise resilience of popular activation functions in DNNs including $tanh$, $sigmoid$, and $ReLU$. For clarity, we denote the pre-activation and the noise by x and δ, respectively. Generally, an activation function f is resilient to a noise δ when we have $|f(x+\delta) - f(x)| < \theta$, where θ is a threshold.

For $tanh$, when $|x| \gg 0$, we have

$$|tanh(x+\delta) - tanh(x)| \approx 2e^{-2|x|}|\delta|. \tag{3.1}$$

Similarly, for *sigmoid*, when $|x| \gg 0$ we have

$$|sigmoid(x+\delta) - sigmoid(x)| \approx e^{-|x|}|\delta|. \tag{3.2}$$

While for $ReLU$, we have

$$|ReLU(x+\delta) - ReLU(x)| \begin{cases} = 0 & x \leq -|\delta| \\ \leq |\delta| & otherwise \end{cases}. \tag{3.3}$$

We define the sub-domain of f that is resilient to the noise δ as the insensitive region of f. For a given θ, the insensitive regions of the above three activations are listed as follows

$$\begin{cases} tanh: & |x| > \frac{1}{2}\ln\frac{2|\delta|}{\theta} \\ sigmoid: & |x| > \ln\frac{|\delta|}{\theta} \\ ReLU: & x < \theta - |\delta| \end{cases}. \tag{3.4}$$

While the insensitivity of $ReLU$ is quite straightforward, we can obtain similar conclusions on *sigmoid* and *tanh* with a single LSTM layer for language modeling over PTB dataset. The baseline perplexity (PPL) is 80.64. For each gate, we consider two cases: adding Gaussian noise to the pre-activations before passing through the gate in the sensitive region; in contrast, adding Gaussian noise to the insensitive region. The sensitive regions have 50% of activations based on the magnitudes; vice verse, for the insensitive regions.

Table 3.1 lists the PPL results on the testing set and the average cosine similarity between the activations of the baseline model and the noise-introduced model. Before applying the nonlinear activation functions, the cosine similarity of two cases – adding noise in the sensitive region or the insensitive region – are at the same level. However,

Table 3.1: Comparison of adding Gaussian noises to the sensitive or insensitive region of LSTM gates.

Case	Cosine similarity before gates				Cosine similarity after gates				PPL
	input	forget	cell	output	input	forget	cell	output	
Sensitive	0.953	0.859	0.952	0.932	0.934	0.946	0.882	0.940	85.70
Insensitive	0.944	0.929	0.943	0.947	0.968	0.987	0.969	0.977	81.79

we can observe that after the nonlinear gates, the cosine similarity in the insensitive case is much closer to one, i.e., fewer output errors, than that in the sensitive case. When comparing the PPL results of these two cases, we can further observe that introducing noise in the insensitive region causes little quality degradation.

The selection of which output neurons should be in the (in)sensitive region is dynamic and input-dependent. Unlike the static weight sparsity that we can prune the ineffectual connections offline in advance, the dynamic region speculation requires a very lightweight criterion for real-time processing. Taking all these into account, the proposed dual-model inference (DMI) algorithm can efficiently determine (in)sensitive region and significantly save the memory accesses and computations.

3.3 Approach

First, we explain the DMI algorithm by taking a fully-connected (FC) layer as an example and then extend it to LSTM, GRU, and CNN. For an FC layer with batch size of one, the operation is typically formulated as $z = \varphi(y), y = Wx + b$, where W is a weight matrix ($W \in \mathbb{R}^{n \times d}$), x is an input vector ($x \in \mathbb{R}^d$), b is a bias vector ($b \in \mathbb{R}^n$), y is a pre-activated output vector ($y \in \mathbb{R}^n$), z is an activated output vector ($z \in \mathbb{R}^n$), and φ is an activation function.

3.3.1 Overview of Dual-module Philosophy

Section 3.2 shows that not all values in z need accurate computation, and those belonging to the insensitive region can afford some extent of approximation. In other words, we only need accurate computations and expensive memory accesses in the sensitive region of y and can skip the ones in the insensitive region. With that, we still need approximated results in the insensitive region. Therefore, the proposed method is to learn a lightweight *little* module from the original trained layer, referred as the *big* module. Essentially, the *little* module has low-volume parameters and low bit-width, thus termed as LL module; in contrast, the original *big* module has high-volume parameters and high precision is called HH module. Let the outputs from these two modules be y^{LL} and y^{HH}, respectively. If the LL module approximates the HH module well, the final output vector – a mixture of results from the HH and the LL modules – can be assembled by

$$y = y^{HH} \odot m + y^{LL} \odot (1 - m) \tag{3.5}$$

where $m \in \{0, 1\}^n$ is a binary mask vector for the output switching. m_i equals 1 in the sensitive region while it switches to 0 in the insensitive region. The overall saving comes from skipping memory accesses and computations of the *big* module while paying smaller overhead in accessing and computing the *little* module.

Applying dual-module inference introduces two challenges: first, how to efficiently construct the LL module; second, how to predict which output neurons belong to the (in)sensitive regions.

3.3.2 Construct the LL Module

As the HH module is the original pre-trained layer, we only need to construct an extra LL module. Delivering a lightweight *little* module at inference time is crucial to

achieving real wall-clock time speedup. Two objectives need to consider when designing the LL module: firstly, achieving much lower overhead in terms of computation and memory access than the HH module; secondly, approximating the outputs of HH module accurately.

Lightweight Linear Transformation. Making W^{HH} sparse and using low bit-width data are the two ways to construct a lightweight approximation of $W^{HH}x$. Inspired by random projection, a common technique for dimension reduction while preserving the distances in Euclidean space [120, 121, 122, 123], we first initialize a smaller dense matrix $W^{LL} \in \mathbb{R}^{n \times k}$ where $k < d$, and then we transform it by multiplying it with a $k \times d$ sparse random projection matrix P in which

$$P_{ij} = \sqrt{\frac{3}{k}} \times \begin{cases} +1 & \textit{with probability } 1/6 \\ 0 & \textit{with probability } 2/3 \\ -1 & \textit{with probability } 1/6 \end{cases} . \tag{3.6}$$

In other words, $W^{HH}x$ is replaced with $W^{LL}Px$. The original layer takes $O(n \times d)$ MACs (multiply-and-accumulate operations), while the sparse kernel only needs $O(\frac{1}{3} \times k \times d + n \times k)$ MACs. Therefore, using a smaller k can greatly reduce the memory and compute costs.

However, there still should be a lower bound of k (i.e. $\inf(k)$) to maintain the approximation accuracy. The hypothesis is that the minimum number of parameters in W^{LL} is approximate to the minimum number of parameters in $W^{HH}P^T$ that preserves the Euclidean distance between W^{HH}'s row vectors. The intuition behind is that while each row vector in W^{HH} defines the linear transformation of each output channel, once the Euclidean distance is not preserved, there might be fewer effectual channels, which hurts the accuracy.

According to the Theorem 1.1 in prior work [120], given constant β and ϵ, as long as k satisfies

$$k > k_0 = \frac{4+2\beta}{\epsilon^2/2 - \epsilon^3/3} log(n) \tag{3.7}$$

with probability at least $1 - n^{-\beta}$, for all row vectors $u, v \in Row(W^{HH})$, we have

$$(1-\epsilon)||u^T - v^T||_2^2 \leq ||uP^T - vP^T||_2^2 \leq (1+\epsilon)||u^T - v^T||_2^2. \tag{3.8}$$

The optimal ϵ and β can be obtained experimentally on validation set. As increasing β is somehow equivalent with decreasing ϵ, we simplify Equation (3.7) as follows

$$k = \frac{4}{\epsilon^2/2 - \epsilon^3/3} log(n). \tag{3.9}$$

To further reduce the complexity, we also apply the quantization technique to reduce the bit-width of parameters. Specifically, we apply a one-time uniform quantization on W^{LL} and b^{LL} to avoid complicated calculations. Although some other accurate quantization methods are available as well, we find that one-time quantization works well in our DMI. Besides, the input x is also quantized to x_Q during run-time to reduce the compute cost.

Knowledge Distillation. Training the LL module to be a good approximation of the HH module is critical. The Knowledge Distillation method is helpful to obtain effective approximation by taking the HH module as the teacher network and the LL module as a student network. Transferring the distilled knowledge as the flow of the solution procedure (FSP) matrix between two layers can increase the convergence rate and get better performance [119].

Let the outputs of two layers be x and z. The FSP matrix is in the form: $G = xy^T$ [119]. In order to transfer the knowledge, we want the FSP matrix of the student network

G_s to approximate to the FSP matrix of teach network G_t. The loss function is defined as

$$L = ||G_t - G_s||_F^2. \tag{3.10}$$

In the DMI method, we have

$$\begin{aligned} &G_t = x\left(y^{HH}\right)^T, \quad G_s = x\left(y^{LL}\right)^T, \\ &||G_t - G_s||_F^2 = Tr\left(xx^T\right)||y^{HH} - y^{LL}||_2^2. \end{aligned} \tag{3.11}$$

As a result, during fine-tuning, the parameters of the HH module (i.e., W^{HH} and b^{HH}) are kept frozen while the parameters of the LL module (i.e., W^{LL} and b^{LL}) are updated by stochastic gradient descent (SGD) to minimize the following loss function:

$$\begin{aligned} &L = \frac{1}{S}\sum_s ||y^{HH} - y^{LL}||_2^2 = \\ &\frac{1}{S}\sum_s ||(W^{HH}x + b^{HH}) - (W^{LL}Px + b^{LL})||_2^2, \end{aligned} \tag{3.12}$$

where S is the mini-batch size.

Insensitive Region Prediction. Whether a pre-activation belongs to the insensitive region is predicted based on whether y_i^{LL} is in the insensitive region. Without loss of generality, we can assume that $\forall i, y_i^{HH}$ follows some distribution p_{HH} with a probability density function $p_{HH}(x)$. As the y^{LL} is the output of an FC layer, according to the central limit theorem, we can then assume that each of its entries follow Gaussian distribution as follows

$$p_{LL} = N(y_\mu^{LL},\ \sigma_{LL}^2) \tag{3.13}$$

where σ_{LL} is a constant value and y_μ^{LL} gives the prediction of the mean. Similarly, the probability density function is $p_{LL}(x)$.

The difference between these two distributions can be measured by their Kullback-Leibler (KL) divergence, i.e.,

$$\begin{aligned} D_{KL}(p_{HH}||p_{LL}) &= \int_{-\infty}^{\infty} p_{HH}(x) ln\left(\frac{p_{HH}(x)}{p_{LL}(x)}\right) dx \\ &= \mathbb{E}_{x\sim p_{HH}}\left[ln\left(p_{HH}(x)\right) - ln\left(p_{LL}(x)\right)\right] \end{aligned} \tag{3.14}$$

where $\mathbb{E}$ represents the expectation. To make an accurate estimation, we can minimize the above KL divergence, which is equivalent to maximizing $\mathbb{E}_{x\sim p_{HH}}\left[ln(p_{LL}(x))\right]$, and we have

$$\begin{aligned} \mathbb{E}_{x\sim p_{HH}}\left[ln(p_{LL}(x))\right] &= \mathbb{E}\left[ln\left(\frac{1}{\sigma_{LL}\sqrt{2\pi}} e^{-\frac{(y^{HH}-y_{\mu}^{LL})^2}{2\sigma_{LL}^2}}\right)\right] \\ &\approx -ln(\sigma_{LL}) - \frac{1}{2}ln(2\pi) - \frac{1}{2\sigma_{LL}^2}\mathbb{E}\left[\left(y^{HH}-y_{\mu}^{LL}\right)^2\right]. \end{aligned} \tag{3.15}$$

Because of

$$\mathbb{E}\left[\left(y^{HH}-y_{\mu}^{LL}\right)^2\right] \approx \frac{1}{S}\sum_{i=1}^{S}\left(y_i^{HH}-y_i^{LL}\right)^2 = ||y^{HH}-y^{LL}||_2^2, \tag{3.16}$$

$D_{KL}(p_{HH}||p_{LL})$ can be equivalently minimized by minimizing $||y^{HH}-y^{LL}||_2^2$ that is just the loss function used in training the LL model (see Equation (3.12)).

3.3.3 Determine the Insensitive Region

Given y^{LL}, the binary mask m in Equation (3.5) is generated by predicting which output neurons belong to the insensitive region. Specifically, based on Section 3.2, we have

$$\begin{cases} sigmoid/tanh: & if\ |y_i^{LL}| > \theta_{th},\ m_i = 0;\ else\ m_i = 1 \\ ReLU: & if\ y_i^{LL} < \theta_{th},\ m_i = 0;\ else\ m_i = 1 \end{cases} \tag{3.17}$$

where θ_{th} is the threshold can be obtained in the ways as follows.

Fixed Threshold. We can simply assign a constant value to θ_{th} and tune it on validation set.

Adaptive Filter. With a global fixed threshold θ like the fixed threshold, we can assign a threshold adaptive to each layer based on Equation (3.4), i.e.,

$$\theta_{th} = \begin{cases} \frac{1}{2}\ln\frac{2|\delta|}{\theta} & for\ tanh \\ \ln\frac{|\delta|}{\theta} & for\ sigmoid \\ \theta - |\delta| & for\ ReLU \end{cases} . \tag{3.18}$$

$|\delta|$ is approximated by $\frac{1}{n}||y^{HH} - y^{LL}||_2$, where n is the length of y^{LL}. Compared with fixed threshold, the adaptive filter can allow more aggressive thresholds.

Top-K Mask. We can also specifically control the acceleration ratio by taking exactly K elements out of output neurons, where K can be a hyper-parameter tuned on validation set. Intuitively, these K output neurons should be taken from the sensitive region, so we have

$$\theta_{th} = \begin{cases} top_K(|y^{LL}|) & for\ tanh/sigmoid \\ top_K(y^{LL}) & for\ ReLU \end{cases} . \tag{3.19}$$

We introduce an insensitive ratio as the number of outputs using the results of the *little* module over the entire outputs. The ratio can be interpreted as the zero ratio in the binary mask m. The higher insensitive ratio will have fewer computations and memory accesses in the *big* module. The choice of an accurate ratio determines the model inference quality, and it is a knob to trade-off the inference quality vs. latency at run-time.

3.3.4 Overview of the Dual-Module Algorithm

Figure 3.1 summarizes the overall implementation of our dual-module algorithm during fine-tuning and inference.

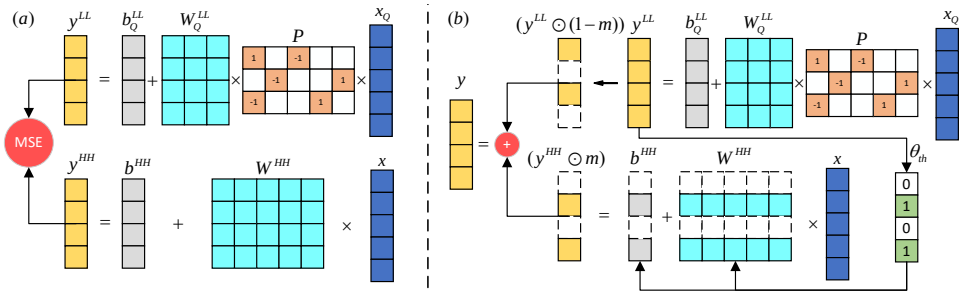


Figure 3.1: Overview of the Dual-Module Algorithm. (a) Fine-tuning: the LL module is trained with the loss function in Equation (3.12). (b) Inference: first, the quantized input activation x_Q is injected into the LL module to generate y^{LL}; second, mask m is obtained based on y^{LL} and the threshold θ_{th}; third, based on m, the elements in y^{HH} are selectively computed to produce $y^{HH} \odot m$; at last, y is obtained using Equation (3.5).

Fine-tuning. As illustrated in Figure 3.1(a), the LL modules in different layers are fine-tuned individually with the loss function in Equation (3.12). The detailed implementation is in Algorithm 1.

Algorithm 1: Dual-Module Fine-tuning Algorithm

Data: HH module parameters W^{HH}, b^{HH}; random projection matrix P; input batch $X = [x_1, ..., x_S]$.

Result: quantized LL module parameters: W_Q^{LL} and b_Q^{LL}.

for *it* $\in$ *all iterations* **do**

- $X_Q = Q(X)$;
- $\left[y_1^{LL}, ..., y_S^{LL}\right] = W_Q^{LL} P X_Q + b_Q^{LL}$;
- $\left[y_1^{HH}, ..., y_S^{HH}\right] = W^{HH} X + b^{HH}$;
- $L_{MSE} = \frac{1}{S} \sum_s ||y_s^{HH} - y_s^{LL}||_2^2$;
- update W_Q^{LL}, b_Q^{LL} with $SGD(\min L_{MSE})$;

end

Inference. The dual-module inference (DMI) is illustrated in Figure 3.1(b) based on Algorithm 2. After obtaining fine-tuned W_Q^{LL} and b_Q^{LL}, dual-module inference takes

Algorithm 2: Dual-Module Inference Algorithm

Data: HH module parameters W^{HH}, b^{HH}; quantized LL module parameters W_Q^{LL}, b_Q^{LL}; threshold θ_{th} to determine m; random projection matrix P; current input x.

Result: Final output y

(1) $x_Q = Q(x)$;
(2) $y^{LL} = W_Q^{LL} P x_Q + b_Q^{LL}$;
(3) Generating m according to Section 3.3.3;
(4-5) **foreach** $m_i \in m$ **do**
    if $m_i == 1$ **then** $y_i = y_i^{HH} = \varphi(W^{HH}[i,:]x + b_i^{HH})$;
    else $y_i = y_i^{LL}$;
end

the following steps: (1) quantize input x with $x_Q = Q(x)$, where $Q(\cdot)$ is a quantization function; (2) obtain the approximated output y^{LL} by performing $y^{LL} = W_Q^{LL} P x_Q + b_Q^{LL}$; (3) generate the binary mask m according to Section 3.3.3; (4) calculate the elements y_i^{HH} $s.t.$ $m_i = 1$ with $y_i^{HH} = W^{HH}[i,:]x + b_i^{HH}$; (5) produce the final output y according to the assembling in Equation (3.5).

3.3.5 Apply to Various Types of Neural Networks

The above example on FC layer can easily generalize to various types of neural networks, such as LSTM and CNNs.

Recurrent Neural Networks. There are two major differences between an LSTM layer and an FC layer: (1) the computation of each gate involves two GEMV operations; (2) there is an additional temporal dimension in LSTM. For the former, we can apply the lightweight linear transformation in Section 3.3.2 to both GEMVs. For the latter, we can modify the loss function to guarantee the approximation performance of the LL module at all time-steps. Taking the forget gate as an example, the loss function L_{MSE}

in Algorithm 1 is modified to

$$\begin{aligned} L_{MSE} &= \frac{1}{ST}\sum_{s}\sum_{t}||y_f^{HH}(t) - y_f^{LL}(t)||_2^2, \\ y_f^{LL}(t) &= b_{fQ}^{LL} + W_{fxQ}^{LL} P_x x_Q(t) + W_{fhQ}^{LL} P_h h_Q(t-1), \\ y_f^{HH}(t) &= b_f + W_{fx}x(t) + W_{fh}h(t-1). \end{aligned} \tag{3.20}$$

Convolutional Neural Networks. For a CONV layer, we can apply the dual-module algorithm to CNN by first doing the *im2col* transformation on input tensor [124]. Then, the input and output become matrices rather than vectors, but the overall algorithm is the same as in Section 3.3.4.

Batch Normalization. Batch normalization (BN) [125] is widely applied in DNNs. During inference, BN normalizes the input activations with

$$\hat{x} = \gamma(\frac{x-\mu}{\sigma}) + \beta, \tag{3.21}$$

where γ and β are trainable parameters, and μ and σ are the moving average of the mean and standard deviation of activations collected during training.

The dual-module algorithm is compatible with BN. When BN is applied before the activation function [126], i.e., $\varphi(BN(Wx+b))$, BN can be merged into the linear transformation as follows

$$\varphi\left(BN\left(Wx+b\right)\right) = \varphi\left(\frac{\gamma}{\sigma}Wx + \left(b + \beta - \frac{\gamma}{\sigma}\mu\right)\right). \tag{3.22}$$

We can have $\hat{W} = \frac{\gamma}{\sigma}W$ and $\hat{b} = b + \beta - \frac{\gamma}{\sigma}\mu$, then the DMI algorithm is directly applicable. When BN is applied after the activation function, the $\varphi\left(Wx+b\right)$ structure is not influenced by BN.

3.4 Evaluation

The dual-module inference (DMI) method is generally applicable to both RNNs and CNNs to improve the inference efficiency. The evaluation part is on a representative set of RNN and CNN models to demonstrate the effectiveness of the DMI method. In Appendix B, more extensive results, as well as evaluation methodology and experimental settings, can strengthen the conclusion. Training the *little* module and evaluating for the inference quality uses the PyTorch framework. When training the *little* module, the parameters of the *big* module are frozen, i.e., excluded from training process, and the same training set and validation set are used to run the SGD optimization.

3.4.1 Experimental Results on RNNs

As memory access is the bottleneck in RNN-based inference, DMI focuses on reducing overall memory access while keeping the overhead of executing the *little* module small. As shown in Figure 3.2, compared with parameters and operations of the single-module, i.e., the baseline case, using a set of LSTM and GRU layers, the *little* module needs much fewer parameters and operations. On average, the *little* module accounts only 8% memory overhead and 35% operation overhead compared with the baseline. Note that we count the number of operations in Figure 3.2 regardless of precision. The computation overhead of the *little* module can be further reduced using a low-precision implementation.

The DMI method is evaluated on CPU-based server platform (Intel(R) Xeon(R) CPU E5-2698 v4) as most inference workloads run on CPUs [127]. The baseline implementation is the PyTorch CPU version with Intel MKL (version 2019.4) as the back-end BLAS kernel library. The custom implementation uses a multi-threaded MKL dot-product kernel at BLAS level-1 to perform the *big* module instead of BLAS level-2 or level-3 kernels. The kernel implementation does not explicitly generate masks, but it directly compares little

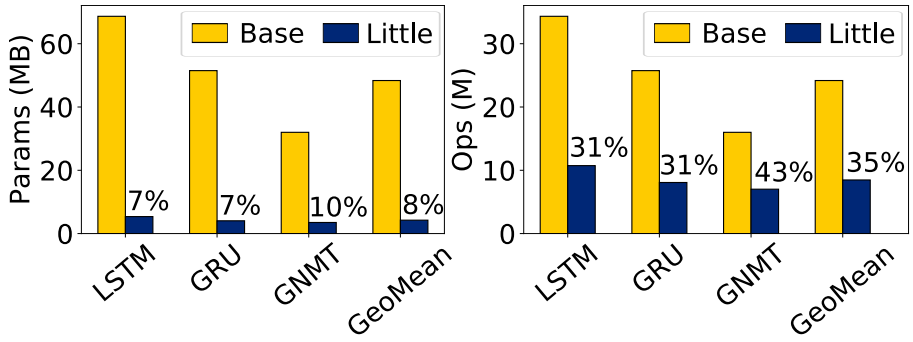


Figure 3.2: Comparison of the amount of accesses and operations between baseline layers and the *little* module of dual-module enhanced RNN-based models.

module results with predetermined thresholds in *OpenMP* parallel-for loops. The kernel-wise performance is measured as wall-clock time and averaged with 1000 runs, assuming cold cache at the execution of each RNN cell. Accessing weights from off-chip memory at each time-step represents the real-world cases, for example, the decoder execution of sequence-to-sequence modeling.

Language Modeling. The implementations of LSTMs/GRUs are adapted from the word-level language modeling example from PyTorch using the same hyper-parameters to train baseline models. Word-level perplexity (PPL) is the measure of model quality. Table 3.2 lists the quality-performance trade-off by varying the insensitive ratio. The larger insensitive ratio indicates more results are from the *little* module and less memory overhead to perform the *big* module. As the insensitive ratio increases, we can observe the degradation of quality as the PPL increases during a gradual reduction in execution time. When the insensitive ratio is 50%, the PPL is slightly increased to 81.36, which is negligible in language modeling tasks, while resulting in 1.67x inference speedup.

Table 3.2 further reports the results using single-layer GRUs on word-level language modeling tasks. Using dual-module method on GRUs expresses the similar quality-performance trade-off as on LSTMs.

Neural Machine Translation. Given the promising results on language modeling,

Table 3.2: RNN quality and execution time (ms). Ln means a LSTM layer with h hidden units; G is short for GRU.

Insensitive	LM, L1500				LM, G1500				GNMT, L1024			
Ratio	PPL	Diff.	Time	Speedup	PPL	Diff.	Time	Speedup	BLEU	Diff.	Time	Speedup
Baseline	80.64	n/a	1.477	1.00x	85.48	n/a	1.182	1.00x	24.32	n/a	0.838	1.00x
10%	80.72	-0.08	1.315	1.12x	85.62	-0.14	1.024	1.15x	24.33	0.01	0.679	1.23x
30%	80.56	0.08	1.095	1.35x	86.01	-0.53	0.869	1.36x	24.18	-0.14	0.541	1.55x
50%	81.36	-0.72	0.885	1.67x	88.73	-3.25	0.726	1.63x	23.73	-0.59	0.480	1.75x
70%	87.48	-6.83	0.641	2.30x	98.09	-12.61	0.545	2.17x	21.92	-2.40	0.360	2.33x
90%	109.37	-28.73	0.380	3.89x	122.75	-37.27	0.350	3.38x	11.77	-12.55	0.243	3.45x

Neural Machine Translation (NMT) is further investigated, which is a popular end-to-end learning approach for automated translation [2] and a standard benchmark model for inference as in MLPerf[1]. The experiments show the de-tokenized BLEU score to measure the model quality on the public WMT16 English-German dataset. The base model[2] consists of a four-layer stacked LSTM in both the encoder and the decoder of the sequence-to-sequence modeling. The focus is on the speedup of the decoder since it is the most memory intensive and the most time-consuming part (about 95%).

The DMI-enabled LSTM layers can replace the standard LSTM layers in the decoder. Similar to the single-layer LSTM results, using the *little* module computed results in the insensitive region can reduce the overall memory access while maintaining the model quality. As listed in Table 3.2, the DMI method can achieve imperceptible BLEU score degradation while accelerating inference by 1.75x. When compromising more translation quality, e.g., decreasing the BLEU score by 2.4, the DMI method can achieve more than 2x speedup.

[1]https://mlperf.org/inference-overview/
[2]From https://github.com/NVIDIA/DeepLearningExamples

3.4.2 Experimental Results on CNNs

Using DMI on CNNs can be regarded as pursuing output sparsity. Table 3.3 compares the classification accuracy and FLOPs reduction of the DMI method with other state-of-the-art methods on predicting *ReLU*-induced output sparsity when ResNet-18 is used for ImageNet classification. The results show that DMI outperforms other methods in delivering better trade-off between the accuracy drop and the FLOPs reduction. With accuracy degrading only 0.5%, the DMI method can achieve 3.02x FLOPs reduction.

Table 3.3: Comparison of the Top-1 accuracy and FLOPs reduction of our method with prior work on dynamic sparsity. The baseline model is ResNet-18 on ImageNet.

Method	Acc. (%)	Diff. (%)	FLOPs reduction
Dense (*torchvision*)	69.7	n/a	1.00x
LCL [53]	66.3	-3.4	1.53x
FBS [54]	68.2	-1.5	1.98x
SeerNet [52]	69.3	-0.4	1.67x
CGNet [55]	68.8	-0.9	1.93x
DMI (Ours)	**69.2**	**-0.5**	**3.02x**

Pixel-wise dynamic output sparsity can be accelerated by either customized GEMM kernel [128] or specialized hardware [101]. Using commodity processors or hardware accelerators could translate the FLOPs reduction on CNNs by DMI to practical speedup. Energy efficiency is another important criterion to evaluate any CNN inference execution, especially for mobile and edge devices. The energy consumption of DMI is shown here, normalized to the energy of running the baseline dense layers of each residual block in ResNet-18. The purpose of Figure 3.3 is to estimate the energy consumption with a focus on the portion of MAC operations which consume the majority of total energy in compute-bound CNNs. The methodology is multiplying the total number of MAC operations of big/little modules with the energy (J) per MAC operation accordingly. The energy/op numbers are from synthesized hardware evaluation. As shown in Figure

3.3, the energy efficiency improvement of using DMI is from 1.7x to 4.9x; on average, the DMI method can improve the energy efficiency by 2.9x.

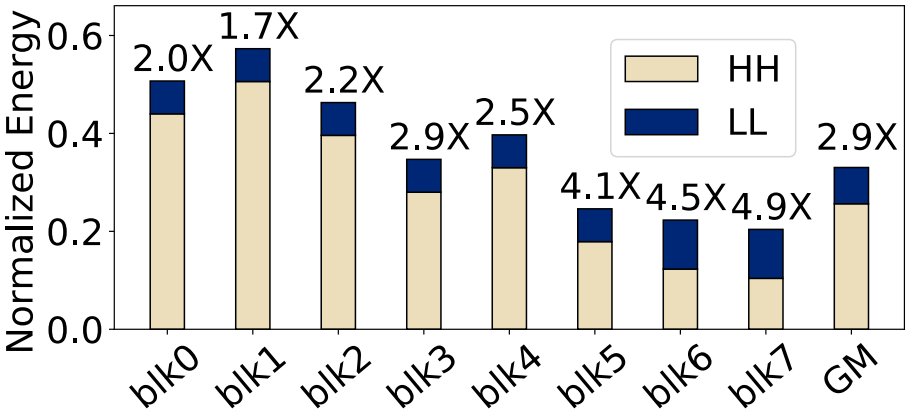


Figure 3.3: Energy efficiency of each residual block in ResNet-18.

3.4.3 Discussion on Dimension Reduction

Dimension reduction is an integral part of the DMI method to reduce the number of parameters of the *little* module. Here shows a study on the impact of different levels of dimension reduction on the model quality and performance. Experiments are conducted on language modeling using a single-layer LSTM of 1500 hidden units. The *little* module is quantized to INT8 and reduced in terms of hidden dimension from 1500 to three different levels, which are calculated by Equation (3.9). The insensitive ratio is fixed at 50% across this set of experiments. As in Table 3.4, the higher dimension of the *little* module, the better approximation the *little* module can perform. More aggressive dimension reduction can further gain more speedup at the cost of more quality degradation: hidden dimension reduced to 417 and 266 can have 1.67x and 1.71x speedup but increase the PPL by 0.72 and 2.87, respectively.

The overhead of performing the computations of the *little* module is discussed here. The last two columns in Table 3.4 measure the execution time of computing the *little*

Table 3.4: Sensitivity study on dimension reduction.

Dimension	PPL	Speedup	*little*	*big*
1500 (baseline)	80.64	1.00x	0%	100%
966 ($\epsilon = 0.3$)	80.40	1.37x	22%	44%
417 ($\epsilon = 0.5$)	81.36	1.67x	12%	47%
266 ($\epsilon = 0.7$)	83.51	1.71x	8%	46%

module and the operation-reduced *big* module. The execution time is normalized to the baseline case, i.e., the execution time of the standard LSTM, to highlight the percentage of overheads. When the hidden dimension is reduced to 966, the overhead of the *little* module accounts 22% while the execution time of the *big* module is cut off by half[3]. In the experiments, $\epsilon = 0.5$ is chosen as the default parameter as it demonstrates good trade-off between quality and speedup in this study. When further reducing the hidden dimension to 266, there is only a slight improvement on speedup compared with the hidden size of 417 in the *little* module, where the overhead of the *little* module is already small enough, but the quality drop is significant.

3.4.4 Discussion on Quantization

Weight quantization of the *little* module is another integral part of constructing the *little* module, showing the impact of different quantization levels on the model quality and the parameter size. After training the *little* module, using the same settings as in Section 3.4.1, we can quantize the weights to lower precision to reduce the memory access on top of the dimension reduction. As listed in Table 3.5, more aggressive quantization leads to smaller parameter size that can reduce the overhead of computing the *little* module; on the other hand, the approximation of the *little* module is compromised by

[3]The execution time is measured with multi-threading.

quantization. The *little* module can be quantized down to INT4 without significant quality degradation. Using lower precision would degrade the quality while decreasing the parameter size. Normally, using INT8 as the quantization level is a good choice for CPU-based implementations when leveraging off-the-shelf INT8 GEMM kernel in MKL. We can expect more speedup once the *little* module overhead can be further reduced by leveraging INT4 compute kernels or running on specialized hardware.

Table 3.5: Inference quality and parameter size comparison under different levels of quantization on the *little* module

Precision	Base	FP32	INT16	INT8	INT4	INT2
PPL	80.64	81.28	81.18	81.36	81.47	82.43
MSE	n/a	0.408	0.425	0.444	0.451	0.68
Params.	68.7	19.1	9.6	4.8	2.4	1.2

3.5 Conclusion

This Chapter describes the *big-little* dual-module inference (DMI) method to boost the execution efficiency of DNNs. Leveraging the noise resilience of nonlinear activation functions, DMI consists a lightweight *little* module to compute for the insensitive region and using the *big* module with skipped memory access and computation to compute for the sensitive region. The DMI method can reduce overall memory access by near half for the memory-bound RNNs and achieve 1.54× to 1.75× wall-clock time speedup without significant degradation on model quality. For the compute-bound CNNs, DMI can achieve 3.02× operation reduction with only a 0.5% accuracy drop.

Chapter 4

Dynamic Sparse Graph

This Chapter presents the application of *elastic processing* in DNN training. The hypothesis is similar with what discussed in Chapter 3 but in a different context: we can also find critical vs. trivial neurons during DNN training.

4.1 Introduction

DNN training, which demands much more hardware resources in terms of both memory capacity and computation volume, is far more challenging than inference. Firstly, activation data in training will be stored for back-propagation, significantly increasing the memory consumption. Secondly, iterative training updates model parameters using mini-batched stochastic gradient descent. We almost always expect larger mini-batches for higher throughput (Figure 4.1(a)), faster convergence, and better accuracy [129]. However, memory capacity is often the limitation factor (Figure 4.1(b)) that may cause performance degradation or even make large models with deep structures or targeting high-resolution vision tasks hard to train [130, 131].

It is difficult to apply existing sparsity techniques towards inference phase to training

phase because of the following reasons: 1) Prior arts mainly compress the pre-trained and fixed weight parameters to reduce the off-chip memory access in inference [33, 86], while instead, the dynamic neuronal activations turn out to be the crucial bottleneck in training [132], making the prior inference-oriented methods inefficient. Besides, during training we need to stash a vast batched activation space for the backward gradient calculation. Therefore, neuron activations creates a new memory bottleneck (Figure 4.1(c)). In this Chapter, we will sparsify the neuron activations for training compression. 2) The existing inference accelerations usually add extra optimization problems onto the critical path [48, 49, 41, 44, 46, 133, 47, 43, 42], i.e., "complicated training ⇒ simplified inference", which embarrassingly complicates the training phase. 3) Moreover, previous studies reveal that batch normalization (BN) is crucial for improving accuracy and robustness (Figure 4.1(d)) through activation fusion across different samples within one mini-batch for better representation [134, 125]. BN almost becomes a standard training configuration; however, inference-oriented methods seldom discuss BN and treat BN parameters as scaling and shift factors in the forward pass. We further find that BN will damage the sparsity due to the activation reorganization (Figure 4.1(e)). Since this Chapter focuses on training, the BN compatibility problem should be addressed.

From the view of information representation, the activation of each neuron reflects its selectivity to the current stimulus sample [134], and this selectivity dataflow propagates layer by layer forming different representation levels. Fortunately, there is much representational redundancy, for example, lots of neuron activations for each stimulus sample are so small and can be removed (Figure 4.1(f)). Motivated by above comprehensive analysis regarding memory and compute, this work proposes to search critical neurons for constructing a sparse graph at every iteration. By activating only a small amount of neurons with a high selectivity, we can significantly save memory and simplify computation with tolerable accuracy degradation. Because the neuron response dynamically changes under

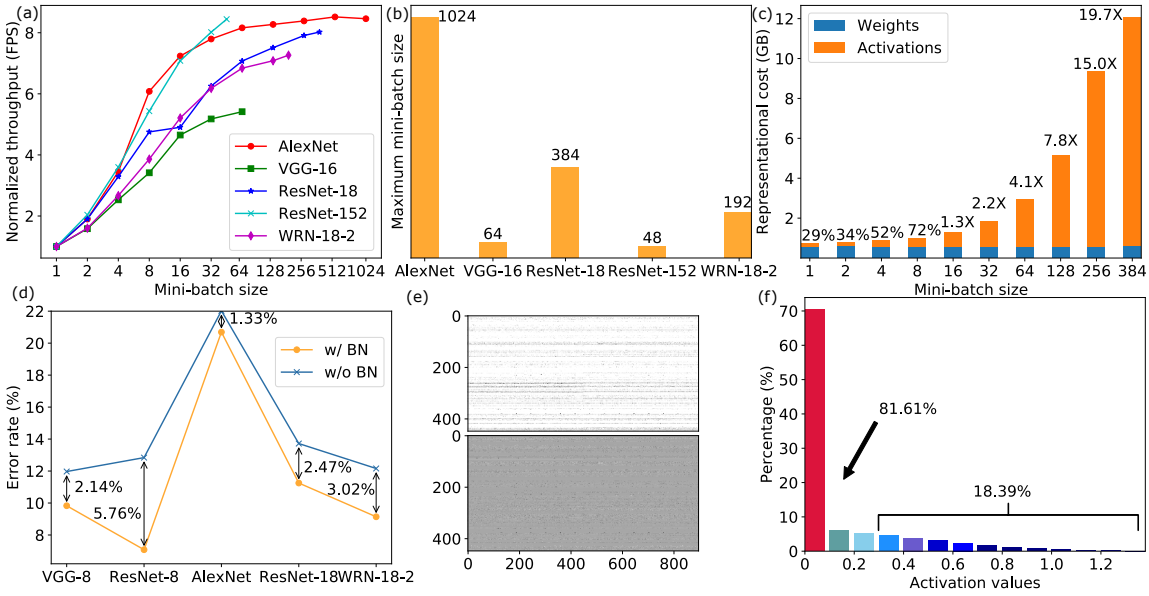


Figure 4.1: Comprehensive motivation illustration. (a) Using larger mini-batch size helps improve throughput until it is compute-bound; (b) Limited memory capacity on a single computing node prohibits the use of large mini-batch size; (c) Neuronal activation dominates the representational cost when mini-batch size becomes large; (d) BN is indispensable for maintaining accuracy; (e) Upper and lower one are the feature maps before and after BN, respectively. However, using BN damages the sparsity through information fusion; (f) There exists such great representational redundancy that more than 80% of activations are close to zero.

different stimulus samples, the sparse graph is variable. The neuron-aware dynamic and sparse graph (DSG) is fundamentally distinct from the static one in previous work on permanent weight pruning since we never prune the graph but activate part of them each time. Therefore, we can maintain the model expressive power as much as possible. A graph selection method, dimension-reduction search, is designed for both compressible activations with element-wise unstructured sparsity and accelerative vector-matrix multiplication (VMM) with vector-wise structured sparsity. Through double-mask selection design, it is also compatible with BN. We can use the same selection pattern and extend the DSG approach to inference. In a nutshell, this Chapter presents a compressible and accelerative DSG approach supported by dimension-reduction search and double-mask selection. It can achieve 1.7-4.5x memory compression and 2.3-4.4x computation reduc-

tion with minimal accuracy loss. The work presented in this Chapter simultaneously pioneers the approach towards efficient online training and offline inference, which can benefit the deep learning in both the cloud and the edge.

4.2 Approach

The proposed method constructs computational graphs dynamically for different inputs, being accelerative and compressive, as shown in Figure4.2(a). On the one hand, choosing a small number of critical neurons to participate in computation, DSG can reduce the computational cost by eliminating calculations of non-critical neurons. On the other hand, it can further reduce the representational cost via compression on sparsified activations. Different from previous methods using permanent pruning, the proposed approach does not prune any neuron and the associated weights; instead, it activates a sparse graph according to the input sample at each iteration. Therefore, DSG does not compromise the expressive power of the model.

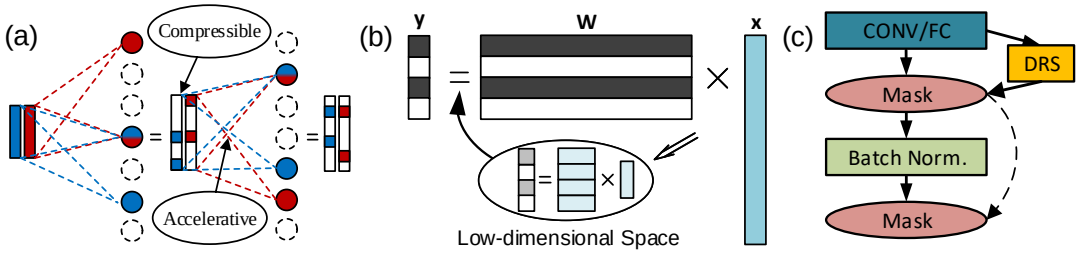


Figure 4.2: (a) Illustration of dynamic and sparse graph (DSG); (b) Dimension-reduction search for construction of DSG; (c) Double-mask selection for BN compatibility. 'DRS' denotes dimension-reduction search.

Constructing DSG needs to determine which neurons are critical. A naive approach is to select critical neurons according to the output activations. If the output neurons have a small or negative activation value, i.e., not selective to current input sample, they

can be removed for saving representational cost. Because these activations will be small or absolute zero after the following ReLU non-linear function (i.e., ReLU(x) = max(0, x)), it's reasonable to set all of them to be zero. However, this naive approach requires computations of all VMM operations within each layer before the selection of critical neurons, which is very costly.

4.2.1 Dimension-reduction Search

To avoid the costly VMM operations in the mentioned naive selection, we propose an efficient method, i.e., dimension reduction search, to estimate the importance of output neurons. As shown in Figure4.2(b), we first reduce the dimensions of $\mathbf{X}$ and $\mathbf{W}$, and then execute the lightweight VMM operations in a low-dimensional space with minimal cost. After that, we estimate the neuron importance according to the virtual output activations. Then, a binary selection mask can be produced in which the zeros represent the non-critical neurons with small activations that are removable. We use a top-k search method that only keeps largest k neurons, where an inter-sample threshold sharing mechanism is leveraged to greatly reduce the search cost [1]. Note that k is determined by the output size and a pre-configured sparsity parameter γ. Then we can just compute the accurate activations of the critical neurons in the original high-dimensional space and avoid the calculation of the non-critical neurons. Thus, besides the compressive sparse activations, the dimension-reduction search can further save a significant amount of expensive operations in the high-dimensional space.

In this way, a vector-wise structured sparsity can be achieved, as shown in Figure 4.3(b). The ones in the selection mask (marked as colored blocks) denote the critical neurons, and the non-critical ones can bypass the memory access and computation of their corresponding columns in the weight matrix. Furthermore, the generated sparse

[1]Implementation details are shown in Appendix B.

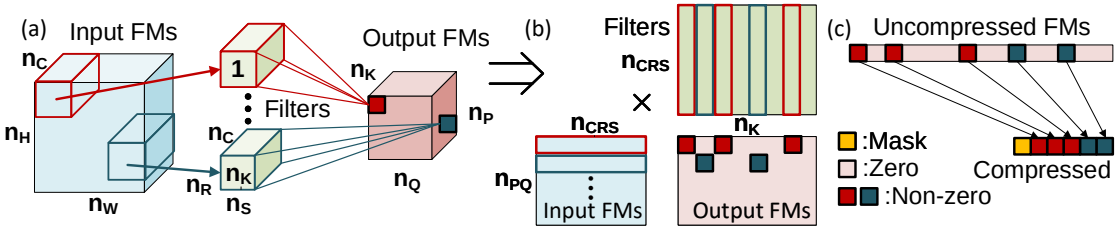


Figure 4.3: Compressive and accelerative DSG. (a) Original dense convolution; (b) Converted accelerative VMM operation; (c) Zero-value compression.

activations can be compressed via the zero-value compression [135, 136, 137] (Figure 4.3(c)). Consequently, it is critical to reduce the vector dimension but keep the activations calculated in the low-dimensional space as accurate as possible, compared to the ones in the original high-dimensional space.

4.2.2 Sparse Random Projection for Efficient Dimension-reduction Search

Notations: Each CONV layer has a four dimensional weight tensor (n_K, n_C, n_R, n_S), where n_K is the number of filters, i.e., the number of output feature maps (FMs); n_C is the number of input FMs; (n_R, n_S) represents the kernel size. Thus, the CONV layer in Figure 4.3(a) can be converted to many VMM operations, as shown in Figure 4.3(b). Each row in the matrix of input FMs is the activations from a sliding window across all input FMs ($n_{CRS} = n_C \times n_R \times n_S$), and after the VMM operation with the weight matrix ($n_{CRS} \times n_K$) it can generate n_K points at the same location across all output FMs. Further considering the $n_{PQ} = n_P \times n_Q$ size of each output FM and the mini-batch size of m, the whole $n_{PQ} \times m$ rows of VMM operations has a computational complexity of $O(m \times n_{PQ} \times n_{CRS} \times n_K)$. For the FC layer with n_C input neurons and n_K output neurons, this complexity is $O(m \times n_C \times n_K)$. Note that here we switch the order of BN and ReLU layer from 'CONV/FC-BN-ReLU' to 'CONV/FC-ReLU-BN', because it's

hard to determine the activation value of the non-critical neurons if the following layer is BN (this value is zero for ReLU). As shown in previous work, this reorganization could bring better accuracy [138].

For the sake of simplicity, we just consider the operation for each sliding window in the CONV layer or the whole FC layer under one single input sample as a basic optimization problem. The generation of each output activation y_j requires an inner product operation, as follows:

$$y_j = \varphi(\langle \mathbf{X}_i, \mathbf{W}_j \rangle) \tag{4.1}$$

where $\mathbf{X}_i$ is the i-th row in the matrix of input FMs (for the FC layer, there is only one $\mathbf{X}$ vector), $\mathbf{W}_j$ is the j-th column of the weight matrix W, and $\varphi(\cdot)$ is the neuronal transformation (e.g., ReLU function, here we abandon bias). Now, according to equation (4.1), the preservation of the activation is equivalent to preserve the inner product.

A dimension-reduction lemma is introduced, namely Johnson-Lindenstrauss Lemma (JLL) [139], to implement the dimension-reduction search with inner product preservation. This lemma states that a set of points in a high-dimensional space can be embedded into a low-dimensional space in such a way that the Euclidean distances between these points are nearly preserved. Specifically, given $0 < \epsilon < 1$, a set of N points in $\mathbb{R}^d$ (i.e., all $\mathbf{X}_i$ and $\mathbf{W}_j$), and a number of $k > O(\frac{log(N)}{\epsilon^2})$, there exists a linear map $f : \mathbb{R}^d \Rightarrow \mathbb{R}^k$ such that

$$(1-\epsilon)\|\mathbf{X}_i - \mathbf{W}_j\|^2 \leq \|f(\mathbf{X}_i) - f(\mathbf{W}_j)\|^2 \leq (1+\epsilon)\|\mathbf{X}_i - \mathbf{W}_j\|^2 \tag{4.2}$$

for any given $\mathbf{X}_i$ and $\mathbf{W}_j$ pair, where ϵ is a hyper-parameter to control the approximation error, i.e., larger $\epsilon \Rightarrow$ larger error. When ϵ is sufficiently small, one corollary from JLL

is the following norm preservation:

$$P[\ (1-\epsilon)\|\mathbf{Z}\|^2 \leq \|f(\mathbf{Z})\|^2 \leq (1+\epsilon)\|\mathbf{Z}\|^2\ ] \geq 1 - O(\epsilon^2) \tag{4.3}$$

where $\mathbf{Z}$ could be any $\mathbf{X}_i$ or $\mathbf{W}_j$, and P denotes a probability. It means the vector norm can be preserved with a high probability controlled by ϵ. Given these basics, we can further get the inner product preservation:

$$P[\ |\langle f(\mathbf{X}_i), f(\mathbf{W}_j)\rangle - \langle \mathbf{X}_i, \mathbf{W}_j\rangle| \leq \epsilon\ ] \geq 1 - O(\epsilon^2). \tag{4.4}$$

The detailed proof can be found in Appendix A.1.

Random projection [140, 141] is widely used to construct the linear map $f(\cdot)$. Specifically, the original d-dimensional vector is projected to a k-dimensional ($k \ll d$) one, using a random $k \times d$ matrix $\mathbf{R}$. Then we can reduce the dimension of all $\mathbf{X}_i$ and $\mathbf{W}_j$ by

$$f(\mathbf{X}_i) = \frac{1}{\sqrt{k}}\mathbf{R}\mathbf{X}_i \in \mathbb{R}^k, \quad f(\mathbf{W}_j) = \frac{1}{\sqrt{k}}\mathbf{R}\mathbf{W}_j \in \mathbb{R}^k. \tag{4.5}$$

The random projection matrix $\mathbf{R}$ can be generated from Gaussian distribution [140]. The work discussed here adopts a simplified version, termed as sparse random projection [141, 121, 122] with

$$P(\mathbf{R}_{pq} = \sqrt{s}) = \frac{1}{2s}; \quad P(\mathbf{R}_{pq} = 0) = 1 - \frac{1}{s}; \quad P(\mathbf{R}_{pq} = -\sqrt{s}) = \frac{1}{2s} \tag{4.6}$$

for all elements in $\mathbf{R}$. This $\mathbf{R}$ only has ternary values that can remove the multiplications during projection, and the remained additions are very sparse. Therefore, the projection overhead is negligible compared to other high-precision operations involving multiplication. Here we set $s = 3$ with 67% sparsity in statistics.

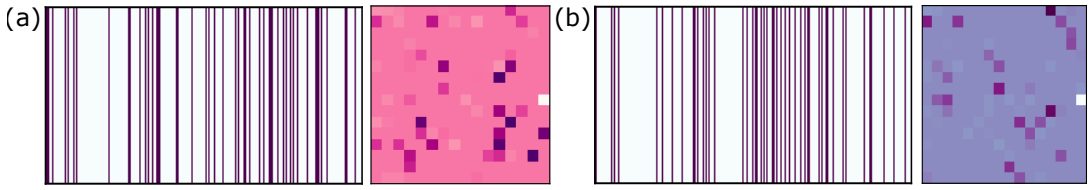


Figure 4.4: Structured selection via dynamic dimension-reduction search for producing sparse pattern of neuronal activations.

Equation (4.4) indicates the low-dimensional inner product $\langle f(\mathbf{X}_i), f(\mathbf{W}_j)\rangle$ can still approximate the original high-dimensional one $\langle \mathbf{X}_i, \mathbf{W}_j\rangle$ in equation (4.1) if the reduced dimension is sufficiently high. Therefore, it is possible to calculate equation (4.1) in a low-dimensional space for activation estimation, and select the important neurons. As shown in Figure 4.3(b), each sliding window dynamically selects its own important neurons for the calculation in high-dimensional space, marked in red and blue as two examples. Figure 4.4 visualizes two sliding windows in a real network to help understand the dynamic process of dimension-reduction search. Here the neuronal activation vector (n_K length) is reshaped to a matrix for clarity. Now For the CONV layer, the computational complexity is only $O[\ m \times n_{PQ} \times n_K \times (k + (1-\gamma) \times n_{CRS})\ ]$, which is less than the original high-dimensional computation with $O(m \times n_{PQ} \times n_{CRS} \times n_K)$ complexity because we usually have $[\ k + (1-\gamma) \times n_{CRS}\ ] \ll n_{CRS}$. For the FC layer, we also have $O[\ m \times n_K \times (k + (1-\gamma) \times n_C)\ ] \ll O(m \times n_C \times n_K)$.

4.2.3 Double-mask Selection for BN Compatibility

To deal with the important but intractable BN layer, we propose a double-mask selection method presented in Figure 4.2(c). After the dimension-reduction search based importance estimation, we produce a sparsifying mask that removes the unimportant neurons. The ReLU activation function can maintain this mask by inhibiting the negative activation (actually all the activations of the CONV layer or FC layer after the selection

mask are positive with reasonably large sparsity). However, the BN layer will damage this sparsity through inter-sample activation fusion. To address this issue, we copy the same selection mask before the BN layer and directly use it on the BN output. It is straightforward but reasonable because we find that although BN causes the zero activation to be non-zero (Figure 4.1(f)), these non-zero activations are still very small and can also be removed. This is because BN just scales and shifts the activations that won't change the relative sort order. In this way, we can achieve fully sparse activation dataflow. The back propagated gradients will also be forcibly sparsified every time they pass a mask layer.

4.3 Evaluation

The overall training algorithm is presented in Appendices A.1. Going through the dataflow where the red color denotes the sparse tensors, a widespread sparsity in both the forward and backward passes is demonstrated. The projection matrices are fixed after a random initialization at the beginning of training. The projected weights in the low-dimensional space are updated every 50 iterations to reduce the projection overhead. The detailed search method and the computational complexity of the dimension-reduction search are provided in Appendix A.1. The evaluation network models include LeNet [142] and a multi-layered perceptron (MLP) on small-scale FASHION dataset [143], VGG8 [144, 145]/ResNet8 (a customized ResNet-variant with 3 residual blocks and 2 FC layers)/ResNet20/WRN-8-2 [146] on medium-scale CIFAR10 dataset [147], VGG8/WRN-8-2 on another medium-scale CIFAR100 dataset [147], and AlexNet [148]/VGG16 [149]/ResNet18, ResNet152 [130]/WRN-18-2 [146] on large-scale ImageNet dataset [150] as workloads. The programming framework is PyTorch and the training platform is based on NVIDIA Titan Xp GPU. The zero-value compression method [135, 136, 137]

is used for memory compression and MKL compute library [151] on Intel Xeon CPU for acceleration evaluation.

4.3.1 Accuracy Analysis

This section provides a comprehensive analysis regarding the influence of sparsity on accuracy and explore the robustness of MLP and CNN, the graph selection strategy, the BN compatibility, and the importance of width and depth.

Accuracy using DSG. Figure 4.5(a) presents the accuracy curves on small and medium scale models by using DSG under different sparsity levels. Three conclusions are observed: 1) The proposed DSG affects little on the accuracy when the sparsity is <60%, and the accuracy will present an abrupt descent with sparsity larger than 80%. 2) Usually, the ResNet model family is more sensitive to the sparsity increasing due to fewer parameters than the VGG family. For the VGG8 on CIFAR10, the accuracy loss is still within 0.5% when sparsity reaches 80%. 3) Compared to MLP, CNN can tolerate more sparsity. Figure 4.5(b) further shows the results on large scale models on ImageNet. Because training large model is time costly, here only presents several experimental points. Consistently, the VGG16 shows better robustness compared to the ResNet18, and the WRN with wider channels on each layer performs much better than the other two models. The topics of width and depth are discussed later.

Graph Selection Strategy. To investigate the influence of graph selection strategy, we repeat the sparsity vs. accuracy experiments on CIFAR10 under different selection methods. Two baselines are used here: the oracle one that keeps the neurons with top-k activations after the whole VMM computation at each layer, and the random one that randomly selects neurons to keep. The results are shown in Figure 4.5(c), in which we can see that the dimension-reduction search and the oracle one perform much better

than the random selection under high sparsity condition. Moreover, dimension-reduction search achieves nearly the same accuracy with the oracle top-k selection, which indicates the proposed random projection method can find an accurate activation estimation in the low-dimensional space. In detail, Figure 4.5(d) shows the influence of parameter ϵ that reflects the degree of dimension reduction. Lower ϵ can approach the original inner product more accurately, that brings higher accuracy but at the cost of more computation for graph selection since less dimension reduction. With $\epsilon = 0.5$, the accuracy loss is within 1% even if the sparsity reaches 80%.

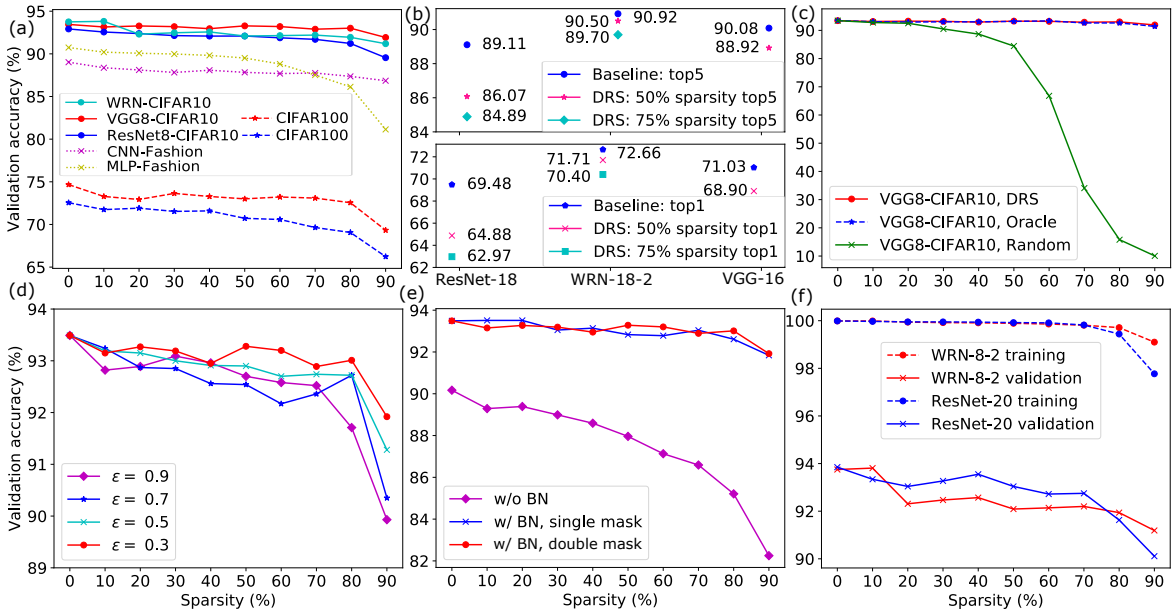


Figure 4.5: Comprehensive analysis on sparsity v.s. accuracy. (a) & (b) Accuracy using DSG; (c) Influence of the graph selection strategy; (d) Influence of the dimension-reduction degree; (e) Influence of the double-mask selection for BN compatibility; (f) Influence of the network depth and width. DRS denotes dimension-reduction search.

BN Compatibility. Figure 4.5(e) focuses the BN compatibility issue. Here we use dimension-reduction search for the graph sparsifying, and compare three cases: 1) removing the BN operation and using single mask; 2) keeping BN and using only single mask (the first one in Figure 4.2(c)); 3) keeping BN and using double masks (i.e. double-mask

selection). The one without BN is very sensitive to the graph ablation, which indicates the importance of BN for training. Comparing the two with BN, the double-mask selection even achieves better accuracy since the regularization effect. This observation indicates the effectiveness of the proposed double-mask selection for simultaneously recovering the sparsity damaged by the BN layer and maintaining the accuracy.

Width or Depth. Furthermore, we investigate an interesting comparison regarding the network width and depth, as shown in Figure 4.5(f). On the training set, WRN with fewer but wider layers demonstrates more robustness than the deeper one with more but slimmer layers. On the validation set, the results are a little more complicated. Under small and medium sparsity, the deeper ResNet performs better (1%) than the wider one. While when the sparsity increases substantial (>75%), WRN can maintain the accuracy better. This indicates that, in medium-sparse space, the deeper network has stronger representation ability because of the deep structure; however, in ultra-high-sparse space, the deeper structure is more likely to collapse since the accumulation of the pruning error layer by layer. In reality, we can determine which type of model to use according to the sparsity requirement. In Figure 4.5(b) on ImageNet, the reason why WRN-18-2 performs much better is that it has wider layers without reducing the depth.

Convergence. DSG does not slow down the convergence speed, which can be seen from Figure A.2(a)-(b) in Appendix A.3. This owes to the high fidelity of inner product when using random projection to reduce the data dimension, as shown in Figure A.2(c). Interestingly, Figure A.3 (also in Appendix A.3) reveals that the selection mask for each sample also converges as training goes on, however, the selection pattern varies across samples. To save the selection patterns of all samples is memory consuming, which is the reason why we do not directly suspend the selection patterns after training but still do on-the-fly dimension-reduction search in inference.

4.3.2 Representational Cost Reduction

This section presents the benefits from DSG on representational cost, measured by the memory consumption over five CNN benchmarks on both the training and inference phases. The zero-value compression algorithm [135, 136, 137] is used for data compression. Figure 4.6 shows the memory optimization results, where the model name, mini-batch size, and the sparsity are provided. In training, besides the parameters, the activations across all layers should be stashed for the backward computation. Consistent with the observation mentioned above that the neuron activation beats weight to dominate memory overhead, which is different from the previous work on inference. We can reduce the overall representational cost by average 1.7x (2.72 GB), 3.2x (4.51 GB), and 4.2x (5.04 GB) under 50%, 80% and 90% sparsity, respectively. If only considering the neuronal activation, these ratios could be higher up to 7.1x. The memory overhead for the selection masks is minimal (<2%).

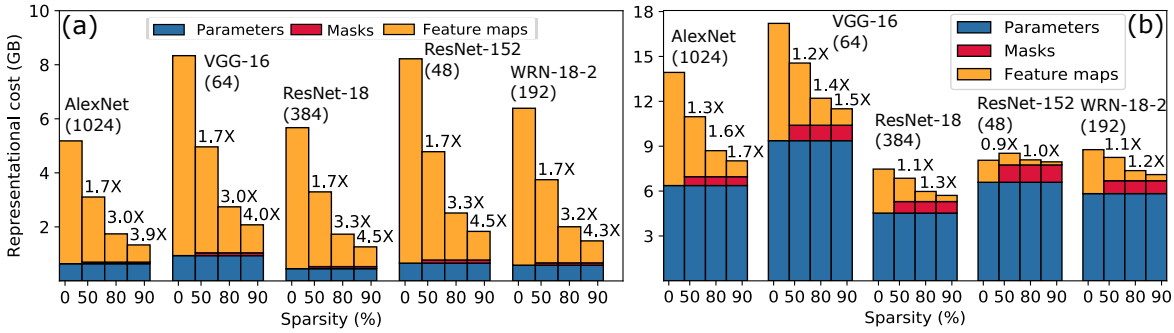


Figure 4.6: Memory footprint comparisons for (a) training and (b) inference.

During inference, only memory space to store the parameters and the activations of the layer with maximum neuron amount is required. The benefits in inference are relatively smaller than that in training since weight is the dominant memory. On ResNet152, the extra mask overhead even offsets the compression benefit under 50% sparsity, whereas, we can still achieve up to 7.1x memory reduction for activations and 1.7x for overall

memory. Although the compression is limited for inference, it still can achieve noticeable acceleration that will be shown in the next section. Moreover, reducing costs for both training and inference is the major contribution.

4.3.3 Computational Cost Reduction

Here shows the results on reducing the computational cost for both training and inference. As shown in Figure 4.7, both the forward and backward pass consume much fewer operations, i.e., multiply-and-accumulate (MAC). On average, 1.4x (5.52 GMACs), 1.7x (9.43 GMACs), and 2.2x (10.74 GMACs) operation reduction are achieved in training under 50%, 80% and 90% sparsity, respectively. For inference with only forward pass, the results increase to 1.5x (2.26 GMACs), 2.8x (4.22 GMACs), and 3.9x (4.87 GMACs), respectively. The overhead of the dimension-reduction search in the low-dimensional space is relatively larger (<6.5% in training and <19.5% in inference) compared to the mask overhead in memory cost. Note that the training demonstrates less improvement than the inference, which is because the acceleration of the backward pass is partial. The error propagation is accelerative, but the weight gradient generation is not because of the irregular sparsity that is hard to obtain practical acceleration. Although the computation of this part is also very sparse with much fewer operations [2], we do not include its GMACs reduction for practical concern.

Finally, the execution time on CPU is evaluated using Intel MKL kernels ([151]). Figure 4.8(a) evaluates the execution time of these layers after the dimension-reduction search on VGG8. Comparing to VMM baselines, the DSG approach can achieve 2.0x, 5.0x, and 8.5x average speedup under 50%, 80%, and 90% sparsity, respectively. When the baselines change to GEMM (general matrix multiplication), the average speedup decreases to 0.6x, 1.6x, and 2.7x, respectively. The reason is that DSG generates dynamic

[2] See Algorithm 5 in Appendices A.2

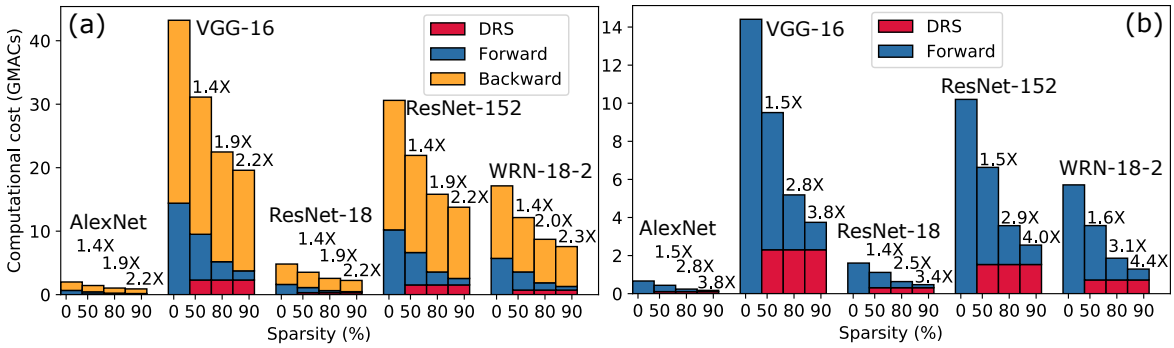


Figure 4.7: Computational complexity comparisons for (a) training and (b) inference. 'DRS' denotes dimension-reduction search.

vector-wise sparsity, which is not well supported by GEMM. A potential way to improve GEMM-based implementation, at workload mapping and tiling time, is reordering executions at the granularity of vector inner-product and grouping non-redundant executions to the same tile to improve local data reuse.

On the same network, DSG is further compared with smaller dense models which could be another way to reduce the computational cost. As shown in Figure 4.8(b), comparing with dense baseline, DSG can reduce training time with little accuracy loss. Even though the equivalent smaller dense models with the same effective nodes, i.e., reduced MACs, save more training time, the accuracy is much worse than the DSG approach. Figure A.4 in Appendix A.4 gives more results on ResNet8 and AlexNet.

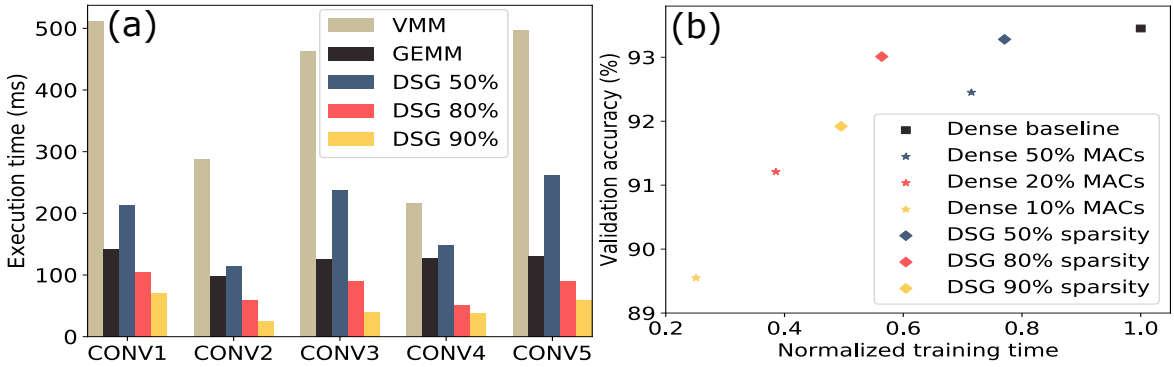


Figure 4.8: On VGG8: (a) Layer-wise execution time comparison; (b) Validation accuracy v.s. training time of different models: large-sparse ones and smaller-dense ones with equivalent MACs.

4.4 Conclusion

This Chapter discusses DSG (dynamic and sparse graph) structure for efficient DNN training and inference through a dimension-reduction search based sparsity forecast for compressive memory and accelerative execution and a double-mask selection for BN compatibility without sacrificing model's expressive power. It can be easily extended to the inference by using the same selection pattern after training. The experiments over various benchmarks demonstrate significant memory saving (up to 4.5x for training and 1.7x for inference) and computation reduction (up to 2.3x for training and 4.4x for inference). Through significantly boosting both forward and backward passes in training, as well as in inference, DSG promises efficient deep learning in both the cloud and edge.

Chapter 5

Dual-Module Architecture

Chapter 3 presents the dual-module inference method for computational savings of DNNs. This Chapter introduces a dedicated dual-module accelerator named DUET, to enable the proposed dual-module processing scheme with better performance and energy efficiency. The architecture of DUET is further devised to support different dataflow and mapping strategies optimized for CNNs and RNNs.

5.1 Architecture Design

The top-level block diagram of DUET is shown in Figure 5.1. Overall, the accelerator consists of three major components: an on-chip global buffer (GLB), a specialized 2D PE array called the Executor, and a decoupled Speculator to handle approximate module processing. In general, the Speculator and the Executor run in parallel. On the one hand, the Speculator uses outputs from the Executor to perform speculation, i.e., running approximate modules in the Speculator, generating approximate results and dynamic switching maps. On the other hand, Executor leverages the switching maps to reduce computations as well as memory access. The decoupled architecture of Executor and

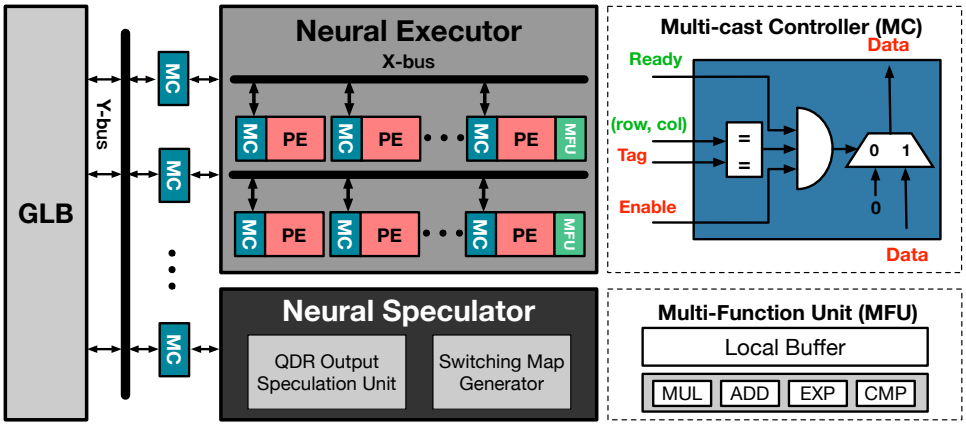


Figure 5.1: Overall architecture with an Executor running accurate modules and a Speculator running approximate modules.

Speculator further enables a fine-grained pipeline design of the dataflow, which will be demonstrated in Section 5.2.

Control: DUET has two-level of control logic. The global control configures the whole system and is responsible for handling traffic between on-chip GLB and off-chip DRAM, traffic between GLB and Executor, and traffic between GLB and Speculator. The lower-level control consists of the control logic inside each PE and the Speculator, which runs independently.

Global Buffer: DUET has a 1MB GLB that can communicate with both the off-chip DRAM module and with the on-chip computation resources (Executor and Speculator) through the NoC. Besides the data needed for computation (input, weight, and output), GLB also stores the data required and generated by the Speculator. These data include the weights for the Speculator, the switching maps to be used to reduce computations for both CNNs and RNNs), the mapping configuration to balance PE workloads for CNNs (see Section 5.2.1) and finally the approximate speculation results (only for RNNs, see Section 5.2.2). GLB provides a total bandwidth of 512B/cycle to feed the Executor and the Speculator sufficiently. The parameters are chosen through design space exploration and are validated via a cycle-accurate simulator.

Network-on-Chip: As shown in Figure 5.1, the NoC in DUET applies a similar design as appeared in Eyeriss [84], which has two dimensions: Y-bus and X-bus. The vertical Y-bus interacts 17 X-buses, with 16 for the Executor and one for the Speculator. Each PE in the Executor has a reconfigurable *(row, col)* ID, and different X-buses or PEs have the same ID if they are receiving the same data. During the data transmission, each data loaded from the GLB is given a specific *(row, col)* ID. In order to correctly deliver the data to its destination, 17 Multi-cast Controllers (MC) as shown in Figure 5.1 are used to compare the *row* ID with the *row* ID of each X-buses. Another 16 MCs will match the *col* ID of the data with the destination's *col* ID tag. The unmatched X-buses and PEs are deactivated to save energy.

5.1.1 Hardware Efficient Speculator Design

Figure 5.2 shows the overall architecture of the Speculator. The design target of the Speculator is – with small area and energy consumption – to provide sufficient throughput for generating approximate results and switching maps that supply the Executor to reduce computations with negligible loss of model quality. Each index in the switching map is one-bit, indicating whether a specific neuron should use the approximate results or need to be updated later by the accurate results from the Executor. In other words, the Executor only needs to compute the output neurons with switching index equals to 1 in the switching map.

As discussed in Chapter 3 and illustrated in Figure 5.2, the dual-module processing method introduces three auxiliary steps to generate switching maps before the layer execution: the quantization step, the dimension reduction step, and the speculation step. The first two steps together make the speculation computation to be both low-dimension and low precision (INT4). The approximate results will be further passed through the

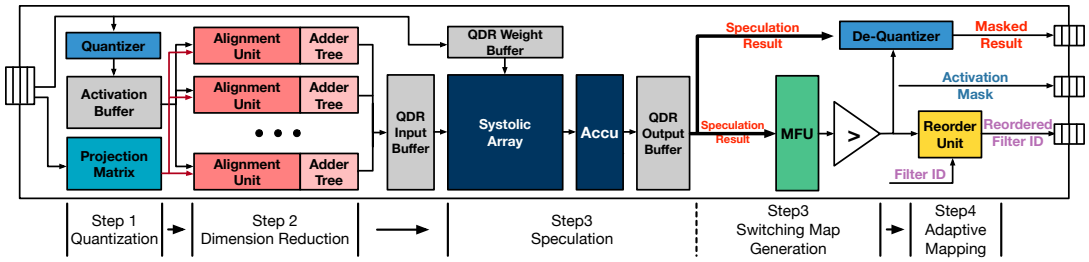


Figure 5.2: Illustration of the Speculator that runs approximate modules. The Switching Map Generator computes the dynamic switching indices for computation skipping in the Executor while balance workloads.

Multi-Function Unit (MFU) to perform non-linear activation (e.g., *ReLU*, *tanh*, and *sigmoid*) and generate final approximate results. Speculation output that falls in the sensitive region will be considered as important output and have its switching index set to one in the switching map. Finally, after generating switching maps, an optional analyzing step called adaptive mapping is added afterward to process the information. This step is essential for CNN execution to help balance the PEs' workloads (See Section 5.2.1). We can find the process of the Speculator step by step below.

Pre-Step: Data preparing. At the beginning of running the Speculator, the ternary projection matrix $\mathbf{P}$ and quantized & dimension-reduced (QDR) weights are loaded into on-chip buffers, i.e., the Projection Matrix buffer and the QDR Weight Buffer as shown in Figure 5.2.

Step 1: Quantization. The Executor's high-dimensional execution uses 16-bit fixed-point data, where the fixed-point data are essentially INT16 with a scale in FP32. Therefore, the output of the Executor will also be the same format. In order to use these results as the input activation and multiply them with the low-precision QDR weights, we need to first quantize them from INT16 to INT4 values with the Quantizer and then stash them into the Activation Buffer. The conversion from 16-bit to 4-bit is realized by simply truncating the 12 least-significant bits (LSBs) and keeping the four most-significant bits (MSBs). Accordingly, the scaling factor also needs to be increased

by 4096 (2^{12}) to maintain the same quantization range. The rounding error caused by 16b-to-4b quantization is inevitable, but such aggressive quantization applied in the Speculator has negligible impact on model quality as the values are only used in the insensitive regions.

Step 2: Dimension reduction. The dimension-reduction step multiplies the quantized input activation with the projection matrix $\mathbf{P}$ to further reduce the dimension. Since all the values in $\mathbf{P}$ come from the set $\{-1, 0, 1\}$, we can efficiently implement this step in hardware with addition and accumulation instead of using multipliers. As shown in Figure 5.2, the Alignment Units first change the signs of input activations according to the element values of $\mathbf{P}$. Then the Adder Trees perform the accumulation of sign-modified input activations. This carry-save-adder tree structure operating in pipeline provides high throughput for dimension reduction. The dimension-reduced inputs are buffered in *QDR* Input Buffer for the next step.

Step 3: Speculation computation & Switching map generation. As shown in Figure 5.2, after quantization and dimension-reduction, the inputs and weights in low dimension and low precision are stored in QDR Input Buffer and QDR Weight Buffer. We can then conduct the INT4 inner-product operations using a 16×32 Systolic Array. The Speculator uses systolic array rather than another 2D PE array to perform regular dense GEMM operation because systolic array designs achieve better energy efficiency with simple control. The size of the systolic array is searched based on design space exploration in Section 5.3.6. The results from the Systolic Array will be accumulated with the partial sum and sent to the Multi-Function Unit (MFU) to calculate the final activated output. The MFU implements different activation functions, including *ReLU*, *tanh* and *sigmoid*.

Dynamic Switching. Given the results (y') from the approximate module and accurate results (y). The final output vector – a mixture of results from approximate

and accurate modules – can be assembled by

$$y = y \odot m + y^{'} \odot (1 - m) \tag{5.1}$$

where $m \in \{0, 1\}^n$ is the switching map for determining which output activations belong to the insensitive region, and $\odot$ is point-wise multiplication. We have

$$\begin{cases} sigmoid/tanh: & if\ |y_i^{'}| > \theta_{th},\ m_i = 0;\ else\ m_i = 1 \\ ReLU: & if\ y_i^{'} < \theta_{th},\ m_i = 0;\ else\ m_i = 1 \end{cases} \tag{5.2}$$

where θ_{th} is the threshold can be obtained by tuning with the training or validation set to reach targeted saving.

Equation (5.2) represents how to generate the switching map, and we further compare these approximate results with the predetermined thresholds from the fine-tuning phase. All the values that fall within the sensitive region will have their switching indices to be one.

Step 4: Adaptive Mapping. As mentioned above, with the switching maps and speculation results, the Executor only needs to compute a small portion of accurate activations. However, processing with sparsity information causes different PEs to have imbalanced workloads for CNNs [90, 100]. To tackle this problem, we further propose the **Adaptive Mapping** strategy to change the order of the computation between different output channels so that output tiles with similar workloads are computed together. Such reordering is very hardware efficient that can be handled directly inside the Speculator using a dedicated Reordering Unit. Section 5.2.1 will further demonstrate the proposed approach. As for RNNs, the designed dataflow guarantees that there is no workload imbalance between different PEs, and we can bypass the reordering operation. How-

ever, we do need to store the results for approximate activations; these 4-bit results are dequantized to 16-bit data with the Speculator's dequantizer and sent to GLB.

To sum up, the Speculator produces three types of information: firstly, the switching maps indicating which output can be skipped by the Executor; secondly, the reordered filter ID that the Executor follows when computing the output feature map; thirdly, the approximate activations required by RNN models. With decoupled Speculator and Executor design and the fine-grained pipeline between these two components, we can hide the latency of speculation as much as possible to increase the overall performance. The hardware efficient quantization and dimension reduction also greatly reduces the area and energy overhead to process approximate modules.

5.1.2 Specialized Executor Design

The Executor processes accurate modules using 16-bit fixed-point arithmetic. Overall, the Executor applies a typical spatial 2D PE architecture to explore different data reuse types. Furthermore, we add specialized hardware components inside each PE to leverage the dynamic switching maps to reduce computation and memory access. As shown in Figure 5.3, each PE consists of dedicated buffers for different types of data, a 16-bit pipelined multiplier and adder, and a local multiply-accumulate micro-instruction lookup table (MAC Instruction LUT) which is the key of skipping unnecessary computations.

On the PE level, we can regard processing DNNs as orchestrating and executing multiple MAC operations. Each MAC operation requires two loads for input and weight values, one load for partial sum, and one store for the updated partial sum. As illustrated in Figure 5.3, the MAC operations are represented with micro-instructions (μinst) that stored in the MAC Instruction LUT. Each μinst contains the input activation (IA) index, weight (W) index, and output activation (OA) index indicating the address of the

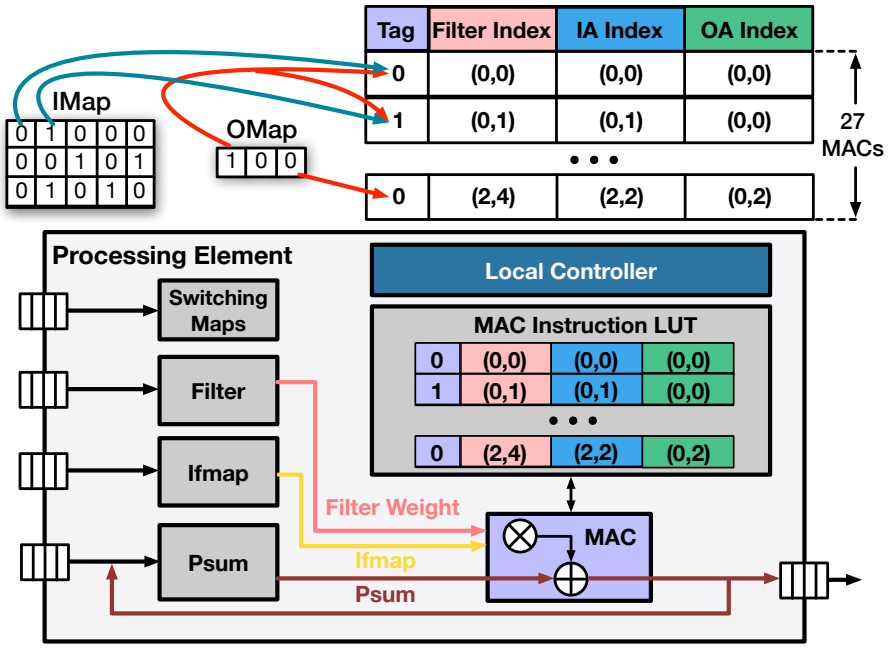


Figure 5.3: Specialized Processing Element that handles dynamic switching to skip computations. MAC operations are stored into a dedicated instruction Lookup Table. OMap and IMap are the switching maps which help to assign validation Tags to each MAC instruction. Instructions with Tag value to be 0 will be skipped.

load/store operations. Besides, an extra tag-bit is added to enable computation saving. MAC operations with tag bit being zero will be directly skipped. We only use the indices to locate the relative positions of the values in the input/weight/output tile. As long as the tile's shape is kept the same, these indices do not change. For a given NN layer, the PEs are processing a static shape of input, weight, or output tile. Therefore, the μinst's indices only need to be generated once at the beginning of layer configuration, and remain unchanged and shared by all the PEs throughout the execution of the whole layer. The dynamic switching maps will be used to configure the tag bits for different tiles to enable computation skipping.

We can further use a simple example of CNN to demonstrate how the PE utilizes switching maps to skip unnecessary computations. As for RNNs, the case is even simpler. In the example shown in Figure 5.3, every step the PE is processing a $3 \times 5 \times 1$ input tile,

and a $3 \times 3 \times 1$ filter tile to generate a $1 \times 3 \times 1$ psum in the ofmap. Therefore, each PE is mapped with $3 \times 3 \times 3 = 27$ MAC operations for each step. To configure the tag-bit, the Executor loads the corresponding IMap and OMap from GLB and stored them in PE's local buffer. The OMap shows whether an output activation needs to be computed by the executor, and IMap shows whether the input activation is zero or not. We only mentioned output switching maps (OMap) in the previous content, but for CNNs, the ineffectual neurons are set to zero, making the OMap become the input sparsity maps (IMap) for the **next** layer. In this way, we only need to pay the overhead of dynamic switching once, but the switching map is used twice for the current layer's OMap and the next layer's IMap. Besides, if a predicted effectual neuron turns out to be ineffectual after $ReLU$, we will update the switching index of that neuron from 1 to 0 and then send it to the GLB. With this **correction** step, when the OMap is loaded as IMap for next layer, it will have even higher sparsity to save more computations.

Based on the switching maps, we can easily decide whether a specific instruction can be skipped or not. For example, as shown in Figure 5.3, the OMap shows that only the first element in the $1 \times 3 \times 1$ output tile needs to be computed. Therefore, all the MAC operations related to the other two output neurons can be discarded, leaving only nine necessary MAC operations. Moreover, since the IMap shows that 2/3 of the input activations are zero, we can further reduce six MAC operations. Finally, each MAC operation is augmented with a 1-bit tag using simple Boolean logic. PE's local control will locate the valid MAC operation in the instruction buffer to perform necessary computation.

5.2 Dataflow and Mapping

DUET is able to support both CNNs and RNNs with a unified architecture. The efficient hardware design presented in Section 5.1 guarantees that different units can well handle their required tasks. However, to achieve the expected performance speedup and energy saving, we still need to have fine-grained dataflow design and hardware mappings. Essentially, we decouple the Speculator with the Executor so that they can run in parallel and let switching maps to be generated prior to run the Executor. Nevertheless, the data dependencies between the current layer's output and the next layer's run-ahead approximate module execution makes it challenging to orchestrate the execution dataflow of the Executor and the Speculator in pipeline. Therefore, this Section separately describes how DUET handles CNN models and RNN models while addressing different bottlenecks during the run-time.

5.2.1 Processing CNNs with Balanced Execution

DUET computes a CNN model layer by layer. The accelerator is configured once for each layer to sequentially process batches of ifmap. The configuration bits are generated offline based on the layer structure and hardware constraints. During runtime, they are loaded as a long scan chain to configure the accelerator to process a layer in a certain tiling shape and set up the mappings for the Executor and the Speculator.

Overall Pipeline For illustrative purposes, suppose we have a CNN layer with a $5 \times 5 \times 3$ ifmap and six $3 \times 3 \times 3$ filters, as shown in Figure 5.4(a). Therefore, the ofmap would be of size $3 \times 3 \times 6$, assuming no padding. In this example, the Executor consists of $3 \times 3 = 9$ PEs. Each line of PEs will together computes a specific channel of the ofmap. Within the same PE line, different PE loads different tiles of ifmap and filter

that contribute to the same area in a specific ofmap channel. Thus, the output partial sum will be horizontally accumulated. Since different lines of PEs compute different output channels, input feature maps are shared vertically within the same column of PEs.

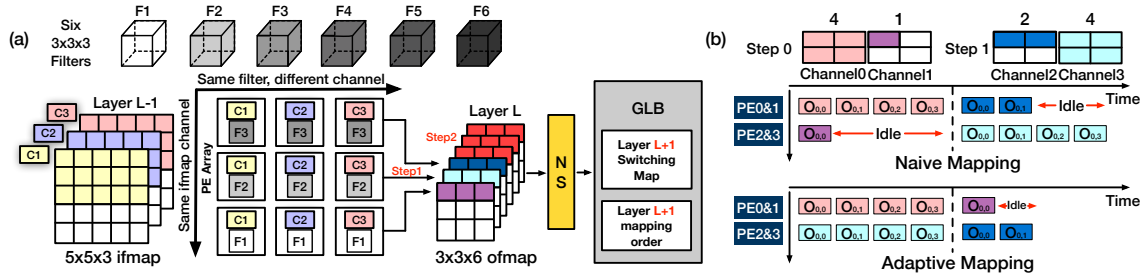


Figure 5.4: Illustration of (a) CNN Dataflow and (b) Adaptive Mapping Strategy. Ifmap data are shared vertically in Executor, and each PE-row computes for a specific ofmap channel. The Speculator uses the Executor's output to perform output speculation for the next layer. After generating switching maps, the Speculator directly performs adaptive mapping to balance PE's workload before execution.

During the execution, the ofmap is computed step by step. Each step the PE array generates a $1 \times 3 \times 3$ output tile, each line generates a $1 \times 3 \times 1$ tile. Therefore, it takes 6 steps to finish computing the ofmap. After finishing a step, we first move along the channel dimension of the ofmap to compute the next tile. In this example, after the first tile is generated, we then compute the **red** tile as shown on the ofmap in Figure 5.4(a). In this way, when these two tiles are sent to the Speculator as input activations to perform speculation, the speculation results using tile1 and tile2 can be further accumulated together. Otherwise, if tile1 and tile2 are different areas in the same channel, the speculation output of tile1 and tile2 will be also at different areas that cannot be accumulated. This will increase the memory footprint of the speculation.

Therefore, the Executor computes each layer's output tile by tile, while the Speculator uses computed tiles to perform sparsity speculation for the **next** layer. In this way, we can pipeline the speculation with execution, hiding the latency of speculation while lowering

the memory overhead. In the example shown in Figure 5.4, while the Executor is still computing layer **L**'s output, the Speculator is already using existing results to generate switching maps for layer **L+1**. Therefore, when the accelerator start to process layer **L+1**, it already has the OMap to be used to skip a considerable amount of unnecessary computations. Also, as mentioned above, the OMap of the previous layer will be updated by the Executor and serve as the IMap for the next layer to further reduce computations.

Adaptive Mapping for Balanced Execution The key challenge of computation skipping in CNNs is the workload imbalance caused by irregular sparsity distribution, which further results in PE under utilization and performance degradation. Specifically, different PEs can have different numbers of necessary MAC operations in the same computation step. Therefore, PEs with less computations will finish earlier and have to wait for the other PEs.

Depending on the sparsity type, there have been several ways to balance the workloads. Prior work like [88, 90]focus on input and weight sparsity. Since the weight sparsity is static and generated offline during the training phase, they adopt offline sorting based on the weight density to reorder the computation sequence prior to the inference. This idea is not suitable for dynamic output switching (OS), as the switching map is generated dynamically during run-time. Online sorting will incur longer latency and energy consumption.

Other work [101, 100] target on output sparsity. These work are based on a coupled executor/predictor design with *early termination* mechanism. Specifically, the prediction is indeed part of the execution process. If the prediction results indicate the output to be zero, the execution will be stopped here. Otherwise it will be completed. To tackle the workload imbalance caused by output sparsity, they mainly use two approaches. The first idea is to enable asynchronous PE execution, so that whenever a PE finishes its current

computation, it can initiate a new computation step. However, this approach causes significantly design and energy overhead due to extra memory access and complicated NoC/buffer design. The second idea is to apply an empirical approach by expecting the computations to be evened out along a certain input dimension. For instance in *Predict*[100], the authors claim that the summation of the computations with the same coordinate across different ofmap channels is nearly the same. However, in order to achieve this balance, they need to increase the tile size of each computation step, which requires larger local buffer and memory footprint. We believe these approaches are sub-optimal and energy inefficient, and the reason is because their dataflow is based on single-module architecture which cannot generate the switching maps prior to the execution.

DUET solves the imbalance issue by the decoupled speculation/execution and hardware efficient dynamic **adaptive mapping**. The decoupled design ensures the switching maps to be generated prior to its execution, which also gives us extra time to perform online sorting using the proposed adaptive mapping. A dedicated Reorder Unit is designed in the hardware to reduce the mapping latency.

Figure 5.4(b) demonstrates the proposed approach. Suppose in each step, we can generate two 2×2 switching maps corresponded to two ofmap channels. Besides, the Executor has 4 PEs that are organized in a 2×2 square. In the normal dense case, output channel 0 and 2 will be mapped to row 0 (PE0&1), and output channel 1 and 3 will be mapped to row 1 (PE2&3). Therefore, Channel 0 and 1 will be later processed at the same time, while channel 2 and 3 are computed together. However, due to output sparsity, the workload of these channel are unbalanced as shown in the figure. Thus, such naive mapping strategy will cause different row of PEs to be under-utilized within the same step. In this case, a better computation sequence would be to compute 0 and 3 first, followed by channel $1, 2$.

Therefore, DUET achieves more balanced PE workloads by changing the order of

computing different ofmap channels. Since the switching maps are generated in advance, we can examine the total workloads for different output channels within several tiles. Based on these numbers, we can further group the output channels that have comparable operations together. After this, a new sequence of ofmap channel is generated. Later when the accelerator processes next layer, the Executor will load filter weights according to this new sequence to have balanced computation time.

Note that, adaptive mapping only changes the order of computing the ofmap channels. In other words, it only affects the sequence of loading the filter data, while the ifmap access and data reuse pattern are not influenced. Also, the output are sequentially stored in the GLB according to their original sequence. For example, even though Channel **0,3** are computed together at first, and Channel **1,2** comes later, in the GLB they are still stored as Channel **0,1,2,3**. In this way we can keep the ID unchanged for the next layer when loading the ifmap data.

The adaptive mapping only considers the imbalance issue caused by output sparsity so that different rows of PEs are balanced. Inside each row, there will still be imbalance within the PEs due to input sparsity. However, we can observe in the experiments that, it is negligible compared with output imbalance. Besides, further enabling more complicated mapping and reordering strategy will considerably increase the sorting overhead.

Architecture Support for Adaptive Mapping In DUET design, the adaptive mapping is done inside the Speculator's Reorder Unit. As tiles of switching maps are sequentially generated, we send them to the Reorder Unit in addition to writing them back to the GLB. The design of the Reorder Unit is shown in Figure 5.5, it consists of multiple 1-bit adder trees that will sum up all the switching indices corresponded to a specific output channel. Each output channel will have a total count of estimated computations. Note that, this number does not represent the workloads for the whole channel, but for

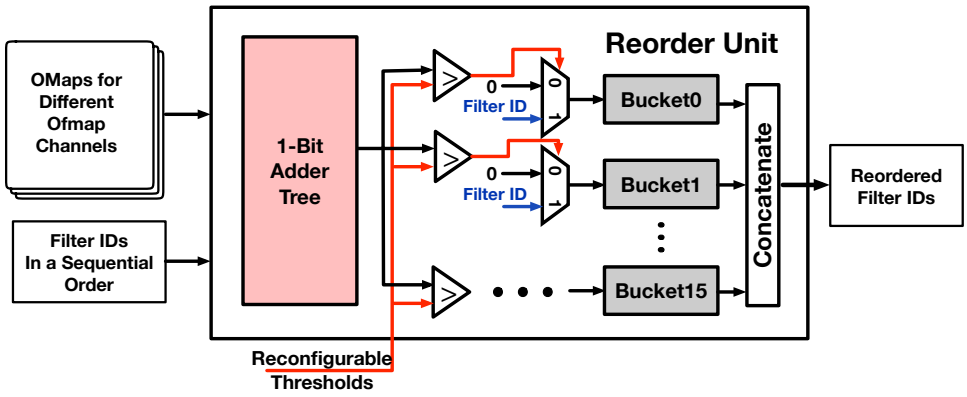


Figure 5.5: Reorder Unit implements adaptive mapping for balanced execution. The output switching maps for different channels are summed up separately and divided into several groups. The IDs of different filters with similar workloads are stored together in a same bucket and sent out together.

the tile that will be processed within one computation step. Then, we compare the sum with preset interval thresholds and write the channel IDs to the corresponding buffers, i.e., Buckets shown in Figure 5.5.

Using the same example in Figure 5.4(b), each of the four output channels will have a sum indicating its computation quantity. In this case, the sums are $4, 1, 2, 4$ for channel $0, 1, 2, 3$. Since there are two PE lines in the Executor, there will be two Buckets in the Reorder Unit. In this case, the channel that has more than two valid output elements will be stored in Bucket0, and others that contain less valid output will be stored in Bucket1. To do so, we only need to compare the four sums with a preset threshold 2, and channel ID $0, 3$ are grouped together, while ID $1, 2$ will be stored in the second Bucket. During execution, the Executor will load the filter data in the order of Bucket0, Bucket1, which gives us the optimal computation sequence as demonstrate above.

5.2.2 Reducing Memory Accesses and Computations on RNNs

Finally, we can see how DUET handles RNN models. Compared with CNN models, processing RNNs is more memory-bound. For instance, given an input vector of length

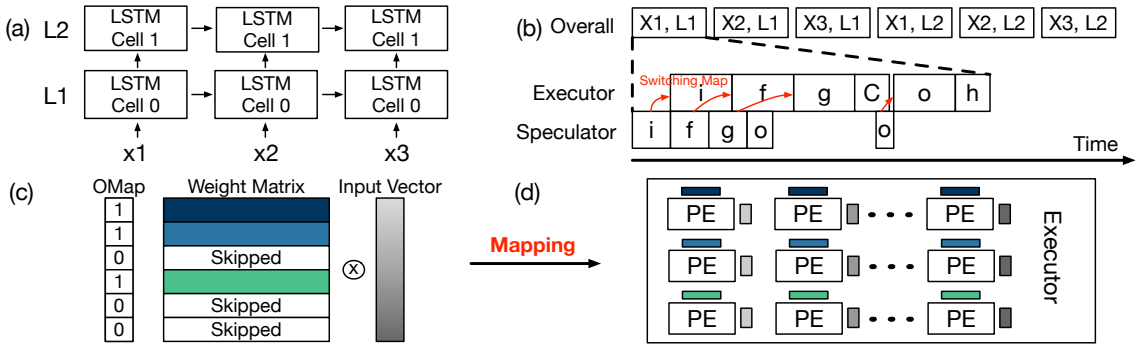


Figure 5.6: Illustration of (a) sample LSTM network, (b) RNN Dataflow and (c)&(d) Executing GEMV with reduced computations and memory access. We apply a gate level speculation/execution pipeline to hide the speculation latency. Each PE-row in the Executor is mapped with a dot product between a row of weight matrix and the input vector. Each PE-row will finally generate a single output value. Therefore, if the switching index is zero, we can directly skip a complete dot-product operation.

1024, the weight matrix used for computing each gate in an RNN cell would be 1024×1024, which requires a 2MB of memory space. Furthermore, although the weights are shared between different input elements, the recurrent data dependency requires us to compute the previous hidden state before we can process the next one. Therefore, during the execution, we have to constantly and cyclically load each weight matrix from the off-chip DRAM. This motivates us to focus on reducing the off-chip memory access with the proposed mechanism, while keeping the dataflow simple enough to ease the control overhead and avoid workload imbalance.

Overall Dataflow For illustrative purpose, we consider an LSTM network with two recurrent LSTM Cells L_1, L_2 and an input sequence with 3 elements x_1, x_2, x_3, as shown in Figure 5.6(a). The inference is executed element by element and then layer by layer. To be more specific, as demonstrated by Figure 5.6(b), for the first element x_1, the weights of L_1 are fetched from DRAM. Due to the limited on-chip memory capacity, each time we can only load part of the weight matrix corresponded to a specific gate. After the first hidden state of L_1 is computed, we use the results and x_2 to compute the second hidden

state. To do so, we need to reload L_1's weight from DRAM. Finally, after all three input elements are passed through L_1, the same process is repeated for L_2.

Inside each layer, the computation is also handled in a sequential pattern. Using the same example in Figure 5.6(b), suppose we are passing x_1 through L_1. We can first compute the input gate i and then the forget gate f, followed by the update gate g to compute cell state C, and finally, we compute the output gate o to get the hidden state h. The computation is mainly matrix-vector multiplication and vector accumulation and activation functions.

Gate-level Dual-Module Pipeline With the baseline dataflow, we further introduce how to utilize the Speculator to perform speculations for RNN models and hide the speculation latency. In DUET, we speculate the output of each gate before its execution. As illustrated by Figure 5.6(b), we start with x_1 and L_1 and perform quantization and dimension reduction for the input gate i. Similar to CNN, this will give us a binary switching map indicating the important neurons that need to be updated with accurate results from the Executor. What's different is that, apart from the switching maps, we also store the approximated results for those *ineffectual* output neurons in the GLB. After the sparse high-precision computation for gate i is finished in the Executor, we add the two vectors together. As a result, in the final output vector, the approximate activations are generated by the Speculator, while the effectual activations are computed in the Executor. With the switching maps, for the weight matrix and bias vector, only the rows related to the accurate output activations need to be fetched from DRAM. Our approach saves memory accesses and computations.

During the Executor's execution of input gate i, the Speculator can start the speculation for the forget gate f, since we only need x_1 and h_0 as our input to perform the speculation. Similarly, the speculation of the other gates can also be hidden with the

Executor's computation. Throughout the processing of each layer, only the speculation for input gate i cannot be hidden due to data dependencies.

Reducing Off-chip Memory Access and On-chip Computations with OMap As shown by Figure 5.6(c)&(d), each PE line in the Executor is mapped with a specific row of the weight matrix to be multiplied with the input vector to generate a single output value. Therefore, if the switching map indicates that a specific output neuron is ineffectual, then we can completely skip the computations related to this neuron during the execution. More importantly, there is no need to load the corresponding row in the weight matrix from the DRAM. Saving weight data access is especially crucial as the profiling results show that off-chip memory access greatly influences the overall performance and energy consumption. With the proposed dual-module processing mechanism, we can directly reduce the amount of data to be loaded from DRAM to GLB and from GLB to Executor. The overall computation complexity and processing time are also reduced, further boosting the performance of executing RNN models.

5.3 Evaluation

This section presents the evaluation of the dual-module processing algorithm and the supporting accelerator architecture of DUET co-design for improved inference efficiency of DNNs. At the algorithm level, we present the trade-off between inference quality and efficiency in terms of FLOPs reduction and data access reduction. At the architecture level, we demonstrate the improved efficiency with DUET and compare DUET with state-of-the-art DNN accelerators with computation skipping.

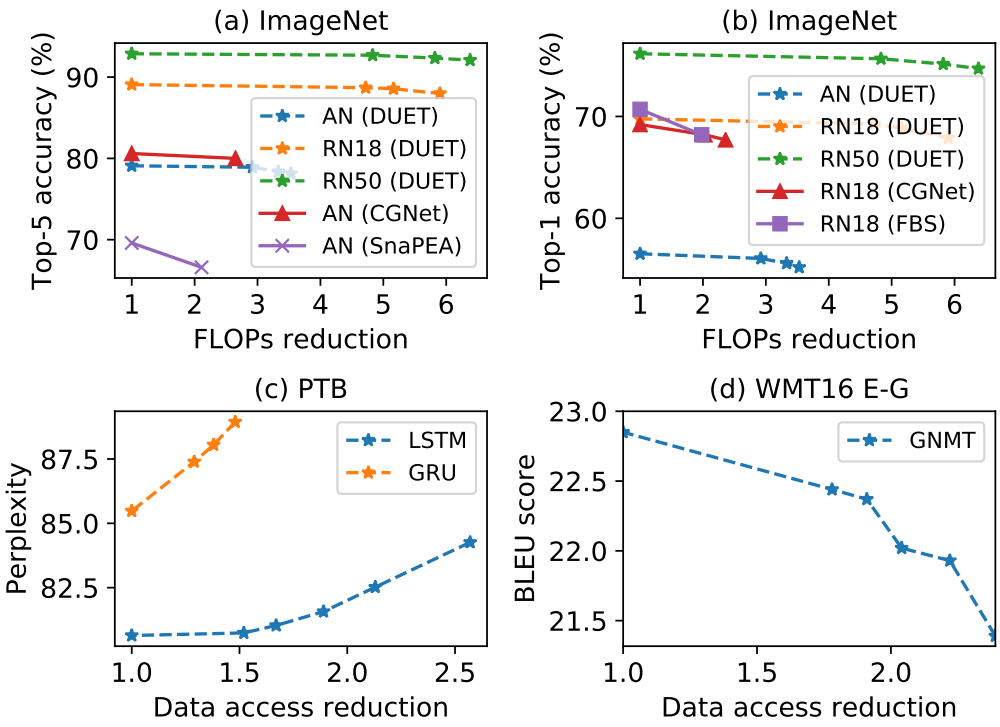


Figure 5.7: Model inference quality vs. savings. Dual-module processing of DUET achieves better quality and saving trade-off and supports a wide range of DNN models.

5.3.1 Algorithm Evaluation

Benchmarks. The evaluation uses a set of machine learning tasks: AlexNet, ResNet18, and ResNet50 to perform image classification on the ImageNet dataset; RNN-based models on language modeling with LSTM and GRU using the PTB dataset and machine translation with GNMT using the WMT16 en-de dataset.

Quality and Efficiency trade-off. The dual-module processing method provides a trade-off model inference quality with improved efficiency. With acceptable quality degradation, DUET can boost DNN execution efficiency that could be critical in latency and energy-constrained application scenarios. Figure 5.7 (a) and (b) show the FLOPs reduction at different levels of accuracy loss in both top-1 and top-5. With 1% top-1 accuracy loss according to MLPerf, the method can reduce operations by 3.33x and 5.15x

using AlexNet and ResNet18, respectively. Comparing with prior methods computation skipping in CNNs, namely SnaPEA [101], FBS [54], and CGNet [55], the proposed approach can achieve better quality and operation reduction trade-off.

As shown in Figure 5.7(c), the method can reduce off-chip weight data access by 1.89x while achieving negligible quality degradation with one perplexity increase from baseline. Similarly, as shown in Figure 5.7(d), serving online translation with GNMT is latency-sensitive because of the memory-bound nature of accessing weights of different LSTM layers at each time-step. With unnoticeable translation quality by users such as one BLEU score loss, our method can achieve 2.22x reduction on off-chip weight data access of the four-layer decoder in GNMT.

5.3.2 Architecture Evaluation Methodology

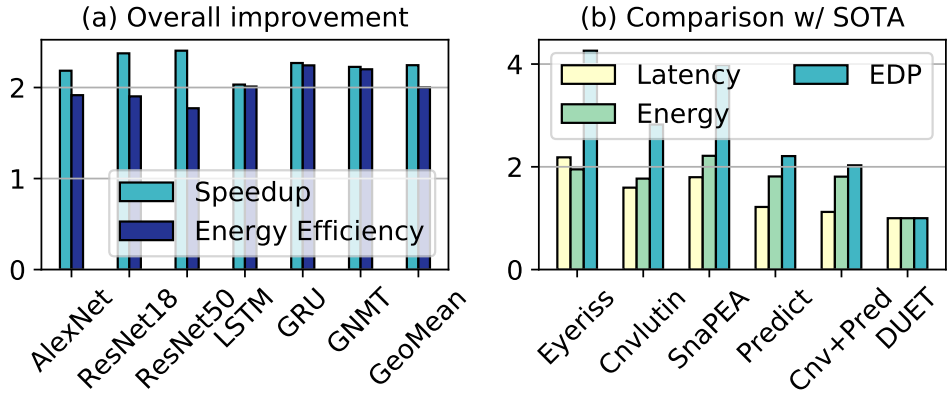


Figure 5.8: (a) Overall performance speedup and energy efficiency; (b) Comparison with other accelerators.

The evaluation of DUET design uses a cycle-level simulator based on the architecture as in the state-of-the-art CNN accelerator [84]. Because dual-module processing is data-dependent, we adopt a hybrid simulation methodology: we integrate the cycle-level simulator with a deep learning framework, i.e., PyTorch. The input data to the simulator,

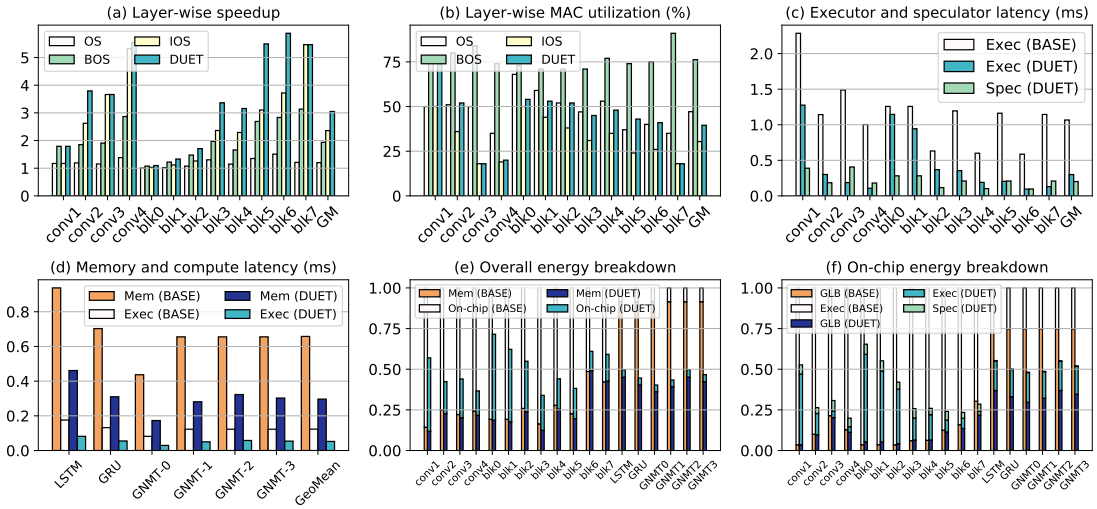


Figure 5.9: Breakdown analysis for DUET. (a) Layer-wise speedup improvement applying different techniques in DUET. (b) Layer-wise MAC utilization improvement. (c) Latency comparison between the Executor, the Speculator, and the baseline single Executor design. (d) Memory and compute latency for RNN models. (e) Overall energy breakdown (w/ off-chip memory access). (f) On-chip energy breakdown (w/o off-chip memory access).

including input activations, model parameters for Executor, and Speculator parameters, are all extracted from PyTorch. We use results from RTL synthesis by DesignCompiler under 45nm technology for DUET control logic. We use CACTI and Micron Power Calculators for SRAM and DRAM estimation. Table 5.1 lists the area of major components in DUET design. The primary area consumption comes from the on-chip memory buffers, while the Executor accounts for 40.0% of the total chip area, and the Speculator only accounts for 6.6% of the area.

Comparison baselines. Firstly, single-module architecture with only the Executor is the baseline architecture. DUET is compared with state-of-the-art accelerators in terms of performance, energy consumption, and energy-delay-product (EDP), when running the same workloads. Specifically, the developed simulator is extended and validated to support Eyeriss[84], Cnvlutin[95], SnaPEA[101], and Prediction[100]. These architectures span over dataflow optimization and computation skipping of sparse activations, which

sufficiently supports the evaluation of DUET. As illustrated here, neuron sparsity causes imbalanced execution and the proposed adaptive mapping can mitigate it. The proposed dual-module architecture design outperforms other sparsity-oriented designs.

5.3.3 Performance speedup analysis

Overall speedup. Figure 5.8(a) shows the overall speedup. Compared with the single-module baseline design, DUET achieves 2.24x average speedup on typical CNN and RNN models. The performance improvements mainly come from three aspects: firstly, total computations are reduced with dual-module processing; secondly, hardware-efficient adaptive mapping ensures balanced execution and high PE utilization in the Executor; finally, advanced switching map generation greatly reduced off-chip memory access for memory-bound workloads.

Layer-wise breakdown for different techniques. To further give more insights to the evaluation results, we provide a layer-wise breakdown for the CONV layers of AlexNet and ResNet18, and we show the effectiveness of our schemes in four stages. The comparison baseline is to only use the Executor with the same mapping and basic dataflow. As shown in Figure 5.9(a), skipping computations given output switching map (OS) without adaptive mapping can only obtain 1.20x speedup on average due to imbalanced execution. Enhanced by adaptive mapping, i.e., Balanced Output Switching (BOS), the performance speedup can achieve 1.93x. Moreover, with integrated input and output switching maps (IOS), the performance can be boosted to 2.36x as more computations can be skipped. Finally, the integrated input and output switching maps (DUET) design with adaptive mapping can achieve a 3.05x average speedup.

Figure 5.9(b) uses the layer-wise MAC utilization of CONV layers, from AlexNet and VGG16, as the metrics to evaluate execution efficiency. For example, CONV5 in AlexNet

has 65.5% computation sparsity when using OS, which could have 2.9x speedup, yet only achieves 1.36x. The gap between actual speedup and theoretical computation reduction indicates the severe imbalanced execution caused by coupled OS speculation. The average MAC utilization of OS only is less than 50%, again, due to imbalanced execution. While integrating the input sparsity with output sparsity (IOS) could have more computation reduction than using OS only, PEs are more under-utilized – on average, 30% in Figure 5.9(b). We can observe higher speedups on CONV layers with more channels. DUET can have more balanced execution with output switching and adaptive mapping. Compared with imbalanced OS, the average utilization of balanced OS can be improved from 47% to 76%; the average performance speedup is increased from 1.20x to 1.93x. Similarly, IOS boosted with our adaptive mapping (*DUET*), the average MAC utilization and the speedup of CONV layers increase from 30% to 39% and from 2.36 to 3.05x, respectively.

As Executor and Speculator compute for critical neurons and trivial neurons, respectively, to deliver final activated results, balancing the processing time of Executor and Speculator is critical to achieving better performance rather than having the Speculator become the new bottleneck. The latency results of Executor and Speculator are shown in Figure 5.9(c). Compared with baseline Executor without computation skipping enabled by the dynamic switching from Speculator, DUET can reduce Executor average latency from 1.06 ms to 0.29 ms with dynamic switching and adaptive mapping. On average, Speculator latency is 0.20 ms that can be hidden with the latency of Executor with pipelined processing.

For memory-bound RNN layers, we focus on the latency of off-chip memory access and on-chip computations. As shown in Figure 5.9(d), BASE processing is severely bounded by accessing weight data from off-chip memory. Enabled by dynamic switching in DUET, the off-chip weight data accessing latency is reduced to 0.30 ms from 0.65 ms.

5.3.4 Energy efficiency analysis

Overall energy savings. As shown in Figure 5.8(a), DUET can achieve 1.95x energy saving on average using Executor-Speculator dual-module processing compared with baseline single-module processing. The energy saving is achieved by cutting on-chip computations and buffer access as well as cutting off-chip data access. Specifically, for Executor, the computations and local buffer access are greatly reduced by taking advantage of the prepared switching maps. For the Speculator, although we need to load QDR weights and store the computed approximate data, we make sure the computations are low-dimension and low-precision, which keeps the cost of these extra memory accesses as low as possible. Besides, the approximated results for the insensitive activations are reused to save energy consumption further.

Energy Breakdown. To further interpret the energy efficiency of DUET, Figure 5.9(e) and (f) show the layer-wise energy breakdown. For compute-bound layers such as CONV layers, the energy saving benefits are mostly from the reduction of MAC computations and local buffer accesses in the Executor. For memory-bound layers such as RNNs, reducing weight data accesses from off-chip memory enabled by DUET helps energy efficiency. These results support the above analysis and demonstrated our initial optimization targets. Figure 5.9(f) shows the on-chip energy breakdown. The Speculator's energy consumption only consumes a small portion, ranging from 3.5% to 6.3% for CONV layers and less than 1% for RNNs compared with the total on-chip energy of baseline.

5.3.5 Comparison with SOTA CNN Accelerators

A special case of dua-module processing is ReLU-based output sparsity prediction typically appeared in CNNs. While the focus of DUET is supporting computation and

Table 5.1: Area breakdown.

Units	Parameters	Area (mm^2)
Global Buffer	1MB	5.655
Executor	256 (16-bit)	4.236
Speculator		0.699
Align. & Adder Trees	4096 (4-bit)	0.378
Systolic Array	16x32 (4-bit)	0.318
Activation Buffer	8KB	0.023
Projection Buffer	32KB	0.090
QDR Input Buffer	4KB	0.011
QDR Weight Buffer	128KB	0.352
QDR Output Buffer	4KB	0.011
Multi-Func. Unit	16 (4-bit)	0.034
Reorder Unit	16 Buckets (32B)	0.014

memory accesses reduction in general DNN models, we compare with state-of-the-art CNN accelerators, i.e., *Eyeriss*[84], *Cnvlutin* [95], *SnaPEA* [101], and *Predict* [100], with computation skipping to demonstrate the benefits of DUET's architecture design choices. All designs are scaled to have the same number of MACs and similar on-chip memory size, and all resul are normalized to DUET, as shown in Figure 5.8(b). DUET achieves better results in terms of latency, energy, and energy-delay-product (EDP).

Performance comparison. Performance-wise speaking, Eyeriss equals a dense baseline as it only supports power-gating to save energy but computation skipping to improve performance; thus, it has the worst latency among others. Equipped with either input sparsity detection or output sparsity prediction mechanisms, Cnvlutin, SnaPEA, and Predict can reduce processing latency from computation skipping, the performance improvements are limited by only single-source computation skipping from either input or output. The workload imbalance caused by irregular sparse activations as in Cnvlutin and SnaPEA compromises the performance.

Energy efficiency comparison. *Cnvlutin*, *SnaPEA*, and *Prediction* use only one

level of on-chip buffer and have no local data reuse, thus those designs consume 1.77x, 2.21x, and 2.21x more energy than DUET, as shown in Figure 5.8(b). Even though those three designs support computation skipping, the energy consumption is at the same level as Eyeriss. Since buffer accessing is the major source of on-chip energy consumption [84], accelerators without data reuse at the local buffer level would inevitably consume much more energy to access global buffer. DUET uses the same two-level on-chip memory hierarchy as in Eyeriss with local data reuse to improve energy efficiency.

Energy-delay-product comparison. To better compare different architectures, Figure 5.8(b) show the comparison on energy-delay-produce (EDP). The EDP of *SnaPEA* and *Predict* are 3.98x and 2.21x more than DUET, respectively. While other designs with computation skipping from both input and output activations, i.e., *Predict+Cnvlutin*, can achieve comparable performance, DUET demonstrates 1.81x and 2.03x better in energy efficiency and EDP, respectively.

5.3.6 Design Space Exploration

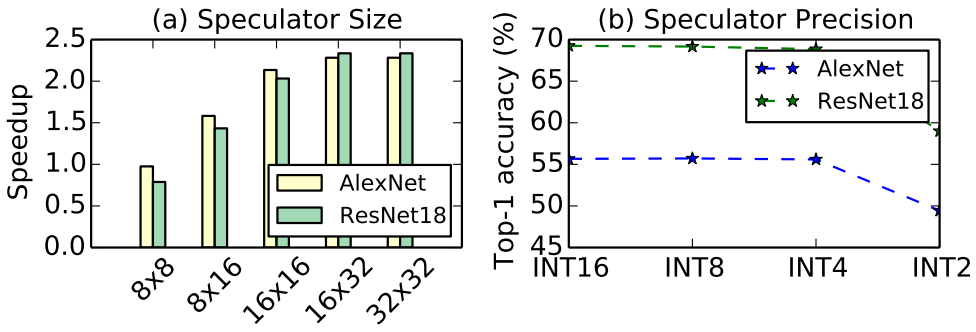


Figure 5.10: Design space exploration of (a) Speculator size and (b) Speculator Precision.

Speculator Size. Here we investigate the impact on performance when choosing different sizes of the Speculator while fixing the size of the Executor. Specifically, when

modifying the systolic array size, other components in the Speculator are scaled accordingly. The two benchmarks used are AlexNet and ResNet18. As shown in Figure 5.10(a), when the Speculator is small, e.g. 8x8, 8x16, the performance improvements are sub-optimal. This is because the Speculator cannot provide sufficient throughput to support the Executor, processing of Speculator becomes the performance bottleneck. Besides, when increasing the size of Speculator to 32x32, the performance merely improves, meaning the latency of the Speculator is already hidden by the Executor. We need to increase the size of Executor if we want to have more speedups. Therefore, the chose systolic array size is 16x32.

Speculator Precision. We also study how compute precision affects the approximation quality of the Speculator, which helps us decide the trade-off between hardware consumption and model accuracy. With the same benchmarks, as shown in Figure 5.10, INT4 is a preferred precision with negligible accuracy loss indicating good approximation quality. With 4-bit data representation and computation, we are able to achieve area and energy-efficient approximation, as demonstrated above.

5.4 Conclusion

This chapter presents an algorithm-architecture co-design to boost the execution efficiency of DNNs. Firstly, as discussed in Chapter 3, the dual-module processing uses lightweight approximate modules to compute insensitive activations and seeks to accurate modules to compute sensitive activations with skipped computations and data accesses. Secondly, the DUET design with specialized and decoupled Executor and Speculator supports balanced execution and memory accesses reduction. Compared with standard single-module processing, DUET can achieve 2.24x performance speedup and 1.97x energy efficiency improvement.

Chapter 6

Near-Memory Classification

Extreme classification is the essential component of large-scale ML models for a wide range of application domains, including image recognition, language modeling, and product recommendation. This Chapter presents a study on the memory-intensive problem of extreme classification and demonstrates that *elastic processing* in the form of candidate-level redundancy can greatly improve the performance on near-memory architectures.

6.1 Introduction

Recent advances in many machine intelligence areas, such as natural language processing (NLP) [152, 1, 153], image recognition [5, 7, 154], and recommendation[155, 156, 6], involve tackling the extreme classification problem, where classification category size is extreme large. For example, in the NLP domain, making predictions is basically classifying the words with high probabilities. Similarly, for image recognition tasks and recommendation tasks, the features generated from hidden neural network layers need to go through the classification layer to output predictions. As shown in Figure 6.1, extreme classification is the essential component to deal with large-scale problems.

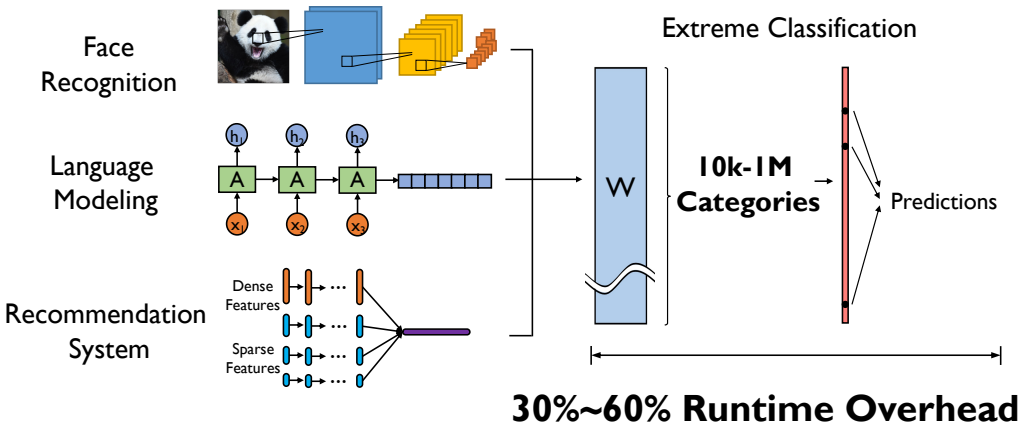


Figure 6.1: Extreme Classification serves as the common component of large-scale Deep Learning applications. The classifier processes with hidden representations from application-specific hidden layers and generates predictions as used in recognition, language, and recommendation.

As classification categories keep scaling in real-world applications, the classifier's parameters could reach hundreds of gigabytes, far beyond the on-chip memory capacity. For large-scale NLP models, the vocabulary sizes are in the range of hundreds of thousands, contributing hundreds of megabytes data [1, 4]. For recommendation systems, using commodity datasets to solve industry-level problems would require classification on the scale of 100M categories [7, 5], consuming around 190GB memory.

Due to the large memory footprint of extreme classification, accessing system memory for the classifier's weight data becomes the bottleneck of system performance. This work characterizes the state-of-the-art Transformer-based language model [157] and shows that the final classification layer consumes 50% of overall model inference time. While GPUs and specialized accelerators can boost the performance of DNN layers [9, 84], they suffer from inter-device data movements when executing the memory-intensive classification layer, as the memory usage exceeds device memory capacity.

Emerging Near-Memory Processing (NMP) technologies [25, 24] have the potential to address the memory-bound problem of extreme classification. However, naive NMP

designs cannot support the computational complexity of full classification due to the area and power limitations. Even the classifier weight data are stored and processed near-memory, the low operational intensity of linear transformation, which is basically matrix-vector multiplication, is still causing performance degradation when accessing weight data from DRAM modules.

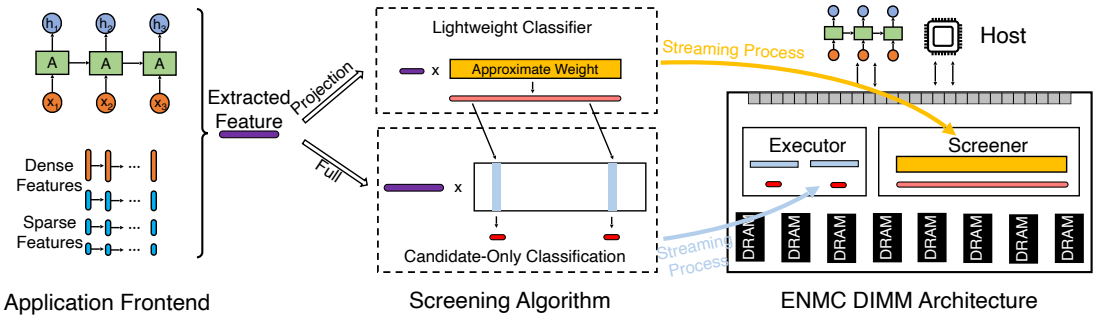


Figure 6.2: The overview of our Approximate Screening algorithm and NMP architecture co-design. Instead of full classification, ENMC co-design essentially performs candidates-only classification, where the candidates are based on the screening method. ENMC's NMP architecture design features a Screener and an Executor to collaboratively process candidates-only classification.

Therefore, this chapter presents the first end-to-end solution to address the memory-bound problem of extreme classification with NMP architecture. Figure 6.2 gives an overview of the proposed software and hardware co-design. The proposed approximate screening algorithm directly reduces the required computations and data accesses involved in linear transformations. As demonstrated in Figure 6.2, given the extracted feature vectors from the application front-end, a learned lightweight classifier firstly performs approximate classification to efficiently identify the set of important candidates in the category space. Afterwards, the classifier will trigger candidates-only computation to generate accurate classification results, while the rest can directly utilize the approximate results computed from the screening phase. Therefore, a large amount of computations and data loading of classification are saved. Experimental results in Section 6.6.1 show that the proposed screening method achieves better trade-off for classification accu-

racy and computation saving, compared with conventional low-rank approximation-based method [153].

Fully leveraging the approximate screening method further motivates the design of the Extreme Near-Memory Classification architecture, namely *ENMC*. Here lists the key features of the ENMC design:

Firstly, as shown in Figure 6.2, ENMC has a dual-module architecture that contains a Screener module and an Executor module that run in parallel. The Screener performs approximate screening efficiently, as described in Section 6.4, and predicts the classification candidates in advance. For each candidate selected in a batch, the ENMC controller will generate instructions for accurate computations handled by the Executor. The computing modules are deployed at rank-level such that there is no need to invade the DRAM chips.

Secondly, the ENMC instruction set facilitates the workloads accommodation between host processor and ENMC, and it supports the communications between the Screener and the Executor. The instruction format that is defined by leveraging the reserved command space is compatible with the commodity DDR interface. Thus, the ENMC DIMM can also support regular memory requests.

Finally, the system-level design, including application workflow and program compiler support, makes the ENMC architecture cooperate with the software framework. The design could be easily extended no matter the host processors are CPUs, GPUs, or domain-specific accelerators.

6.2 Motivation

As discussed in Section 2.2, the classification layer is the essential component in NLP tasks and large-scale recommendation systems. Figure 6.3 shows the breakdown of model

parameters and operations into classification and non-classification, i.e., input embedding and hidden layers. For the three NLP tasks, classifiers consume a significant amount of parameters and operations. When classification category sizes scale to millions as in large-scale recommendation, classification layers become the major bottleneck. We can observe similar breakdown on execution time.

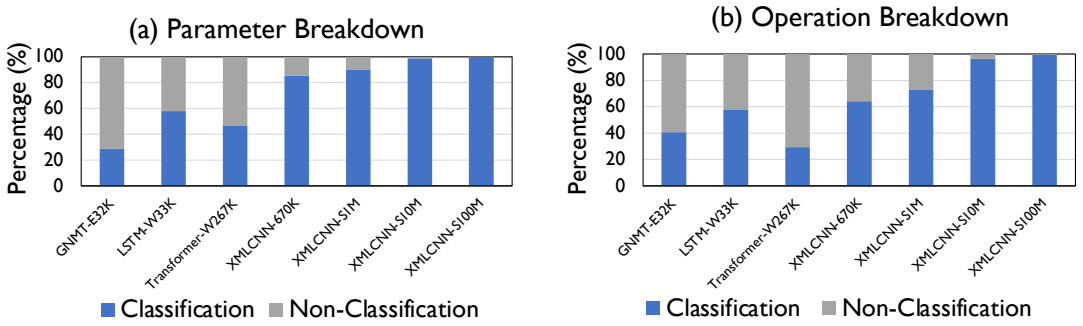


Figure 6.3: The breakdown of parameters and operations into classification and non–classification. Classification layers consume a large portion and become the bottleneck when categorize sizes scale.

6.2.1 Opportunity

The root cause for extreme classification being the bottleneck is from the large memory footprint and the low operational intensity. As shown in Figure 6.4(a), classifiers consume memory in the order of hundreds megabytes or even gigabytes, far beyond the on-chip memory capacity of modern GPUs or NPUs. The execution time of classification increases linearly with category size and hidden dimensions. From the perspective of DL practitioners and algorithm developers, using larger vocabulary or category and hidden dimensions is almost always a way to improve model quality. However, the increasing memory usage will worsen the memory-bounded execution problem. For recommendation systems, the increasing need for an enormous number of items results in even more challenging requirement to accommodate the classifier.

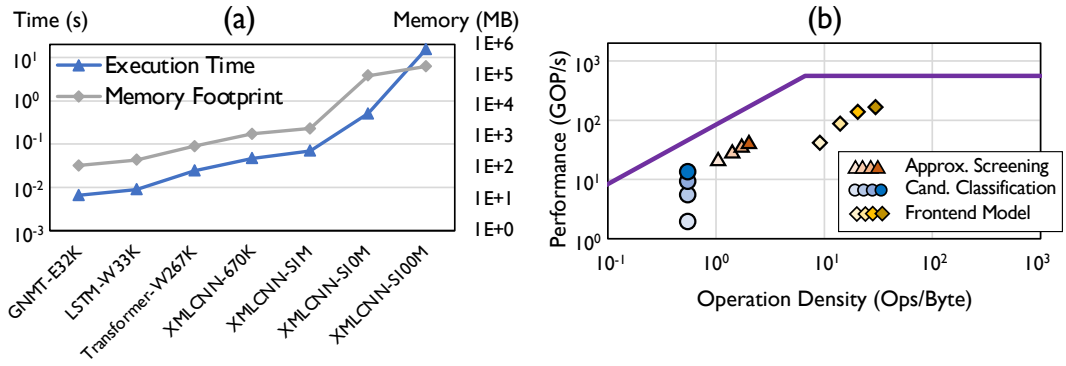


Figure 6.4: (a) The memory footprint and the execution time on CPU of classification layers scale linearly with the number of categories. (b) Roofline analysis of the major components. Darker color indicates larger batch size.

Opportunity of approximation: In extreme classification, outputs from classifier are probabilities. While we should compute all the outputs of the linear transformation using all classifier parameters, many applications require only the probabilities of the most top words. For example, in neural machine translation, we only use the top-K values of softmax-normalized probabilities to select the translated words, where K is the beam search size when applied. Therefore, we could have only the top-K probabilities to be accurate, then having the rest to be approximate, aiming at significantly reduced computations and data accesses. The next section explores the opportunity of using approximation to achieve efficient extreme classification.

Opportunity of NMP: Although approximation can greatly reduce the computation amount in extreme classification, approximate screening is still bounded by the memory bandwidth. Figure 6.4(b) plots the data points for approximate screening, candidate-only classification, and front-end neural networks in a CPU's roofline model. Both screening and classification exhibit low operation intensity after eliminating redundant computations and reducing hidden dimensions. Candidate-only classification presents sparsity characteristics due to the random candidate lookup within a batch (similar to embedding lookup in recommendation model), and thus also locate at the

memory-bound region in the roofline. Therefore, different from the front-end models that are often bounded by computation capability, approximate screening and candidate-only classification can benefit from the large bandwidth of NMP architectures.

6.2.2 Limitations of Existing NMP

As mentioned above, due to the memory-bounded execution pattern of classification, NMP-enabled systems could leverage the near-data capability to avoid significant amount of off-chip memory traffic. However, existing NMPs often employ a homogeneous architecture equipped with unified floating-point and integer compute units [158, 159, 25]. The proposed screening method explores a heterogeneous computation pattern that includes a low-precision approximate screening phase and a full-precision candidate-only classification phase. Therefore, the NMP architecture features a dedicate resource management of both phases and a customized pipeline design.

6.3 Approach

Section 6.2 discusses the potential of using NMP to alleviate the memory pressure of executing extreme classification. However, the limited computing capability of NMP logic cannot afford the computations of extreme classification. In other words, the execution of full classification on NMP core becomes the bottleneck.

We can find that not all computations in classification are useful. In fact, only a small portion of classification results contribute to model predictions. For example, in language modeling tasks, only output probabilities of the most important words need to be accurate. Thus, the proposed efficient approximation method can estimate the subset of output probabilities that need accurate computations and then populate the rest probabilities with approximate results. Similarly, for other classification-involved

tasks, we only need accurate computations for a small number of key candidates and use approximate results for the remaining outputs.

6.3.1 Screening Method Overview

Given a d-dimensional vector ($h \in \mathbb{R}^d$) from hidden DNN layers, where d is the hidden dimension, the softmax classification transforms the hidden vector h to a l-dimensional probability space. he output probability vector is denoted as $z \in \mathbb{R}^l$, where l is the vocabulary size. The transformation is essentially matrix-vector multiplication as

$$z = Wh + b \tag{6.1}$$

where $W \in \mathbb{R}^{l \times d}$ is the classifier weight matrix and $b \in \mathbb{R}^l$ is the bias vector. Then, the softmax function normalize the output vector z into probability distribution as

$$p_i = softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)} \tag{6.2}$$

where p_i is the i-th element of output probability vector p. The probability vector is then used to perform next word predictions as in language modeling or translation. While softmax is the most common normalization function used in classification, the method is capable to other non-linear functions used in classification such as sigmoid [6].

As discussed in Section 6.2, the memory-intensive transformation is a good candidate of NMP architectures. However, the computational complexity is not affordable for NMP. ENMC seeks redundancy in extreme classification and uses low-cost approximated computations to mitigate the computational burden. The introduced low-dimensional and low-precision screening module can approximate the original classifier. Next, we will discuss how to reduce computations at inference time given the screening module. After

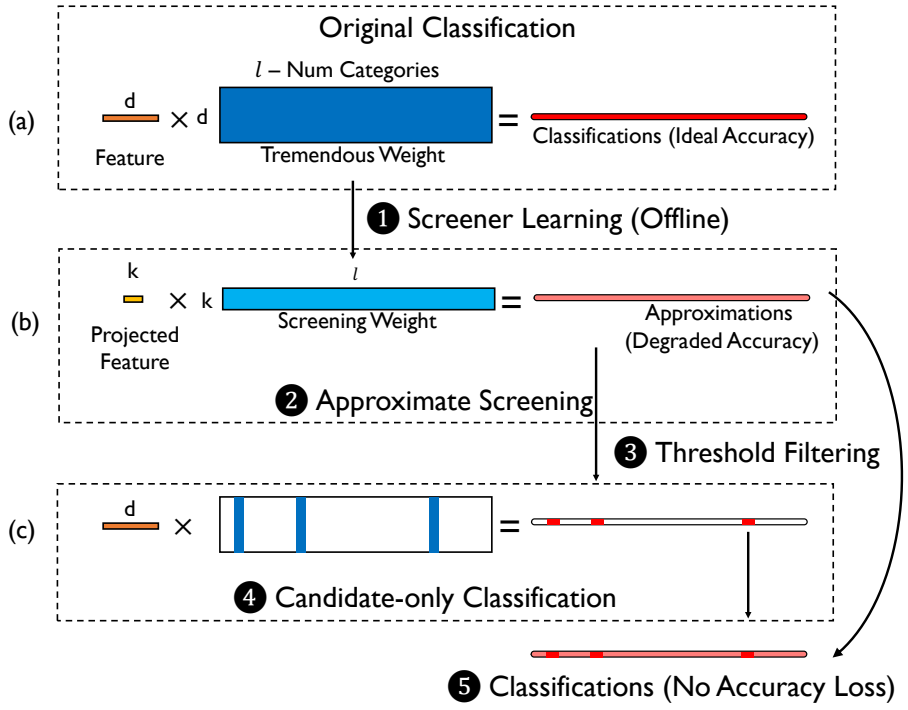


Figure 6.5: Illustration of approximate screening: (1) the screener learns from full classifier at the offline learning phase; (2) the screening step computes approximate results, involving lightweight Screener weights and the projection matrix, and selects candidates among approximate results; (3) the threshold filtering step selects key candidates; (4) only the corresponding vectors in the full classification weights are used to compute candidates-only accurate results; (5) the final results before softmax normalization combine both approximate and accurate results.

that, we can see the details of the learning process that obtains the screening module.

6.3.2 Inference Process

As shown in Figure 6.5(a), the standard classification is essentially matrix-vector multiplication followed by softmax normalization. The execution is bounded by accessing W from DRAM modules.

We can construct the approximate screening module with a projection matrix P and a reduced-hidden-dimension weight matrix $\tilde{W}$. The initialization of the projection matrix is according to standard sparse random projection [141], and the overhead is

negligible (less than 0.1%) compared with classifier weights as the projection matrix P can be represented in 2-bit format. The process of computing approximate results can be expressed as

$$\tilde{z} = \tilde{W}Ph + \tilde{b} \tag{6.3}$$

where $\tilde{W} \in \mathbb{R}^{l \times k}$ and $P \in \sqrt{\frac{3}{k}} \cdot \{-1, 0, 1\}^{k \times d}$.

Figure 6.5(b) illustrates the process: the d-dimensional hidden vector h is first projected to a lower k-dimensional space, and the low-dimensional vector multiplies $\tilde{W}$ to get approximated output $\tilde{z}$. Compared with full classification, the accessed approximate weight volume is significantly reduced since $k << d$. Furthermore, we can reduce the precision of running the screening module to further reduce accessed data.

After obtaining the approximate results, i.e., $\tilde{z}$, we can estimate the importance of all l values and select the most important m values, referred as candidates, that require accurate computations. The estimation can be done with top-m searching or thresholding, where the threshold value can be tuned on validation sets.

Only for the candidates that need accurate computations, the proposed method then need to access full classifier weights W, i.e., a small portion of totally l weight vectors. These weight vectors then multiply with the original hidden vector to produce the accurate results for the candidates, as shown in Figure 6.5(c). The final outputs before softmax function is a mixed vector with approximate values from screening and accurate values from full W.

6.3.3 Learning Algorithm

Here, we discuss the learning procedure to obtain screening module. The goal for screening is to approximate the classifier well. Therefore, the outputs z from full classifier is regarded as the learning target and train the screening module weights $\tilde{W}$ to fit. The

Algorithm 3: Training algorithm for the parameters of the Screener

Data: Batched context vectors $\{h_i\}_{i=1}^{S}$, where $h_i \in \mathbb{R}^d$ from hidden layers; trained classifier weights $W \in \mathbb{R}^{l \times d}$ and bias $b \in \mathbb{R}^l$; projection matrix P.
Result: Screener weights $\tilde{W} \in \mathbb{R}^{l \times k}$ and bias $\tilde{b} \in \mathbb{R}^l$.

```
Initialize projection matrix P ∈ sqrt(3/k) · {−1, 0, 1}^{k×d};
for it ∈ all iterations do
    Compute loss according to Eq. (6.4);
    Update W̃, b̃ with SGD(min Loss);
end
```

optimization objective function is

$$L = \frac{1}{s} \sum_{s} ||(Wh + b) - (\tilde{W}Ph + \tilde{b})||_2^2 \tag{6.4}$$

where s is the mini-batch size of training samples. During training, the classifier parameters, i.e., W and b, as well as the parameters of hidden layers are fixed and will not be changed. We only update the screening module's parameters $\tilde{W}$ and $\tilde{b}$. The projection matrix P is constructed and initialized before distillation and stays constant during distillation and inference.

The learning algorithm uses the default training and validation datasets and does not need extra training data. The convergence happens in a several training epochs, much faster than original model training. Algorithm 3 gives the overall training of screening parameters.

6.4 Architecture

This section introduces the architecture design of the ENMC: firstly, a glimpse of the design overview, followed by the micro-architecture details; then, the ENMC instruction set and system-level design.

6.4.1 Design Overview

We have yet exploited the opportunity of eliminating the redundancy in the extremely-large weight and forecasting the classification results with much smaller overhead using lightweight screening algorithm. Although the computation bottleneck is alleviated with our proposed approximate screening framework, the tremendous classification dimension is still bandwidth-hungry, and conventional processor-memory systems are hardly able to overcome the memory throughput wall. Therefore, this section further presents the co-design the near-data processing subsystem, Extreme Near-Memory Classifier (ENMC), that can facilitate the processor computing the extreme classification. The design goal of such near-data architecture is to leverage the large bandwidth provided by rank-level parallelism in a DRAM channel, and process the classification in data stream through dedicate on-DIMM hardware.

Specifically, here highlights the features of the ENMC design: First, a dual-module architecture is used that contains a Screener module and an Executor module that runs in parallel. The Screener performs fixed-point screening as described in Section 6.3, and predicts the classification candidates in advance. Since the classification weight is low-dimensional and quantized, the Screener is able to process the data in a streaming manner, such that the large rank-level bandwidth can be leveraged. For each candidate found in a batch, the ENMC Controller will generate instructions for further full-precision computations which are completed by the Executor. We can put these computation logic at the rank level such that there is no need to invade the DRAM chips.

Second, the ENMC instruction set is designed to facilitate the workloads accommodation from host processors and support the communications between the Screener and Executor modules. The instruction format is defined by leveraging the unused address line and data line in the PRECHARGE command to ensure the compatibility with the

commodity DDR interface. Thus, regular memory requests can also be served with our ENMC DIMM.

Third, the system-level design is provided, including the program compiler support and application workflow, to make the ENMC architecture cooperate with the software framework. Our design could be easily extended to support different scenarios where the host processors could be CPU, GPU, or domain-specific accelerators.

6.4.2 ENMC Micro-architecture

We now introduce the micro-architecture of ENMC using the design overview, followed by the implementation details of each component.

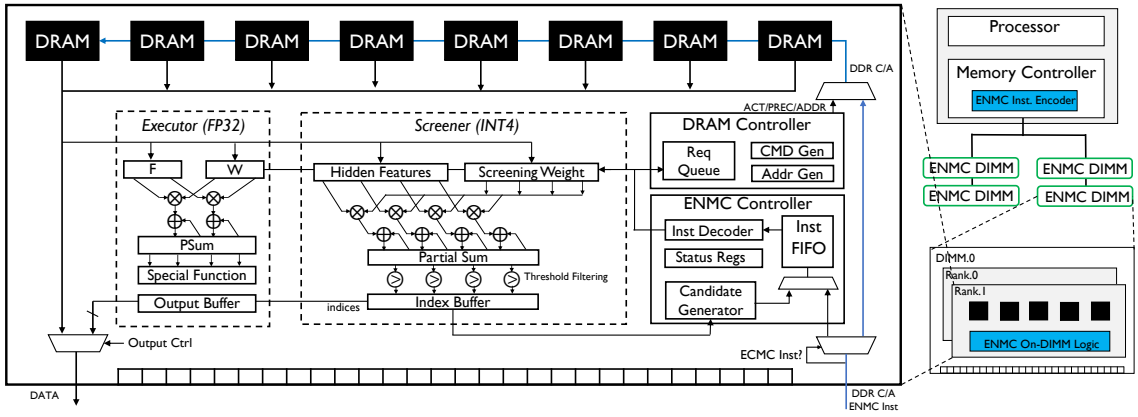


Figure 6.6: The architecture overview of an ENMC DIMM. The ENMC logic is located at each rank to leverage the large rank-level bandwidth. The ENMC mainly consists of a controller to decode instructions, a DRAM controller to generate DDR C/A commands, a screener to perform approximation, and an executor to process full-precision classification.

Overview. ENMC is on the DIMM board between the DRAM devices and the DDR PHY, such that the host processor could interface with ENMC through standard memory channels. Figure 6.6 illustrates the details of the proposed ENMC architecture. The host processor contains several memory channels, which are deployed as the ENMC DIMMs. The ENMC logic locates at each rank of a ENMC DIMM, and thus enjoys

scaling bandwidth offered by larger number of ranks. The on-DIMM ENMC architecture consists of a ENMC controller, a DRAM controller, and two processing units: the Executor and the Screener. The ENMC controller buffers the instruction from the host processor for approximate screening. It also generates instructions for full-precision computation according to the candidate indices provided by the Screener. Then, it decodes the formatted instructions to generate control signals for data access, computation, and output transmission. The DRAM controller works as a simplified memory controller that processes data access requests in ENMC instructions and generates the standard DDR C/A signals to the DRAM chips. The Screener and Executor take charge of the approximate screening and the full-precision computation as described in Section 6.3.2, respectively. The Screener performs dimension-reduced INT4 computations to efficiently approximate the classifier's output. A preloaded threshold is used to filter out the important candidates based on the approximate results. Apart from floating-point arithmetic, the Executor is also equipped with a special-function unit to process the non-linear activation in the final layer. The two computation modules works in parallel and write results to the output buffer that returns them to the host processor asynchronously.

ENMC Controller. The ENMC controller has two main functionalities: processing the instructions from host processor (i.e., screening computation) and generating instructions for the Executor (i.e., candidate-only computation). It is made of status register files, an instruction buffer, an instruction decoder, and an instruction generator. The status register files are used for ENMC initialization and stores information such as addresses and sizes of input features, vocabulary, and screening weight. It also includes the instruction counter. The instruction buffer is a FIFO, and both the host processor and instruction generator could push instructions into it. The instruction decoder sequentially reads from the FIFO and generates control signals to corresponding ENMC components. For example, an instruction of accessing a piece of tiled screening weight

would result in a read request to the DRAM controller and a select signal to the top DEMUX that chooses the integer weight buffer. Meanwhile, a full-precision computation instruction would lead to a triggering signal to the floating-point MAC array, which reads data from two input buffers and writes results to the partial sum (PSUM) buffer. The instruction generator receives the indices of classification candidates from the Screener (*batch_id*, *candidate_id*) , and then reads the constant reg to generate corresponding instruction for candidate-only computation in full-precision.

DRAM Controller. The DRAM controller employs a similar architecture as the host-side memory controller and consists of a request queue, a command generator, and an address generator. The request buffer takes memory request from the ENMC controller. The command and address generators initiate standard DDR4 C/A signals that are sent to all the DRAM chips. For hardware simplicity, we do not deploy unnecessary features like queue prioritizing, request coalescing, etc.

Screener. The Screener processes the approximate screening phase in the approximate screening algorithm with fixed-point precision. We put two input buffers (feature buffer and screening weight buffer), a fixed-point multiply-accumulate (MAC) array, a partial sum (PSUM) buffer, a threshold filter, and an instruction translator in the Screener. The MAC array performs the screening computation over the two input buffers and accumulates with the intermediate results in the PSUM buffer. After a tiled screening is finished, the data in the PSUM buffer are filtered with a comparator array. The indices of values larger than the threshold are buffered and later sent to the ENMC controller.

Executor. The Executor computes candidate-only classification under full-precision. Compared with the Screener, it applies floating-point MAC array and has an extra special-function unit that performs the non-linear activation such as Softmax and Sigmoid. An output buffer placed below the special-function unit caches both the results

from the Screener and the Executor. The output buffer keeps the state of the data with status reg files and notifies the ENMC controller (by pushing a RETURN instruction) when finishing a batched/tiled data.

6.4.3 ENMC Instruction Set

The design goal of the ENMC instruction set is to make the host processor able to communicate with ENMC DIMM through standard DDR4 memory channels. Inspired by FIRDRAM [22], we can issue ENMC instructions from the memory controller with PRECHARGE command combining special addresses and data. For example, according to the DDR4 JEDEC specification, for a 4Gb DIMM with 8 ×8 DRAM chips, the row address space consumes 14 bits, i.e., A0-A13 in the C/A bus, and the data bus is 64-bit. Normal PRECHARGE command sets all the row address bits to be low, since no row information is needed. Therefore, an ENMC instruction could be accommodated with sending a PRECHARGE command but turning on the row address signals. Given this insight, the ENMC instructions are formatted in 13-bit command and 64-bit data that transmits through signal A0-A12 and D/Q bus.

Table 6.1: The ENMP instruction set

ENMC Instruction Set		
Type	Instruction	Description
Initialization	INIT reg, data	Initialize the ENMC module by writing a particular register

Data Transfer	LDR buffer, addr
	STR buffer, addr
	MOVE buffer1, buffer2
		load/store the quantized feature data into/from the INT4 feature buffer (weight buffer, with specified address addr

Compute	ADD_INT4 buffer1, buffer2 MUL_INT4 buffer1, buffer2 ADD_FP32 buffer1, buffer2 MUL_FP32 buffer1, buffer2	add/multiply the data in two specified buffer buffer1, buffer2
	MUL_ADD_INT4 MUL_ADD_FP32	multiply the data in feature buffer and weight buffer, and accumulate they with the partial sum buffer
	FILTER buffer	filter the data in the specific buffer and write the results to the index buffer
	SIGMOID, SOFTMAX	special functions such as Sigmoid and Softmax that run on specialized hardware for the data in the FP32 partial sum buffer
Control	BARRIER, NOP	synchronization and bubble instruction to let the controller wait for memory accesses, compute operation, data movement, etc.
	QUERY reg	query the value in the specific reg
	RETURN	return the data in the output buffer
	CLR	clear and reset all buffers and registers

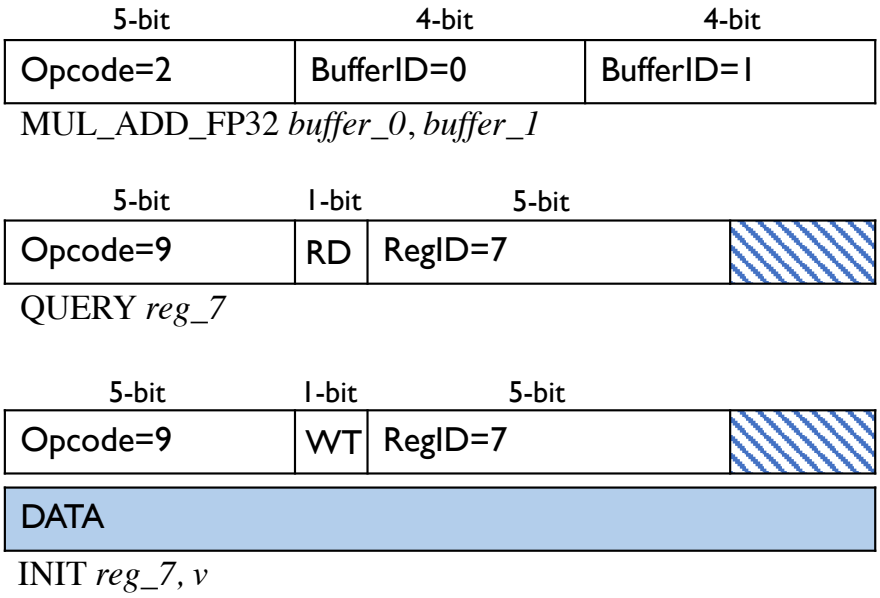


Figure 6.7: Instruction Format

Instruction Specification. As shown in Table 6.1, the ENMP instruction set consists of four types of instructions: Initialization, Data Transfer, Compute, and Control. (a) *Initialization.* The initialization instruction is used to write the status reg files in the ENMC controller, in order to initiate the parameters of a classification task. It specifies which reg to write and the corresponding value. (b) *Data Transfer.* The data transfer instructions are used to access the on-DIMM buffers, such as loading data to the input feature buffer or writing back the results to the PSUM buffer with specific addresses. Also, the MOVE instruction is used to transfer data in two buffers, such as storing results in the PSUM buffer to the output buffer. (c) *Compute.* The compute instructions corresponds to the computation operations in the two computing units, including ADD, MUL, MUL_ADD, and denotes the operation precision. FILTER instruction is used to filter out the candidates. There are also instruction for special functions such as SOFTMAX and SIGMOID that operate on the PSUM buffer in the Executor. (d) *Control.* The control instructions include BARRIER for synchronization, NOP for stalling, RETURN to send

back the output buffer data, and CLR to reset the ENMC. The QUEUE instruction is designed for the host processor to pull the status counters in each component.

Instruction Format. As shown in the Figure 6.7, a typical ENMC command without data or address takes 13 bits, where the opcode is 5 bits and the rest 8 bits are used to specify which buffer to operate on. For example, Figure 6.7(a) shows the instruction format for performing multiply-accumulate in the Screener. For the status register accessing instruction, QUERY and INIT shares the same opcode, and they use one bit after opcode to specify the read or write operation, and 5 bits to specify the register index, as shown in Figure 6.7(b). Moreover, for instructions that involves values (i.e., data or address) that exceeds the length of row addresses, the DQ bus is further utilized. For example, when the host processor tries to write the status reg in the ENMC controller, the command address bus specifies the write operand and the ID of target regwith INIT instruction, and the DQ bus transmits the desired data in burst manner following the ENMC command.

6.4.4 System Design

Here further shows how to architect the system-level design to facilitate existing software solutions running on the ENMC memory. Firstly, the programming support can wraps up ENMC instructions into high-level APIs such that a program could call the ENMC kernels directly. Secondly, the execution flow demonstrates how the host processor interacts with the ENMC DIMM.

Programming Support. Following previous NMP solutions [24, 25], we divide the application code into kernels running on the host processor and ENMC in a heterogeneous manner. Therefore, the host processor calls the provided APIs to offload specific classification tasks. Figure 6.8(a) shows an illustrative application code in Python style.

Illustrative Application Program

```
import ENMC

//Host Preprocess
model = app_model()
model = load(path)
...

//Initialize ENMC Params
ENMC.initial_classifier(\
    model.classifier.w, model.classifier.b)
ENMC.initial_screener(\
    model.proj_M, model.screen_w, model.screen_b)

for (batch_id, x) in batched_dataset:
    //Host Inference
    x = model.lstm(x)

    //ENMC Classification
    x = model.classifier(batch_id, x)
```

ENMC Instructions

```
//Initialization
INIT rx0, addr_w
INIT rx1, addr_b
INIT rx2, dim0_w
INIT rx3, dim1_w
...

//Run Classification
INIT r0, batch_id
INIT r1, batch_size
LOAD buffer0, 0x4cea4fe2
LOAD buffer1, 0xa1ebea62
MUL_ADD_INT4 buffer0, buffer1
LOAD buffer0, 0x4cea50e2
...
FILTER psum0
BARRIER
LOAD buffer2, 0xb2eef3a2
LOAD buffer3, 0xa97fac44
MUL_ADD_FP32 buffer2, buffer3
...
```

Figure 6.8: An illustrative example of programming support of ENMC. The ENMC APIs are wrapped as high-level function libraries, which are further compiled into ENMC instructions.

The functions can run on ENMC DIMM as wrapped up into a Python package, such as initializing the Screener and screening-based classification. Therefore, a programmer could build a machine-learning model transparently using the ENMC package. The approximate screening algorithm is implemented inside an ENMC object of classifier. Furthermore, when translating the applications into ENMC instructions, the compiler tiles the operation with initialized parameters and hardware configurations and executes the instruction in a loop. The ENMC instructions are further packed into a memory request packet and routed to the memory controller, which transmits them to the ENMC DIMM, as shown in Figure 6.8(b).

Execution Flow. Figure 6.9 presents the ENMC workflow compared with a host only system. The execution of front-end feature extraction (DNN-based or non-DNN-based) and the classification can be treated in a decoupled way. To be more specific, the host in the ENMC system is dedicated to run the feature extraction and offloads the

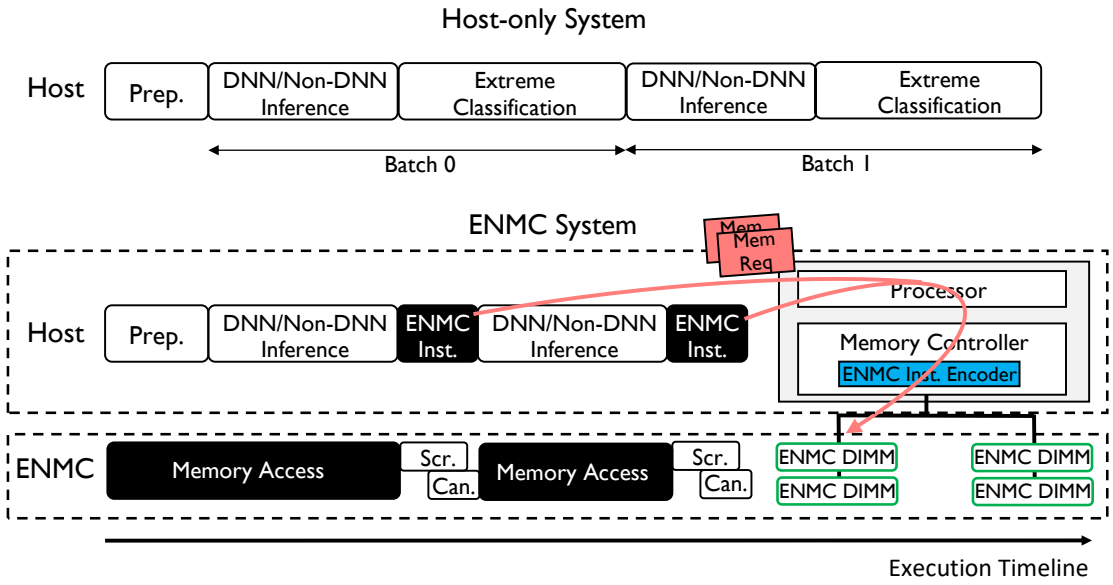


Figure 6.9: The ENMC workflow compared with a host-only system. ENMC offloads the classification tasks to the ENMC DIMMs by sending the instructions as memory requests through the memory controller.

classification tasks to the ENMC memory. The ENMC memory works as a regular main memory for data accessing in the first phase, and performs screening approximation and candidate-only classification in the second phase.

6.5 Methodology

This section discusses the methodology of evaluating the ENMC co-design, including the implementation details and performance metrics.

6.5.1 Software Evaluation

The approximate screening algorithm is implemented on top of existing pre-trained models in the PyTorch machine learning framework [160]. The screening parameters are trained under mean-square-error (MSE) loss using the original training and validation

datasets till convergence. Both the input features and the screening parameters are further quantized at inference time. The number of candidates, screening parameters size, and quantization precision are adjustable for sensitivity studies.

Table 6.2: Evaluated models and datasets.

Application	Dataset	Dataset Type	Num. Categories	Inference Model	Model Type	Hidden Size	Abbr.
NLP	Wikitext-2	Language Modeling	33,278	LSTM	RNN	1500	LSTM-W33K
NLP	Wikitext-103	Language Modeling	267,744	Transformer	DNN	512	Transformer-W268K
NMT	WMT16, en-de	Translation	32,317	GNMT	DNN	1024	GNMT-E32K
Recommendation	Amazon-670k	Multi-label Classification	670,091	XMLCNN	CNN	512	XMLCNN-670K

Workloads. The evaluated tasks include Language Modeling (LM) [161], Neural Machine Translation (NMT) [1], and product-to-product recommendation [162]. LM tasks use the Wikitext-2 and Wikitext-103 datasets [3] and evaluate on both long short-term memory networks (LSTM) and Transformer networks. NMT tasks use the WMT16 English-to-German dataset and evaluate on Google's Neural Machine Translation System (GNMT) [2]. Product recommendation tasks use the Amazon670K dataset [10] and evaluate on a Convolutional Neural Network based model [6]. Table 6.2 lists the applications, the models, and the datasets used in the evaluation, as well as the number of categories and the hidden dimensions, including three synthesized datasets with 1 million, 10 million, and 100 million categories to study the scalability of ENMC (namely S1M, S10M, and S100M).

Baselines. Two other approximation methods for classification: SVD-softmax [153] and FGD [163] are compared with approximate screening. The SVD-softmax method leverages singular value decomposition (SVD) to approximate the classification weight with principle singular values; the FGD method uses graph-based nearest neighbor search to approximate top-k classification results. Both baselines are implemented in the PyTorch-based framework.

Table 6.3: ENMC Configurations

DRAM Configuration			
Spec	DDR4-2400MHz	DRAM Chip	8Gb×8
Channels	8	Ranks/CH	8
Queue	64-entry	Capacity/CH	64GB
Timing	CL-tRCD-tRP: 16-16-16 tRC=55, tCCD=4, tRRD=4, tFAW=6		
ENMC Configuration			
Tech Node	28nm	Frequency	400MHz
Executor Buffer	256B+256B	Screener Buffer	256B+256B
FP32 MAC	16	INT4 MAC	128

6.5.2 Hardware Evaluation

The ENMC logic is implemented in RTL and synthesized with Design Compiler for hardware parameters including timing, power, and area. A cycle-accurate simulator is developed for the ENMC DIMM that interfaces with Ramulator [164] to derive the DRAM timing information. Since the host processor and the ENMC DIMM execute the feature extraction phase and the classification phase separately without complicated feature interactions in between, we can simulate a simple host model that only issues ENMC instructions regularly according to the status registers.

Configurations. As shown in Table 6.3, the ENMC DIMM is based on DDR4-2400 specifications. Each rank consists of 8×8 DRAM chips that add to a total capacity of 8Gb. We put 8 memory channels for the host processor, and there are 8 ranks per channel, contributing to 64GB capacity and 21.3 GB/s bandwidth per channel. In addition, the ENMC logic is synthesized with TSMC 28nm technology, running on the frequency of 400MHz. The two input buffers and accumulation buffer in both Screener and Executor are 256B. There are 64 INT4 MACs and 16 FP32 MACs on each DIMM. The exponential function, for non-linear activations in the executor, uses Taylor expansion to the 4^{th} order.

Baselines. ENMC is compared with CPU and other NMP architectures, and all

Table 6.4: Comparing ENMC with three NMP baselines, all configured with similar area and power budget.

NMP Designs	Configuration	Est. Area - mm^2	Est. Power - mW
NDA [158]	4*4 Functional Units + 1KB Memory	0.445	293.6
Chameleon [159]	4*4 Systolic Array + 1KB Memory	0.398	249.0
TensorDIMM [25]	16-lane VPU + 512B Queue * 3	0.457	303.5
ENMC (Ours)	FP32 * 16 + INT4 * 128 + 256B Buffer * 4	0.442	285.4

of them are equipped with the approximate screening algorithm. The CPU baseline is Intel Xeon Platinum 8280 @ 2.7GHz. It has 28 physical cores and 6 DDR4-2666 memory channels, contributing to a total memory capacity of 512 GB and 128GB/s ideal bandwidth. Three state-of-the-art DRAM-based NMP architectures are also selected for evaluation:

NDA [158] provides a near-data acceleration solution by stacking coarse-grain reconfigurable accelerators (CGRA) with DRAM devices. The CGRA mainly consists of functional units, switches, and memory.

Chameleon [159] is similar to NDA by employing a 2D architecture and focusing on how to integrate the accelerator with commercial DRAM. As Chameleon could work with any programmable compute unit, a systolic array acts as the accelerator core to distinguish it from NDA.

TensorDIMM [25] is a NMP architecure for deep learning applications, especially for recommendation workloads. It leverages the VPU to accelerate the embedding operations in recommendation systems.

For a fair comparison, ENMC and three baselines are configured with approximately

the same area and power budget, as shown in Table 6.4; the control logic and DRAM device controller are excluded.

6.6 Evaluation

This section evaluates the screening method for extreme classification and the micro-architecture of near-memory processing cores. The method evaluation shows the trade-offs between inference quality and speedup to CPU execution time of full classification. Then, the architecture evaluation presents the speedup of classification enabled by NMP co-design and the system performance improvements.

6.6.1 Algorithm-level Evaluation

Overall model quality. The hypothesis is that extreme classification can afford approximation. Here, the experimental results can support the hypothesis. Overall, the screening method can achieve significant computation saving with negligible model quality degradation. Lowering model inference quality to the acceptable extend can achieve more computation reduction.

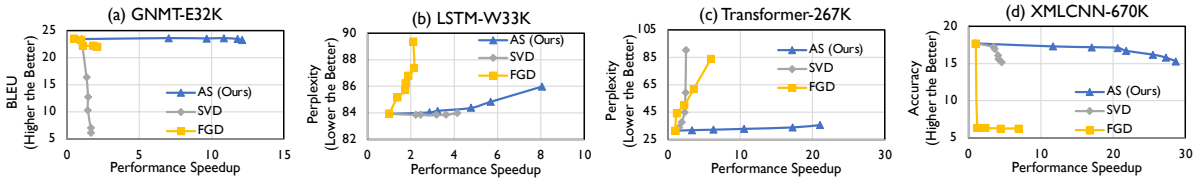


Figure 6.10: Quality vs. Speedup trade-off of Approximate Screening (AS) and two baselines: SVD and FGD.

As shown in Figure 6.10(a), compared with using full classification as in NMT tasks, the proposed method can achieve speedup of 11.8× without any loss in translation quality measured by BLEU score. As for LM tasks, the speedups can reach 5.7× to 6.3× while preserving perplexity results, as shown in Figure 6.10(b) and (c). Similarly, for product

recommendation, the screening method can achieve 17.4× speedup with only 0.5% drop in accuracy, as shown in Figure 6.10(d).

Because of the well approximation that the screening method achieves, the screening phase can effectively select the key candidates for classification. Using the NMT task as an example, at every decoding step, we want the most likely word or a few words if using beam search. With Approximate Screening, we can identify the key candidates and compute the accurate probabilities of these words for translation, saving redundant computations for the remaining words in the vocabulary. We set the overhead of Approximate Screening to be 3.1% of full classification.

Compared with two other approximation methods, the method achieves better quality-speedup trade-off, as shown in Figure 6.10. Besides, the computation overhead of SVD-based approximation is 4× more than ours. We can infer that the improvement of the method is due to the learning-based approximation and no strong requirement for classifier weights to be low-rank.

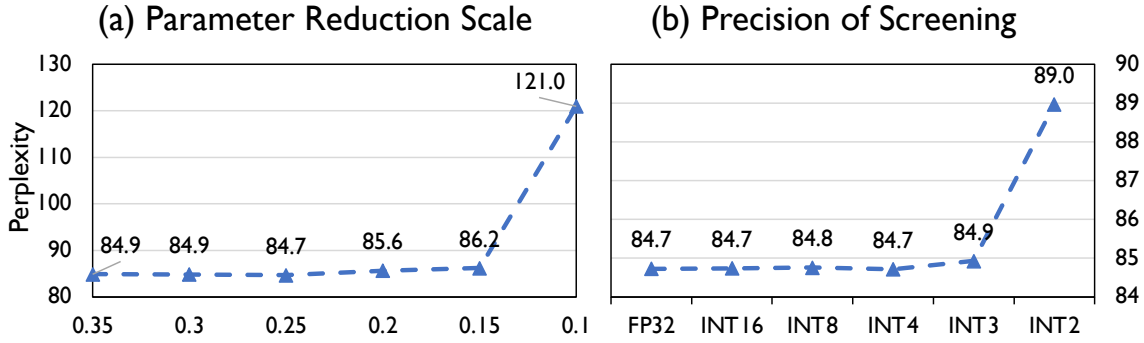


Figure 6.11: Comparing different (a) parameter reduction scales and (b) quantization levels of AS. We choose parameter reduction scale of 0.25 and precision of INT4 in our design.

Sensitivity on Approximate Screening. Intuitively, better approximation costs larger computation and data overhead, while achieving better model quality with screening. Here shows different parameter sizes of the screening module and the corresponding

quality. Figure 6.11(a) shows different parameter reduction scales of the screening module vs. full classifier; the scale is chosen to be 0.25 as the good quality preserving. As shown in Figure 6.11(b), 4-bit fixed-point quantization is used on on the screening module as this quantization level maintains approximation as using single floating-point precision.

6.6.2 Architecture-level Evaluation

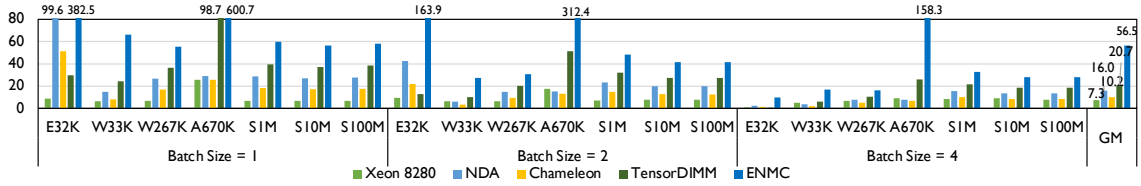


Figure 6.12: The performance results of ENMC, CPU, NDA, Chameleon, and TensorDIMM, normalized to vanilla CPU; all schemes are equipped with approximate screening.

Performance. As described in Section 6.5, ENMC is compared with four baselines. Figure 6.12 shows the performance results of using batch size of 1, 2, 4, normalized to the full-classification CPU baseline for each workload, and the results are arranged according to the size of classification across the x-axis. The approximate screening demonstrates 7.3× performance speedup on average in CPU baseline, and the ENMC offers a total 56.5× speedup over the CPU. Also, 3.5×, 5.6×, and 2.7× average are observed when compared with NDA, Chameleon, and TensorDIMM respectively. First, ENMC provides significant speedups of 55.5×-600.7× when running low-latency inference with batch size of 1, because ENMC processes the inference in a streaming manner over the lightweight classification. The huge performance gain in XMLCNN-670K workload is due to the reduction of the number of candidates by 50×. Second, the three NMP baselines benefit from large internal bandwidth and offer 10.2-20.7× speedup over the CPU baseline. However, ENMC could further boost their performance by 2.7-5.6× with heterogeneous resource management and dataflow customization. This result aligns the assumption that

the performance of naive NMP solutions is bounded by the limited on-DIMM buffers and computation resources. Because they employ homogeneous FP32 computation units and hardly meets the throughput requirement in the screening phase. ENMC eliminates the redundant computation and needs only a small portion of FP32 computations. The entire screening phase is processed with lightweight INT4 units in stream.

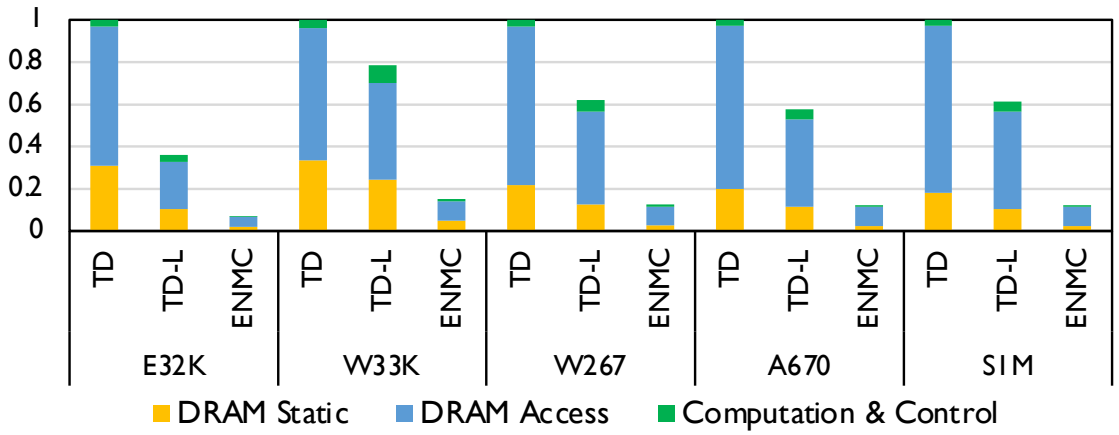


Figure 6.13: Energy breakdown by DRAM static cost, DRAM access, and computation & control logic, normalized to TensorDIMM.

Energy Consumption. The energy results of ENMC are evaluated against TensorDIMM and TensorDIMM-Large for a fair comparison. As shown in Figure 6.13, ENMC can reduce the average energy cost by 5.0× and 8.4× compared with TensorDIMM and TensorDIMM-Large, respectively. Particularly, breaking down the energy consumed by the DRAM static cost, DRAM access, and on-DIMM computation/control logic, we can observe that the significant energy reduction of ENMC comes from two facts: First, the co-designed approximation algorithm greatly reduces the DRAM accesses in ENMC. ENMC performs INT4 and low-dimensional screening during the classification phase, while TensorDIMM and TensorDIMM-Large need to operate over the full classification weight. Moreover, due to the limited logic-side buffer size, TensorDIMM cannot store the intermediate results in a matrix multiplication operation. Thus, the buffer overflow results in frequent DRAM memory accesses. Second, the reduced execution time

leads to the background energy reduction of the DRAM modules. As the DRAM takes a noticeably portion of power for refreshing, ENMC reduces the DRAM static energy consumption by 9.3× and 4.8× compared with TensorDIMM and TensorDIMM-Large.

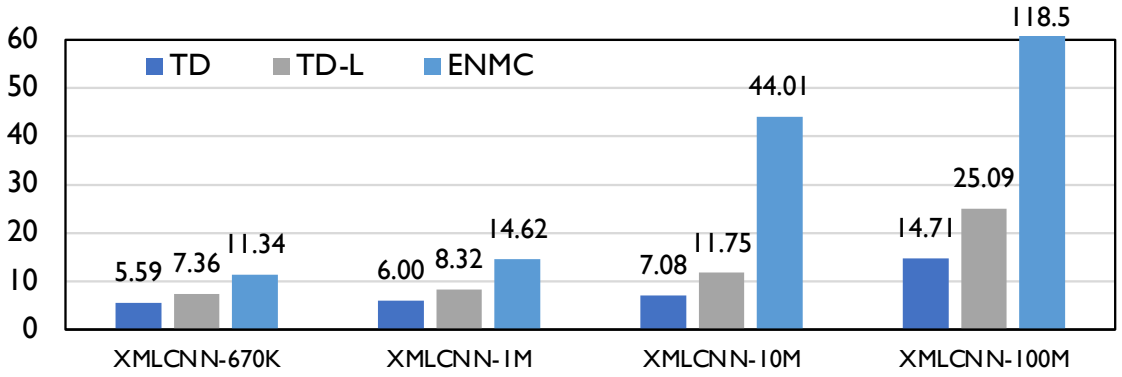


Figure 6.14: The end-to-end performance scalability compared with TensorDIMM and TensorDIMM large. We evaluate them with the same front-end model of XMLCNN, and we normalize the performance to CPU.

End-to-End Scalability. Evaluation on large synthetic datasets shows the scalability of performance considering the end-to-end performance. Figure 6.14 shows the performance of TensorDIMM, TensorDIMM-Large, and ENMC, normalized to the CPU baseline, when restricting the application to the same front-end model of XMLCNN. For comparison, ENMC achieves 4.7× and 2.9× speedup over TensorDIMM and TensorDIMM-Large. Particularly, for the two smaller datasets, ENMC achieves 2.2× and 1.6× speedups, while for the two tremendous datasets, ENMC achieves 7.1× and 4.2× speedups, compared with TensorDIMM and TensorDIMM-Large, respectively. The excellent scalability of ENMC comes from the fact that the ENMC processes the lightweight classification in stream and does not need to buffer large intermediate results back to DRAM.

Area and Power. Table 6.5 shows the breakdown area and power estimation of ENMC. The total area of ENMC logic is $0.388mm^2$, and the total power is 264.6mW, which are comparable to prior NMP architectures such as RecNMP [24]. Specifically, the compute unit (INT4 and FP32 MAC arrays) takes 40.8% of the total area and 25%

Table 6.5: Area and Power Estimation.

	Area (mm^2)	Power (mW)		Area (mm^2)	Power (mW)
INT4 MAC	0.013	10.4	**FP32 MAC**	0.145	58.0
Compute Buffer	0.061	56.8	**Control Buffer**	0.053	49.3
ENMC Ctrl	0.035	32.9	**DRAM Ctrl**	0.135	78.0
Total Area 0.442mm^2***; Total Power 285.4***mW					

of the total power. The buffers made of register files in the Screener and the Executor compose of 23.5% of the total area and 32.2% of the total power. Finally, the ENMC controller and DRAM controller takes 9.0% and 34.8% of the area, and 12.4% and 29.5% of the power, respectively.

6.7 Conclusion

This Chapter addresses the extreme classification problem with NMP-based software-hardware co-design. An approximate screening algorithm is proposed to reduce the computational complexity and memory consumption of extreme classification. Furthermore, a near-memory architecture is designed to utilize efficient candidates-only classification enabled by the screening method. Finally, the approximate screening method achieves 7.3× speedups, and the ENMC architecture further improves the performance by 7.4× and demonstrates 2.7× speedup compared with the state-of-the-art NMP baseline.

Chapter 7

Dynamic Sparse Attention

Transformer Neural Networks powered by the attention mechanism are stunning in sequence modeling tasks. However, the quadratic computational complexity of attention hinders long-sequence modeling. This Chapter investigates *elastic processing* in the attention mechanism and proposes dynamic sparse attention for scalable Transformer acceleration.

7.1 Introduction

Transformers [4] have become the driving force for sequence modeling tasks such as neural machine translation [165], language understanding [166], and generative modeling [167, 168]. Equipped with the self-attention mechanism, Transformers are capable of handling long-range dependencies.

Despite the impressive progress made by Transformers, the computational requirements make the deployment of Transformer-based models difficult at inference time, especially when processing long sequences. The self-attention modules are the execution bottleneck under long sequences. Therefore, many studies propose Transformer variants

to mitigate the quadratic time and space complexity issue. Some approaches are primary for memory footprint reduction during training while efficient inference is being understudied [62, 169, 67, 68]. Other methods use fixed or static sparse attention patterns to save computations [65, 66, 170, 64]. However, as discovered in this work, intrinsic sparse patterns in attention are naturally dynamic, depending on input sequences. Thus, we can exploit the dynamic sparse patterns to save attention computations without sacrificing the representation power of attention. Intuitively, posing static sparsity constraints in attention could be too strong to capture dynamic attention connections.

The proposed Dynamic Sparse Attention (DSA) approach exploits dynamic sparsity to improve efficiency. The challenge is to efficiently search for sparse patterns close to *oracle* sparse patterns that keep all the important attention weights. The searching can be formulated as a prediction problem, and the standard attention mechanism can be augmented with a prediction path. As discussed in Section 7.3, we can first obtain an approximation of attention scores with low computational costs. Then, we can predict the sparse attention patterns using the approximate attention scores. With the predicted sparse attention patterns represented as binary masks, we can save computations involved in full attention scores, $softmax$, and attention outputs.

Compared with static sparse attention methods, the DSA method is dynamic and naturally captures sparse attention patterns of different input sequences. We can observe important tokens that attract a large portion of attention weights from other tokens, similar to the global attention method [170, 64]. However, the positions of global tokens are input-dependent, and the DSA method can effectively identify such varieties, instead of relying on domain knowledge to predetermine certain global tokens in fixed positions. Compared with other low-rank approximation methods, the approximation in DSA is only for sparsity prediction without strict and static constraints on attention positions. Therefore, DSA can maintain the representation power of full attention while reducing

unnecessary attention weights.

Although DSA can save theoretical computations and maintain attention capability, achieving practical speedups and energy savings on real hardware is challenging. Section 7.5 discusses the implications of DSA on existing GPU architectures, and Section 7.6 discusses specialized hardware accelerators. The fine-grained dynamic sparsity as searched by DSA is extended to structural dynamic patterns, such as block-wise and vector-wise. We give the study on structural sparse patterns vs. attention's expressive power and explore the opportunities for dataflow optimization and data reuse from dynamic sparsity.

The evaluation in Section 7.4 shows that DSA can achieve 95% sparsity in attention weights without compromising model accuracy. Under this setting, the overall computational saving is up to 4.35× compared with full attention, while the sparsity prediction only introduces around 1.17% to 1.33% computational overhead. Experiments in Section 7.5 show that, on NVIDIA V100 GPU, applying vector-wise sparsity of 90% ratio on DSA delivers 1.15× speedup on attention score computation, 14.6× speedup on $softmax$ computation, and 1.94× speedup on attention output computation, with only 0.1% of accuracy loss.

Finally, through hardware specialization, we can explore architecture support to translate the theoretical savings to real performance speedup. Three system-level challenges are addressed through the proposed architecture design. Firstly, to support large Transformer models with various configurations, we need to effectively disassemble the algorithm and identify the essential components. Prior accelerators are designed for specific components like the self-attention block [104, 105]. Instead, the DOTA design, as discussed in Section 7.6, provides an efficient abstraction of the model and presents a unified architecture to support all components, achieving better area- and energy-efficiency. Besides, Section 7.6.1 further presents the analysis on different levels of parallelism on top of the proposed abstraction and present a scalable system architecture. Secondly,

low-precision computation is essential to the cost of the attention detection mechanism. To support multi-precision computations, DOTA uses a Reconfigurable Matrix Multiplication Unit (RMMU) that can be dynamically orchestrated to satisfy the throughput requirements of different computation precision (Section 7.6.2). Finally, when computing the attention output with the sparse attention graph, DOTA outperforms prior work by adopting the Token-Parallel dataflow with software-enabled workload balancing and hardware-enabled out-of-order execution.

7.2 Background and Motivation

Before going into details of the DSA method, this Section introduces the preliminaries of the standard attention mechanism used in vanilla Transformers. Then, the challenge of serving long sequences under the quadratic complexity of attention is discussed. Finally, as shows in this Section, redundancy exists in attentions and dynamic sparse patterns are naturally expressed in attention.

7.2.1 Preliminaries of Attention

The attention mechanism is the essential component of Transformers [4]. 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

$$Q, K, V = XW_Q, XW_K, XW_V \tag{7.1}$$

, 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

$$A = \phi(\frac{QK^\top}{\sqrt{d_k}}) \tag{7.2}$$

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

$$Z = AV. \tag{7.3}$$

Serving Transformer-based models is challenging when the input sequence length l is large. When using long sequences, computing Eq. (7.2) and Eq. (7.3) 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.

7.2.2 Intrinsic Sparsity in Attention Weights

A number of efficient Transformer variants have been proposed to mitigate the quadratic complexity of self-attention [65, 170, 64, 171]. One straightforward way to exploit the intrinsic redundancy in attention is forming sparse patterns as in

$$A = \phi(QK^\top - c(1 - M)), \tag{7.4}$$

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 de-

termine sparse patterns is through trainable masks. However, all these methods explore static or fixed sparse patterns, restricting viable attention connections.

7.2.3 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. (7.3). 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, as discovered in this work, sparse patterns in attention are inherently dynamic and data-dependent.

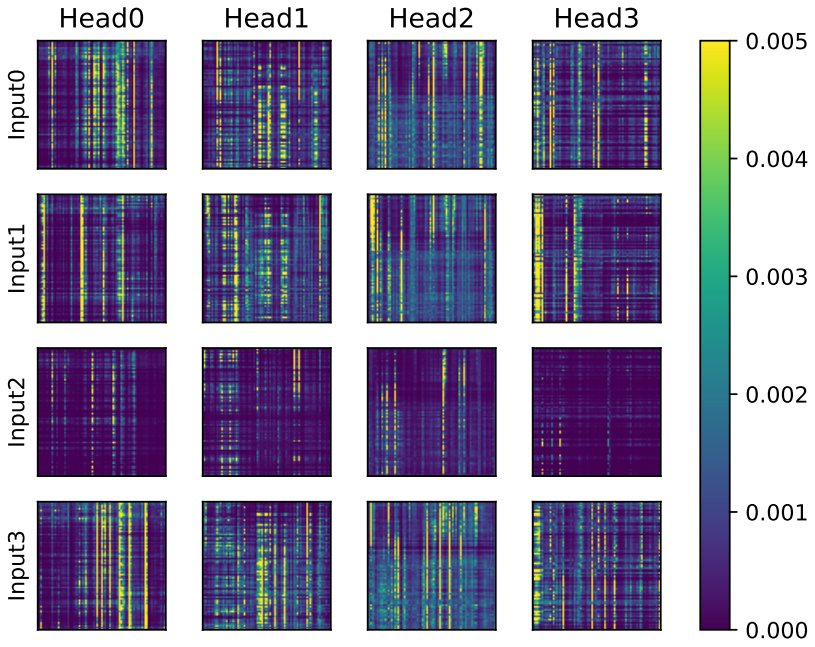


Figure 7.1: Visualization of attention weights from different inputs and attention heads. Only a small amount of attention weights are important. Note values > 0.005 are clamped to show as 0.005.

Here, the hypothesis is supported by showing the original attention weights matrix (after $softmax$ normalization) in Figure 7.1. The model used here is a vanilla Transformer and the benchmark is Text Classification from Google Long-Range Arena [172].

Figure 7.1 indicates that only a small amount of attention weights are with large magnitude and a significant portion is near zero. Note 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 7.1, 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.

7.3 Approach

Section 7.2 shows 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. The process of identifying sparse attention patterns can be formulated 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 presents *Dynamic Sparse Attention* (DSA) that exploits sparsity in attention weights to reduce computations. The principle of the DSA method is to effectively search for dynamic sparse patterns without enforcing strict and static constraints on attention while keeping the searching cost small. DSA leverages trainable approximation to predict

sparse attention patterns. As shown in Figure 7.2, DSA uses 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.

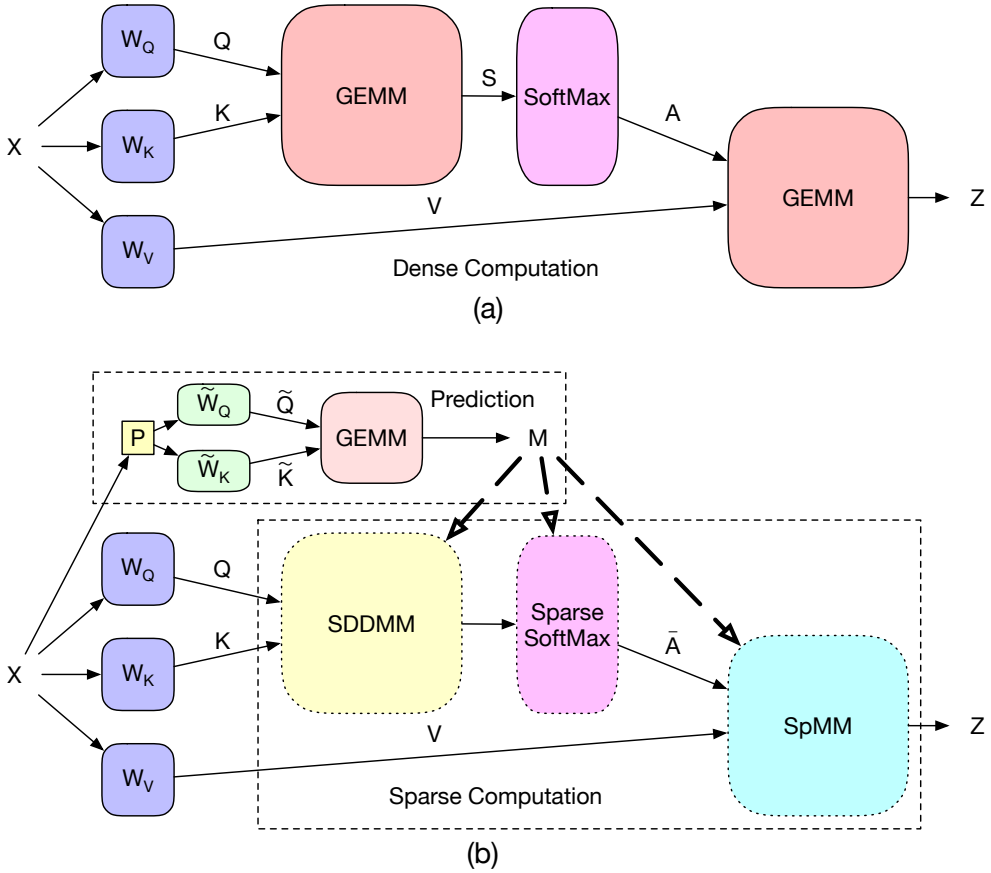


Figure 7.2: (a) Standard full attention; (b) Dynamic sparse attention with approximation-based prediction and sparse computation.

7.3.1 Design of Prediction Path

Attention scores are denoted as $S = QK^\top$ and omit the scaling factor for simplicity. As shown in Figure 7.2(a), two general matrix-matrix multiplication kernels (GEMM) and one $softmax$ kernel consume the majority of computations in self-attention. A pair

of approximate query and key transformations are constructed in the prediction path to compute for approximate score $\tilde{S}$, as in

$$\tilde{Q}, \tilde{K} = XP\tilde{W}_Q, XP\tilde{W}_K. \tag{7.5}$$

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, the approximate attention scores are as $\tilde{S} = \tilde{Q}\tilde{K}^\top$. 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. 7.3.

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

$$L_{MSE} = \frac{1}{B}||S - \tilde{S}||_2^2 = \frac{1}{B}||QK^\top - \tilde{Q}\tilde{K}^\top||_2^2 \tag{7.6}$$

where B is the mini-batch size.

Given the motivation of finding dynamic sparse patterns, the hypothesis of the DSA 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. The results of applying oracle sparse patterns, by directly dropping small-magnitude attention weights during inference without fine-tuning the model, can

further support the hypothesis. As listed in Table 7.1, around 90% (up to 97%) of small attention weights can be dropped with negligible accuracy loss.

Table 7.1: 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

7.3.2 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. A countermeasure is 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.

The computational graph and the loss function of the original model stay the same, 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, DSA jointly fine-tunes the model parameters and parameters of the prediction path as in

$$L = L_{Model} + \lambda L_{MSE} \tag{7.7}$$

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

DSA approximates the original attention score with a low-rank matrix $\tilde{S}$. When training the model with loss function in Eq. 7.6, 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. DSA brings two advantages. First, the rank of S will be automatically adjusted to tasks with different difficulty levels. Hence, DSA 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.

7.3.3 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, the traditional asymptotic analysis can be augmented 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 the settings, α is between 90% and 98% and the developed GPU kernels can achieve practical speedups. The assumption is that 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. DSA can also save the time of $softmax$ kernel with the same sparse attention patterns.

7.3.4 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. Section 7.6 introduces two types of architectures to support multi-precision computations.

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, DSA can extended to support structural sparsity that can improve the data reuse of both SDDMM and SpMM kernels. Customized kernels are developed that take advantage of the sparsity locality to improve kernel performance, achieving practical runtime speedup on NVIDIA V100 GPU. Also, the choice of structural sparsity pattern is demonstrated and 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, more optimization opportunities can be explored when considering the whole process as a two-step SDDMM-SpMM chain. Please refer to Section 7.6 for more architectural design details and experimental results.

7.4 Algorithmic Evaluation

This section evaluates the performance of DSA over representative benchmarks from Long-Range Arena [172]. First, the model accuracy results of DSA are compared with dense vanilla transformers and other efficient transformer models. Then, a sensitivity study is presented that using 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, the model efficiency of DSA is investigated by analyzing the computational cost (MACs) and relative energy consumption.

7.4.1 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 [173] API and optimized with just-in-time ($jax.jit$) compilation. DSA is implemented on top of the vanilla transformer provided by LRA

and compared 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 7.3. All the other model configurations are kept the same for a fair comparison.

The experiments incorporate three tasks from the LRA benchmark, including Text Classification, Document Retrieval, and Image Classification. The Long ListOps and Pathfinder tasks are excluded. Appendix C provides benchmark descriptions and experiment configurations.

7.4.2 Accuracy

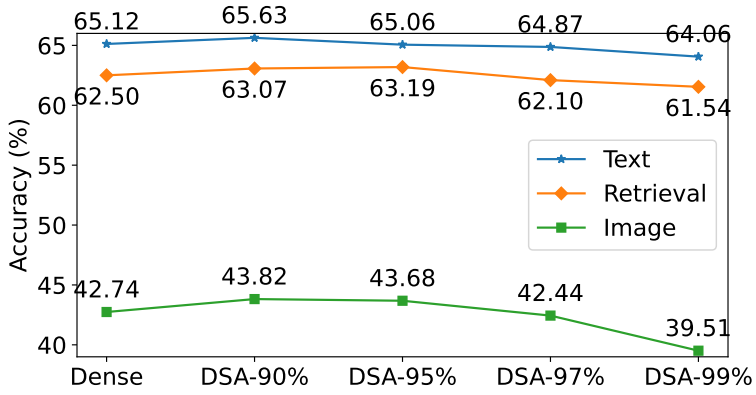


Figure 7.3: Overall model accuracy of DSA (**fine-tuned** from a pretrained checkpoint) compared with vanilla dense transformer.

Figure 7.3 presents the overall model accuracy of DSA on different LRA tasks. In this experiment, the DSA model is fine-tuned from a pretrained 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 in the DSA models. For instance, **DSA-90%** means that keeping 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.

Table 7.2: Accuracy of different Transformer models on the LRA benchmark suite [172]. For a fair comparison, we follow the instructions in LRA and train our model **from scratch**. **DSA-90%** uses projection scale $\sigma = 0.25$ and INT4 quantization.

Model	Text	Retrieval	Image	Avg
Transformer	65.12	<u>62.5</u>	42.74	56.79
Local Attention	52.98	53.39	41.46	50.89
Sparse Trans.	63.58	59.59	**44.24**	<u>55.80</u>
Longformer	62.85	56.89	42.22	53.99
Linformer	53.94	52.27	38.56	48.26
Reformer	56.10	53.40	38.07	49.19
Sinkhorn Trans.	61.20	53.83	41.23	52.09
Synthesizer	61.68	54.67	41.61	52.65
BigBird	64.02	59.29	40.83	54.71
Linear Trans.	**65.90**	53.09	42.34	53.78
Performer	65.40	53.82	42.77	54.00
DSA-90%	<u>65.62</u>	**63.07**	<u>43.75</u>	**57.48**

As shown in Figure 7.3, for all the evaluated tasks, dense transformer possesses a considerable amount of redundancy in the attention matrix under the long-sequence condition, which supports the previous claim in Section 7.2. 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.

Several training constraints during the experiments are used to fairly compare with other transformer variants provided by LRA. For example, instead of fine-tuning from a pretrained baseline, the DSA model used in the comparison is obtained from a randomly initialized model, i.e., training from scratch. We also fix other model parameters (e.g., number of layers, number of heads, hidden dimension) and training configurations (e.g., total training steps). The results are shown in Table 7.2. We use **DSA-90%** with quanti-

zation precision to be INT4, and let the random projection dimension scale $\sigma{=}k/d{=}0.25$. As we can see from the table, DSA achieves first-tier performance in all three tasks and delivers a leading average score on the LRA benchmarks.

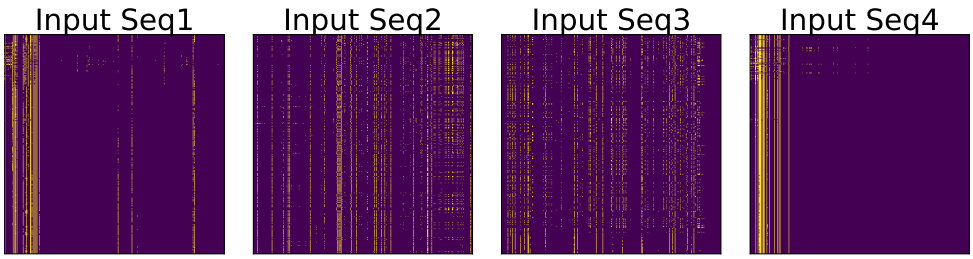


Figure 7.4: *Oracle* attention mask generated by *top-k* selection.

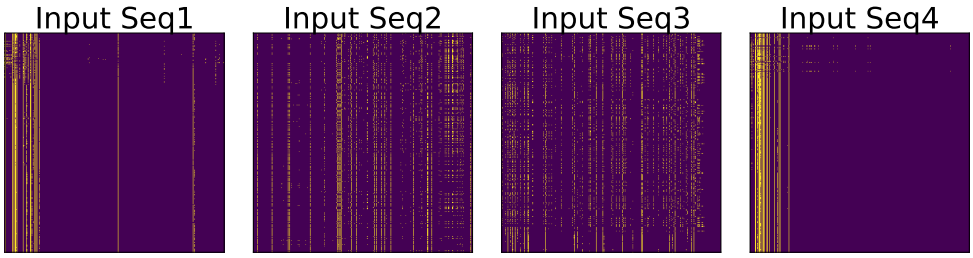


Figure 7.5: Sparse attention mask generated by DSA prediction.

This encouraging performance mainly comes from two aspects. Firstly, joint 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 7.4 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 7.5 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 the experiments, the prediction accuracy is around $85 \sim 95\%$.

Two cases on the Text Classification dataset are tested to make sure the high performance of DSA comes from the proposed approach rather than the task itself. First is applied with a 99% sparsity constraint on the vanilla transformer, but with a static local attention pattern. Second is used 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 the previous discussion.

7.4.3 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 can involve trainable parameters for prediction, we also need to 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 to use low-precision to reduce the computation overhead. Here presents the sensitivity results regarding different choices of the reduced dimension size and quantization precision.

Different sizes of k are used to evaluate the accuracy of **DSA-90%** on the LRA Text Classification task. Here, $\sigma = k/d \in (0, 1]$ represents 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 7.3, 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. Because we use predictor to indicate the positions of the important attention weights, while passing the

accurate attention weights to the output. Therefore, the predictor module can tolerate highly approximate computation as long as it can capture the relative importance in the attention matrix.

Table 7.3: 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

To further study the performance and the impact of the predictor, another experiment is conducted to sweep over different quantization precision, while fixing σ to be 0.25. As shown in Table 7.3, **DSA-90%** achieves good accuracy with 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 7.6, 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. The reason is that, 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

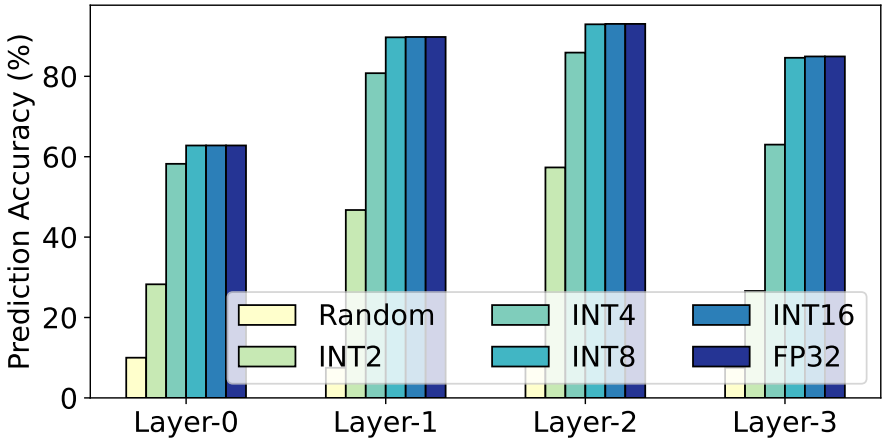


Figure 7.6: The prediction accuracy of DSA in a 4-layer **DSA-90%** model with different quantization precision.

the overall model accuracy. Finally, Figure 7.6 and Table 7.3 include a special case of randomly selecting 10% important positions. With this random mask applied to the model, the prediction accuracy is less than 10%, and overall model accuracy directly drops to 60.42%. This result supports the previous analysis.

7.4.4 Model Efficiency

As mentioned earlier, DSA has the potential to significantly reduce computation and memory consumption of the self-attention layer, which is especially beneficial for deploying a long sequence transformer model at inference time. While we acknowledge that the actual runtime performance and memory footprint are largely depending on the underlying hardware implementation, this subsection sheds light on this problem by quantitatively analyzing the cost of DSA.

What presented are the number of required MAC operations for each attention layer. The number of MAC is used as the computational cost metric because the majority of the operations in the self-attention layer are matrix multiplications. The total MAC operations can be divided into three parts: (1) Linear: General Matrix-matrix Multi-

plication(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 introduced earlier, the two GEMM operations in the part (2) scale quadratically with the sequence length, and these GEMM operations can be transformed to SDDMM and SpMM in DSA-enabled 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 7.7. 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.

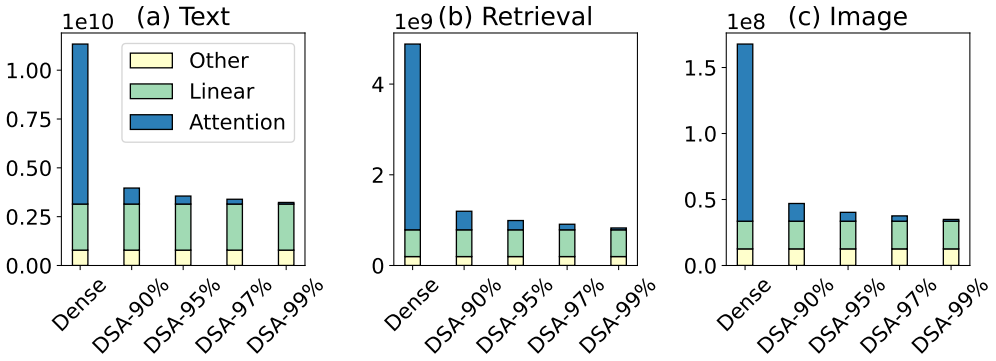


Figure 7.7: Computational cost measured in the number of MACs.

Note that the computation overhead of the prediction path for generating the sparsity mask is not included. 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. Therefore, the relative energy consumption is used to illustrate the overall cost of DSA-augmented attention.

Figure 7.8 shows the relative energy consumption of **DSA-95%** with $\sigma = 0.25$ 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 [174] 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.

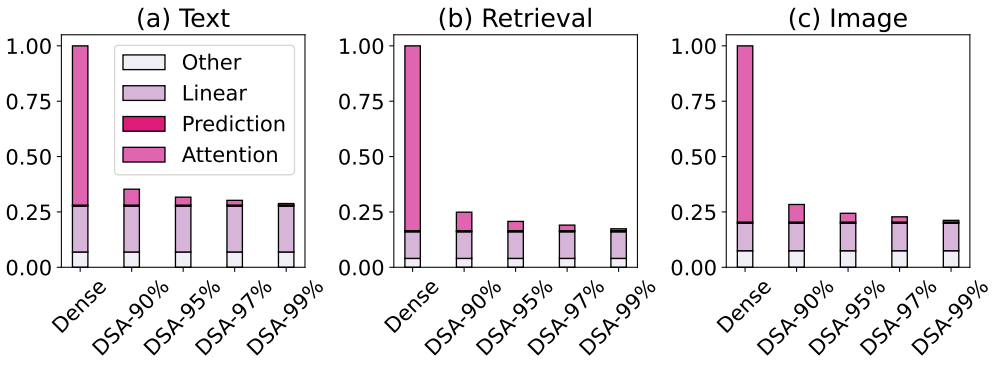


Figure 7.8: Relative energy consumption projected to vanilla transformer.

7.5 GPU Acceleration

In Section 7.4.4, we discussed 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 will dive deeper into this problem as we discuss the implementation of DSA on GPUs. Specifically, mapping DSA onto GPU architectures is a challenge, and DSA is flexible as enabling efficient algorithm-hardware co-designs.

Given the predicted sparse patterns, we can reformulate $QK^\top$ as the sampled dense dense matrix multiplication (SDDMM) and AV as the sparse matrix-matrix multipli-

cation (SpMM). Under fine-grained sparsity, a recent work [175] proposes SpMM and SDDMM kernel that outperforms dense GEMM kernel under $> 71\%$ and $> 90\%$ sparsity, respectively. Besides, *cusparse* [176] also achieves practical speedup at $> 80\%$ sparsity for single precision data. As we presented in Section 7.4, DSA can easily deliver a sparsity ratio of more than 90% with zero accuracy degradation, therefore enabling faster kernel implementations on GPUs.

7.5.1 Vector Sparse Patterns

While fine-grained sparse GPU kernels are able to outperform the dense counterparts on relatively high sparsity ratios, the speedup is significantly limited due to low data reuse. Moreover, when half precision (FP16) is used for computation, above fine-grained kernels can hardly compete with GEMM kernel, as NVIDIA Tensor Core provides much higher throughput for half precision matrix multiplication. Thus, the performance gain on sparse matrix multiplication can hardly mitigate the overhead of computing the prediction path in DSA, especially for half precision scenarios that commonly appeared at inference. To tackle this problem, structural dynamic sparsity can be introduced to the attention selection. Specifically, instead of selecting $top{-}k$ independent attention weights, we can enforce block-wise and vector-wise constraints. Also, trade-off can be made by adjusting the block size, as larger blocks deliver higher speedup but can potentially cause accuracy loss.

In this work, the experiments on vector sparsity use the Text Classification benchmark. As shown in Figure 7.9, column-vector sparse encoding is in use, 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 [177]. Table 7.4 gives the corresponding kernel speedup and model

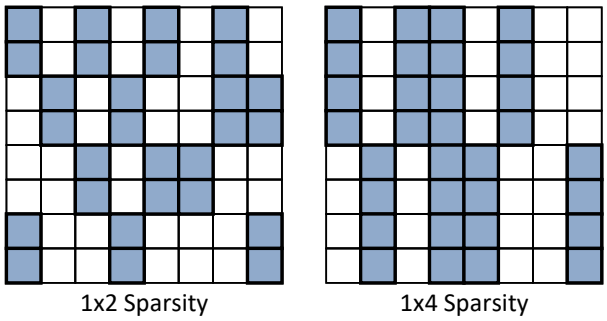


Figure 7.9: Column-vector sparse encoding [177]

accuracy under 90% sparsity ratio. The data type is FP16 for $1 \times 4/1 \times 8$ 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.

Table 7.4: Model accuracy and kernel speedup over *cuBLASHgemm*. Customized SDDMM/SpMM kernel for $1 \times 4/1 \times 8$ sparsity and reused the kernel in [175] for fine-grained sparsity. Experiments are done on NVIDIA V100 GPU.

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

To shed some light on the results, we can trace back to the visualizations of the attention matrix in Figure 7.1. 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 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 [64, 170], DSA illustrates the dynamic distribution which motivates us to propose the prediction path to efficiently locate these important connections.

7.5.2 Sparse Softmax Computation

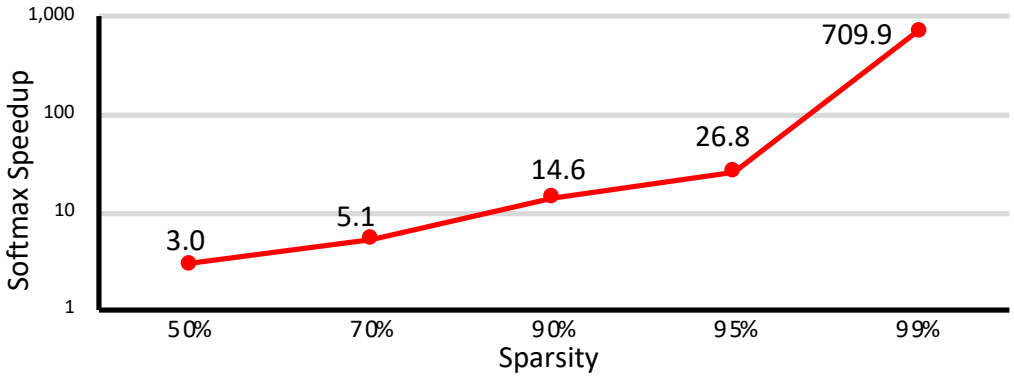


Figure 7.10: Speedup of softmax with different sparsity ratios.

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. The 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. The latency of the PyTorch-implemented softmax function is evaluated on NVIDIA V100 GPU. From the configuration in Text Classification Benchmark, batch size=16, $h = 4$, $l = 2000$, with different sparsity ratios. Figure 7.10 shows that the reduced softmax achieves $3.0 \sim 709.9\times$ speedup compared with dense softmax function.

7.5.3 Overall Performance Speedup

Here is demonstration of the overall performance of DSA on Nvidia V100 GPU. The results in Table 7.5 show that combining all proposed techniques, DSA can achieve 1.24x-1.97x speedup on self-attention block and 1.19-1.71x end-to-end speedup with negligible accuracy degradation. As we can see, while DSA does achieve practical speedup on existing GPUs, it is still hard to fully utilize DSA's computation savings. On one hand, GPU architectures like V100 do not support ultra low-precision computations like INT4 and INT2. Latest Ampere GPU provides lower precision computation, which shows DSA's impact on future GPUs. On the other hand, SpMM and SDDMM operators have sub-optimal performance on GPUs, which results in limited performance speedup. Therefore, we further discuss the opportunities to apply hardware specialization to DSA.

Table 7.5: End-to-End Performance Speedup

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

7.6 Hardware Specialization

While adding structural constraints can potentially benefit GPU kernel implementation, the expressive power of the model is still inevitably affected. For instance, as shown in Table 7.4, the 4×1 vector encoding achieves comparable accuracy with full-attention, but is lower than the accuracy of using fine-grained sparsity under the same sparsity ratio. Thus, an alternative approach is to use hardware specialization to fully exploit the potential saving from DSA.

Here presents the design of DOTA, Detect and Omit attention sparsity in Trans-

former Accelerator, which is capable of performing scalable Transformer inference by efficiently utilizing DSA. This design specifically addresses 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. The abstraction of the model helps designing a scalable and unified architecture for different Transformer layers, achieving good area- and power-efficiency. (Section 7.6.1). Second, apart from implementing normal precision arithmetic, DOTA also needs to support low-precision computations required by the attention detection. Instead of separately implementing all the arithmetic precision, a reconfigurable design would be preferred as it can dynamically balance the computation throughput of multi-precision computations. (Section 7.6.2). Finally, to efficiently compute over the detected attention graph, we should tackle the workload imbalance and irregular memory access caused by attention sparsity (Section 7.6.3).

7.6.1 Overall System Architecture

Figure 7.11 illustrates 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.

Each encoder can be splited 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 mul-

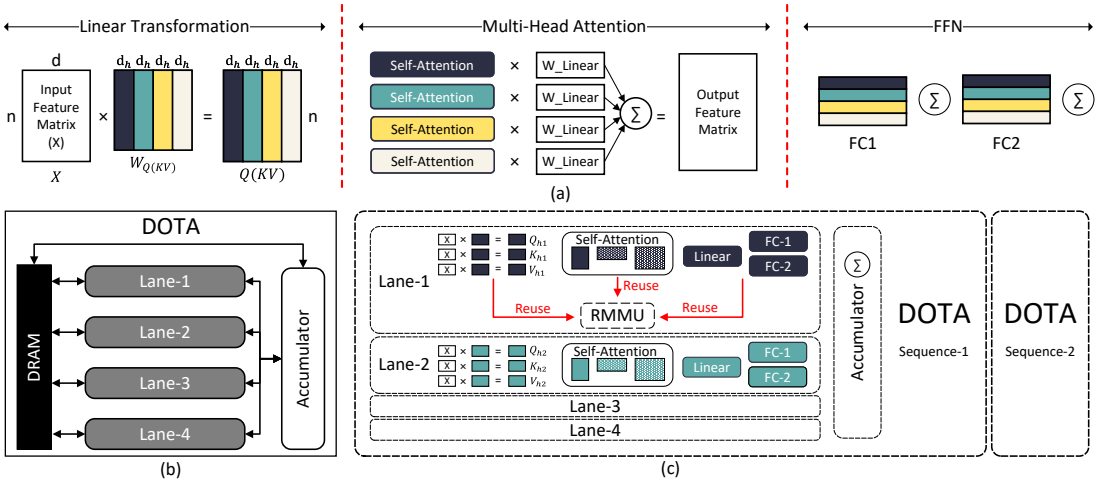


Figure 7.11: DOTA system design. (a) The abstraction of a single encoder block. We divide each encoder into three sequential stages. Each stage contains multiple GEMM operations that can be further cut into chunks (represented by different colors) and mapped to different compute Lanes. (b) Overall system 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).

tiple chunks and processed in parallel. Therefore, as shown in Figure 7.11, the DOTA accelerator has four compute lanes, and each Lane is dedicated 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. The chunk's size is 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.

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 can 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. One DOTA accelerator has four lanes because four is the least common multiple of the attention head numbers across all the benchmarks evaluated. More Lanes can be implemented for higher chunk-level parallelism.

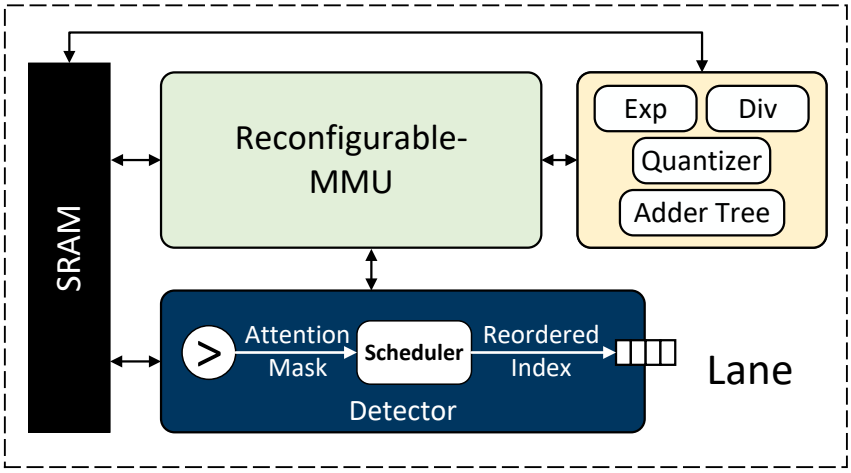


Figure 7.12: Architecture of each compute Lane.

Inside each Lane, as shown in Figure 7.12, there is an SRAM buffer, a Reconfigurable Matrix Multiplication Unit (RMMU), a Detector for attention selection, and a Multi-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 (INT2/4) 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 7.6.3).

7.6.2 Reconfigurable Matrix Multiplication Unit Design

As presented in Figure 7.12, 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.

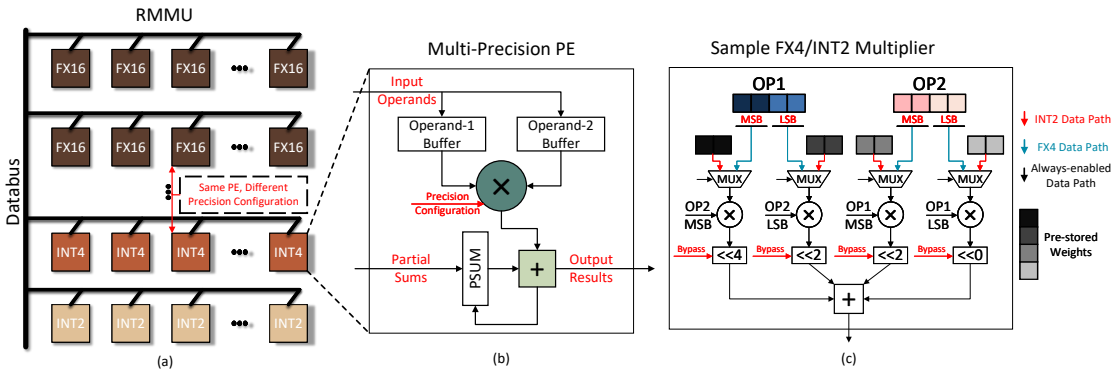


Figure 7.13: Design of the Reconfigurable Matrix Multiplication Unit. (a) RMMU is composed of a 2D PE array, where each row can be configured to a specific computation precision. (b) Each PE is a multi-precision MAC unit. (c) A sample FX4/IN2 multi-precision multiplier. The key is to build up high precision 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.

To tackle this problem, the RMMU design is shown in Figure 7.13. The key idea is to design computation engine with configurable precision. As we can see from Figure 7.13,

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.

The multi-precision multiplier design is 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 [178]. Without loss of generality, the implementation of an FX4/INT2 multiplier is shown in Figure 7.13 (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). This problem is addressed 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, the multiplier is utilized as a tiny input-stationary MAC unit which can perform 4 INT2 multiplications and accumulations each cycle.

To summarize, multi-precision PEs are used in the RMMU and ensure scalable com-

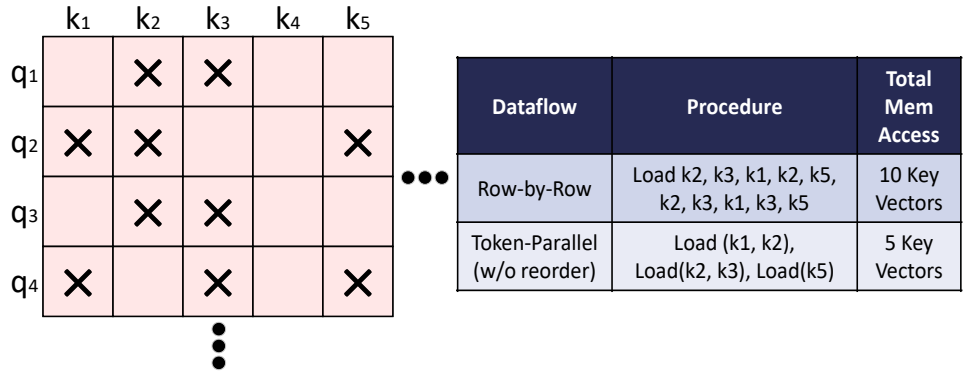
putation throughput when using low-precision data. The final design implements FX-16 multiplier built up from low-precision INT multipliers as discussed above.

7.6.3 Token-parallel Sparse Attention Computation

After RMMU generates estimated attention scores, the Detector unit selects important attention connections. Specifically, as depicted in Figure 7.12, 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. This is named as token-parallel dataflow, which can improve Key/Value data reuse and reduce total memory access. This subsection uses three different examples to demonstrate the benefits, the challenges, and the solutions to compute the attention output with the detected attention graph and Token parallelism.

Token-Parallel Dataflow. As shown by the example in Figure 7.14, 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 7.14, 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 can observe similar locality in real 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.



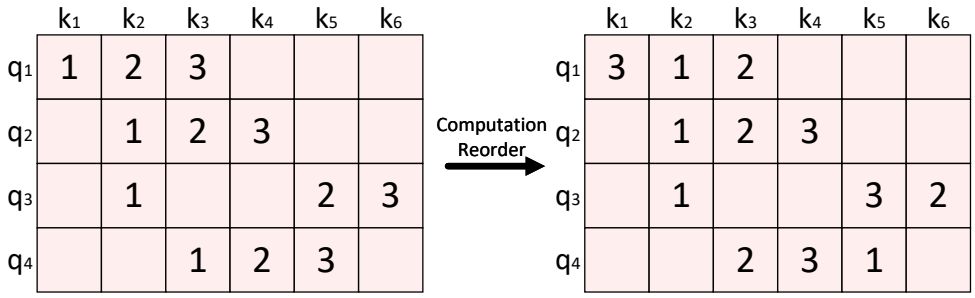
Dataflow	Procedure	Total Mem Access
Row-by-Row	Load k2, k3, k1, k2, k5, k2, k3, k1, k3, k5	10 Key Vectors
Token-Parallel (w/o reorder)	Load (k1, k2), Load(k2, k3), Load(k5)	5 Key Vectors

Figure 7.14: Token-level parallelism reduces key/value vector memory access.

Workload Balancing. One challenge of parallel token processing is the workload imbalance issue among different rows. Figure 7.14 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, this problem is tackled directly from algorithm perspective without affecting the underlying hardware. Specifically, a constraint is added to enforce all the rows in the attention matrix to have the same number of selected attention connections.

Out-of-Order Execution. Finally, hardware-enabled out-of-order execution is proposed to further improve key/value reuse and reduce total memory access. As shown in Figure 7.15, suppose all four queries have balanced workload and are processed in parallel. The left-to-right computation order firstly computes (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.



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

To address this problem, a locality-aware scheduling algorithm is proposed to reorder the computation of each query. As shown in Figure 7.15 and 7.16, we can 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. Doing so need to first look for keys that belong to q_4 alone. If not found, moving 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 can go to the second best choice, which is k_5. Thus, in the first round, 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 4. 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.

Algorithm 4: 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.
```

A Scheduler is designed to implement the scheduling algorithm. As shown in Figure 7.16, 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 Figure 7.15, '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, token-level parallelism is explored 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 7.8. (SoftMax is optional.)

$$O = (Q * K) * V = A * V \tag{7.8}$$

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 [179].

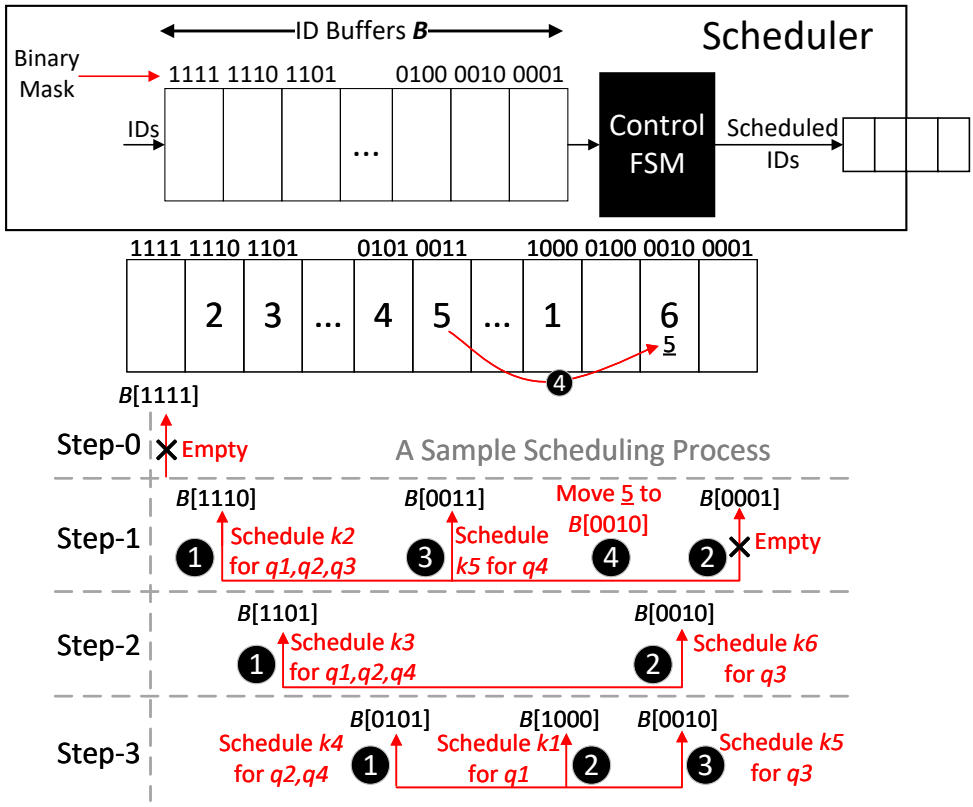


Figure 7.16: Design of the Scheduler and the scheduling process of Figure 7.15.

7.6.4 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. A custom simulator is developed 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 need to make sure the SRAM bandwidth meets the need of the computation-bounded layers, while leaving embedding and decoder to be memory-bounded.

7.6.5 Performance Speedup

Appendix C includes the hardware evaluation methodology. Figure 7.17 presents the speedup of DOTA over the baselines. Both stand along attention block and the end-to-end performance improvements are evaluated. Two versions of DOTA are provided 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, the original settings [105] are used, and the retention ratio is 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

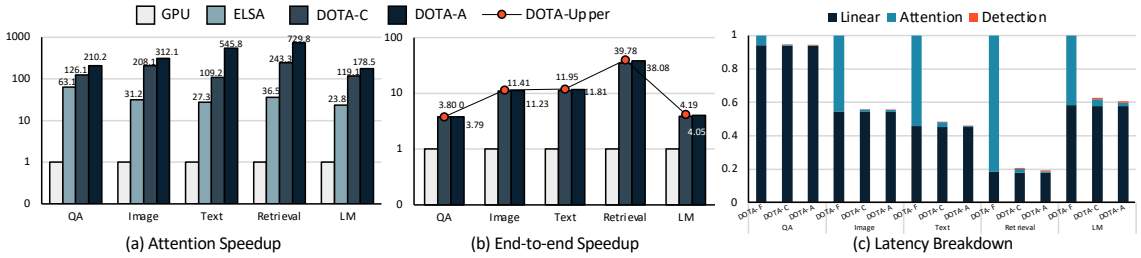


Figure 7.17: (a) Speedup of DOTA over GPU and ELSA on attention block. (b) End-to-end speedup over GPU. Red dots indicate the theoretical performance upper-bound of an accelerator. (c) Normalized latency breakdown of DOTA. DOTA-F means to compute the *Full* attention graph with DOTA without detection 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%.

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. Another baseline is included by assuming the accelerator always works at its peak throughput, and the attention computation has a negligible cost. Combining this peak throughput assumption and Amdahl's law [180], 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 7.17 (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 7.17 (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.

7.7 Conclusion

This Chapter presents Dynamic Sparse Attention (DSA), a novel method that exploits dynamic sparse patterns in attention to reduce computational cost when serving Transformers. Specifically, it is shown that the DSA method can achieve up to 95% attention sparsity without model inference quality loss. Other than prior art that uses static sparse patterns in attention, DSA explores dynamic sparse patterns that are inherent in attention when processing different input sequences. Instead of replacing standard attention with other variants such as low-rank approximation methods, DSA can augment standard attention with a prediction path as the means to locate dynamic sparsity. On one hand, attention approximation can be very efficient when only used for sparsity prediction. On the other hand, the expressive power of full attention is preserved as the important attention weights from full attention are effective in model inference. Experimental results on the LRA benchmark demonstrate superior performance and model efficiency of DSA. Furthermore, this Chapter demonstrates the potential of using DSA to improve hardware performance and efficiency. With customized kernel design and

structural sparsity, DSA delivers practical speedup on GPU. The algorithm benefit can be further exploited with specialized architecture, as the hardware can fully benefit from low-precision prediction, fine-grained sparse computation, and data locality.

Chapter 8

Conclusion

This dissertation presents how to explore the elasticity in Machine Learning (ML) processing and hardware architectures to enhance efficiency.

8.1 Summary of Contributions

By exploring the elasticity of approximate-accurate activations, Chapter 3 presents the dual-module inference method that is applicable to various types of neural networks. For the memory-bound RNNs, with overall memory accesses reduced by 40% on a commodity CPU-based server platform, the method can achieve 1.5× to 1.7× wall-clock time speedup with negligible impact on model quality. In addition, the method can reduce the operations of the compute-bound CNNs by 3.0×, with only a 0.5% accuracy drop. The high-level view of DMI is using learned auxiliary modules for speculative computation skipping. Future large-scale ML models could benefit from such a paradigm where only a subset of components are activated during inference.

As discussed in Chapter 4, the proposed dynamic sparse graph (DSG) approach, supported by dimension-reduction search and double-mask selection, can achieve 1.7-

4.5× memory compression and 2.3-4.4× computation reduction with minimal accuracy loss across a set of CNNs. This approach is distinct from static model compression techniques on permanent weight pruning since we never prune the graph but activate partially. Therefore, we can maintain the model's expressive power as much as possible. This work simultaneously pioneers the approach towards efficient online training and offline inference, which can benefit the DL in both the cloud and the edge.

As proposed in Chapter 5, with acceptable model inference quality degradation, dual-module accelerator design can achieve 2.2× speedup and 2.0× energy reduction. Leveraging heterogeneous computing capabilities of modern hardware is a critical question to address, and our design proposes an orchestrated execution at the level of specialized hardware. Future ML accelerators can embrace the idea of leveraging heterogeneous computing units tailored for components that have different computational requirements.

Chapter 6 presents a software-hardware co-design that features: a novel screening method to reduce the computation and memory consumption by efficiently approximating the classification output; a specialized NMP architecture design for both the screening phase and the classification phase. Overall, the approximate screening method achieves 7.3× speedup over the CPU baseline, and the ENMC architecture design further improves the performance by 7.4× and demonstrates 2.7× speedup compared with the state-of-the-art NMP baseline.

Finally, in Chapter 7, Dynamic Sparse Attention (DSA) is proposed to accelerate long-range Transformer Neural Networks. The DSA method can achieve up to 95% attention sparsity without model inference quality loss. Instead of using static sparse patterns, DSA explores explores dynamic sparse patterns that are inherent in attention when processing different input sequences. Experimental results demonstrate superior performance and model efficiency of DSA. Furthermore, the potential of using DSA to improve hardware performance and efficiency is demonstrated. With customized kernel

design and structural sparsity, DSA delivers practical speedup on GPU. The algorithm benefit can be further exploited with specialized architecture, as the hardware can fully benefit from low-precision prediction, fine-grained sparse computation, and data locality.

8.2 Future Research

The design principles of elastic processing and architectures can be expanded to three aspects in the future: *design metrics* (e.g., robustness, privacy, and environmental impact), *computing paradigms* (e.g., near-memory, in-storage, and network-centric computing), and *application domains* (e.g., autonomous, graph processing, and biomedical).

8.2.1 Co-Designing Robustness and Efficiency

Computer systems are vulnerable to erroneous inputs and execution-time errors. Specifically, for ML applications, adversarial examples can fool the models, and device-level faults can degrade the accuracy or crash the system in reliability-critical scenarios. While many studies have been proposed to improve the robustness of ML systems, designing defense and detection mechanisms is often separated from architectural properties. Using robust countermeasures at the cost of performance and efficiency degradation is not favorable in real-time applications and resource-limited platforms. Elastic processing can be expanded to jointly design robust ML methods and hardware architectures in several potential ways: 1) reusing approximate computing units designed for efficiency to enhance robustness from adversarial attack; 2) architectural support for detecting adversarial examples without hurting performance; 3) incorporating redundancy as to improve system reliability while elastically removing unnecessary computations.

8.2.2 Incorporating Emerging Computing Paradigms

For data-intensive workloads, such as large-scale classification training, embedding operation, and bioinformatics, accessing the memory hierarchy for data is the bottleneck of system performance and energy consumption. We are in search of offloading data-intensive workloads to smart storage systems with Near-Data Processing (NDP) technologies. However, native NDP design cannot support the computational complexity of data-intensive workloads due to the area and power limitations. Leveraging the design principles in elastic processing and near-memory processing, future research can bridge the gap between data-intensive workloads and NDP-enabled storage systems.

A typical large-scale ML training system consists of CPUs and accelerators connected by traditional PCIe interconnects. The communication limits the training throughput due to high latency from passive network fabrics, inflexible network topology, and coherence management. Building from the work on CNN training with dynamic sparse graph in Chapter 4, future research could explore elastic communication methods in large-scale training and seek co-designs of active network fabrics with caching and computing capabilities and specialized network for adaptive communication with topology reconfiguration.

8.2.3 Expanding Application Domains

Future research can explore elastic processing and domain-specific architectures in a wide range of application domains, including robotics, graph analytics, and bioinformatics. While specialization plays a key role in scaling performance and efficiency, non-recurring engineering (NRE) cost, development time, and programmability remain challenging. On one hand, highly specialized ASIC designs can deliver the best performance and efficiency, but lack programmability and flexibility. Once the algorithms

evolve, the ASIC designs become obsolete. On the other hand, general-purpose processors and programmable devices have the flexibility but cannot meet performance and efficiency requirements. Thus, we need to balance specialization and generality in next-generation architectures.

One potential direction is on software/hardware co-optimization towards both efficiency and flexibility, as this raises many challenges. For example, how can we achieve near ASIC performance and efficiency while having flexibility? How can we enable rapid design space exploration in accelerators? Can we have abstractions for specialized components in an application domain such that extended accelerator designs can deliver high efficiency with full programming capability? How can we enable automated design flow for accelerators and jointly search optimization space in both software and hardware? Future accelerator design to be essentially parallel programming where applications and algorithms will be adapted and mapped to hardware simultaneously.

Cross-domain adaptation can potentially reuse past specialized designs for new problems and new algorithms with much reduced cost and development time than from-scratch approaches. Recent integration technology advancements such as multi-chip-module and fine-grained 3D stacking could facilitate cross-domain design reuse. This agile co-design paradigm could enable efficient and flexible next-generation intelligent applications

Appendix A

Supplemental Materials for Dynamic Sparse Graph

A.1 Proof of the Dimension-reduction Search for Inner Product Preservation

Theorem 1. Given a set of N points in $\mathbb{R}^d$ (i.e. all $\mathbf{X}_i$ and $\mathbf{W}_j$), and a number of $k > O(\frac{log(N)}{\epsilon^2})$, there exist a linear map $f : \mathbb{R}^d \Rightarrow \mathbb{R}^k$ and a $\epsilon_0 \in (0, 1)$, for $0 < \epsilon \leq \epsilon_0$ we have

$$P[\ |\langle f(\mathbf{X}_i), f(\mathbf{W}_j)\rangle - \langle \mathbf{X}_i, \mathbf{W}_j\rangle| \leq \epsilon\ ] \geq 1 - O(\epsilon^2). \tag{A.1}$$

for all $\mathbf{X}_i$ and $\mathbf{W}_j$.

Proof. According to the definition of inner product and vector norm, any two vectors $\mathbf{a}$ and $\mathbf{b}$ satisfy

$$\begin{cases} \langle \mathbf{a}, \mathbf{b}\rangle = (\|\mathbf{a}\|^2 + \|\mathbf{b}\|^2 - \|\mathbf{a} - \mathbf{b}\|^2)/2 \\ \langle \mathbf{a}, \mathbf{b}\rangle = (\|\mathbf{a} + \mathbf{b}\|^2 - \|\mathbf{a}\|^2 - \|\mathbf{b}\|^2)/2 \end{cases}. \tag{A.2}$$

It is easy to further get

$$\langle \mathbf{a}, \mathbf{b} \rangle = (\|\mathbf{a}+\mathbf{b}\|^2 - \|\mathbf{a}-\mathbf{b}\|^2)/4. \tag{A.3}$$

Therefore, we can transform the target in equation (A.1) to

$$\begin{aligned} &| \langle f(\mathbf{X}_i), f(\mathbf{W}_j) \rangle - \langle \mathbf{X}_i, \mathbf{W}_j \rangle | \\ =& | \|f(\mathbf{X}_i)+f(\mathbf{W}_j)\|^2 - \|f(\mathbf{X}_i)-f(\mathbf{W}_j)\|^2 - \|\mathbf{X}_i+\mathbf{W}_j\|^2 + \|\mathbf{X}_i-\mathbf{W}_j\|^2 |/4 \\ \leq& | \|f(\mathbf{X}_i)+f(\mathbf{W}_j)\|^2 - \|\mathbf{X}_i+\mathbf{W}_j\|^2 |/4 + | \|f(\mathbf{X}_i)-f(\mathbf{W}_j)\|^2 - \|\mathbf{X}_i-\mathbf{W}_j\|^2 |/4, \end{aligned} \tag{A.4}$$

which is also based on the fact that $|u-v| \leq |u|+|v|$. Now recall the definition of random projection in equation (5) of the main text

$$f(\mathbf{X}_i) = \frac{1}{\sqrt{k}}\mathbf{R}\mathbf{X}_i \in \mathbb{R}^k, \quad f(\mathbf{W}_j) = \frac{1}{\sqrt{k}}\mathbf{R}\mathbf{W}_j \in \mathbb{R}^k. \tag{A.5}$$

Substituting equation (A.5) into equation (A.4), we have

$$\begin{aligned} &| \langle f(\mathbf{X}_i), f(\mathbf{W}_j) \rangle - \langle \mathbf{X}_i, \mathbf{W}_j \rangle | \\ \leq& | \|\tfrac{1}{\sqrt{k}}\mathbf{R}\mathbf{X}_i + \tfrac{1}{\sqrt{k}}\mathbf{R}\mathbf{W}_j\|^2 - \|\mathbf{X}_i+\mathbf{W}_j\|^2 |/4 + | \|\tfrac{1}{\sqrt{k}}\mathbf{R}\mathbf{X}_i - \tfrac{1}{\sqrt{k}}\mathbf{R}\mathbf{W}_j\|^2 - \|\mathbf{X}_i-\mathbf{W}_j\|^2 |/4 \\ =& | \|\tfrac{1}{\sqrt{k}}\mathbf{R}(\mathbf{X}_i+\mathbf{W}_j)\|^2 - \|\mathbf{X}_i+\mathbf{W}_j\|^2 |/4 + | \|\tfrac{1}{\sqrt{k}}\mathbf{R}(\mathbf{X}_i-\mathbf{W}_j)\|^2 - \|\mathbf{X}_i-\mathbf{W}_j\|^2 |/4 \\ =& | \|f(\mathbf{X}_i+\mathbf{W}_j)\|^2 - \|\mathbf{X}_i+\mathbf{W}_j\|^2 |/4 + | \|f(\mathbf{X}_i-\mathbf{W}_j)\|^2 - \|\mathbf{X}_i-\mathbf{W}_j\|^2 |/4 \end{aligned} \tag{A.6}$$

Further recalling the norm preservation in equation (3) of the main text: there exist a linear map $f : \mathbb{R}^d \Rightarrow \mathbb{R}^k$ and a $\epsilon_0 \in (0,1)$, for $0 < \epsilon \leq \epsilon_0$ we have

$$P[\ (1-\epsilon)\|\mathbf{Z}\|^2 \leq \|f(\mathbf{Z})\|^2 \leq (1+\epsilon)\|\mathbf{Z}\|^2\ ] \geq 1 - O(\epsilon^2). \tag{A.7}$$

Substituting the equation (A.7) into equation (A.6) yields

$$\begin{aligned} P[\ |\ \|f(\mathbf{X}_i+\mathbf{W}_j)\|^2-\|\mathbf{X}_i+\mathbf{W}_j\|^2\ |/4+|\ \|f(\mathbf{X}_i-\mathbf{W}_j)\|^2-\|\mathbf{X}_i-\mathbf{W}_j\|^2\ |/4... \\ \leq \tfrac{\epsilon}{4}(\|\mathbf{X}_i+\mathbf{W}_j\|^2+\|\mathbf{X}_i-\mathbf{W}_j\|^2)=\tfrac{\epsilon}{2}(\|\mathbf{X}_i\|^2+\|\mathbf{W}_j\|^2)\ ]... \\ \geq P(\ |\ \|f(\mathbf{X}_i+\mathbf{W}_j)\|^2-\|\mathbf{X}_i+\mathbf{W}_j\|^2\ |/4\leq \tfrac{\epsilon}{4}\|\mathbf{X}_i+\mathbf{W}_j\|^2\ )... \\ \times P(\ |\ \|f(\mathbf{X}_i-\mathbf{W}_j)\|^2-\|\mathbf{X}_i-\mathbf{W}_j\|^2\ |/4\leq \tfrac{\epsilon}{4}\|\mathbf{X}_i-\mathbf{W}_j\|^2\ )... \\ \geq [1-O(\epsilon^2)]\cdot[1-O(\epsilon^2)]=1-O(\epsilon^2). \end{aligned} \tag{A.8}$$

Combining equation (A.6) and (A.8), finally we have

$$P[\ |\ \langle f(\mathbf{X}_i),f(\mathbf{W}_j)\rangle-\langle\mathbf{X}_i,\mathbf{W}_j\rangle\ |\leq\tfrac{\epsilon}{2}(\|\mathbf{X}_i\|^2+\|\mathbf{W}_j\|^2)\ ]\geq 1-O(\epsilon^2)\ . \tag{A.9}$$

It can be seen that, for any given $\mathbf{X}_i$ and $\mathbf{W}_j$ pair, the inner product can be preserved if the ϵ is sufficiently small. Actually, previous work [141, 121] discussed a lot on the random projection for various big data applications, here we re-organize these supporting materials to form a systematic proof. We hope this could help readers to follow this paper. In practical experiments, there exists a trade-off between the dimension reduction degree and the recognition accuracy. Smaller ϵ usually brings more accurate inner product estimation and better recognition accuracy while at the cost of higher computational complexity with larger k, and vice versa. Because the $\|\mathbf{X}_i\|^2$ and $\|\mathbf{W}_j\|^2$ are not strictly bounded, the approximation may suffer from some noises. Anyway, from the abundant experiments in this work, the effectiveness of our approach for training dynamic and sparse neural networks has been validated.

Algorithm 5: DSG training

Data: A mini-batch of inputs & targets ($\mathbf{X}_0$, $\mathbf{X}^*$), previous weights $\mathbf{W}^t$, previous BN parameters θ^t.

Result: Update weights $\mathbf{W}^{t+1}$, update BN parameters θ^{t+1}.

Random projection: $f(\mathbf{W}_k^t) \Leftarrow \mathbf{W}_k^t$;

Step 1. Forward Computation;

for *k=1 to L* **do**

  if *k<L* **then**

    Projection: $f(\mathbf{X}_{k-1}) \Leftarrow \mathbf{X}_{k-1}$;

    Generating $Mask_k$ via dimension-reduction search according to $f(\mathbf{X}_{k-1})$ and $f(\mathbf{W}_k^t)$;

    $\mathbf{S}_k \Leftarrow \varphi[\ Mask_k(\mathbf{X}_{k-1}\mathbf{W}_k^t)\ ]$;

    $\mathbf{X}_k \Leftarrow Mask_k[\ BN(\mathbf{S}_k, \theta_k^t)\ ]$;

  else

    $\mathbf{X}_L \Leftarrow linear(\mathbf{X}_{L-1}\mathbf{W}_L^t)$;

Step 2. Backward Computation;

Compute the gradient of the output layer $\mathbf{G}_{\mathbf{X}_L} = \frac{\partial C(\mathbf{X}_L, \mathbf{X}^*)}{\partial \mathbf{X}_L}$;

for *k=L to 1* **do**

  if *k==L* **then**

    $\mathbf{G}_{\mathbf{X}_{L-1}} \Leftarrow Mask_{k-1}(\mathbf{G}_{\mathbf{X}_L}(\mathbf{W}_L^t)^T)$;

    $\mathbf{G}_{\mathbf{W}_L} \Leftarrow \mathbf{G}_{\mathbf{X}_L}^T \mathbf{X}_{L-1}$;

  else

    $(\mathbf{G}_{\mathbf{S}_k}, \mathbf{G}_{\theta_k}) \Leftarrow Mask_k[\ BN\_grad(\mathbf{G}_{\mathbf{X}_k}, \mathbf{S}_k, \theta_k^t)\ ]$;

    $\mathbf{G}_{\mathbf{W}_k} \Leftarrow (\mathbf{G}_{\mathbf{S}_k} \odot \varphi\_grad)^T \mathbf{X}_{k-1}$;

    if *k>1* **then**

      $\mathbf{G}_{\mathbf{X}_{k-1}} \Leftarrow Mask_{k-1}[\ (\mathbf{G}_{\mathbf{S}_k} \odot \varphi\_grad)(\mathbf{W}_k^t)^T\ ]$;

Step 3. Parameter Update;

for *k=1 to L* **do**

  $\mathbf{W}_k^{t+1} \Leftarrow Optimizer(\mathbf{W}_k^t, \mathbf{G}_{\mathbf{W}_k})$;

  $\theta_k^{t+1} \Leftarrow Optimizer(\theta_k^t, \mathbf{G}_{\theta_k})$;

A.2 Implementation and overhead

The training algorithm for generating DSG is presented in Algorithm 5. The generation procedure of the critical neuron mask based on the virtual activations estimated in the low-dimensional space is presented in Figure A.1, which is a typical top-k search. The k value is determined by the activation size and the desired sparsity γ. To reduce the search cost, we calculate the first input sample $X(1)$ within the current mini-batch and then conduct a top-k search over the whole virtual activation matrix for obtaining the top-k threshold under this sample. The remaining samples share the top-k threshold from the first sample to avoid costly searching overhead. At last, the overall activation mask is generated by setting the mask element to one if the estimated activation is larger than the top-k threshold and setting others to zero. In this way, we greatly reduce the search cost. Note that, for the FC layer, each sample $X(i)$ is a vector.

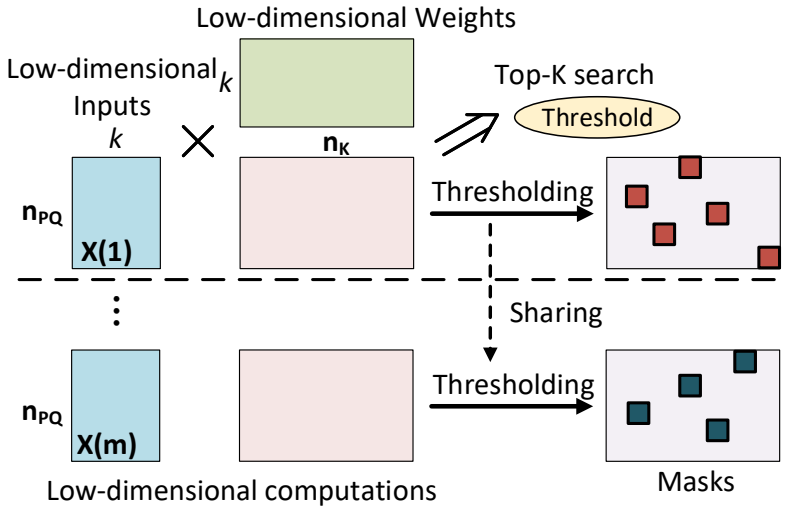


Figure A.1: Selection mask generation: using a top-k search on the first input sample $X(1)$ within each mini-batch to obtain a top-k threshold which is shared by the following samples. Then, we apply thresholding on the whole output activation tensor to generate the importance mask for the same mini-batch.

Furthermore, we investigate the influence of the ϵ on the computation cost of dimension-

Table A.1: Computational complexity of dimension-reduction search. MMACs denotes mega-MACs and BL denotes baseline.

Layers	Dimension					Operations (MMACs)				
n_{PQ}, n_{CRS}, n_K	BL	0.3	0.5	0.7	0.9	BL	0.3	0.5	0.7	0.9
1024, 1152, 128	1152	539	232	148	119	144	67.37	29	18.5	14.88
256, 1152, 256	1152	616	266	169	136	72	38.5	16.63	10.56	8.5
256, 2304, 256	2304	616	266	169	136	144	38.5	16.63	10.56	8.5
64, 2304, 512	2304	693	299	190	154	72	21.65	9.34	5.94	4.81
64, 4608, 512	4608	693	299	190	154	144	21.65	9.34	5.94	4.81

reduction search for importance estimation. We take several layers from the VGG8 on CIFAR10 as a case study, as shown in Table A.1. With ϵ larger, the dimension-reduction search can achieve lower dimension with much fewer operations. The average reduction of the dimension is 3.6x ($\epsilon = 0.3$), 8.5x ($\epsilon = 0.5$), 13.3x ($\epsilon = 0.7$), and 16.5x ($\epsilon = 0.9$). The resulting operation reduction is 3.1x, 7.1x, 11.1x, and 13.9x, respectively.

A.3 Convergence Analysis

One interesting question is that whether DSG slows down the training convergence or not, which is answered by Figure A.2. According to Figure A.2(a)-(b), the convergence speed under DSG constraints varies little from the vanilla model training. This probably owes to the high fidelity of inner product when we use random projection to reduce the data dimension. Figure A.2(c) visualizes the distribution of the pairwise difference between the original high-dimensional inner product and the low-dimensional one for the CONV5 layer of VGG8 on CIFAR10. Most of the inner product differences are around zero, which implies an accurate approximation capability of the proposed dimension-reduction search. This helps reduce the training variance and avoid training deceleration.

Another question in DSG is that whether the selection masks converge during training or not. To explore the answer, we did an additional experiment as shown in the Figure

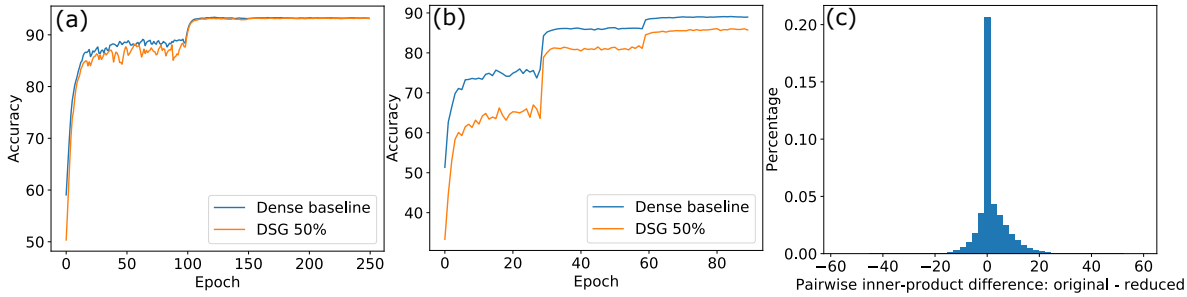


Figure A.2: Accuracy convergence. (a) Training curve with validation accuracy of VGG8 on CIFAR10; (b) Training curve with top-5 validation accuracy of ResNet-18 on ImageNet; (c) Distribution of pairwise difference between the original high-dimensional inner product and the low-dimensional one for the CONV5 layer in VGG8.

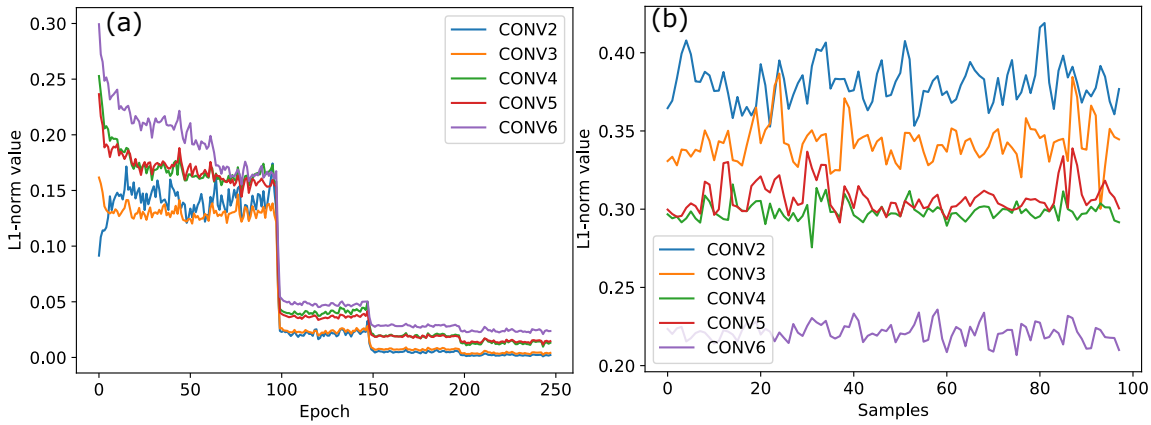


Figure A.3: Selection mask convergence. (a) Average $L1$-norm value of the difference mask tensors between adjacent training epochs across all samples in one mini-batch; (b) Average $L1$-norm value of the difference mask tensors between adjacent samples after training.

A.3. We select a mini-batch of training samples as a case study for data recording. Each curve presents the results of one layer (CONV2-CONV6). For each sample at each layer, we recorded the change of binary selection mask between two adjacent training epochs. Here the change is obtained by calculating the $L1$-norm value of the difference tensor of two mask tensors at two adjacent epochs, i.e., $change = batch\_avg\_L1norm(mask_{i+1} - mask_i)$. Here the $batch\_avg\_L1norm(\cdot)$ indicates the average $L1$-norm value across all samples in one mini-batch. As shown in Figure A.3(a), the selection mask for each sample converges as training goes on.

In our implementation we inherit the random projection matrix from training and

perform the same on-the-fly dimension-reduction search in inference. We didn't try to directly suspend the selection masks, because the selection mask might vary across samples even if we observe convergence for each sample. This can be seen from Figure A.3(b), where the difference mask tensors between adjacent samples in one mini-batch present significant differences (large $L1$-norm value) after training. Therefore, it will consume lot of memory space to save these trained masks for all samples, which is less efficient than conducting on-the-fly search during inference.

A.4 Comparison with Other Methods

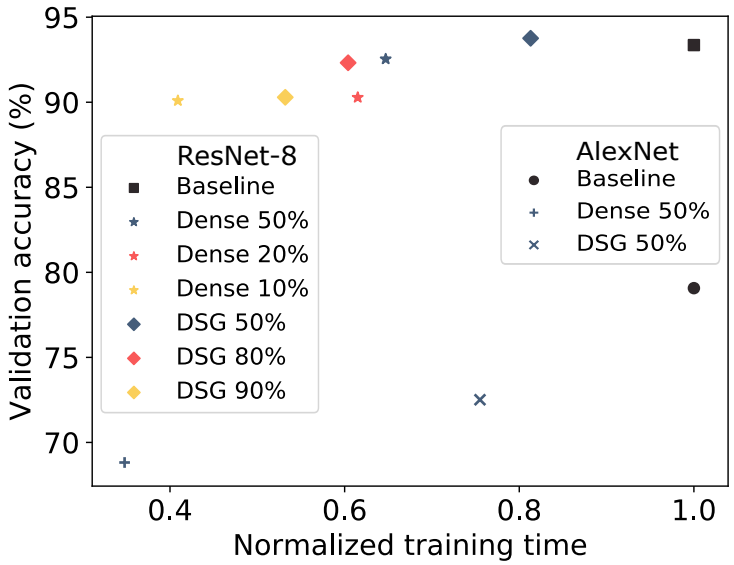


Figure A.4: Comparison with smaller-dense models with equivalent MACs using ResNet8 on CIFAR10 and AlexNet on ImageNet.

Figure A.4 extends Figure 4.8(b) in the main text to more network structures, including ResNet8 on CIFAR10 and AlexNet on ImageNet. The similar observation can be achieved: the equivalent smaller dense models with the same effective MACs are able to save more training time but the accuracy degradation will be increased. Note that in this figure, the DSG training uses a warm-up training with dense model for the first 10

epochs. The overhead of the warm-up training has been taken account into the entire training cost. To make the accuracy results on CIFAR10 and ImageNet comparable for figure clarity, AlexNet reports the top-5 accuracy.

Our work targets at both the training and inference phases while most of previous work focused on the inference compression. In prior methods, the training usually becomes more complicated with various regularization constraints or iterative fine-tuning/retraining. Therefore, it is not very fair to compare with them during training. For this reason, we just compare with them on the inference pruning. Different from doing DSG training from scratch, here we utilize DSG for fine-tuning based on pre-trained models.

Table A.2: Comparison with other structured sparsification methods for inference. All the results are from VGG16 on ImageNet, and the default accuracy is top-1 accuracy. The baseline methods are Taylor Expansion [49], ThinNet [44], Channel Pruning [47], AutoPrunner [43], and AMC [39].

Methods	Taylor Expansion	ThinNet	Channel Pruning	AutoPrunner	AMC	**DSG**
Operation Sparsity	62.86%	69.81%	69.32%	73.6%	80%	62.92%
Accuracy	87%(top-5)	67.34%	70.42%	68.43%	69.1%	71.44%(top-1) 90.56%(top-5)

To guarantee the fairness, all the results are from the same network (VGG16) on the same dataset (ImageNet). Since our DSG produces structured sparsity, we also select structured sparsity work as comparison baselines. Different from the previous experiments in this paper, we further take the input sparsity at each layer into account rather than only count the output sparsity. This is due to the fact that the baselines consider all zero operands. The results are listed in Table A.2, from which we can see that DSG is able to achieve a good balance between the operation amount and model accuracy.

Appendix B

Supplemental Materials for Dual-Module Inference

B.1 Motivation for Dual-Module Inference

As shown in Figure B.1, the nonlinear activation functions – *sigmoid* and *tanh* – have insensitive regions where the output activations are resilient to errors introduced in pre-activation accumulation results.

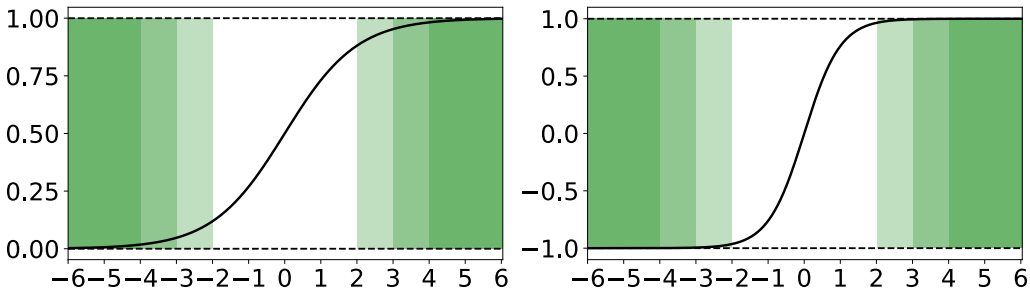


Figure B.1: Insensitive (green shaded) and sensitive (white) regions of *sigmoid* (left) and *tanh* (right) nonlinear functions.

The selection of which neurons should be in the (in)sensitive region is dynamic and input-dependent, which can be seen in Figure B.2. Unlike the static weight sparsity that

we can prune the unused connections offline in advance, the dynamic region speculation requires a very lightweight criterion for real-time processing. Taking all these into account, we propose a dual-model inference method that efficiently determines (in)sensitive region and significantly saves the memory access and computational cost.

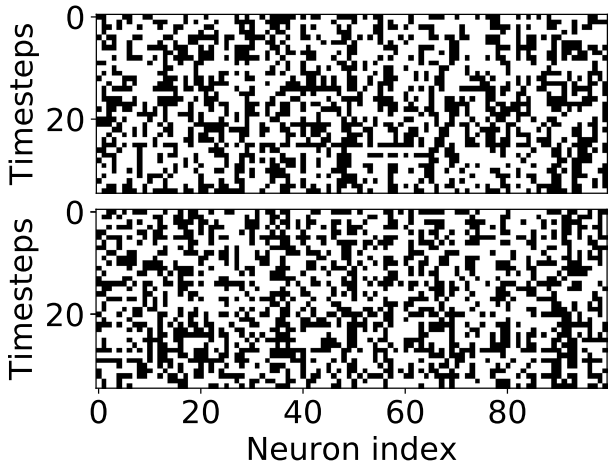


Figure B.2: Dynamic region distribution across time-steps and inputs. The white and black colors denote neurons in the insensitive and sensitive regions, respectively. The upper and lower patterns are from different inputs.

B.2 Experiments

Settings. Our evaluation for single-layer RNNs is adapted from PyTorch's word language modeling example, where the dataset has 10K tokens. We do not use dropout when training the LL module(s); the starting learning rate is 5, and we decay it by four if no loss descent has been seen on the validation dataset. The RNNs used in language modeling have 35 timestamps; the maximum generated sequence length in GNMT is 80. For language modeling, we choose 1500 hidden units following the word language modeling example, and we compare our method which dynamically reduces 50% of weight accesses to the static case where only 750 hidden units are used. Besides single-layer LSTM and GRU, we also evaluate four-layer stacked LSTMs as in GNMT. For GNMT

experiments, we use the same set of parameters when training the base model [1]. We train the four *little* modules in the four-layer stacked LSTM in a total of 4 epochs.

Our experiments on ResNet-18 is adapted from the image classification example of PyTorch.

Table B.1: LSTM perplexity and execution time (ms).

Insensitive	hidden size: 1500				hidden size: 750			
ratio	PPL	Diff.	Time	Speedup	PPL	Diff.	Time	Speedup
Base	80.64	n/a	1.477	1.00x	84.32	n/a	0.546	1.00x
10%	80.72	-0.08	1.315	1.12x	84.42	-0.10	0.448	1.22x
30%	80.56	0.08	1.095	1.35x	84.43	-0.11	0.415	1.32x
50%	81.36	-0.72	0.885	1.67x	84.29	0.03	0.342	1.60x
70%	87.48	-6.83	0.641	2.30x	84.89	-0.57	0.287	1.90x
90%	109.37	-28.73	0.380	3.89x	88.44	-4.12	0.216	2.53x

Additional results. In addition to the results of LSTMs using 1500 hidden units in the main text, We observe a similar quality-performance trade-off for LSTM with 750 hidden units as shown in Table B.1. Comparing the case of base LSTM with 750 hidden units with dual-module LSTM with 1500 hidden units and 50% insensitive ratio, although the memory access reduction is at the same level, our proposed dual-module approach achieves much better model quality because we kept the expressive power of a larger LSTM layer.

Table B.2: GRU perplexity and execution time (ms).

Insensitive	hidden size: 1500				hidden size: 750			
ratio	PPL	Diff.	Time	Speedup	PPL	Diff.	Time	Speedup
Base	85.48	n/a	1.182	1.00x	89.64	n/a	0.466	1.00x
10%	85.62	-0.14	1.024	1.15x	89.81	-0.17	0.383	1.22x
30%	86.01	-0.53	0.869	1.36x	89.63	0.01	0.334	1.40x
50%	88.73	-3.25	0.726	1.63x	89.69	-0.05	0.302	1.54x
70%	98.09	-12.61	0.545	2.17x	92.51	-2.87	0.284	1.64x
90%	122.75	-37.27	0.350	3.38x	102.37	-12.73	0.198	2.35x

We further report the results using single-layer GRU on word-level language modeling

[1] From https://github.com/NVIDIA/DeepLearningExamples

tasks as in Table B.2. Using dual-module inference on GRUs expresses the similar quality-performance trade-off as of LSTMs. Our dual-module method is generally applicable to both LSTMs and GRUs. We also measured the execution time of GNMT layer with the hidden size of 1024 and the input size of 2048 in Table B.3.

Table B.3: GNMT BLEU score and execution time (ms). (1024, 2048) indicates the hidden size is 1024 and the input size is 2048; similarly for (1024, 1024).

Insensitive	Quality		(1024, 1024)		(1024, 2048)	
ratio	BLEU	Diff.	Time	Speedup	Time	Speedup
Base	24.32	n/a	0.838	1.00x	1.092	1.00x
10%	24.33	0.01	0.679	1.23x	0.962	1.14x
30%	24.18	-0.14	0.541	1.55x	0.803	1.36x
50%	23.73	-0.59	0.480	1.75x	0.642	1.70x
70%	21.92	-2.40	0.360	2.33x	0.479	2.28x
90%	11.77	-12.55	0.243	3.45x	0.307	3.56x

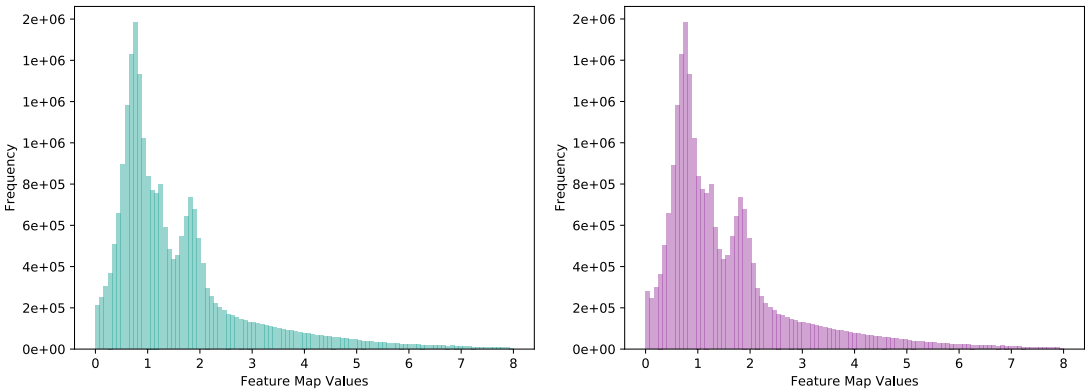


Figure B.3: The distribution of the *little* module, on the right, exhibits the same as the original module on the left.

Evaluation on the *little* module. Using one layer in ResNet-18, we first show the histogram of feature map values of both the original module and the *little* module learned and approximated from the original module. As shown in Figure B.3, the distribution of the little module exhibits the same as the original module. We then show the visualization of feature maps of both the original, i.e., the *big* module, and of the *little* module in Figure

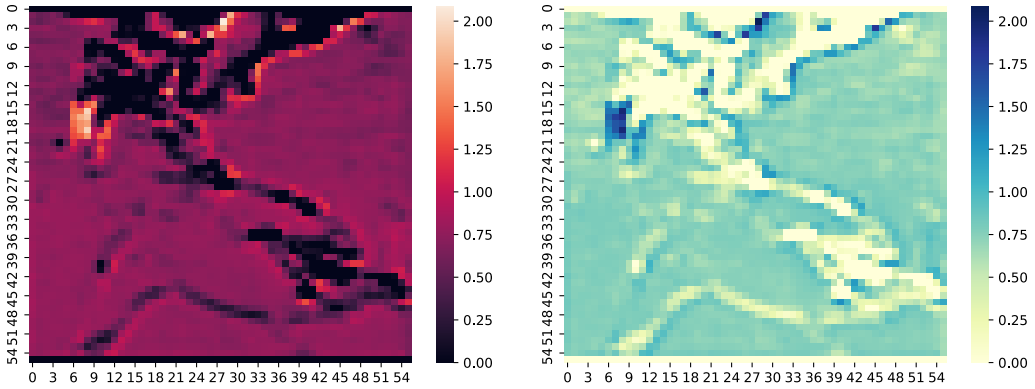


Figure B.4: Visualization of the feature maps of the original layer, i.e., the *big* module on the left, and of the *little* module on the right. The *little* module can approximate the original layer well.

B.4. From the visualized feature map comparison, we can see that the *little* module approximates the original module well and represents the almost same features.

B.3 Comparison with Weight Pruning Method

Table B.4: Comparison of our proposed dual-module inference (DMI), using 50% insensitive ratio, with weight pruning using one LSTM layer with 1500 units in word language modeling task on WikiText-2 dataset.

Method	PPL w/o DMI	PPL w/ DMI
Dense	85.52	86.21
80% weight sparsity	86.42	88.46
90% weight sparsity	88.75	90.96

As shown in Table B.4, we compare our proposed dual-module inference approach with the automated gradual pruning method [27], which is a popular pruning method with open implementation[2]. Firstly, compared with weight pruning, our method achieves better quality with practical speedup – 1.54x to 1.75x reduction on wall-clock time – on commodity CPUs while element-wise weight pruning requires specialized hardware to

[2]From https://github.com/NervanaSystems/distiller

gain real speedup of computation given irregular sparsity. Moreover, our dual-module inference method can be further applied on top of pruned models to reduce execution time by reducing memory access.

Appendix C

Supplemental Materials for Dynamic Sparse Attention

C.1 Benchmark Descriptions and Experiment Configurations

In our experiments, we choose Text Classification, Image Classification and Document Retrieval from Long-Range Arena, while excluding Long ListOps and Pathfinder. This is because the ListOps results in LRA exhibit significant divergence without much explanation. And for Pathfinder, we are unable to reproduce the baseline results with the given training configurations from LRA.

C.1.1 Text Classification

The Text Classification task, as introduced in LRA [172], is a binary classification that uses real-world data to benchmark the ability of the models to deal with compositionality. IMDb review [181] is selected as the dataset, which is a common choice for document

classification. Moreover, to make the problem more challenging, this task takes a byte-level setup instead of the normal character-level setup for language modeling. Therefore, the model needs to learn from the unsegmented data and make compositional decisions.

For model configuration, we use the original hyperparameters given in the LRA repository [1]. Specifically, the baseline transformer consists of 4 attention layers, each with 4 heads. The hidden dimension size is 256 and the positional FFN layer has a dimension size of 1024. The learning rate is 0.05 with a weight decay of 0.1. Finally, the baseline model is trained for $20K$ steps where the first $8K$ are warmup steps and the batch size is 32.

When compared with the dense baseline in Figure 2 of the full paper, the **DSA-x%** models are obtained from fine-tuning the dense model for $5K$ steps with different levels of sparsity constraints. During fine-tuning, parameters from both original model and the predictor are updated simultaneously using the combination of cross-entropy loss and MSE loss. The weight factor λ of the MSE loss is 0.01 and the learning rate is uniformly set as 0.0002.

When compared with other efficient transformers as shown in Table 1 of the full paper, we directly train the DSA prediction path from scratch. The overall training step is still $20K$, but we use the first $15K$ to train the original model and freeze the predictor module. Therefore, the first $15K$ steps are the same as training a dense baseline. After this, we jointly optimize the model and the predictor module during the last $5K$ steps with the same MSE loss factor and learning rate as above.

Finally, to limit the training cost, we set the sequence length to be 2000 for the baseline comparison and sensitivity study, while only set the length to be 4000 when comparing with other models.

[1] `https://github.com/google-research/long-range-arena`

C.1.2 Document Retrieval

Document Retrieval is a binary classification task that serves as a test to evaluate how well a model compresses long sequences into representations for similarity-based matching. This task uses ACL Anthology Network [182] and aims to identify if two papers have a citation link. Similar to Text Classification, byte-level setup is used to increase the difficulty of the problem.

We use a uniform sequence length of 4000 in this task. The baseline transformer consists of 4 attention layers. Each attention layer has 4 heads, 128 hidden dimensions, and 512 FFN dimensions. The learning rate is 0.05 with a weight decay of 0.1. The model is trained for $5K$ steps with Adam optimizer and a batch size of 32. Similar to the strategy in the Text Classification task, we use fine-tuning for baseline comparison and training-from-scratch for cross model comparison. The $5K$ steps are equally divided into $2.5K$ for dense training and $2.5K$ for joint training in the training-from-scratch experiment. When jointly optimizing all the parameters, the weight factor λ of the MSE loss is 0.01 and the learning rate is 0.0002.

C.1.3 Image Classification

The final task we include in our evaluation is image classification using CIFAR-10 [147]. Each 32×32 input image is flattened as a sequence of pixels. Therefore, the sequence length of this task is 1024. The input images are mapped to a single gray-scale channel where each pixel is represented with an 8-bit integer value. Following the given settings, the baseline transformer model contains one attention layer with 8 heads, 64 query/key/value hidden dimensions, and 128 FFN dimensions.

There are in total 45,000 training samples and 15,000 validation samples. We train the model for 200 epochs with a learning rate of 0.0005 and a batch size of 128. Same

as above, we use finetuning for baseline comparison and training-from-scratch for cross model comparison. The 200 steps are divided into 150 for dense training and 50 for joint training in the training-from-scratch experiment. When jointly optimizing all the parameters, the weight factor λ of the MSE loss is 0.01 and the learning rate is 0.0002.

C.2 Evaluation

In this section we present the evaluation results of DOTA.

C.2.1 Evaluation Methodology

Benchmarks. Our experiments include series of representative Transformer benchmarks with challenging long-sequence tasks. We first run BERT (large) [166] on question answering task (QA) using the Stanford Question Answering Dataset (SQuAD) [183] v1.1 with a sequence length of 384. To scale our evaluation to longer sequences, we further select three tasks from Long-Range-Arena [172] (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, 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 [181] with a sequence length of 2k. The third task aims to identify if two papers in the ACL Anthology Network [182] contain a citation link. The papers are modeled as 4k input sequences to the Transformer model. Finally, we use GPT-2 to evaluate causal language modeling (LM) on Wikitext-103 [3] 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 [105]) and pure software Transformer models presented in LRA.

Table C.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

Hardware Experiment Methodology. The system configuration and consumption of DOTA is shown in Table C.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. We 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, while qualitatively discuss the difference between DOTA and other customized hardware (See Section 2.5). 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.

Bibliography

[1] I. Sutskever, O. Vinyals, and Q. V. Le, *Sequence to sequence learning with neural networks*, in *Advances in Neural Information Processing Systems*, vol. 4, pp. 3104–3112, 2014. arXiv:1409.3215.

[2] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, J. Klingner, A. Shah, M. Johnson, X. Liu, Ł. Kaiser, S. Gouws, Y. Kato, T. Kudo, H. Kazawa, K. Stevens, G. Kurian, N. Patil, W. Wang, C. Young, J. Smith, J. Riesa, A. Rudnick, O. Vinyals, G. Corrado, M. Hughes, and J. Dean, *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, *arXiv preprint arXiv:1609.08144* (2016) [arXiv:1609.0814].

[3] S. Merity, C. Xiong, J. Bradbury, and R. Socher, *Pointer sentinel mixture models*, in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017. arXiv:1609.0784.

[4] 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*, vol. 2017-Decem, pp. 5999–6009, 2017. arXiv:1706.0376.

[5] L. Song, P. Pan, K. Zhao, H. Yang, Y. Chen, Y. Zhang, Y. Xu, and R. Jin, *Large-Scale Training System for 100-Million Classification at Alibaba*, in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 2909–2917, 2020. arXiv:2102.0602.

[6] J. Liu, W. C. Chang, Y. Wu, and Y. Yang, *Deep learning for extreme multi-label text classification*, in *SIGIR 2017 - Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, (New York, NY, USA), pp. 115–124, Association for Computing Machinery, Inc, aug, 2017.

[7] T. Medini, Q. Huang, Y. Wang, V. Mohan, and A. Shrivastava, *Extreme classification in log memory using count-min sketch: A case study of amazon*

search with 50M products, in *Advances in Neural Information Processing Systems*, vol. 32, 2019. arXiv:1910.1383.

[8] J. L. Hennessy and D. A. Patterson, *A new golden age for computer architecture*, *Communications of the ACM* **62** (feb, 2019) 48–60.

[9] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. Richard Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, *In-datacenter performance analysis of a tensor processing unit*, in *Proceedings - International Symposium on Computer Architecture*, vol. Part F1286 of *ISCA '17*, (New York, NY, USA), pp. 1–12, Association for Computing Machinery, 2017. arXiv:1704.0476.

[10] K. Bhatia, K. Dahiya, H. Jain, A. Mittal, Y. Prabhu, and M. Varma, *The extreme classification repository: Multi-label datasets and code*, .

[11] P. Knebel, D. Berkram, A. Davis, D. Emmot, P. Faraboschi, and G. Gostin, *Gen-Z Chipsetfor Exascale Fabrics*, in *2019 IEEE Hot Chips 31 Symposium, HCS 2019*, pp. 1–22, 2019.

[12] NVIDIA Corporation, *Gpudirect : Integrating the Gpu With a Network Interface*, in *GPU Technology Conference*, 2015.

[13] S. Van Doren, *Abstract - HOTI 2019: Compute Express Link*, in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*, pp. 18–18, 2020.

[14] L. Nai, R. Hadidi, H. Xiao, H. Kim, J. Sim, and H. Kim, *CooLPIM: Thermal-Aware source throttling for efficient PIM instruction offloading*, in *Proceedings - 2018 IEEE 32nd International Parallel and Distributed Processing Symposium, IPDPS 2018*, pp. 680–689, 2018.

[15] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, *A scalable processing-in-memory accelerator for parallel graph processing*, in *Proceedings - International Symposium on Computer Architecture*, vol. 13-17-June, pp. 105–117, 2015.

[16] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, *GraphP: Reducing Communication for PIM-Based Graph Processing with Efficient Data Partition*, in *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2018-Febru, pp. 544–557, 2018.

[17] G. Kim, N. Chatterjee, M. O'Connor, and K. Hsieh, *Toward Standardized near-data processing with unrestricted data placement for GPUs*, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017*, pp. 1–12, 2017.

[18] C. D. Kersey, H. Kim, and S. Yalamanchili, *Lightweight SIMT core designs for intelligent 3D stacked DRAM*, in *ACM International Conference Proceeding Series*, vol. Part F1311, pp. 49–59, 2017.

[19] P. Gu, X. Xie, Y. Ding, G. Chen, W. Zhang, D. Niu, and Y. Xie, *IPIM: Programmable In-Memory Image Processing Accelerator Using Near-Bank Architecture*, in *Proceedings - International Symposium on Computer Architecture*, vol. 2020-May, pp. 804–817, 2020.

[20] P. Gu, X. Xie, S. Li, D. Niu, H. Zheng, K. T. Malladi, and Y. Xie, *DLUX: A LUT-Based Near-Bank Accelerator for Data Center Deep Learning Training Workloads*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **40** (2021), no. 8 1586–1599.

[21] H. Shin, D. Kim, E. Park, S. Park, Y. Park, and S. Yoo, *McDRAM: Low latency and energy-efficient matrix computations in DRAM*, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **37** (2018), no. 11 2613–2622.

[22] Y. C. Kwon, S. H. Lee, J. Lee, S. H. Kwon, J. M. Ryu, J. P. Son, O. Seongil, H. S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H. S. Shin, J. Kim, B. S. Phuah, H. M. Kim, M. J. Song, A. Choi, D. Kim, S. Y. Kim, E. B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. H. Song, J. Youn, K. Sohn, and N. S. Kim, *25.4 A 20nm 6GB Function-In-Memory DRAM, Based on HBM2 with a 1.2TFLOPS Programmable Computing Unit Using Bank-Level Parallelism, for Machine Learning Applications*, in *Digest of Technical Papers - IEEE International Solid-State Circuits Conference*, vol. 64, pp. 350–352, 2021.

[23] A. Yazdanbakhsh, C. Song, J. Sacks, P. Lotfi-Kamran, H. Esmaeilzadeh, and N. Sung Kim, *In-DRAM near-data approximate acceleration for GPUs*, in *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, pp. 1–14, 2018.

[24] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H. H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov,

M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C. J. Wu, M. Hempstead, and X. Zhang, *RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing*, in *Proceedings - International Symposium on Computer Architecture*, vol. 2020-May, pp. 790–803, 2020. arXiv:1912.1295.

[25] Y. Kwon, Y. Lee, and M. Rhu, *TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning*, in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, pp. 740–753, 2019. arXiv:1908.0307.

[26] Y. Kwon, Y. Lee, and M. Rhu, *Tensor Casting: Co-Designing Algorithm-Architecture for Personalized Recommendation Training*, in *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2021-Febru, pp. 235–248, 2021. arXiv:2010.1310.

[27] C. Zhu, H. Mao, S. Han, and W. J. Dally, *Trained ternary quantization*, in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017. arXiv:1612.0106.

[28] C. Xu, J. Yao, Z. Lin, W. Ou, Y. Cao, Z. Wang, and H. Zha, *Alternating multi-bit quantization for recurrent neural networks*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1802.0015.

[29] Y. Xu, Y. Wang, A. Zhou, W. Lin, and H. Xiong, *Deep neural network compression with single and multiple level quantization*, in *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 4335–4342, 2018. arXiv:1803.0328.

[30] P. Wang, X. Xie, L. Deng, G. Li, D. Wang, and Y. Xie, *HitNet: Hybrid ternary recurrent neural network*, in *Advances in Neural Information Processing Systems*, vol. 2018-Decem, pp. 604–614, 2018.

[31] J. Choi, Z. Wang, S. Venkataramani, P. I.-J. Chuang, V. Srinivasan, and K. Gopalakrishnan, *PACT: Parameterized Clipping Activation for Quantized Neural Networks*, *arXiv preprint arXiv:1805.06085* (2018) [arXiv:1805.0608].

[32] S. Han, J. Pool, J. Tran, and W. J. Dally, *Learning both weights and connections for efficient neural networks*, in *Advances in Neural Information Processing Systems*, vol. 2015-Janua, pp. 1135–1143, 2015. arXiv:1506.0262.

[33] S. Han, H. Mao, and W. J. Dally, *Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding*, in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016. arXiv:1510.0014.

[34] S. Narang, G. Diamos, S. Sengupta, and E. Elsen, *Exploring sparsity in recurrent neural networks*, in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017. arXiv:1704.0511.

[35] X. Dai, H. Yin, and N. K. Jha, *Grow and Prune Compact, Fast, and Accurate LSTMs*, *IEEE Transactions on Computers* **69** (2020), no. 3 441–452, [arXiv:1805.1179].

[36] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, *Exploring the Granularity of Sparsity in Convolutional Neural Networks*, in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, vol. 2017-July, pp. 1927–1934, 2017.

[37] W. Wen, Y. Chen, H. Li, Y. He, S. Rajbhandari, M. Zhang, W. Wang, F. Liu, and B. Hu, *Learning intrinsic sparse structures within long short-term memory*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1709.0502.

[38] A. Polino, R. Pascanu, and D. Alistarh, *Model compression via distillation and quantization*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1802.0566.

[39] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, *AMC: AutoML for Model Compression and Acceleration on Mobile Devices*, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* **11211 LNCS** (feb, 2018) 815–832, [arXiv:1802.0349].

[40] T.-W. Chin, C. Zhang, and D. Marculescu, *Layer-compensated Pruning for Resource-constrained Convolutional Neural Networks*, *arXiv preprint arXiv:1810.00518* (2018) [arXiv:1810.0051].

[41] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, *Learning Efficient Convolutional Networks through Network Slimming*, in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, pp. 2755–2763, 2017. arXiv:1708.0651.

[42] J. Ye, X. Lu, Z. Lin, and J. Z. Wang, *Rethinking the smaller-norm-less-informative assumption in channel pruning of convolution layers*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1802.0012.

[43] J. H. Luo and J. Wu, *AutoPruner: An end-to-end trainable filter pruning method for efficient deep model inference*, *Pattern Recognition* **107** (2020) [arXiv:1805.0894].

[44] J. H. Luo, J. Wu, and W. Lin, *ThiNet: A Filter Level Pruning Method for Deep Neural Network Compression*, in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, pp. 5068–5076, 2017. arXiv:1707.0634.

[45] Y. He, X. Zhang, and J. Sun, *Channel Pruning for Accelerating Very Deep Neural Networks*, in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, pp. 1398–1406, 2017. arXiv:1707.0616.

[46] L. Liang, L. Deng, Y. Zeng, X. Hu, Y. Ji, X. Ma, G. Li, and Y. Xie, *Crossbar-aware neural network pruning*, *IEEE Access* **6** (2018) 58324–58337, [arXiv:1807.1081].

[47] Y. Hu, S. Sun, J. Li, X. Wang, and Q. Gu, *A novel channel pruning method for deep neural network compression*, *arXiv preprint arXiv:1805.11394* (2018) [arXiv:1805.1139].

[48] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, *Learning structured sparsity in deep neural networks*, in *Advances in Neural Information Processing Systems*, pp. 2082–2090, 2016. arXiv:1608.0366.

[49] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, *Pruning convolutional neural networks for resource efficient inference*, in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017. arXiv:1611.0644.

[50] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, *Adaptive neural networks for efficient inference*, in *34th International Conference on Machine Learning, ICML 2017*, vol. 2 of *ICML'17*, pp. 812–821, JMLR.org, 2017. arXiv:1702.0781.

[51] Y. Lin, C. Sakr, Y. Kim, and N. Shanbhag, *PredictiveNet: An energy-efficient convolutional neural network via zero prediction*, in *Proceedings - IEEE International Symposium on Circuits and Systems*, pp. 1–4, 2017.

[52] 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 Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2019-June, pp. 11208–11217, 2019.

[53] X. Dong, J. Huang, Y. Yang, and S. Yan, *More is less: A more complicated network with less inference complexity*, in *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 1895–1903, 2017. arXiv:1703.0865.

[54] X. Gao, Y. Zhao, L. Dudziak, R. Mullins, and X. Cheng-Zhong, *Dynamic channel pruning: Feature boosting and suppression*, in *7th International Conference on Learning Representations, ICLR 2019*, 2019. arXiv:1810.0533.

[55] W. Hua, Y. Zhou, C. de Sa, Z. Zhang, and G. Edward Suh, *Channel gating neural networks*, in *Advances in Neural Information Processing Systems*, vol. 32, pp. 1884–1894, 2019. arXiv:1805.1254.

[56] D. Neil, J. H. Lee, T. Delbrück, and S. C. Liu, *Delta networks for optimized recurrent network computation*, in *34th International Conference on Machine Learning, ICML 2017*, vol. 5, pp. 3971–3980, 2017. arXiv:1612.0557.

[57] X. Zhang, C. Xie, J. Wang, W. Zhang, and X. Fu, *Towards memory friendly long-short term memory networks (lstms) on mobile gpus*, in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 162–174, IEEE, 2018.

[58] V. Campos, B. Jou, X. Giró-I-Nieto, J. Torres, and S. F. Chang, *SkIp RNN: Learning to skip state updates in recurrent neural networks*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1708.0683.

[59] R. Spring and A. Shrivastava, *Scalable and sustainable deep learning via randomized hashing*, in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. Part F1296, pp. 445–454, 2017. arXiv:1602.0819.

[60] X. Sun, X. Ren, S. Ma, and H. Wang, *MeProp: Sparsified back propagation for accelerated deep learning with reduced overfitting*, in *34th International Conference on Machine Learning, ICML 2017*, vol. 7, pp. 5080–5089, 2017. arXiv:1706.0619.

[61] Y. Lin, S. Han, H. Mao, Y. Wang, W. J. Dally, and B. Dally, *Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training*, in *International Conference on Learning Representations*, 2018. arXiv:1712.0188.

[62] L. Gong, D. He, Z. Li, T. Qin, L. Wang, and T. Y. Liu, *Efficient training of BERT by progressively stacking*, in *36th International Conference on Machine Learning, ICML 2019*, vol. 2019-June of *Proceedings of Machine Learning Research*, pp. 4202–4211, PMLR, feb, 2019.

[63] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D. C. Juan, *Sparse sinkhorn attention*, in *37th International Conference on Machine Learning, ICML 2020*, vol. PartF16814, pp. 9380–9389, 2020. arXiv:2002.1129.

[64] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontanon, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, *Big bird: Transformers for longer sequences*, in *Advances in Neural Information Processing Systems*, vol. 2020-Decem, 2020. arXiv:2007.1406.

[65] R. Child, S. Gray, A. Radford, and I. Sutskever, *Generating Long Sequences with Sparse Transformers*, arXiv:1904.1050.

[66] J. Qiu, H. Ma, O. Levy, W. T. Yih, S. Wang, and J. Tang, *Blockwise self-attention for long document understanding*, in *Findings of the Association for Computational Linguistics Findings of ACL: EMNLP 2020*, pp. 2555–2565, 2020. arXiv:1911.0297.

[67] N. Kitaev, Ł. Kaiser, and A. Levskaya, *Reformer: The Efficient Transformer*, *International Conference on Learning Representations* (2020) [arXiv:2001.0445].

[68] 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, [arXiv:2003.0599].

[69] Y. Tay, M. Dehghani, D. Bahri, and D. Metzler, *Efficient Transformers: A Survey*, *arXiv* (2020), no. August 2020 1–28, [arXiv:2009.0673].

[70] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, *Linformer: Self-Attention with Linear Complexity*, *arXiv preprint arXiv:2006.04768* (2020) [arXiv:2006.0476].

[71] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention*, in *37th International Conference on Machine Learning, ICML 2020*, vol. PartF16814, pp. 5112–5121, 2020. arXiv:2006.1623.

[72] 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*, *International Conference on Learning Representations (ICLR)* (sep, 2021) [arXiv:2009.1479].

[73] H. Peng, N. Pappas, D. Yogatama, R. Schwartz, N. A. Smith, and L. Kong, *Random Feature Attention*, *International Conference on Learning Representations* (2021) [arXiv:2103.0214].

[74] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, *A dynamically configurable coprocessor for convolutional neural networks*, in *Proceedings - International Symposium on Computer Architecture*, vol. 38, pp. 247–257, ACM, 2010.

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

[76] 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*, in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, vol. 49, pp. 269–283, ACM, 2014.

[77] 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, 2014.

[78] 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 SIGPLAN Notices*, vol. 50, pp. 369–381, 2015.

[79] 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 *Proceedings - International Symposium on Computer Architecture*, vol. 13-17-June, pp. 92–104, 2015.

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

[81] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, *Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory*, in *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, pp. 380–392, 2016.

[82] P. Judd, J. Albericio, and A. Moshovos, *Stripes: Bit-Serial Deep Neural Network Computing*, *IEEE Computer Architecture Letters* **16** (2017), no. 1 80–83.

[83] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G. Y. Wei, and D. Brooks, *Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators*, in *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, pp. 267–278, 2016.

[84] 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.

[85] M. Gao, J. Pu, X. Yang, M. Horowitz, and C. Kozyrakis, *TETRIS: Scalable and efficient neural network acceleration with 3D memory*, *ACM SIGPLAN Notices* **52** (2017), no. 4 751–764.

[86] 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 *FPGA 2017 - Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 75–84, 2017. arXiv:1612.0069.

[87] 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 *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2016-Decem, p. 20, 2016.

[88] 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 *Proceedings - International Symposium on Computer Architecture*, vol. Part F1286, pp. 27–40, 2017. arXiv:1708.0448.

[89] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, *UCNN: Exploiting computational reuse in deep neural networks via weight repetition*, in *Proceedings - International Symposium on Computer Architecture*, pp. 674–687, 2018. arXiv:1804.0650.

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

[91] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, *Scalpel: Customizing DNN pruning to the underlying hardware parallelism*, in *Proceedings - International Symposium on Computer Architecture*, vol. Part F1286, pp. 548–560, 2017.

[92] H. T. Kung, B. McDanel, and S. Q. Zhang, *Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining under Joint Optimization*, in *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 821–834, 2019. arXiv:1811.0477.

[93] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan, *PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices*, in

Proceedings of the Annual International Symposium on Microarchitecture, MICRO, vol. 2018-Octob, pp. 189–202, 2018. arXiv:2004.1093.

[94] X. Zeng, T. Zhi, X. Zhou, Z. Du, Q. Guo, S. Liu, B. Wang, Y. Wen, C. Wang, X. Zhou, L. Li, T. Chen, N. Sun, and Y. Chen, *Addressing irregularity in sparse neural networks through a cooperative software/hardware approach*, *IEEE Transactions on Computers* **69** (2020), no. 7 968–985.

[95] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, *Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing*, in *Proceedings - 2016 43rd International Symposium on Computer Architecture, ISCA 2016*, pp. 1–13, 2016.

[96] J. Zhu, J. Jiang, X. Chen, and C. Y. Tsui, *Sparsenn: An energy-efficient neural network accelerator exploiting input and output sparsity*, in *Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018*, vol. 2018-Janua, pp. 241–244, 2018. arXiv:1711.0126.

[97] 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* **30** (2019), no. 3 644–656, [arXiv:1706.0140].

[98] M. Mahmoud, K. Siu, and A. Moshovos, *Diffy: A déjà vu-free differential deep neural network accelerator*, in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2018-Octob, pp. 134–147, 2018.

[99] J. F. Zhang, C. E. 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 *IEEE Symposium on VLSI Circuits, Digest of Technical Papers*, vol. 2019-June, pp. C306–C307, 2019.

[100] M. Song, J. Zhao, Y. Hu, J. Zhang, and T. Li, *Prediction based execution on deep neural networks*, in *Proceedings - International Symposium on Computer Architecture*, pp. 752–763, 2018.

[101] 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 *Proceedings - International Symposium on Computer Architecture*, pp. 662–673, 2018.

[102] 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.

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

[104] 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 *Proceedings - 2020 IEEE International Symposium on High Performance Computer Architecture, HPCA 2020*, pp. 328–341, 2020. arXiv:2002.1094.

[105] 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 *Proceedings - International Symposium on Computer Architecture*, vol. 2021-June, pp. 692–705, 2021.

[106] H. Wang, Z. Zhang, and S. Han, *SpAtten: Efficient Sparse Attention Architecture with Cascade Token and Head Pruning*, in *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2021-Febru, pp. 97–110, 2021. arXiv:2012.0985.

[107] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2015 Inter, pp. 1026–1034, 2015. arXiv:1502.0185.

[108] G. Giacinto and F. Roli, *Design of effective neural network ensembles for image classification purposes*, *Image and Vision Computing* **19** (2001), no. 9-10 699–707.

[109] W. Zheng, J. Lu, and J. Zhou, *Hardness-Aware Deep Metric Learning*, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **43** (2021), no. 9 3214–3228, [arXiv:1903.0550].

[110] D. Bahdanau, K. H. Cho, and Y. Bengio, *Neural machine translation by jointly learning to align and translate*, in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015. arXiv:1409.0473.

[111] A. Graves, A. R. Mohamed, and G. Hinton, *Speech recognition with deep recurrent neural networks*, in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pp. 6645–6649, 2013. arXiv:1303.5778.

[112] Y. He, T. N. Sainath, R. Prabhavalkar, I. McGraw, R. Alvarez, D. Zhao, D. Rybach, A. Kannan, Y. Wu, R. Pang, Q. Liang, D. Bhatia, Y. Shangguan, B. Li, G. Pundak, K. C. Sim, T. Bagby, S. Y. Chang, K. Rao, and A. Gruenstein, *Streaming End-to-end Speech Recognition for Mobile Devices*, in *ICASSP, IEEE*

International Conference on Acoustics, Speech and Signal Processing - Proceedings, vol. 2019-May, pp. 6381–6385, 2019. arXiv:1811.0662.

[113] Y. Wang, R. J. Skerry-Ryan, D. Stanton, Y. Wu, R. J. Weiss, N. Jaitly, Z. Yang, Y. Xiao, Z. Chen, S. Bengio, and Others, *Tacotron: Towards end-to-end speech synthesis*, *arXiv preprint arXiv:1703.10135* (2017).

[114] T. N. Kipf and M. Welling, *Semi-supervised classification with graph convolutional networks*, in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017. arXiv:1609.0290.

[115] 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*, in *Advances in Neural Information Processing Systems*, vol. 2015-Janua, pp. 2224–2232, 2015. arXiv:1509.0929.

[116] J. Klicpera, A. Bojchevski, and S. Günnemann, *Predict then propagate: Graph neural networks meet personalized PageRank*, in *7th International Conference on Learning Representations, ICLR 2019*, 2019. arXiv:1810.0599.

[117] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, *Characterizing and Understanding GCNs on GPU*, *IEEE Computer Architecture Letters* **19** (2020), no. 1 22–25, [arXiv:2001.1016].

[118] J. L. McKinstry, S. K. Esser, R. Appuswamy, D. Bablani, J. V. Arthur, I. B. Yildiz, and D. S. Modha, *Discovering Low-Precision Networks Close to Full-Precision Networks for Efficient Inference*, in *Proceedings - 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing, EMC2-NIPS 2019*, pp. 6–9, 2019. arXiv:1809.0419.

[119] J. Yim, D. Joo, J. Bae, and J. Kim, *A gift from knowledge distillation: Fast optimization, network minimization and transfer learning*, in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4133–4141, 2017.

[120] D. Achlioptas, *Database-friendly random projections: Johnson-Lindenstrauss with binary coins*, in *Journal of Computer and System Sciences*, vol. 66, pp. 671–687, Elsevier, 2003.

[121] E. Bingham and H. Mannila, *Random projection in dimensionality reduction: Applications to image and text data*, in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 245–250, 2001.

[122] P. Li, T. J. Hastie, and K. W. Church, *Very sparse random projections*, in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 2006, pp. 287–296, 2006.

[123] L. Liu, L. Deng, X. Hu, M. Zhu, G. Li, Y. Ding, and Y. Xie, *Dynamic sparse graph for efficient deep learning*, in *7th International Conference on Learning Representations, ICLR 2019*, 2019. arXiv:1810.0085.

[124] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, *cuDNN: Efficient Primitives for Deep Learning*, *arXiv preprint arXiv:1410.0759* (2014) [arXiv:1410.0759].

[125] S. Ioffe and C. Szegedy, *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, in *32nd International Conference on Machine Learning, ICML 2015*, vol. 1, pp. 448–456, 2015. arXiv:1502.0316.

[126] Z. Chen, L. Deng, G. Li, J. Sun, X. Hu, L. Liang, Y. Ding, and Y. Xie, *Effective and Efficient Batch Normalization Using a Few Uncorrelated Data for Statistics Estimation*, *IEEE Transactions on Neural Networks and Learning Systems* **32** (2021), no. 1 348–362.

[127] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. Khudia, J. Law, P. Malani, A. Malevich, S. Nadathur, J. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy, *Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications*, *arXiv preprint arXiv:1811.09886* (2018) [arXiv:1811.0988].

[128] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan, *Sampled Dense Matrix Multiplication for High-Performance Machine Learning*, in *Proceedings - 25th IEEE International Conference on High Performance Computing, HiPC 2018*, pp. 32–41, 2019.

[129] S. L. Smith, P. J. Kindermans, C. Ying, and Q. V. Le, *Don't decay the learning rate, increase the batch size*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1711.0048.

[130] K. He, X. Zhang, S. Ren, and J. Sun, *Deep residual learning for image recognition*, in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 770–778, 2016. arXiv:1512.0338.

[131] S. Wu, G. Li, F. Chen, and L. Shi, *Training and inference with integers in deep neural networks*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1802.0468.

[132] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, *GIST: Efficient data encoding for deep neural network training*, in *Proceedings - International Symposium on Computer Architecture*, pp. 776–789, 2018.

[133] T. Zhang, S. Ye, K. Zhang, X. Ma, N. Liu, L. Zhang, J. Tang, K. Ma, X. Lin, M. Fardad, and Y. Wang, *StructADMM: A Systematic, High-Efficiency Framework of Structured Weight Pruning for DNNs*, *arXiv preprint arXiv:1807.11091* (2018) [arXiv:1807.1109].

[134] A. S. Morcos, D. G. Barrett, N. C. Rabinowitz, and M. Botvinick, *On the imporance of single directions for generalization*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1803.0695.

[135] Y. Zhang, J. Yang, and R. Gupta, *Frequent value locality and value-centric data cache design*, *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)* **35** (2000), no. 11 150–159.

[136] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, *A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps*, in *Proceedings - International Symposium on Computer Architecture*, vol. 13-17-June, pp. 41–53, 2015.

[137] M. Rhu, M. O'Connor, N. Chatterjee, J. Pool, Y. Kwon, and S. W. Keckler, *Compressing DMA Engine: Leveraging Activation Sparsity for Training Deep Neural Networks*, in *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2018-Febru, pp. 78–91, 2018. arXiv:1705.0162.

[138] D. Mishkin and J. Matas, *All you need is a good init*, in *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016. arXiv:1511.0642.

[139] W. B. Johnson and J. Lindenstrauss, *Extensions of Lipschitz mappings into a Hilbert space*, in *Contemporary mathematics*, vol. 26, pp. 189–206. 1984.

[140] N. Ailon and B. Chazelle, *The fast johnson-lindenstrauss transform and approximate nearest neighbors*, in *SIAM Journal on Computing*, vol. 39, pp. 302–322, SIAM, 2009.

[141] D. Achlioptas, *Database-friendly random projections*, in *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 274–281, 2001.

[142] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, *Gradient-based learning applied to document recognition*, *Proceedings of the IEEE* **86** (1998), no. 11 2278–2323.

[143] H. Xiao, K. Rasul, and R. Vollgraf, *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*, *arXiv preprint arXiv:1708.07747* (2017) [arXiv:1708.0774].

[144] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, *Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1*, *arXiv preprint arXiv:1602.02830* (2016) [arXiv:1602.0283].

[145] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, *GXNOR-Net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework*, *Neural Networks* **100** (2018) 49–58, [arXiv:1705.0928].

[146] S. Zagoruyko and N. Komodakis, *Wide Residual Networks*, in *British Machine Vision Conference 2016, BMVC 2016*, vol. 2016-Septe, pp. 87.1–87.12, 2016. arXiv:1605.0714.

[147] A. Krizhevsky and G. Hinton, *Learning multiple layers of features from tiny images*, *Cs.Toronto.Edu* (feb, 2009) 1–58.

[148] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet classification with deep convolutional neural networks*, *Communications of the ACM* **60** (2017), no. 6 84–90.

[149] K. Simonyan and A. Zisserman, *Very deep convolutional networks for large-scale image recognition*, in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015. arXiv:1409.1556.

[150] Jia Deng, Wei Dong, R. Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, *ImageNet: A large-scale hierarchical image database*, in *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pp. 248–255, 2009.

[151] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, *Intel Math Kernel Library*, in *High-Performance Computing on the Intel® Xeon Phi™*, pp. 167–188. Springer, 2014.

[152] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, *Distributed representations ofwords and phrases and their compositionality*, in *Advances in Neural Information Processing Systems*, vol. cs.CL, pp. 1–9, 2013. arXiv:1310.4546.

[153] K. Shim, M. Lee, I. Choi, Y. Boo, and W. Sung, *SVD-softmax: Fast softmax approximation on large vocabulary neural networks*, in *Advances in Neural Information Processing Systems*, vol. 2017-Decem, pp. 5464–5474, 2017.

[154] X. Zhang, L. Yang, J. Yan, and D. Lin, *Accelerated training for massive classification via dynamic class selection*, in *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, vol. 32, pp. 7566–7573, 2018. arXiv:1801.0168.

[155] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, *Graph convolutional neural networks for web-scale recommender systems*, in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 974–983, 2018. arXiv:1806.0197.

[156] Y. Prabhu, A. Kag, S. Harsola, R. Agrawal, and M. Varma, *Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising*, in *The Web Conference 2018 - Proceedings of the World Wide Web Conference, WWW 2018*, pp. 993–1002, 2018.

[157] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, *Fairseq: A fast, extensible toolkit for sequence modeling*, in *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Demonstrations Session*, pp. 48–53, 2019. arXiv:1904.0103.

[158] A. Farmahini-Farahani, J. H. Ahn, K. Morrow, and N. S. Kim, *NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules*, in *2015 IEEE 21st International Symposium on High Performance Computer Architecture, HPCA 2015*, pp. 283–295, 2015.

[159] H. Asghari-Moghaddam, Y. H. Son, J. H. Ahn, and N. S. Kim, *Chameleon: Versatile and practical near-DRAM acceleration architecture for large memory systems*, in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2016-Decem, pp. 1–13, 2016.

[160] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An imperative style, high-performance deep learning library*, in *Advances in Neural Information Processing Systems* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, and R. Garnett, eds.), vol. 32, pp. 8024–8035, Curran Associates, Inc., 2019. arXiv:1912.0170.

[161] S. Merity, N. S. Keskar, and R. Socher, *Regularizing and optimizing LSTM language models*, in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, 2018. arXiv:1708.0218.

[162] J. McAuley and J. Leskovec, *Hidden factors and hidden topics: Understanding rating dimensions with review text*, in *RecSys 2013 - Proceedings of the 7th ACM Conference on Recommender Systems*, pp. 165–172, 2013.

[163] M. Zhang, X. Liu, W. Wang, J. Gao, and Y. He, *Navigating with graph representations for fast and scalable decoding of neural language models*, in *Advances in Neural Information Processing Systems*, vol. 2018-Decem, pp. 6308–6319, 2018. arXiv:1806.0418.

[164] Y. Kim, W. Yang, and O. Mutlu, *Ramulator: A fast and extensible DRAM simulator*, *IEEE Computer Architecture Letters* **15** (2016), no. 1 45–49.

[165] M. Ott, S. Edunov, D. Grangier, and M. Auli, *Scaling Neural Machine Translation*, in *WMT 2018 - 3rd Conference on Machine Translation, Proceedings of the Conference*, vol. 1, pp. 1–9, 2018. arXiv:1806.0018.

[166] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, *BERT: Pre-training of deep bidirectional transformers for language understanding*, in *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, vol. 1, pp. 4171–4186, 2019. arXiv:1810.0480.

[167] N. Parmar, A. Vaswani, J. Uszkoreit, L. Kaiser, N. Shazeer, A. Ku, and D. Tran, *Image transformer*, in *35th International Conference on Machine Learning, ICML 2018*, vol. 9, pp. 6453–6462, 2018. arXiv:1802.0575.

[168] 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*, in *Advances in Neural Information Processing Systems*, vol. 2020-Decem, 2020. arXiv:2005.1416.

[169] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, *Transformer-XL: Attentive language models beyond a fixed-length context*, in *ACL 2019 - 57th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference*, pp. 2978–2988, 2020. arXiv:1901.0286.

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

[171] H. Shi, J. Gao, X. Ren, H. Xu, X. Liang, Z. Li, and J. T.-Y. Kwok, *Sparsebert: Rethinking the importance analysis in self-attention*, in *International Conference on Machine Learning*, pp. 9547–9557, PMLR, 2021.

[172] 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*, *International Conference on Learning Representations* (2020) [arXiv:2011.0400].

[173] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne, *JAX: composable transformations of Python+NumPy programs*, 2018.

[174] 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 *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 2021-Febru, pp. 841–853, 2021.

[175] T. Gale, M. Zaharia, C. Young, and E. Elsen, *Sparse gpu kernels for deep learning*, in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, vol. 2020-Novem, 2020. arXiv:2006.1090.

[176] C. NVIDIA, *CUSPARSE library*, *NVIDIA Corporation, Santa Clara, California* (2011), no. July.

[177] Z. Chen, Z. Qu, L. Liu, Y. Ding, and Y. Xie, *Efficient tensor core-based gpu kernels for structured sparsity under reduced precision*, in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2021.

[178] 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 networks*, in *Proceedings - International Symposium on Computer Architecture*, pp. 764–775, 2018. arXiv:1712.0150.

[179] 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 *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, vol. 2020-Octob, pp. 738–750, 2020.

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

[181] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts, *Learning word vectors for sentiment analysis*, in *ACL-HLT 2011 - Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, vol. 1, (Portland, Oregon, USA), pp. 142–150, Association for Computational Linguistics, jun, 2011.

[182] D. R. Radev, P. Muthukrishnan, V. Qazvinian, and A. Abu-Jbara, *The ACL anthology network corpus*, *Language Resources and Evaluation* **47** (feb, 2013) 919–944.

[183] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, *SQuad: 100,000+ questions for machine comprehension of text*, in *EMNLP 2016 - Conference on Empirical*

Methods in Natural Language Processing, Proceedings, pp. 2383–2392, 2016. arXiv:1606.0525.