UNIVERSITY OF CALIFORNIA

Los Angeles

Adaptive AI Algorithms for Generic Hardware & Unified Hardware Acceleration Architecture

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Feng Shi

2021

ABSTRACT OF THE DISSERTATION

Adaptive AI Algorithms
for Generic Hardware &
Unified Hardware Acceleration Architecture

by

Feng Shi
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2021
Professor Song-Chun Zhu, Chair

We are now in an era of the Big Bang of artificial intelligence (AI). In this wave of revolution, both industry and academia have cast numerous funds and resources. Machine learning, especially Deep Learning, has been widely deployed to replace the traditional algorithms in many domains, from the euclidean data domain to the non-euclidean domain. As the complexity and scale of the AI algorithms increase, the system host these algorithms requires more computational power and resources than before. Using the design of the modules of the video analytic platform as the use cases, we analyze the workload cost for computational resource and memory allocation during the execution of the system. The video analytic platform is a complex system that comprises various computer vision and decision-making tasks. Every module accomplishing a specific task is a stage in the pipeline of the video analytic platform. With the analyses mentioned above, we synthesize the adaptive AI algorithms from availability and variability perspectives, such as optimization with tensorization or matricization. We conceive the sparse Transformer and segmented linear

Transformer as the critical components for the human action recognition task. Constraint Satisfaction Problem is employed to assist the decision-making in the scene parsing stage. To facilitate this fulfillment of this task, we designed a hybrid model for graph learning-based SAT solver. Graph matching is employed at the final stage for the scene understanding task. We implemented a hybrid model of GNN and Transformer architecture. Finally, we design the unified hardware acceleration architecture for both dense and sparse data based on the optimizations of algorithms. Our design of the architecture targets the arithmetic operation kernels, such as matrix multiplications, with the help of data transformation and rearrangement. We first transform the inputs and weights with Winograd transform for dense convolution operations, then we feed the transformed data to the matrix multiplication accelerator. For sparse data, we need to utilize the index to nonzero to fetch data; therefore, the indexation, scattering, and gathering are crucial components, effective implementation will dramatically improve the system's overall performance. To improve the matrix multiplication accelerator's efficiency and reduce the number of heavy arithmetic operations and the number of memory accesses, we also conduct the hardware-based recursive algorithm, i.e., Strassen's algorithm for matrix multiplication.

The dissertation of Feng Shi is approved.

To my parents, professors, friends . . .

Jehovah is my Shepherd; I will lack nothing.

He makes me lie down in the green pastures;

He leads me beside waters of rest.

He restores my soul;

He guides me on the paths of righteousness

For his name's sake. [Psalm 23]

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Achieving this thesis was made possible and delightful by many people that have left their footprints in my life. First and foremost, I am deeply grateful to my advisor, Song-Chun Zhu, for supporting me in my Ph.D. studies and guide me through the process. He gave me freedom and vast support in pursuing projects for which I was passionate. His advice and constructive criticism empowered me to overcome many difficulties. I have learned a lot from him on both professional and personal levels as he played both roles of a mentor and a friend.

I had the opportunity of working closely with and learning from some of the top people in my research field. First and foremost, I am incredibly grateful to Professor Jason Cong and Professor Vijaykrishnan Narayanan for sharing many research opportunities and always being available for discussions or sharing ideas through emails and meetings. As the coauthors of several papers, Professor Vijaykrishnan Narayanan also spent enormous time proofreading the articles and provided valuable advice in editing them.

I also thank Professor Yingnian Wu provided guidance and advice in my research, especially concerning theoretical reasoning. During my study at UCLA, I also attended the weekly discussion and paper reading about computer architecture organized by Professor Tony Nowatzki; these activities opened my mind.

The works in this dissertation are also achieved with the collaborators: Chonghan Lee from The Pennsylvania State University, Yizhou Zhao, Liang Qiu from University of California Los Angeles, and Professor Tian Han from Stevens Institute of Technology.

This dissertation was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The grant number is GI18518.156870.

Finally, I would like to thank my family - my parents, my future wife (I always want to give her a bright life) for standing by my side all the time. The journey would not have been fulfilled without you.

VITA

2015–2021	Ph.D. (Computer Science, changed major), UCLA, Los Angeles, California.
2019–2020	Researcher/Engineer intern, DMAI Inc., Los Angeles, California.
2018–2019	Researcher/Engineer intern, CARA Inc., Los Angeles, California.
2016–2016	Research intern, Altera Inc., San Jose, California.
2014–2014	Research intern, Qualcomm Inc., San Diego, California.
2013–2015	Ph.D. (Electrical Engineering), UCLA, Los Angeles, California.
2012–2013	Visiting Scholar, Electrical Engineering Department, UCLA, Los Angeles, California.
2010–2012	Engineer, EDA Center at Institute of Computing of Chinese Academy of Science, Beijing, China.
2010–2012	Engineer, Microprocessor Research Center at Institute of Computing of Chinese Academy of Science, Beijing, China.
2007–2008	Master (Electrical and Computer Engineering), University Pierre and Marie Curie, Paris, France.
2006–2007	Research assistant, INRIA, Rocquencourt, France.
2004–2007	European Engineer degree/ Master, Institut des Sciences et Techniques en Yvelines, France.
2001–2004	Bachelor (Computer Science Department), Paul Sabatier University, Toulouse, France.

PUBLICATIONS

(List according to the number of citations by Google Scholar)

Quanshi Zhang, Ruiming Cao, **Feng Shi**, Ying Nian Wu, and Song-Chun Zhu, "*Interpreting CNN Knowledge via an Explanatory Graph*" in AAAI 2018.

Feng Shi, Haochen Li, Yuhe Gao, Benjamin Kuschner, and Song-Chun Zhu. "*Sparse Winograd Convolutional Neural Networks on Small-scale Systolic Arrays*" in ACM/SIGDA FPGA 2019.

Quanshi Zhang, Xin Wang, Ruiming Cao, **Feng Shi**, Ying Nian Wu, and Song-Chun Zhu, "*Extracting an Explanatory Graph to Interpret a CNN*" in IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 2020. DOI:10.1109/TPAMI.2020.2992207

Liang Qiu, Yizhou Zhao, Weiyan Shi, Yuan Liang, **Feng Shi**, Tao Yuan, Zhou Yu, Song-Chun Zhu. "*Structured attention for unsupervised dialogue structure induction*" in 2020 Empirical Methods in Natural Language Processing (EMNLP)

Quanshi Zhang, Xin Wang, Ruiming Cao, Ying Nian Wu, **Feng Shi**, Song-Chun Zhu, "Explanatory Graphs for CNNs" on arXiv, 2018, arXiv:1812.07997

Wei Yao, **Feng Shi**, Lei He, Siming Pan, Brice Achkir, Li Li, "*Power-bandwidth trade-off on TSV array in 3D IC and TSV-RDL junction design challenges*" in 2012 IEEE 21st Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS).

Feng Shi, Ziheng Xu, Tao Yuan, Song-Chun Zhu, "*HUGE2: a Highly Untangled Generative-model Engine for Edge-computing*" on Arxiv, arXiv:1907.11210.

Mohammad Khairul Bashar, Jaykumar Vaidya, RS Surya Kanthi, Chonghan Lee, **Feng Shi**, Vijaykrishnan Narayanan, Nikhil Shukla, "*Ferroelectric-based Accelerators for Computationally Hard Problems*" in Proceedings of the 2021 on Great Lakes Symposium on VLSI (GLSVLSI), 485-489

Feng Shi, Chonghan Lee, Mohammad Khairul Bashar, Nikhil Shukla, Song-Chun Zhu, Vijaykrishnan Narayanan, "Transformer-based Machine Learning for Fast SAT Solvers and Logic Synthesis," 2022 Asia and South Pacific Design Automation Conference (ASP-DAC) (under review).

Feng Shi, Ahren Yiqiao Jin, Song-Chun Zhu, "*VersaGNN: a Versatile accelerator for Graph neural networks*," in 2021 IEEE Transactions on Computers (under review).

Feng Shi, Chonghan Lee, Liang Qiu, Yizhou Zhao, Tianyi Shen, Shivran Muralidhar, Tian Han, Song-Chun Zhu, Vijaykrishnan Narayanan, "*STAR: Sparse Transformer-based Action Recognition*" in 2021 ACM Multimedia (MM).

Feng Shi, Chonghan Lee, Yizhou Zhao, Tian Han, Vijaykrishnan Narayanan, Song-Chun Zhu, "*Deep Graph Similarity through Linear Attention and Heat Kernel*," in Advances in Neural Information Processing Systems (NeurIPS), 2021 (under review).

Ji-Ye Zhao, Dong Liu, Dan-Dan Huan, Meng-Hao Su, Bin Xiao, Ying Xu, **Feng Shi**, Chen Chen, Song Wang, "Physical Design Methodology for Godson-2G Microprocessor," in 2010 Journal of Computer Science and Technology (JCST).

CHAPTER 1

Introduction

In the artificial intelligence paradigm, the essential step is feature extraction from data, and then the extracted features are utilized for different tasks, such as classification and detection. Data feature extraction has been evolved from traditional algorithms for handcrafted features to automated approaches with machine learning algorithms in recent decades. Significantly, the advent of deep learning, a branch of machine learning, brought efficiency and accuracy even beyond human ability. Many new methods have been developed in such a wave of advancement, from Multi-Layer Perceptron (MLP) to Convolutional Neural Networks (CNN) to Transformer with fully attention mechanism, and AI models are becoming more and more sophisticated. At the same time, researchers have been driven by the increased requests to design domain-specific hardware to accommodate the complexity of machine learning models. In such circumstances, hardware and software co-designing become highly crucial. We target adaptive AI algorithms and unified hardware to construct the system to leverage both performance and cost from both directions.

We employ the video analytic platform, an ensemble of cognitive tasks with various algorithms, as a working example. We analyze the overall pipeline of the video analytic platform from module to module and abstract the standard computational kernels from those modules. We take these kernels as the bridge between software and hardware.

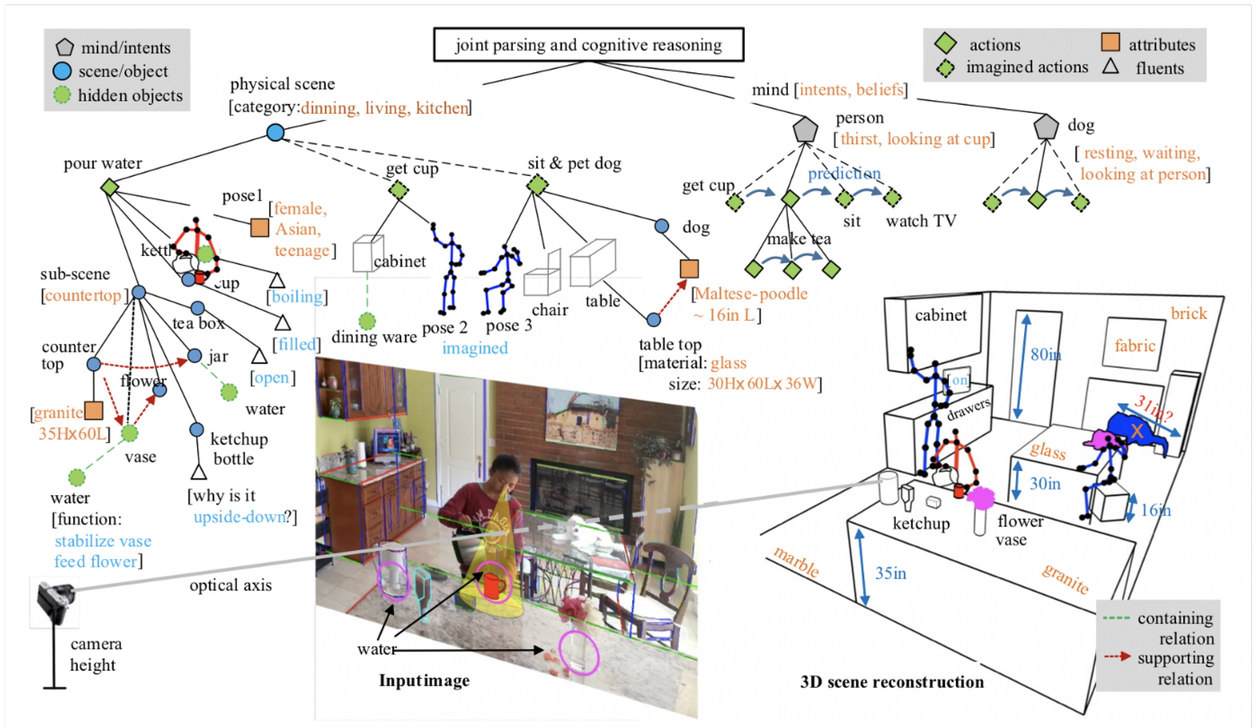


Figure 1.1: Scenario of video analytic platform

1.1 The Need for Adaptive Artificial Intelligence Algorithms

1.1.1 Video Analytic Platform

Video today dominates network traffic, data center storage, and even edge platforms for low-power applications. However, real-time analytic on large video corpora remains out of reach for many tasks, such as object-based search and reconstruction and analysis of 3D scenes. Such real-time capabilities will have many critical commercial and national security applications. To provide a concrete context for developing the architecture and system stack, we target a set of applications that promise to be commercially crucial over the next 10-20 years, that include a rich diversity of computation and data access patterns, and that is fundamentally challenging for existing architecture and systems.

The above Figure 1.1 shows that a robust video analytic platform can understand and

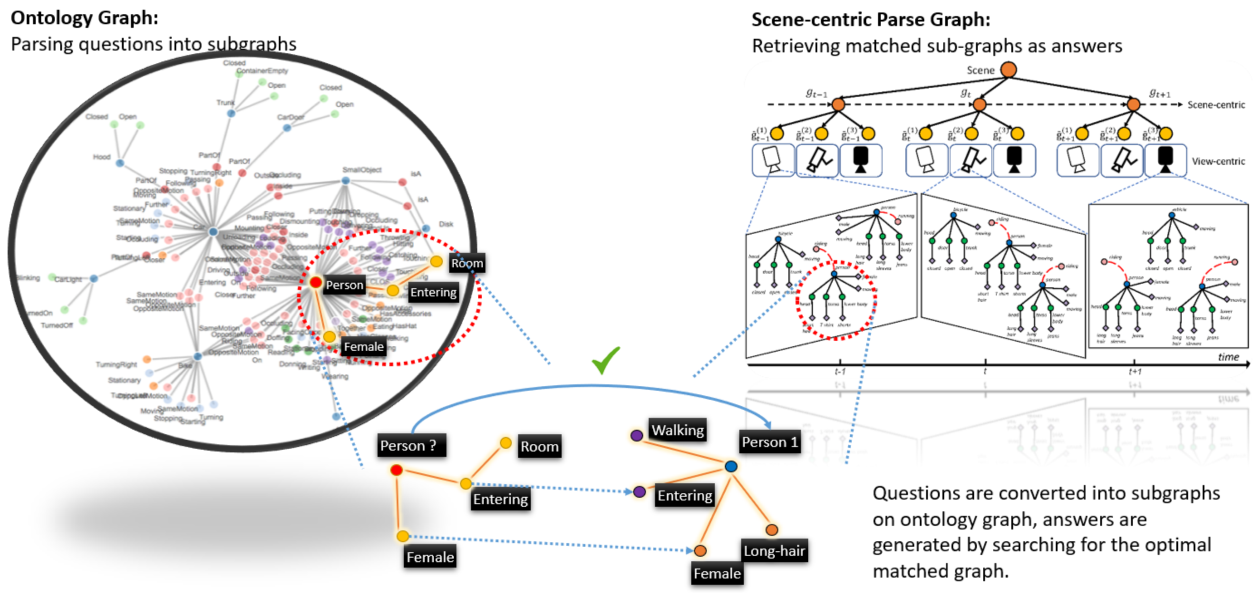

Ontology Graph:
Parsing questions into subgraphs
Scene-centric Parse Graph:
Retrieving matched sub-graphs as answers
Scene
Scene-centric
View-centric
time
Person
Room
Entering
Female
Person ?
Walking
Person 1
Long-hair
Questions are converted into subgraphs on ontology graph, answers are generated by searching for the optimal matched graph.

Figure 1.2: An example of intelligent Q.A. system

explain the scene in the frames captured by the cameras. This platform first detects the existing objects in the frame, then tries to derive the relational connections among these objects. An And-Or Graph/parse graph can depict the decompositional, spatial, and temporal relations (even causal relation) among these objects.

After constructing the And-Or Graph and distilling a parse graph from it, we want to explain what the machine has seen from the picture. A Q&A sub-platform can further investigate the scene understanding ability of the system as a back-end into the video analytic platform. This sub-platform possesses a knowledge database storing the ontology graph. It receives the sub-graph (parse graph or raw And-Or graph) parsed by the front-end of the video analytic platform as the questions (depicted in the previous paragraph). Then it retrieves a matched sub-graph from the ontology graph as answers, as shown in Figure 1.2.

Figure 1.2 displays an overview of the system. Take a concrete example for demonstration, say, we have captured a scene with a graph with leaf nodes as, a girl, a chair, a desk, and a candle; we also have additional links describing, the girl is sitting on the chair, the candle is

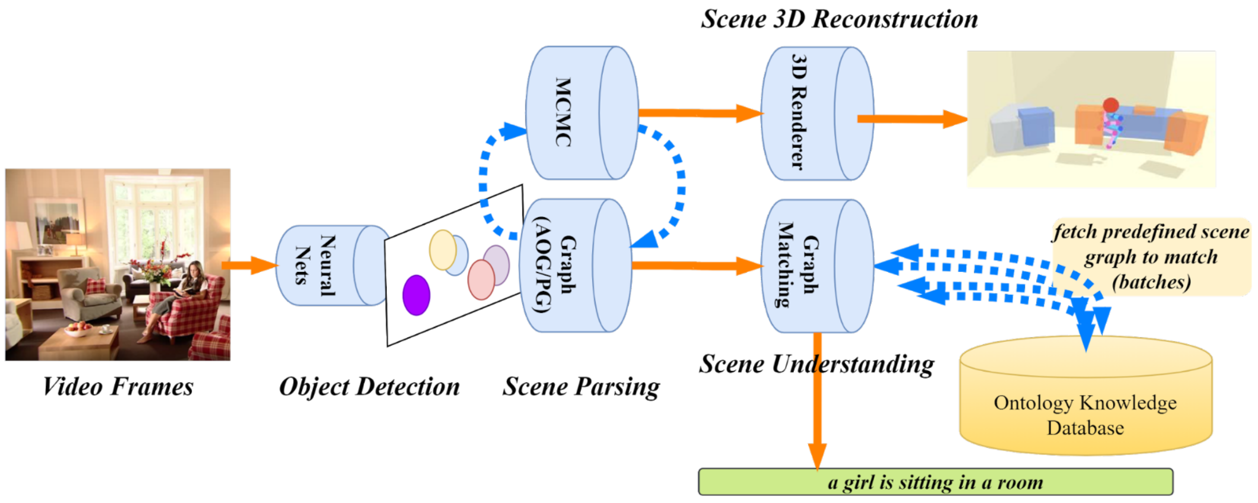


Figure 1.3: Illustration of the pipeline of the scene parsing and understanding

on the desk. The front end concludes a description, saying, “a girl is sitting in a room with one chair, one desk, and one candle.” However, such expressions are still unnatural in real life. The sub-graph, which consists of a chair(s), a desk(s), and candle(s), can be matched to a category “home office,” which is a sub-graph stored in the ontology graph database, though this sub-graph may have a node lamp instead of a candle (the most similar one). If we turn the phrase into “a girl is sitting in a home office,” the expression becomes more natural as a human being. Figure 4.2 and Figure 1.4 illustrate such a scenario.

1.1.2 The analysis of the bottlenecks in the platform

We provide the analysis of the bottlenecks in the platform stage by stage according to Figure 4.2.

1. The object detection stage mainly consists of convolutional neural networks. This stage is at the same time compute- and memory-bound; the data fetched into memory are repeatedly read for the filter windows where the convolution operations are performed to generate new feature maps. In different layers, the amount of computation varies.

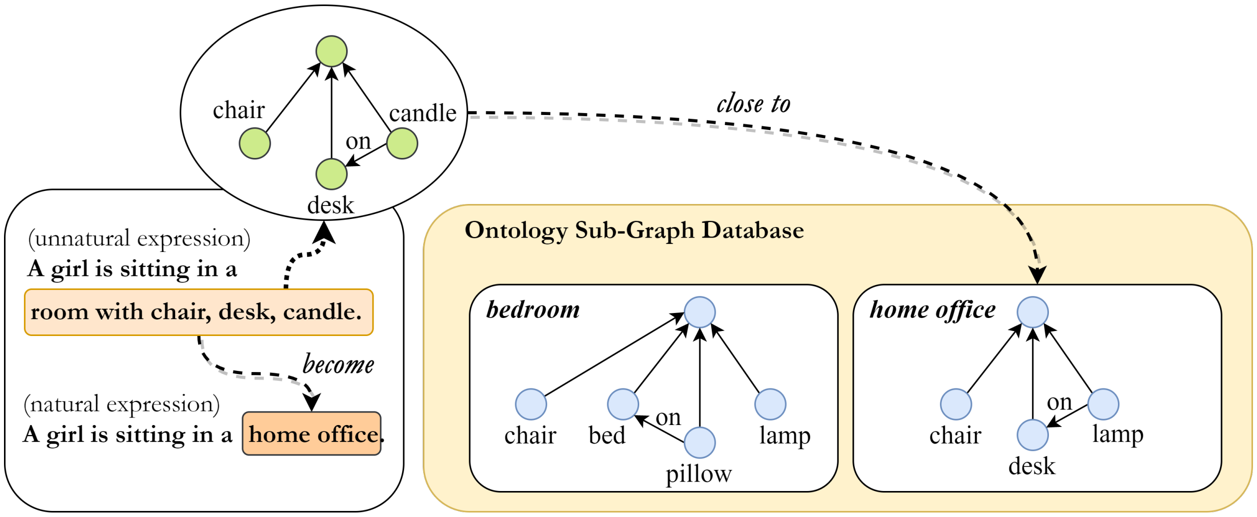


Figure 1.4: Graph matching used for scene understanding

2. The stage for the 3D reconstruction utilizes the MCMC algorithm. Such algorithms usually generate many intermediate results, which causes a high rate of memory accesses and disk accesses where the 3D object models are stored. The trial-and-error sampling procedure also triggers a bunch of function calls. Furthermore, as the 3D environment becomes more prominent and more complex, the complexity of the query-reasoning system proliferates.

3. The graph matching stage is also both compute- and memory-bound. For an iteration of graph matching, the conventional algorithm is an NP-hard, usually a compute-intensive case. The matching phase tries to find a similar graph (or some pattern/configuration) in the database for a computed parse graph from the scene parsing stage. When there are millions of graphs (predefined patterns) stored in the ontology base (knowledge database), the matching problem becomes painful. Therefore, this stage is again both compute- and memory-bound.

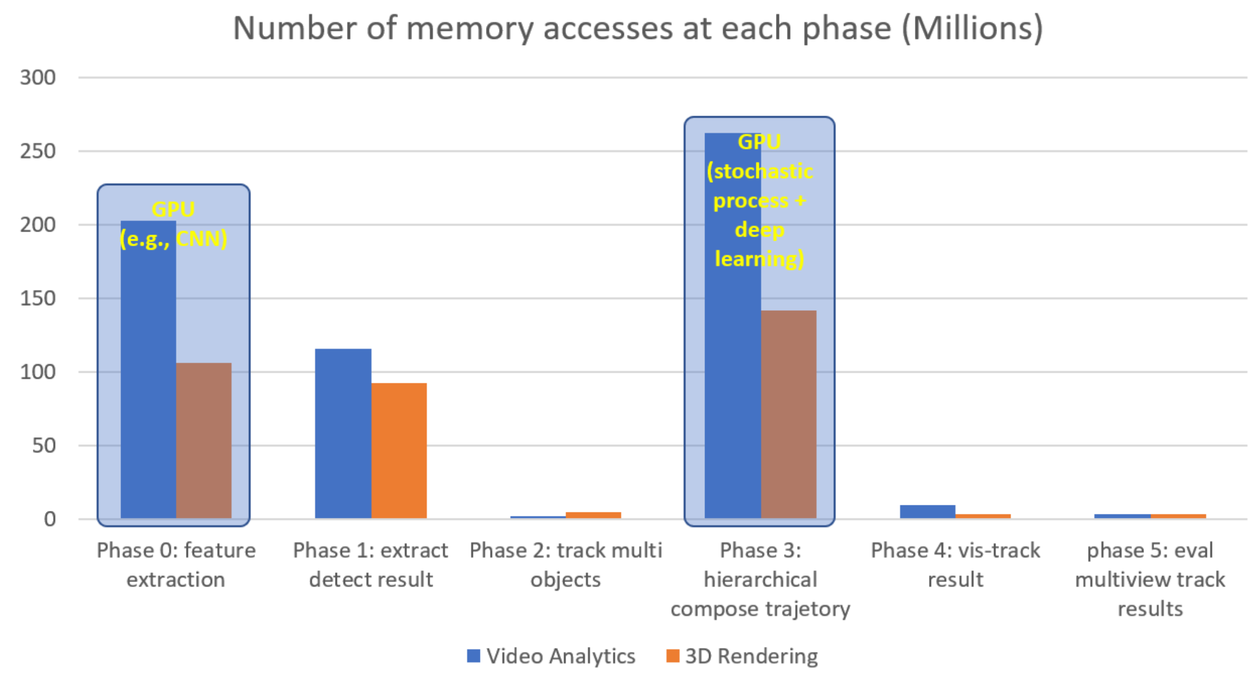


Figure 1.5: The benchmarks of different computational kernels

1.1.3 The need for a new computing paradigm

Video data occupy many physical resources, e.g., network traffic, and storage; it is also the source of many important commercial and national-security needs. For example, content-based video search remains far from the goal of enabling high-level specification of objects of interest and the ability to search across various camera resolutions, perspectives, and viewing conditions. Another important capability is the ability to reconstruct a 3D environment and enable rich Spatio-temporal and causal queries. As the target 3D environment grows, the data volumes increase rapidly; for example, 30 cameras monitoring just one portion of a single building floor generate TBs of data per hour. Increasingly, these tasks must happen at line speeds to keep up with the rate of new data products, and often real-time processing is needed to draw timely inferences. The algorithms currently entail deep learning, dynamic programming, Monte-Carlo iteration, graph analytic, and natural language processing. Deployments of real-time video analytic will need to do as much processing in

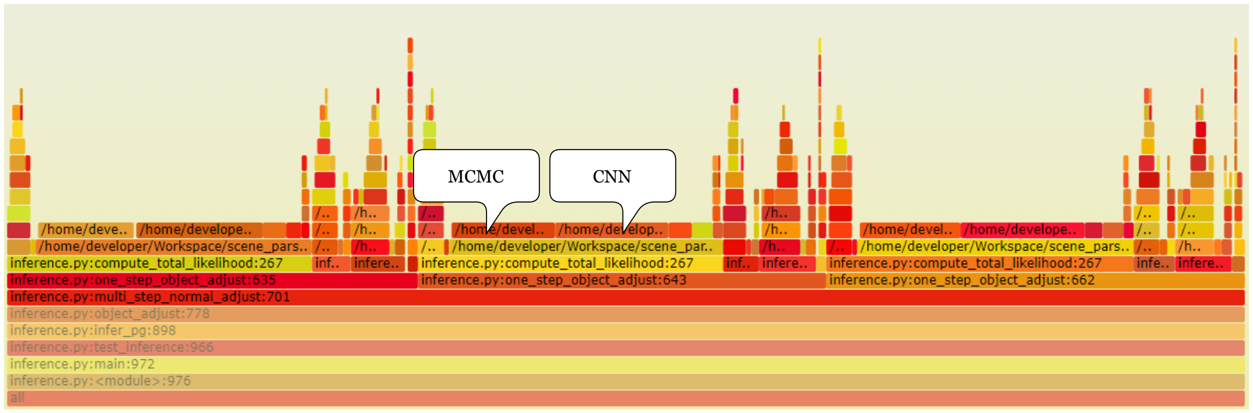


Figure 1.6: Flame graphs illustrate the tree of function calls

the cameras as possible, so will span edge devices to cloud in implementing an end-to-end solution. Figure 1.5 and Figure 1.6 provide the statistics and profiling of the video analytic system.

functions	# of calls	% of time
one_step_object_adjust(): MCMC process	662	18.90%
compute_total_likelihood(): MCMC process	265	33.57%
render_scene(): 3D rendering	226	31.57%
object_detection(): Neural Network	279	15.96%

Table 1.1: Statistics of function calls (Python) in 3D reconstruction platform

For the two example platforms, i.e., video analytic and 3D reconstruction platforms, the computation- and memory-bound components are the feature extraction and the hierarchical compose trajectory phrases. As an example, Table 3.5 summarizes the statistics of function calls in a 3D reconstruction platform. Most deep learning based algorithms, i.e., Deep Neural Networks (DNNs), Convolutional Neural Networks (CNNs), long/short-term memory (LSTM), Recurrent neural networks (RNNs), or Transformers, belong to feature extraction category; while the stochastic process-based planning/decision-making algorithms, i.e., reinforcement learning (RL), Monte-Carlo Markov Chain (MCMC), and constraint satisfaction problems

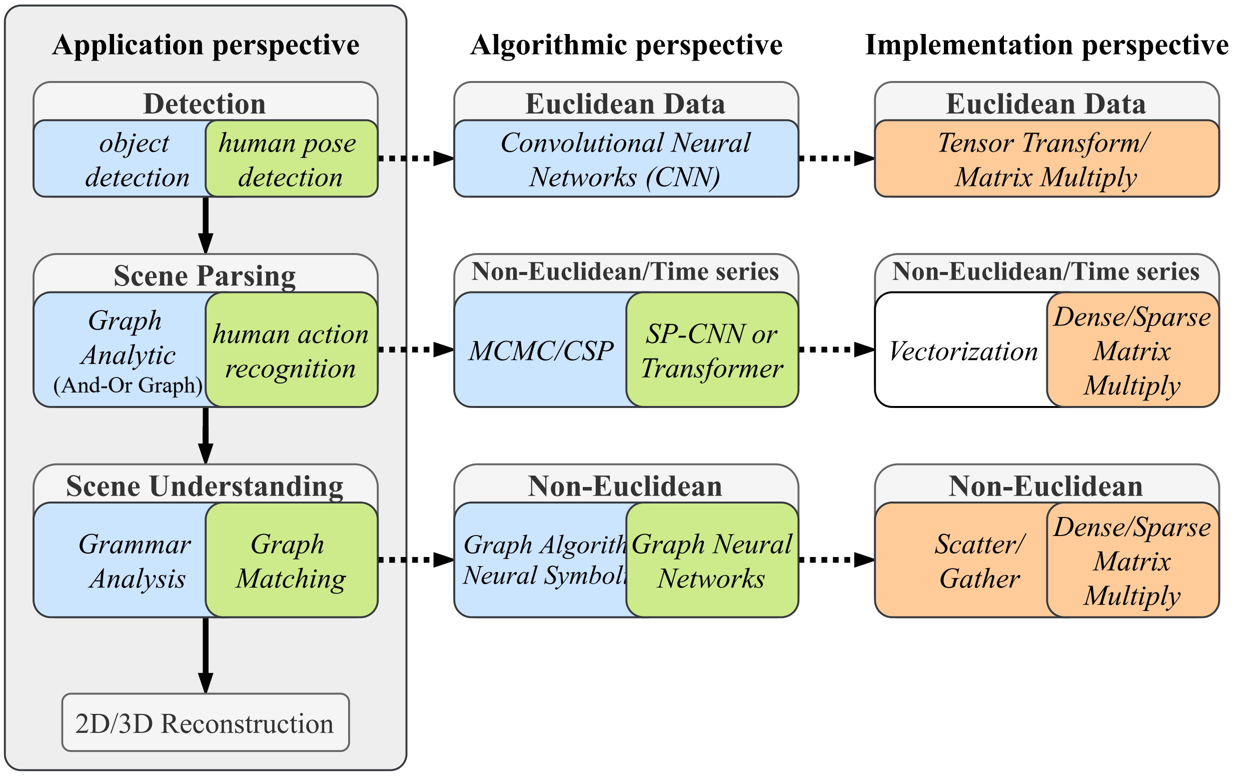
(CSP), fall into the hierarchical compose trajectory category, which need to make decision through the probability distribution (or any stochastic process) of the interaction between actions and states. Sometimes, these two categories are combined to perform the decision-making tasks, especially, in some deep RL models utilize CNNs as policy network to evaluate the rewards and returns.

In the following chapters, in the context of the video analytic platform, we conduct various algorithms for relevant applications in such a platform. Our project collects a diverse set of algorithms into a benchmark suite of challenging applications. However, we primarily target three algorithms/applications that are highly challenging for current systems. Like we are considering, advancements would enable substantially new capabilities, which is of high societal relevance, leading to high impact from our work. Furthermore, these applications use key algorithmic building blocks whose acceleration will benefit many other applications, such as machine learning, graph analytic, etc.

1.2 Algorithm abstraction and their implementation

The left column, Figure 1.7(a), gives an example of the potential pipeline architecture of a video analytic platform. The middle column, Figure 1.7(b), summarizes the kernel algorithms deployed in the corresponding module of the pipeline. The rightmost column of Figure 1.7 provides implementation details from the arithmetic operation perspective. The key to achieving a high degree of parallelism is to vectorize, matricize, or tensorize the data access and processing patterns. The vectorization, matricization, and tensorization are also helpful to bulky access to the data, which results in a higher locality for caching and the possibility of reuse of data. Meanwhile, the vectorization (same for matricization and tensorization) for the irregular data, especially for the non-euclidean data, can be fulfilled with scattering and gathering operation through indexing data.

Chapter 2 shows the human action recognition engine algorithm, an intermediate module



(a) The pipeline of video analytic platform

(b) The algorithms used at each stage

(c) basic arithmetic operations used at each stage

Figure 1.7: The abstraction of algorithms and their implementation

in the video analytic platform for the scene parsing task. This module utilizes the human skeleton data as input, including temporal information derived from the video sequence and the spatial information represented by the coordinates in 2D or 3D space in each frame. The skeleton data are extracted with the help of the human pose detection engine as the front-end pipeline of the video analytic platform. The human action recognition engine in this work is designed with two pipelines; each pipeline is a Transformer encoder structure and treats the skeleton sequence in spatial or temporal domains, respectively. Due to different data properties within spatial and temporal domains, the two encoders are designed to process the sparse data with tree structure and time-series data with the linear causal sequence.

Chapter 3 presents the graph matching algorithm with deep learning solution. Graph matching is a crucial problem in many fields, and it is also an NP-hard problem with very high complexity. The conventional approaches are often time-consuming and lacking enough parallelism. The model proposed consists of the graph neural networks for extract graph nodes' hidden features or embedding. Then used the dot product between pairs of node features to conduct a similarity matrix. The key to such a problem lies in whether the two graphs' topology is consistent. With the help of graph diffusion, which uses the heat kernel to construct graph wavelets, we can plot the energy diffusion map on the graph. An energy diffusion map is an efficient way to depict the topology of graphs by assessing the energy flowing from node to node in a graph. Therefore, we integrate the energy diffusion map into the graph matching problem and translate such kind of problem into an energy diffusion comparison problem.

Chapter 4 proposes a CSP/SAT solver based on deep learning approach. Constraint Satisfaction Problem (CSP) and Boolean Satisfiability Problem (SAT) are ubiquitous and adopted in many fields. Such kinds of problems are also NP-complete. Many heuristic approaches have been invented; however, these algorithms mostly focus on a "trial and error" iterative manner that cannot grasp the overall topological relation between nodes of a bipartite graph constructed by the variables and clauses. They neither can summarize the hidden features embedded inside the topology and those derived from nodes' connectivity. We also introduce the concept "meta-paths" into our model to extract the correlated feature crossing the link between clauses and variables of the bipartite graphs depicting the CSP instance.

Chapter 5 aims to accelerate the convolution operations in convolutional neural networks, the bone architecture used in most modern computer vision tasks. We first apply the Winograd transform on the input tensor and the trained weights (or kernels). This transform can dramatically reduce the number of operations compared with the direct methods of convolution operations. With the rearrangement of data access pattern crossing dimensions, we apply the matricization to those transformed tensors then perform matrix multiplication between input tensors and weight tensors. So the hardware design part is simplified to the optimization to accelerate the performance for matrix multiplications. The complexity of hardware design remains manageable and easy to handle.

Chapter 6 proposes an approach to tackle the complexity of the deconvolution operations. Deconvolution is a key component of the generative model, such as generative adversarial networks (GAN), variational autoencoder (VAE). By discovering the regularity of the data access pattern to the input data and kernels of the deconvolution operations, we decompose the input tensor and weight tensor into several regular repeated combinations. The regular convolution operator then operates on each of those combinations. This method simplifies the hardware design, and also, the hardware tailored for regular convolution operation can be deployed directly without any modification.

Chapter 7 focuses on designing a versatile accelerator for both sparse and dense matrix multiplications. The Graph Neural Network (GNNs) is a very suitable use case for this accelerator since the two phases, feature transform and feature aggregation, of a graph convolution layer are dense and sparse matrix multiplications, respectively. Of course, this accelerator is not limited to the GNN model acceleration, especially the spatial and temporal transformer model in Chapter 2 can also be easily deployed onto it. The hardware algorithm utilized in this accelerator design can be regarded as an extension of the accelerator design in Chapter 5. Based on the recursive matrix multiplication fashion used in Chapter 5 we further reduce the number of heavy arithmetic operations, i.e., multiplications, through Strassen's algorithm when in the dense mode; for the sparse mode, we introduce the auxiliary FIFOs for keeping the index to nonzero in processing. In addition, a software approach algorithm for regrouping the sparse matrices assists in reducing the idled time slots during index alignment.

1.3 Acknowledgment of Funding Source and Disclaimer

The research was funded in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. The grant number is GI18518.156870. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

Part I

Adaptive Artificial Intelligence Algorithms – AI for Chip

CHAPTER 2

STAR: Sparse Transformer-based Action Recognition

This chapter, as shown in Figure 2.1, presents the human action recognition engine algorithm, an intermediate step of the scene parsing in the video analytic platform. It uses the human skeleton data to classify the human action. The skeleton data, extracted with the help of the human pose detection engine as the front-end pipeline of the video analytic platform, comprises a sequence of video frames, the coordinates in 2D or 3D space in each frame presents the spatial information; and each sequence of the same body joint in the frames forms the temporal information. This thesis's human action recognition engine consists of two pipelines; each pipeline is a Transformer encoder structure and treats the skeleton sequence in spatial or temporal domains, respectively. Due to different data properties within spatial and temporal domains, the two encoders process the sparse data with tree structure and time-series data with the linear causal sequence.

2.1 Introduction

Human action recognition plays a crucial role in many real-world applications, such as holistic scene understanding, video surveillance, and human-computer interaction [79, 27]. In particular, skeleton-based human action recognition has attracted much attention in recent years and has shown its effectiveness. The skeleton representation contains a time series of 2D or 3D coordinates of human key-joints, providing dynamic body movement information that is robust to variations of light conditions and background noises in contrast to raw RGB representation.

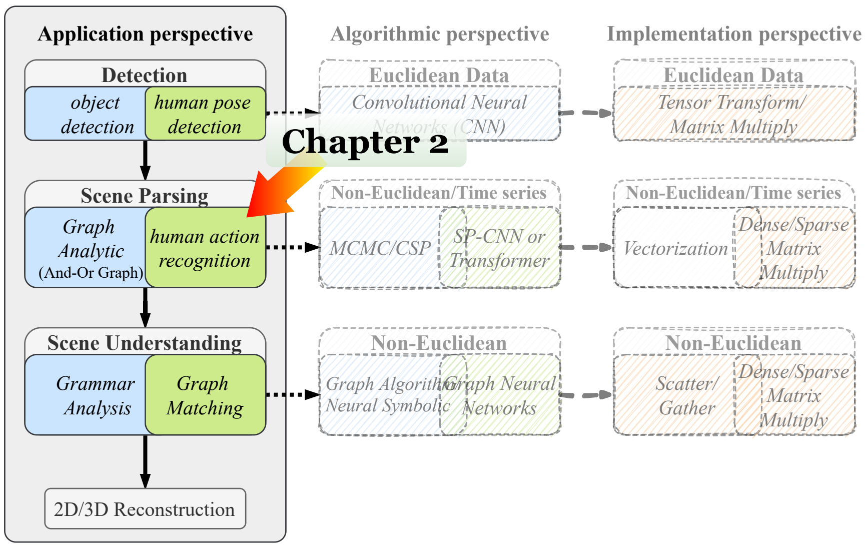


Figure 2.1: The overview of the position of this chapter

Earlier skeleton-based human action recognition methods focus on designing hand-crafted features extracted from the joint coordinates [161, 164] and aggregating learned features using RNNs and CNNs [185, 175, 192, 107, 90]. However, these methods rarely explore the relations between body joints and result in unsatisfactory performance. Recent methods focus on exploring the natural connection of human body joints and successfully adopted the Graph Convolutional Networks (GCNs), especially for non-Euclidean domain data, similar to Convolutional Neural Networks (CNNs) but executing convolutional operations to aggregate the connected and related joints' features. Yan et al. [180] proposed a ST-GCN model to extract discriminative features from spatial and temporal graphs of body joints. Following the success of ST-GCN, many works proposed optimizations to ST-GCN to improve the performance and network capacity [140, 103, 110].

However, the existing GCN-based models are often impractical in real-time applications due to their vast computational complexity and memory usage. The baseline GCN model, e.g., ST-GCN, consists of more than 3.09 million parameters and costs at least 16.2 GFLOPs to run inference on a single action video sample [180]. DGNN, which is composed of incremental GCN modules, even contains 26 million model parameters. [140] Such high model complexity leads to difficulty in model training and inference, makes the model not suitable for deployment on edge devices. Furthermore, these GCN-based models process fixed-size action videos by padding repetitive frames and zeros to match the maximum number of frames and persons depicted in the videos. These additional paddings increase the latency and memory required hindering their adoption in real-time and embedded applications.

This paper proposes a *sparse transformer-based action recognition* (STAR) model as a novel baseline for skeleton action modeling to address the above shortcomings. Transformers have been a popular choice in natural language processing. Recently, they have been employed in computer vision to attain competitive results compared to convolutional networks, while requiring fewer computational resources to train [42, 82]. Inspired by these Transformer architectures, our model consists of spatial and temporal encoders, which apply sparse attention and segmented linear attention on skeleton sequences along the spatial and temporal dimension, respectively.

Our sparse attention module along the spatial dimension performs sparse matrix multiplications to extract correlations of connected joints, whereas previous approaches utilize dense matrix multiplications where most of the entries are zeros, causing extra computation. The segmented linear attention mechanism along temporal dimension further reduces the computation and memory usage by processing variable length of sequences. We also apply segmented positional encoding to the data embedding to provide the concept of time-series ordering along the temporal dimension of variable-length skeleton data. Additionally, segmented context attention performs weighted summarization across the entire video frames, making our model robust compared to GCN-based models with their fixed-length receptive field on the temporal dimension.

Compared to the baseline GCN model (ST-GCN), our model (STAR) achieves higher performance with much smaller model size on the two datasets, NTU RGB+D 60 and 120. The major contributions of this work are listed below:

- We focus on designing an efficient model purely based on self-attention mechanism. We propose *sparse transformer-based action recognition* (STAR) model that process variable length of skeleton action sequence without additional preprocessing and zero paddings. The flexibility of our model is beneficial for real-time applications or edge platforms with limited computational resources.

- We propose a sparse self-attention module that efficiently performs sparse matrix multiplications to capture spatial correlations between human skeleton joints.

- We propose a segmented linear self-attention module that effectively captures temporal correlations of dynamic joint movements across time dimension.

- Experiments show that our model is 5$\sim$7$\times$ smaller than the baseline models while providing 4$\sim$18$\times$ execution speedup.

2.2 Related works

2.2.1 Skeleton-Based Action Recognition

Recently, skeleton-based action recognition has attracted much attention since its compact skeleton data representation makes the models more efficient and free from the variations in lighting conditions and other environmental noises. Earlier methods to skeleton-based action modeling have mainly worked on designing hand-crafted features and relations between joints [27, 161, 164]. Recently, by looking into the inherent connectivity of the human body, Graph Convolutional Networks (GCNs), especially, ST-GCNs have gained massive success in getting satisfactory results in this task. The model consists of spatial and temporal convolution modules similar to conventional convolutional filters used for images [180]. The graph adjacency matrix encodes the skeleton joints' connections and extracts high-level spatial representations from the skeleton action sequence. On the temporal dimension, 1D convolutional filters facilitate extracting dynamic information.

Many following works have proposed improvements to ST-GCN to improve the performance. Li et al. [103] proposed AS-GCN, which leveraged the potential of adjacency matrices to scale the human skeleton's connectivity. Furthermore, they generated semantic links to capture better structural and action semantics with additional information aggregation. Lei et al. [141] proposed Directed Graph Neural Networks (DGNNs), which incorporate joint and bone information to represent the skeleton data as a directed acyclic graph. Liu et al. [110] proposed a unified spatial-temporal graph convolution module (G3D) to aggregate information across space and time for effective feature learning.

Some studies have been focusing on the computational complexity of GCN-based methods. Cheng et al. [24] proposed Shift-GCN, which leverages shift graph operations and point-wise convolutions to reduce the computational complexity. Song et al. [147] proposed multi-branch ResGCN that fuses different spatio-temporal features from multiple branches and used residual bottleneck modules to obtain competitive performance with less number of parameters. Compared to these methods, our spatial and temporal self-attention modules have several essential distinctions: our model can process variable length of skeleton sequence without preprocessing with zero-paddings. Our model can retrieve global context on the temporal dimension by applying self-attention to the input sequence's entire frames.

2.2.2 Transformers and Self-Attention Mechanism

Vaswani et al. [155] first introduced Transformers for machine translation and have been the state-of-the-art method in various NLP tasks. For example, GPT and BERT [125, 40] are currently the Transformer-based language models that have achieved the best performance. The core component of Transformer architectures is a self-attention mechanism that learns the relationships between each element of a sequence. In contrast to recurrent networks that process sequence in a recursive fashion and are limited to attention on short-term context, transformer architectures enable modeling long dependencies in sequence. Furthermore, the multi-head self-attention operations can be easily parallelized. Recently, Transformer-based models have attracted much attention in the computer vision community. Convolution operation has been the core of the conventional deep learning models for computer vision tasks. However, there are downfalls to the operation. The convolution operates on a fixed-sized window, which only captures short-range dependencies. The same applies to GCNs where the Graph Convolution operation is incapable of capturing long-range relations between joints in both spatial and temporal dimensions.

Vision Transformer (ViT) [42] is the first work to completely replace standard convolutions in deep neural networks on large-scale image recognition tasks. Huang et al. [82] explored the sparse attention to study the trade-off between computational efficiency and performance of a Transformer model on the image classification task. A recent study [121] proposed a hybrid model consists of the Transformer encoder and GCN modules on the skeleton-based human action recognition task. Nevertheless, no prior study has completely replaced GCNs with the Transformer architecture to the best of our knowledge.

2.3 Methodology

In this section, we present the algorithms used in our model and the relevant architecture of our model. Section 2.3.1 depicts the **sparse multi-head self-attention** (MHSA) mechanism used in spatial Transformer encoder module; Section 2.3.2 introduces the novel data format and the relevant **linear multi-head self-attention** (MHSA) mechanism for temporal Transformer encoder; Section 2.3.3 shows the overall framework of our model and related auxiliary modules.

2.3.1 Spatial domain: Sparse MHSA

The crucial component of our *spatial transformer encoder* is the *sparse multi-head self-attention* module. GCN-based models and previous Transformer models, such as ST-GCN and ST-TR, utilize dense skeleton representation to aggregate the features of neighboring

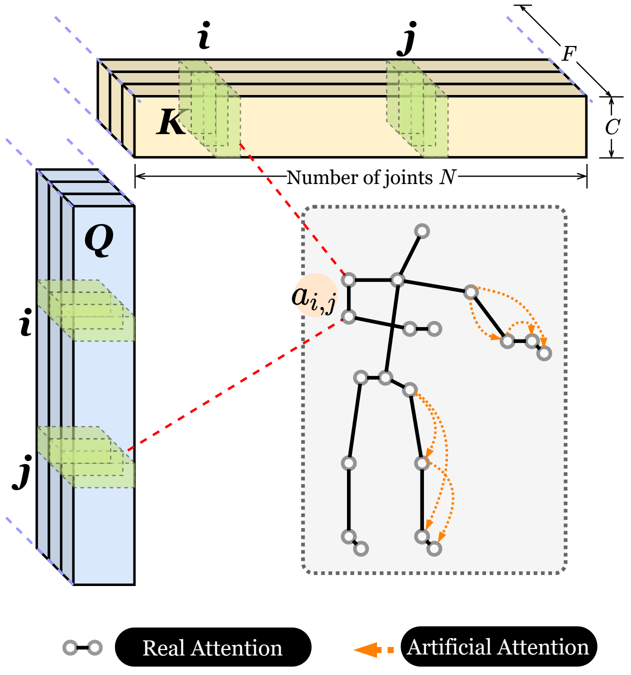


Figure 2.2: Illustration of our Sparse attention module: Given the queries Q and the keys K of the skeleton embedding, feature vectors of joint i and j are correlated with attention weight $\alpha_{i,j}$ The solid black line on the skeleton represents the physical connection of human skeleton. The dashed line connecting two joints represents the artificial attention of joints.

nodes. This dense adjacency matrix representation contains 625 entries for the NTU dataset, while the actual number of joint connections representing the skeletons is only 24. It means that 96% of the matrix multiplications are unnecessary calculations for zero entries. So we propose a sparse attention mechanism, which only performs matrix multiplications on the sparse node connections. This allows each joint to only aggregate the information from its neighboring joints based on the attention coefficients, which are dynamically assigned to the corresponding connections.

The joint connections are based on the topology of skeleton, which is a tree structure. The attentions inherited from this topology are seen as *physical attention* (or *real attention*), as illustrated in Figure 2.2. To augment the attending field, we also artificially add more links between joints according to the logical relations of body parts, and we call these artificially created attentions as *artificial attention*, as the dashed yellow arrows shown in Figure 2.2. For simplicity, suppose that the skeleton adjacency matrix is A, then the artificial links for additional spatial attention are obtained through A^2 and A^3. Hence, in our model, the spatial attention maps are evaluated based on the topology representation of $A + A^2 + A^3$.

The sparse attention is calculated according to the connectivity between joints. As described in below equations: after the embedding in Equation 2.1, the joint-to-joint attention between a pair of connected joints is computed first by an exponential score of the dot product of the feature vectors of these two joints (Equation 2.2), then the score is normalized by the sum of exponential scores of all neighboring joints as described in Equation 2.3.

$$Q = XW_q, K = XW_k, V = XW_v \tag{2.1}$$

$$\alpha_{i,j} = \frac{\langle q_i, k_j \rangle}{\sum_{n \in N(i)} \langle q_i, k_n \rangle} \tag{2.2}$$

$$v'_i = \sum_{j \in N(i)} \alpha_{i,j} v_j, \qquad \text{or } V' = \mathcal{A}V \tag{2.3}$$

where Q, K, and V are queries, keys, and values in Transformer's terminology, respectively; and $q_i = Q(i)$, $k_j = K(j)$, $v_j = V(j)$, and $\langle q, k \rangle = exp\left(\frac{q^T k}{\sqrt{d}}\right)$. Finally, we obtain attention maps $\mathcal{A}$ as multi-dimension (multi-head) sparse matrices sharing the identical topology described by a single adjacency matrix (including links for the artificial attention), where attention coefficients are $\mathcal{A}(i, j) = \alpha_{i,j}$. The sparse operation can be fulfilled with tensor gathering and scattering operations for parallelism.

2.3.2 Temporal domain: Segmented Linear MHSA

The most apparent drawbacks in the previous approaches [180, 142] are utilizing (1) the fixed number of frames for each video clip and (2) zero-filling for the non-existing second person. The first drawback constrains their scalability to process video clips longer than the predefined length and their flexibility on a shorter video clip. The second drawback due to

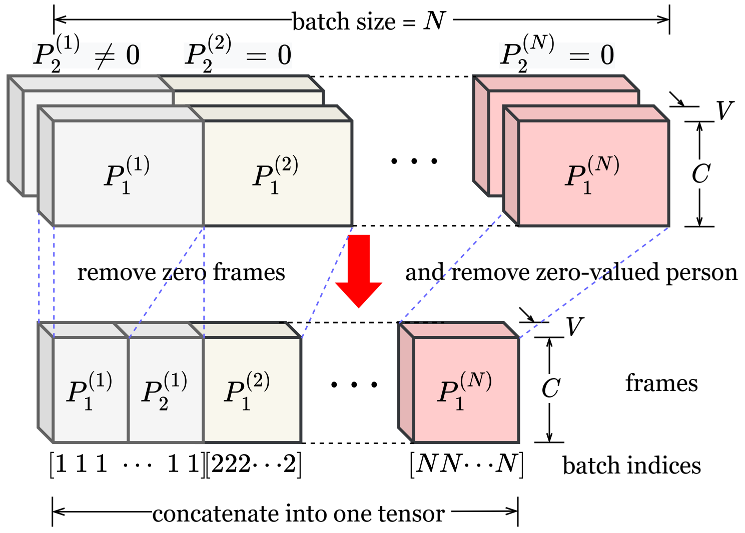


Figure 2.3: Illustration of our data format used in our framework: Previous works used the upper data format, which has fixed-sized time and person dimensions. Our work adopts new data format on the bottom, which has combined batch, person, and time dimensions into a single variable length sequence.

the zero's participation in computation causes latency degradation. Moreover, a significant amount of memory space is allocated to those zero-valued data during the computation. So we propose a compact data format to bypass these drawbacks. Also, we propose Segmented Linear MHSA to process our compact data format.

2.3.2.1 Variable Frame Length Data Format

The Figure 2.3 shows the comparison between our data format and the format used by previous works. In the data format adopted by previous works, longer videos are cut off to the predefined length and shorter videos are padded with repeated frames. Furthermore, the frames with a single person are all zero-padded to match the fixed number of persons. The upper data format from Figure2.3 illustrates the NTU RGB+D data format used by previous works. In each fixed-length video $V^{(i)}$, $P_1^{(i)}$ and $P_2^{(i)}$ represent two persons. In NTU RGB+D 120 dataset, only 26 out of 120 actions are mutual actions, which means that the

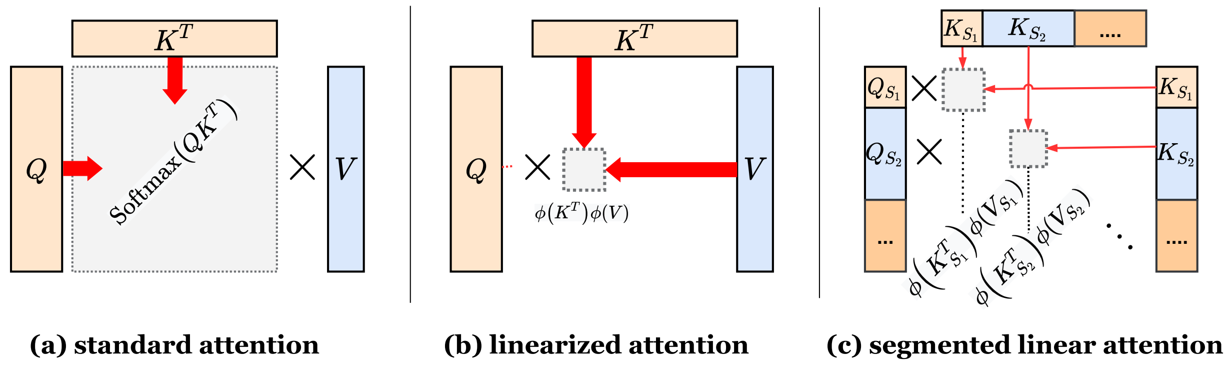
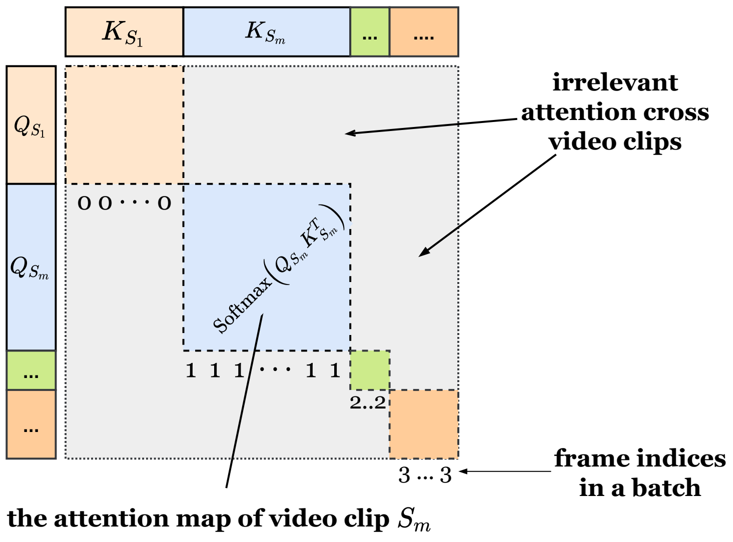


Figure 2.4: Illustration of Different Attention Operations: (a) Standard attention is obtained from $Softmax(QK^T)V$ with a complexity of $\mathcal{O}(n^2)$. (b) Linearized attention $\phi(Q)(\phi(K^T)V)$ with kernel function $\phi(\cdot)$ reduces the complexity to $\mathcal{O}(n)$, (c) we extend the linearized attention (b) to process segments of sequences.

second person's skeleton is just zeros ($P_2^{(i)} = 0$ in Figure 2.3) in most data samples. In contrast to the previous data format, the proposed format maintains the original length of each video clip. Additionally, when a video clip contains two persons, we concatenate them along the frame dimension. Instead of keeping an individual dimension for a batch of video clips, we further concatenate the video clips in a batch along the frame dimension, and the auxiliary vector stores the batch indices to indicate to which video clip a frame belongs, as shown in the bottom data format of Figure 2.3. Moreover, given the new dimensions (N, V, C) as shown in Figure 2.3, where N is the total number of frames after concatenating the video clips along the temporal dimension and V is the number of skeleton's joints, we regard dimension N as the logical batch size for spatial attention and dimension V as the logical batch size for temporal attention.

2.3.2.2 Segmented Linear Attention

With the new data format introduced in the previous section, we propose a novel linear multi-head attention tailored for this data format. We call it a Segmented Linear Attention.



K_{S_1}
K_{S_m}
...
....
Q_{S_1}
Q_{S_m}
...
...
irrelevant attention cross video clips
0 0 ··· 0
$\mathrm{Softmax}\left(Q_{S_m}K_{S_m}^T\right)$
1 1 1 ··· 1 1
2..2
3 ... 3
frame indices in a batch
the attention map of video clip S_m

Figure 2.5: Segmented Attention: directly applying the linearized attention to our new data format will calculate unexpected attention between the two irrelevant video clips, which is error-prone. Therefore, we use segmented attention corresponding to each video sequence.

As stated in the previous sections, Transformers are originally designed for sequential data. In the human skeleton sequence, each joint's dynamic movement across the frames can be regarded as a time series. Therefore, the 3D coordinates, i.e., (x, y, z), of every joint can be processed individually through the trajectory along the time dimension, and the application of attention extracts the interaction among time steps represented by frames.

Linear Attention. Standard dot product attention mechanism [155] (Equation 2.4) with the global receptive field of N inputs are prohibitively slow due to the quadratic time and memory complexity $\mathcal{O}(N^2)$. The quadratic complexity also makes Transformers hard to train and limits the context. Recent research toward the linearized attention mechanism derives the approximation of the *Softmax*-based attention. The most appealing ones are linear Transformers [88, 29, 139] based on kernel functions approximating the *Softmax*. The linearized Transformers can improve inference speeds up to three orders of magnitude without much loss in predictive performance [153]. Given the projected embeddings Q, K, and V for input tensors of queries, keys, and values, respectively, according to the observation from the accumulated value $V_i' \in \mathbf{R}^d$ for the query $Q_i \in \mathbf{R}^d$ in position i, d is the channel dimension, the linearized attention can be transformed from Equation 2.4 to Equation 2.5, the computational complexity is reduced to $\mathcal{O}(Nd)$, when d is much smaller than N, the computational complexity is approaching linear $\mathcal{O}(N)$:

$$V_i' = \frac{\sum_{j=1}^{N} \langle Q_i, K_j \rangle V_j}{\sum_{j=1}^{N} \langle Q_i, K_i \rangle} \tag{2.4}$$

$$\begin{aligned} V_i' &= \frac{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j)} = \frac{\phi(Q_i)^T U}{\phi(Q_i)^T Z} \\ U &= \sum_{j=1}^{N} \phi(K_j) V_j^T, \quad Z = \sum_{j=1}^{N} \phi(K_j) \end{aligned} \tag{2.5}$$

where $\phi(\cdot)$ is the kernel function. In work of [88], kernel function is simply simulated with ELU, $\phi(x) = elu(x) + 1$; while [29] introduces the *Fast Attention via Orthogonal Random Feature* (FAVOR) maps as the kernel function, $\phi(x) = \frac{c}{\sqrt{M}} f(Wx + b)^T$, where $c > 0$ is a constant, and $W \in \mathbf{R}^{M \times d}$ is a Gaussian random feature matrix, and M is the dimensionality of this matrix that controls the number of random features.

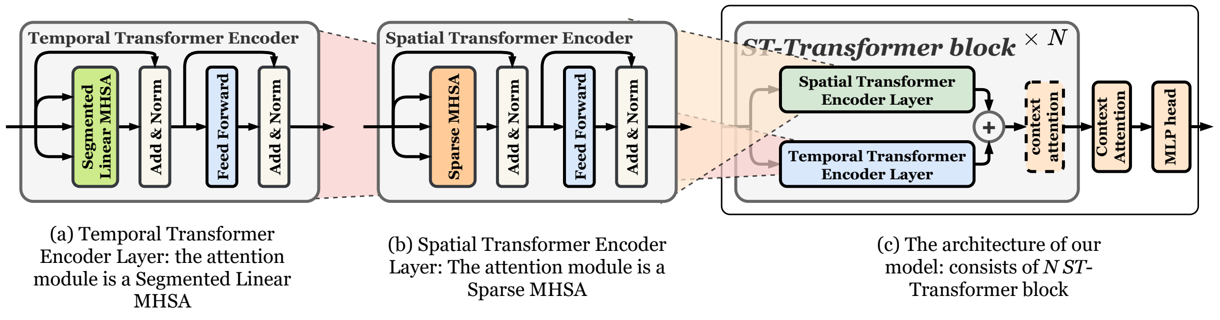

Temporal Transformer Encoder
Segmented Linear MHSA
Add & Norm
Feed Forward
Add & Norm
Spatial Transformer Encoder
Sparse MHSA
Add & Norm
Feed Forward
Add & Norm
ST-Transformer block
× N
Spatial Transformer Encoder Layer
Temporal Transformer Encoder Layer
+
context attention
Context Attention
MLP head
(a) Temporal Transformer Encoder Layer: the attention module is a Segmented Linear MHSA
(b) Spatial Transformer Encoder Layer: The attention module is a Sparse MHSA
(c) The architecture of our model: consists of N ST-Transformer block

Figure 2.6: Illustration of the overall pipeline of our approach (STAR)

Segmented Linear Attention. Since we concatenate the various length of video clips within a single batch along the time dimension, directly applying linear attention will cause the cross clip attention, leading to irrelevant information taken into account from one video clip to another, as shown in Figure 2.5. Therefore, we consider the frames of a video clip arranged as a segment, and then we design the segmented linear attention by reformulating Equation 2.5 with segment index. Therefore, for each V_i in segment $\mathcal{S}_m$, we summarize

$$\begin{aligned} V'_{i\in\mathcal{S}_m} &= \frac{\phi(Q_{i\in\mathcal{S}_m})^T \sum_{j\in\mathcal{S}_m} \phi(K_j)V_j^T}{\phi(Q_{i\in\mathcal{S}_m})^T \sum_{j\in\mathcal{S}_m} \phi(K_j)} \\ &= \frac{\phi(Q_{i\in\mathcal{S}_m})^T U_{S_m}}{\phi(Q_{i\in\mathcal{S}_m})^T Z_{S_m}} \\ U_{S_m} &= \sum_{j\in\mathcal{S}_m} \phi(K_j)V_j^T, \quad Z_{S_m} = \sum_{j\in\mathcal{S}_m} \phi(K_j) \end{aligned} \tag{2.6}$$

where $\mathcal{S}_m$ is the m-th segment, and the reduction operation $\sum_{j\in\mathcal{S}_m} f(x)$ can be easily implemented through the indexation to segments; and with help of the gathering and scattering operations [47], the segmented linear attention maintains the highly-paralleled computation. Figure 2.4 illustrates the comparison of different attention operations.

2.3.3 STAR Framework

In this work, we propose the Sparse-Transformer Action Recognition (STAR) framework. Figure 2.6 (c) shows the overview of our STAR framework. The STAR framework is built upon several Spatial-Temporal Transformer blocks (ST-block) followed by context-aware attention and MLP head for classification. Each ST-block comprises two pipelines: the spatial Transformer encoder and the temporal Transformer encoder. Each Transformer encoder consists of several key components, including the ***multi-head self-attention*** (MHSA), ***skip connection*** (AND & Norm part in Figure 2.6 (c)), and ***feed-forward network***. The spatial Transformer encoder utilizes sparse attention to capture the topological correlation of connected joints for each frame. The temporal Transformer encoder utilizes the segmented linear attention to capture the correlation of joints along the time dimension. The output sum from the two encoder layers is fed to the context-aware attention module to perform weighted summarization on the sequence of frames. Positional encoding is also utilized before ST-block to provide the context of ordering on the input sequence. Below is a brief introduction to each of them.

2.3.3.1 Context-aware attention

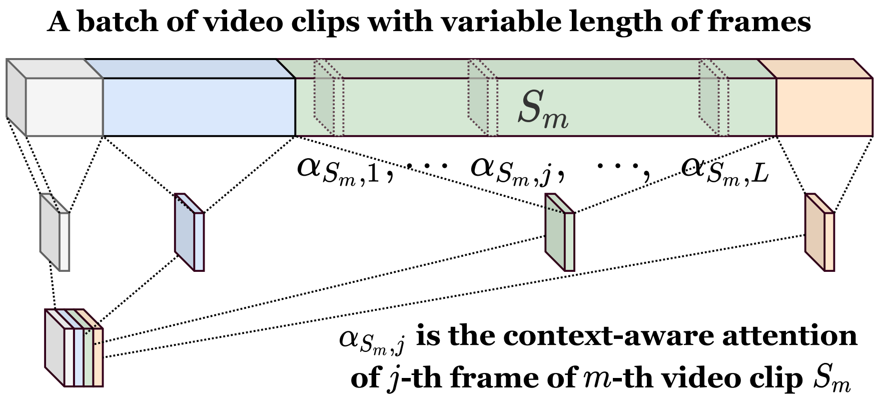

A batch of video clips with variable length of frames
S_m
$\alpha_{S_m,1}, \cdots \alpha_{S_m,j}, \cdots, \alpha_{S_m,L}$
$\alpha_{S_m,j}$ is the context-aware attention of j-th frame of m-th video clip S_m

Figure 2.7: The context-aware attention is utilized to summarize each video clip.

In previous works [180, 142], before connecting to the final fully-connected layer for classification, summarizing the video clip embedding along the temporal dimension is implemented by global average pooling. Alternatively, we utilize a probabilistic approach through *context-aware attention*, which is extended from the work of [13], to enhance this step's robustness, as demonstrated in Figure 2.7. Denote an input tensor embedding of video clip S_m as $V \in R^{F \times N \times D}$, for F is the number of frames in video clip S_m, N is the number of joints of skeleton, and each joint possessing D features, where $v_i \in R^{N \times D}$ is the embedding of frame i of V. First, a *global context* $c \in R^{N \times D}$ is computed, which is a simple average of embedding of frames followed by a nonlinear transformation: $c = tanh\left(\frac{1}{F} W \sum_{i \in S_m}^{F} v_i\right)$, where $W \in R^{D \times D}$ is a learnable weight matrix. The context c provides the global structural and feature information of the video clip that is adaptive to the similarity between frames in video clip S_m, via learning the weight matrix. Based on c, we can compute one attention weight for each frame. For frame i, to make its attention an aware of the global context, we take the inner product between c and its embedding. The intuition is that, frames similar to the global context should receive higher attention weights. A sigmoid function $\sigma(x) = \frac{1}{1+exp(-x)}$ is applied to the result to ensure the attention weights is in the range $(0, 1)$. Finally, the video clip embedding $v' \in R^{N \times D}$ is the weighted sum of video clip embeddings, $v' = \sum_{i \in S_m}^{F} a_i v_i$. The following equations summarize the proposed context-aware attentive mechanism:

$$\begin{aligned} c &= tanh\left(\frac{1}{N} W \sum_{j \in S_m}^{F} v_j\right) = tanh\left(\frac{1}{F}\left(V^T \cdot \mathbf{1}\right) W\right) \\ v' &= \sum_{i \in S_m}^{F} \sigma\left(v_i^T \left[tanh\left(\frac{1}{F} W \sum_{j \in S_m}^{F} v_j\right)\right]\right) v_i \\ &= \sum_{i \in S_m}^{F} \sigma\left(v_i^T c\right) v_i = \left[\sigma(Vc)\right]^T V \end{aligned} \tag{2.7}$$

2.3.3.2 Positional Encoding

As the attention mechanism is order-agnostic to the permutation in the input sequence [155, 154] and treats the input as an unordered *bag* of element. Therefore, an extra positional embedding is necessary to maintain the data order, i.e., time-series data are in the inherently sequential ordering. Then these positional embedding are participating the evaluation of the attention weight and value between token i and j in the input sequence.

Segmented Sequential Positional Encoding However, as we arrange the variable-length video clips into a batch along the temporal dimension, it is not feasible to directly apply positional encoding to the whole batch. Therefore, we introduce the *segmented positional encoding* where each video clip gets its positional encoding according to batch indices. An example of such encoding is shown in Figure 2.8.

Structural Positional Encoding. we also attempt to apply the structural positional encoding, e.g., tree-based positional encoding [143, 116], to the spatial dimension, i.e., the tree topology of skeleton. Experiments show that the current approach which we used cannot improve our model's performance significantly. Hence, to reduce our model's complexity, we decide not to apply the structural positional encoding for this work and leave it for future research.

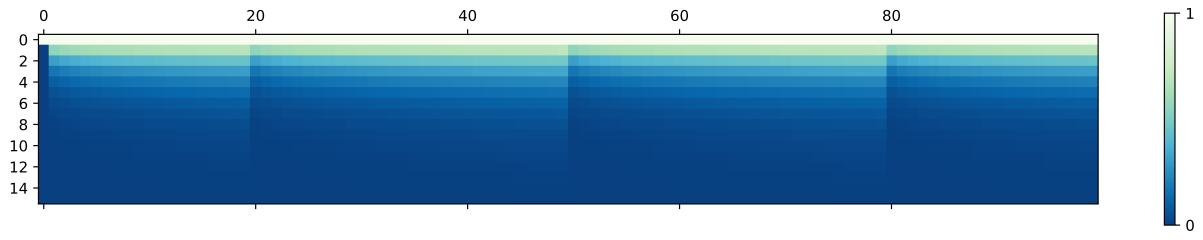


Figure 2.8: Illustration of Segmented Positional Encoding for a batch of 4 video clips. x-axis represents the number of frames and y-axis represents the feature dimension.

	NTU-60		NTU-120	
Method	**X-subject**	**X-view**	**X-subject**	**X-setup**
ST-GCN	81.5	88.3	72.4	71.3
ST-TR	88.7	95.6	81.9.	84.1
STAR-64 (ours)	81.9	88.9	75.4	78.1
STAR-128 (ours)	83.4	89.0	78.3	80.2

Table 2.1: Comparison of models' accuracy on NTU RGB+D 60 and 120 datasets

Model	**CUDA time (ms)**	**num. of parameters**	**GMACs**
ST-GCN	333.89	3.1M	261.49
ST-TR	1593.05	6.73M	197.55
STAR-64 (ours)	86.54	0.42M	15.58
STAR-128 (ours)	191.23	1.26M	73.33

Table 2.2: Comparison of models' efficiency

2.4 Experiments

In this section, we conduct experiments and ablation studies to verify the effectiveness and efficiency of our proposed sparse spatial and segmented linear temporal self-attention operations. The comparison has been made with **ST-GCN** [180], the baseline GCN model, and **ST-TR** [121], one of the state-of-the-art hybrid model, which have utilized full attention operation coupled with graph convolutions. The corresponding analysis demonstrates the potential of our model and the possible room for improvements. ST-TR is a hybrid model, though the its overall architecture is based on Transformer, the embedding still highly relies on convolution-based feature extraction mechanism, i.e., 1D convolution for temporal dimension features, graph/2D convolution for the spatial dimension features. Therefore, convolution still occupies most of the computational resource in their pipeline. The Transformer structure in ST-TR is used to enforce the long-distanced feature similarity calculation while convolution-based feature similarity calculation is captured only in a short range of receptive fields, especially for long time-series alike data, i.e., a joint of skeleton moving along the sequence of frames of video.

2.4.1 Datasets

In the experiments, we evaluate our model on two largest scale 3D skeleton-based action recognition datasets, NTU-RGB+D 60 and 120.

2.4.1.1 NTU RGB+D 60

This dataset contains 56,880 video clips involving 60 human action classes. The samples are performed by 40 volunteers and captured by three Microsoft Kinect v2 cameras [136]. It contains four modalities, including RGB videos, depth sequences, infrared frames, and 3D skeleton data. Our experiments are only conducted with the 3D skeleton data. The length of the action samples vary from 32 frames to 300 frames. In each frame, there are at most 2 subjects and each subject contains 25 3D joint coordinates. The dataset follows two evaluation criteria, which are Cross-Subject and Cross-View. In the Cross-View evaluation (X-View), there are 37,920 training samples captured from camera 2 and 3 and 18,960 test samples captured from camera 1. In the Cross-Subject evaluation (X-Sub), there are 40,320 training samples from 20 subjects and 26,560 test samples from the rest. We follow the original two benchmarks and report the Top-1 accuracy as well as the profiling metrics.

2.4.1.2 NTU RGB+D 120

The dataset [108] extends from NTU RGB+D 60 and is currently the largest dataset with 3D joint annotations. It contains 57,600 new skeleton sequences representing 60 new actions, a total of 114,480 videos involving 120 classes of 106 subjects captured from 32 different camera setups. The dataset follows two criteria, which are Cross-Subject and Cross-Setup. In the Cross-Subject evaluation, similar to the previous dataset, splits subjects in half to training and testing dataset. In the Cross-Setup evaluation, the samples are divided by the 32 camera setup IDs, where the even setup IDs are for training and the odd setup IDs for testing. Similar to the previous dataset, there is no preprocessing to set the uniform video length for all the samples. We follow the two criteria and report the Top-1 accuracy and the profiling metrics.

Unlike GCN-based models, where the length of all the samples and the number of subjects need to be fixed (e.g. 300 frames and 2 subjects), our model can process varying length of input samples and of the number of subjects. So no further preprocessing with padding is done on the samples.

2.4.2 Configuration of experiments

Implementation details. As the original Transformer framework [155] employs the unified model size d for every layer, we follow the same notion and keep the hidden channel size uniform across the attention heads and the feedforward networks. We run the experiments with two different hidden channel sizes, 64 and 128 for our Transformer encoders (STAR-64 and STAR-128), respectively. The hidden channel size of the MLP head is also proportional to that of the attention heads. Our model consists of 5 layers, each layer comprises one spatial encoder and one temporal encoder in parallel and the output sum from the two encoders is fed to the next layer. Drop rates are set to 0.5 for every module. We also replace the ReLU non-linear activation funciton with SiLU (or Swish) [45, 126] to increase the stability of gradients in back-propagation phase (GELU or SELU also bring similar effect). Our model is implemented with the deep learning framework PyTorch [119] and its extension PyTorch Geometric [51]. The scattering/gathering operations and sparse matrix multiplications are based on PyTorch Scatter [47] and PyTorch Sparse [48], respectively.

	MACs	**Parameters**	**Latency**
ST-GCN	*Conv2d*: 260.4 GMACs	*Conv2d*: 3.06M	*Conv2d*: 149.92ms
	BatchNorm2d: 737.3 MMACs	*BatchNorm2d*: 6.4K	*BatchNorm2d*: 19.92ms
	ReLU: 184.3 MMACs	*Linear*: 15.4K	*ReLU*: 4.49ms
ST-TR	*Conv2d*: 810.57 GMACs	*Conv2d*: 2.7M	*Conv2d*: 692.39ms
	MatMul: 161.1 GMACs	*BatchNorm2d*: 10.5K	*MatMul*: 161.38ms
	BatchNorm2d: 138.4 MMACs	*Linear*: 30.8K	*BatchNorm2d*: 38.97ms
STAR-64	MatMul(attention): 24.4 GMACs	Linear:83.2K	MatMul: 25.27ms
	Mul(sparse): 12.3 GMACs	LayerNorm: 1.3K	Mul: 12.81ms
	Linear: 6.2 GMACs		Linear:6.53ms

Table 2.3: The breakdown analysis and top-3 components in each metrics

Training setting. The maximum number of training epochs is set to 100. We used the Adam optimizer [92] with $\beta_1 = 0.9$, $\beta_2 = 0.98$ and $\epsilon = 10^{-9}$. Following the setting of the original Transformer paper [155], the learning rate is adjusted throughout the training:

$$lrate = d^{-0.5} \cdot min(t^{-0.5}, t \cdot w^{-1.5}) \tag{2.8}$$

where d is the model dimension, s is *step number*, and w is the *warmup steps.* According to the equation 2.8, the learning rate linearly increases for the first w training steps and then decreases proportionally to the inverse square root of the step number. We keep the original settings for the baseline models in their papers [180, 121], and use their codes provided online. All our training experiments are performed on a system with two GTX TITAN X GPUs and a system with one TITAN RTX GPU, while the inferences are executed on a single GPU.

2.4.3 Results and Analysis

We evaluate the accuracy and the efficiency of the baseline GCN model (ST-GCN), our model (STAR) and the hybrid model (ST-TR), which utilize both transformer and GCN frameworks.

2.4.3.1 Accuracy

We first evaluate the effectiveness of our Transformer encoder based model compared to ST-TR and ST-GCN models. Each model's accuracy is evaluated with the NTU RGB+D 60 and 120 testing datasets. As shown in the Table 2.1, our model outperforms ST-GCN in both cross-view (cross-set) and cross-subject benchmarks of the two dataset. Our model achieves $3.6 \sim 7.7$ percent lower accuracy compared to ST-TR, which heavily relies on convolution-based key components inherited from ST-GCN and utilizes them in both spatial and temporal pipelines. Our model yields modest performance compared to the state-of-the-art models in NTU RGB+D 60 and 120 when trained from scratch. The Figure 2.12 shows that there exists a performance gap between the training and testing. Transformer architectures' lack of inductive biases, especially translation equivariance and locality that are essential to convolution networks, could result in weak generalization. In NLP, Transformers are usually pretrained on a large corpus of text and fine-tuned on a smaller task-specific dataset to boost the performance. We would like to conduct extensive experiments on pre-training and fine-tuning our model on a larger dataset in the future to improve the accuracy comparable to those of the state-of-the-art models. For our future study, we want to address effective generalization methods for Transformer models, which resolves overfitting issues and improve the overall performance.

2.4.3.2 Efficiency

In this section, we evaluate the efficiency of the different models. As shown in Table 2.2, our model (STAR) is significantly efficient in terms of model size, the number of multiply-accumulate operations (GMACs) and the latency. Each metric for the different models is evaluated by running inference with sample dataset. Our model is fed with the original skeleton sequence of varying length. The other two models are fed with fix-sized skeleton sequence padded to 300 frames and 2 persons. We use the official profiler of PyTorch (v1.8.1) [119], and Flops-Profiler of DeepSpeed [127] to measure the benchmarks. The results are summarized with the following metrics:

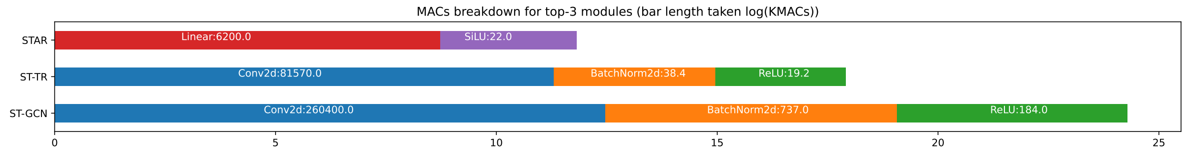


Figure 2.9: The breakdown of MACs for top-3 modules

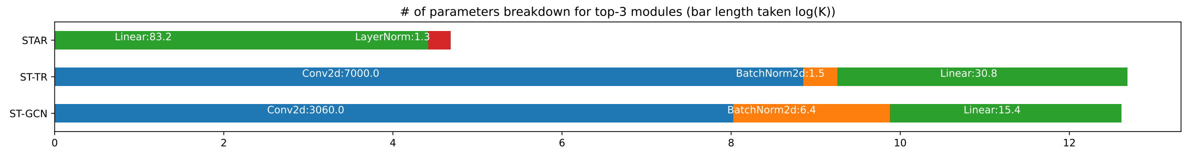


Figure 2.10: The breakdown of # parameters for top-3 modules

MACs. The number of Multiply-Accumulate (MAC) operations is used to determine the efficiency of deep learning models. Each MAC operation is counted as two floating point operations. With the same hardware configuration, more efficient models require fewer MACs

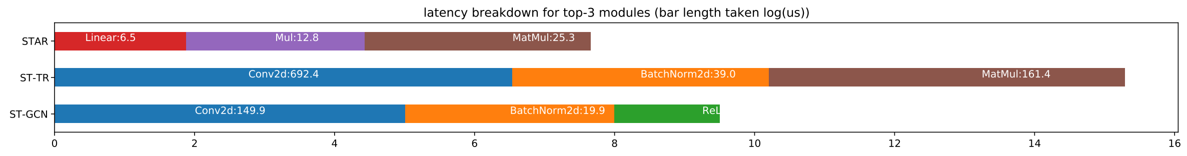


Figure 2.11: The breakdown of latency for top-3 modules

than other models to fulfill the same task. As shown in Table 2.2, both of our model with different channel sizes execute only $\frac{1}{3} \sim \frac{1}{17}$ amount of GMACs (i.e., Giga MACs) compared to ST-GCN and ST-TR models, respectively.

Model size. Model size is another metric to measure the efficiency of a machine learning model. Given the same task, smaller model delivering the same or very close performance is preferable. Smaller model is not only beneficial for the higher speedup and less memory accesses but also gives better energy consumption, especially for embedded systems and edge devices with scarce computational resources and small storage volume. The column of the number of parameters in Table 2.2 depicts the size of the models, these parameters are trainable weights in the model. Among all the model, STAR possesses the smallest model size, 0.42M and 1.26M for STAT-64 and STAR-128, respectively.

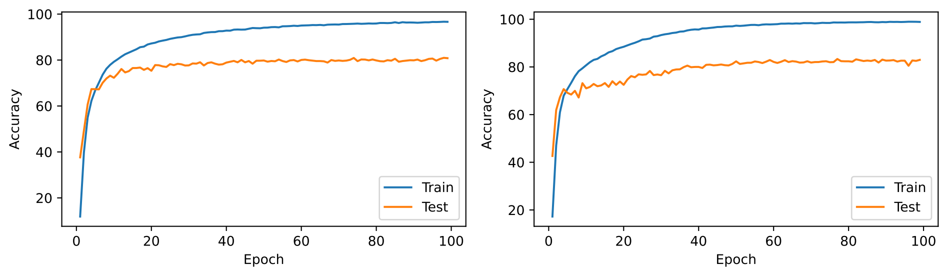


Figure 2.12: The training and testing curve for STAR-64(left) and STAR-128(right) on NTU-RGB+D 60 X-subject benchmark.

Breakdown analysis. The breakdown analysis is used to identify potential bottlenecks within different models (STAR-64, ST-TR, and ST-GCN). Table 2.3 provides the detailed profiling results for the top-3 computation modules that are dominant in each models.

According to Figure 2.9, 2.10 and 2.11, the convolution operations cost significant number of MAC operations and lead to computation bound. ST-GCN and ST-TR mainly consist of the convolution operations followed by batch normalization, which requires relatively large computational resources. Our Transformer model is based on sparse and linear attention mechanisms. It only produces relatively small attention weights from sparse attention; and performs low-rank matrix multiplication for linear attention ($\mathcal{O}(n)$). This replaces huge dynamic weights of attention coefficients from the standard attention mechanism, which has a quadratic time and space complexity ($\mathcal{O}(n^2)$).

2.4.4 Ablation Study

In this section, we evaluate the effectiveness and efficiency of our sparse self-attention operation in spatial encoder compared to the standard transformer encoder with full-attention operation. Table 2.4 and Table 2.5 show that our model with sparse self-attention operation achieves higher accuracy on both X-subject and X-view benchmarks and use significantly less number of GMACs and runtime. This shows that additional correlations of distant joints calculated by full attention do not improve the performance but rather contribute noise to the prediction. To handle such issue, learnable masks, consistent with adjacency matrix of skeleton, can be integrated to the full attention calculation to avoid accuracy degradation. But it requires extra computation involving learnable masks.

	NTU-60		NTU-120	
Method	X-subject	X-view	X-subject	X-setup
STAR (sparse)	83.4	84.2	78.3	78.5
STAR (full)	80.7	81.9	77.4	77.7

Table 2.4: Classification accuracy comparison between Sparse attention and Full attention on the NTU RGB+D 60 Skeleton dataset.

Model	CUDA time (ms)	GMACs
STAR-sparse	105.7	15.58
STAR-full	254.7	73.33

Table 2.5: Efficiency comparison between Sparse attention and Full attention on the NTU RGB+D 60 Skeleton dataset.

2.5 Conclusion

In this work, we propose an efficient Transformer-based model with sparse attention and segmented linear attention mechanisms applied on spatial and temporal dimensions of action skeleton sequence. We demonstrate that our model can replace graph convolution operations with the self-attention operations and yield the modest performance, while requiring significantly less computational and memory resources. We also designed compact data representation which is much smaller than fixed-size and zero padded data representation utilized by previous models. This work was supported in part by Semiconductor Research Corporation (SRC).

CHAPTER 3

Deep Graph Similarity through Linear Attention and Heat Kernel

This chapter presents the graph matching algorithm with deep learning solution, as shown in Figure 3.1. Graph matching, though with a high complexity as an NP-hard problem, is essential in many fields. The conventional approaches are often time-consuming and lacking enough parallelism. The model proposed consists of the graph neural networks for extract graph nodes' hidden features (or embedding) and a graph similarity measurement module. The key to such a problem lies in whether the two graphs' topology is consistent. With the help of graph diffusion, which uses the heat kernel to construct graph wavelets, we can sketch the energy diffusion map on graphs. An energy diffusion map is an effective way to depict the topology of graphs by assessing the energy flowing from one node to another node along the path between them. Therefore, we integrate the energy diffusion map into the graph matching problem and translate such kind of problem into an energy diffusion comparison problem.

3.1 Introduction

Graph similarity learning (or matching) solves the problem of finding structural correspondences between graphs using node and edge features. Since graphs excel at representing connectivity and structural information, graph matching has been extensively studied in social network analysis [72], fake review spammer detection[131], body keypoint identification[195],

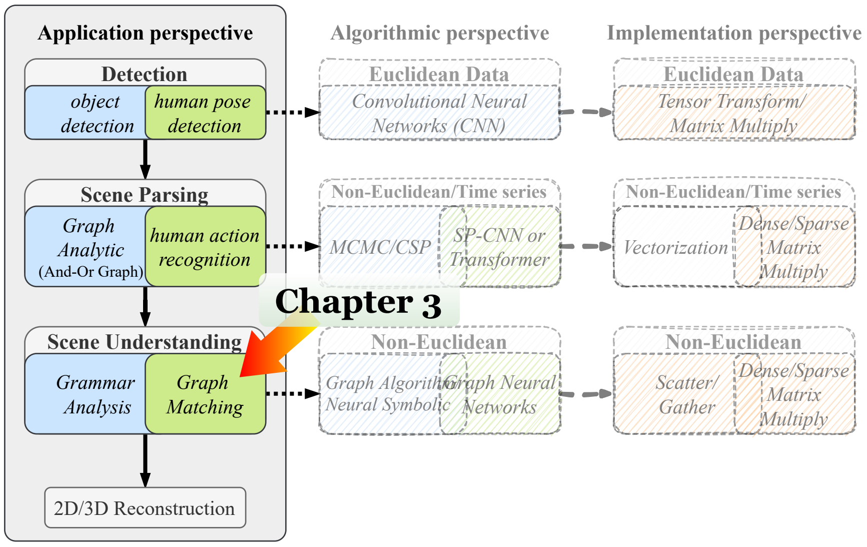


Figure 3.1: The position of this chapter in the platform

image matching in computer vision [186, 167, 55] and matching molecular structures of proteins in bioinformatics [56].

Conventionally, graph matching is formulated as a learning problem, exploiting several metrics for measuring the structural distance between graphs , including *Graph Edit Distance* (*GED*) [17] and *Maximum Common Subgraph* (*MCS*) [18], to evaluate the similarity (or distance) between graphs. However, computing the pairwise graph-graph *GED* or *MCS* score is known to be NP-Hard [188].

This work proposes a Transformer-based Graph Similarity (TRGSim) model combining the heat kernel based graph convolution, which models the probability distributions of energy diffusion according to the graph's topology, and efficient Transformer architecture adopting linear cross attention mechanism, which implicitly evaluates the similarity matrices between two graphs in attention module. The contributions of this work are summarized as:

1. We propose a scalable, ultra-lightweight, and computational efficient model - *TRGSim* - for learning graph structural similarity through integration of heat kernel and linear cross attention.

2. We mathematically reformulate the graph matching from a high-dimensional quadratic assignment problem to a tractable low-dimension problem.

3. We conduct extensive experiments to verify the effectiveness and efficiency of the proposed model, and show the superior matching performance and scalability.

3.2 Related Works

Graph Neural Networks and Transformers There have been various node and graph embedding methods proposed over the years [132, 68, 15, 16, 104] and these graph neural networks have proven useful in a wide range of applications [61, 16, 130, 168, 23]. Generalized CNN architectures with their message-passing scheme in local neighboring nodes especially have gained much attention [61, 16]. A separate line of works [158, 168] introduced graph neural networks with self-attention mechanism [156] to capture long-range dependencies. These works have shown impressive progress on supervised prediction problems in computer vision and graphical models. However, applying neural networks with self-attention mechanism on relational and symbolic domain remain largely unexplored.

Graph Matching There has been ongoing research on relational and symbolic domain especially on graph matching problem [186, 194]. *GraphSim* [12] learns graph-level similarity patterns by directly using node-level embeddings via pairwise node-node similarity scores. *Graph Matching Network* (GMN) [105], based on *Message Passing Network* [61], learns inter-graph information via a cross-graph attention mechanism. These approaches either calculate graph-level similarity scores [105, 14], focus merely on local characteristics of graph nodes, or do not generalize to new instances [167, 39]. Generally, local topological properties of nodes (e.g., node degrees, number of k-cliques) are mainly used to compute node similarities. Graph-level topological properties, which are essential to recovering structurally similar nodes, are typically overlooked. Thus, existing approaches usually yield consistent node alignment results, but fail to align edges and are subject to small perturbations in the graph topology. We argue that instead of localized structures, the topology encodes macro-structural information that is fundamental in determining cross-graph discrepancies, and thus, their *GED* and *MCS*. It is because, especially in graph signal processing, the diffusion of energy is determined by the graph-level topology.

3.3 Preliminaries

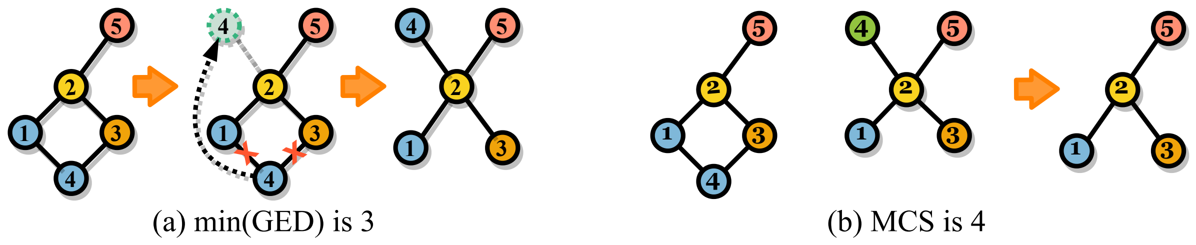


Figure 3.2: (a) The *GED* between the two graphs is 3, as the transformation requires 2 edge deletions, 1 edge insertion. (b) the *MCS* is measured as counting the number of nodes with same indecencies

Let $\mathcal{G} = (V, A, X)$ be a graph possessing a set of n vertices $V = \{v_1, v_2, \cdots, v_n\}$, and a *node feature matrix* $X \in \mathbb{R}^{|V|\times|F|}$, $|F|$ is the dimension of the feature vector. The adjacency matrix $A = \{A_{i,j} = 1 : \text{edge } (i, j) \text{ exists}\}$ is obtained from connectivity of graph $\mathcal{G}$.

The *degree matrix* D is given by $D_{i,i} = \sum_j A_{i,j}$; the normalized *Laplacian matrix* is calculated by $L = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.

Graph matching (or **similarity**) problem indicates finding the vertex and edge correspondences between two graphs. Given a source graph $\mathcal{G}_s = (V_s, A_s, X_s)$ and a target graph $\mathcal{G}_t = (V_t, A_t, X_t)$, it aims to find a *similarity matrix* $S = \{S_{i,i'} = 0 \text{ or } 1 : i \in V_s \text{ and } i' \in V_t\}$ representing the pairwise vertex correspondence between V_s and V_t. The mathematical description of graph matching [62, 65, 26, 55] is given as

$$\arg\max_S \sum_{i\in V_s}\sum_{j\in V_s}\sum_{i'\in V_t}\sum_{j'\in V_t} S_{i,i'}A^{(s)}_{i,j}A^{(t)}_{i',j'}S_{j,j'} \tag{3.1}$$

which is categorized as a *Quadratic Assignment Problem* (QAP).

If the perfect matching has been found between vertices $i \in V_s$ and $i' \in V_t$, also $j \in V_s$ and $j' \in V_t$, then it corresponds to $\mathcal{S}_{i,i'} = 1$ and $\mathcal{S}_{j,j'} = 1$.

The complexity of searching for an exact solution arises in a high-dimensional space.

3.3.1 Metrics used in General Graph Similarity

Graph Edit Distance (*GED*) [17] is a metric of the dissimilarity between two graphs $\mathcal{G}_s$ and $\mathcal{G}_t$, which measures the minimal number of edit operations to transform one graph into another. An edit operation is either an insertion or a deletion of a node or edge, or the relabeling of a node. Figure 3.2(a) demonstrates a concrete example.

The two graphs $\mathcal{G}_s$ and $\mathcal{G}_t$ considered to apply with such measures often have similar sizes.

Maximum Common Subgraph (*MCS*) [18] is another generalized metric, defined as the largest induced graph of a graph pair that contains the maximum number of vertices. For two graphs $\mathcal{G}_s$ and $\mathcal{G}_t$, their pairwise similarity scores is evaluated as the number of nodes in their *MCS*, as shown in Figure 3.2(b).

3.3.2 Graph Fourier Transform and Graph Convolutional Networks

A *Graph Convolutional Network* (GCN) model consists of a stack of graph convolution layers. The graph Fourier transform is defined as $\hat{x} = U^T x$ on node signal $x \in \mathbb{R}^n$ of a graph G, its inverse is $x = U\hat{x}$, where $U = [u_1, u_2, \cdots, u_n]$ is a complete set of orthonormal eignvectors of $L = U\Lambda U^T$ and $\Lambda = diag(\{\lambda_l\}_{l=1}^n)$ for $\lambda_1 \leq \lambda_2 \leq \cdots \lambda_n$. Performing graph Fourier transform on both graph signal x in spatial domain and its filter $f(\cdot)$, the graph convolution operator $*_G$ becomes $x *_G f = U\left(\left(U^T f\right) \odot \left(U^T x\right)\right) = U g_\theta U^T x$,

where $\odot$ is the element-wise Hadamard product, and $g_\theta = diag(\{\theta_i\}_{i=1}^n)$. ChebyNet [35] proposes a polynomial expansion of parameterized $g_\theta = \sum_{k=0}^{K-1} \alpha_k \Lambda^k$, which builds the graph convolution layer as:

$$y = U g_\theta U^T x = U \left(\sum_{k=0}^{K-1} \alpha_k \Lambda^k \right) U^T x = (\alpha_0 I + \alpha_1 L + \cdots + \alpha_{K-1} L^{K-1}) x \tag{3.2}$$

GCN [96] simply takes the first two terms of Equation 3.2 by presuming $K = 2$ and $\alpha = \alpha_0 = -\alpha_1$, and becomes $y = U g_\theta U^T \approx \alpha(I - L)x$.

3.3.3 Linearized Attention

Linearized attention [89, 29, 138], which has linear complexity, is the variant of standard attention of the quadratic complexity. The entry of the attention map $\mathcal{A} \in \mathbb{R}^{L_1 \times L_2}$ is of the form $\mathcal{A}(i,j) = SM(Q_i, K_j)$, where $SM(\cdot)$ is a softmax. Assume a kernel function $\phi(\cdot)$ is provided, then with the assistance of this kernel function, the attention evaluation becomes linearized by low-rank approximation

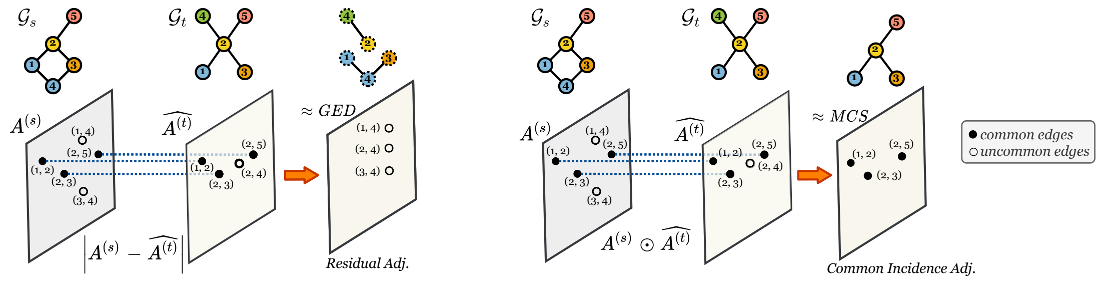
$$SM(Q, K)V = \left(\phi(Q)\phi(K^T)\right) V = \phi(Q) \left(\phi(K)^T V\right) \tag{3.3}$$

3.4 Methodology

In this section, we present *TRGSim*, an end-to-end, deep graph similarity learning framework in detail. Section 3.4.1 describes the reformulation of the graph matching problem; section 3.4.2 discusses the design of *TRGSim*; section 3.4.3 introduces using linear cross attention to estimate similarity matrix of two graphs; section 3.4.4 delivers the graph convolution with heat kernel; and finally we conduct complexity analysis in section 3.4.5.

3.4.1 Reformulating Graph Matching

In order to overcome the curse of dimensionality, *TRGSim* tackles the problem of searching the similarity matrix from a different angle. Instead of estimating the maximum similarity by performing high-order loops on Equation (3.1), we first reformulate the *GED* (similar for



(a) estimate GED with **residual adjacency** between adjacency matrices $A^{(s)}$ and $\widehat{A^{(t)}}$ of G_s and G_t, respectively

(b) estimate MCS with **Hadamard product** of adjacency matrices $A^{(s)}$ and $\widehat{A^{(t)}}$ of G_s and G_t, respectively

Figure 3.3: reformulating the estimation of graph similarity.

MCS) from matrix computation perspective. The evaluation of the *GED* can be generalized as

$$GED \leftarrow \underbrace{\sum_i |\mathcal{S}_{i,i} = 0|}_{\text{number of unmatched nodes}} + \underbrace{\sum_{i,j} (|A^{(s)}_{i,j} - \widehat{A^{(t)}_{i,j}}|)}_{\text{number of unmatched edges}} \text{ , where } \widehat{A^{(t)}} = \mathcal{S}A^{(t)}\mathcal{S}^T. \tag{3.4}$$

Assume we have a perfect similarity matrix $\mathcal{S}$ with exactly one entry of 1 in each row and each column, and 0 elsewhere (in such cases, it becomes a permutation matrix). The first part of Equation (3.4) indicates the number of unmatched nodes (with different labelling) by counting the number of zero entries along the diagonal of the similarity matrix, $\mathcal{S}_{i,i} = 0$. The second part of Equation (3.4) infers the number of unmatched edges between graph $\mathcal{G}_s$ and $\mathcal{G}_t$ by computing the difference between $A^{(s)}$ and $\widehat{A^{(t)}}$, where $\widehat{A^{(t)}} = \mathcal{S}A^{(t)}\mathcal{S}^T$ is both row- and column-reordered with respect to $A^{(s)}$. If the entries of $A^{(s)}$ and $\widehat{A^{(t)}}$ at coordinates (i, j) are not equal, it contributes 1 to the number of unmatched edges. Therefore, by counting the number of nonzero entries in the *residual adjacency matrix* $|A^{(s)} - \widehat{A^{(t)}}|$ as shown in Figure 3.3 (a), we obtain a graph-graph similarity measure. A similar procedure can be applied to *MCS*, except that we utilize the element-wise product of $A^{(s)}$ and $\widehat{A^{(t)}}$. The coordinates (i, j) of $A^{(s)} \odot \widehat{A^{(t)}}$ holds 1 only when a common edge exists between node i and node j in both graphs, as shown in Figure 3.3(b). Looking back to Equation (3.1), it is an optimization of the *Integer Programming Problem*.

However, with the above analysis, we attain a lower-dimensional solution space compared with Equation (3.1) since Equation (3.4) only works on the 2D residual adjacency matrices. Hence, *TRGSim* focuses on obtaining a high-quality similarity matrix $\mathcal{S}$, which makes $|A^{(s)} - \mathcal{S}A^{(t)}\mathcal{S}^T|$ match the optimal value, i.e., the ground truth *GED*. The trend of the first term $\sum_i |\mathcal{S}_{i,i} = 0|$ in Equation (3.4) is consistent with that of the second term since they are controlled by the same $\mathcal{S}$.We further transform the Equation 3.4 by multiplying $\mathcal{S}$ at the right hand side, we get

$$GED \propto |A^{(s)} - \mathcal{S}A^{(t)}\mathcal{S}^T|\mathcal{S} = |A^{(s)}\mathcal{S} - \mathcal{S}A^{(t)}| \tag{3.5}$$

3.4.2 The TRGSim model

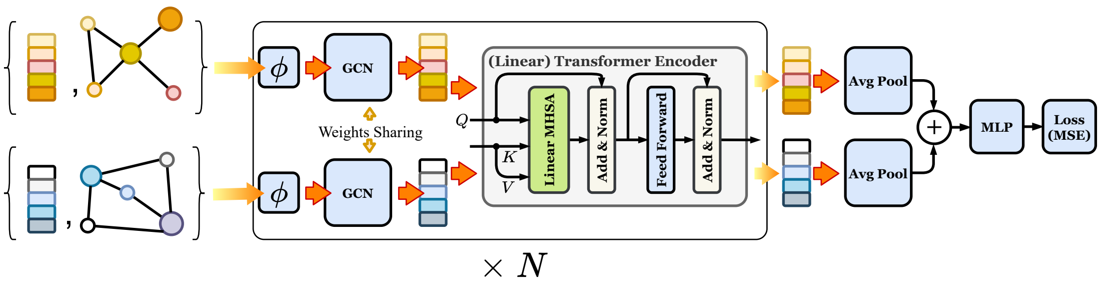


Figure 3.4: An overview of our proposed architecture *TRGSim*. Our model comprises four stages: 1) the graph convolution layers learning graph node encoding through the heat kernel $\phi(\cdot)$; 2) the Transformer encoder computes embedded similarity matrix through linear cross attention;3) average pooling used to summarize the graph embeddings ; 4) the final MLP evaluates the similarity scores.

Figure 3.4 illustrates the overall architecture of our model. The model approximates $|A^{(s)}\mathcal{S} - \mathcal{S}A^{(t)}|$, which is proportional to the ground truth GED. The GCN layer accompanying the heat kernel $g_\theta(x) = e^{-sL}$, which portrays the graph topology, computes the graph node embedding, where normalized Laplacian L is derived from adjacency matrices A. The node embedding from the previous stage also emulates the kernel function $\phi(\cdot)$ utilized by the linear cross attention of the Transformer encoder. Linear cross attention in Transformer encoder embeds the computation of similarity matrix $\mathcal{S}$ into kernel function. The average pooling layer is employed for graph-level embedding, and the final multi-layer perceptron (MLP) layer evaluates the mean square error (MSE) loss.

3.4.3 Linearized Cross Attention as Similarity Matrix

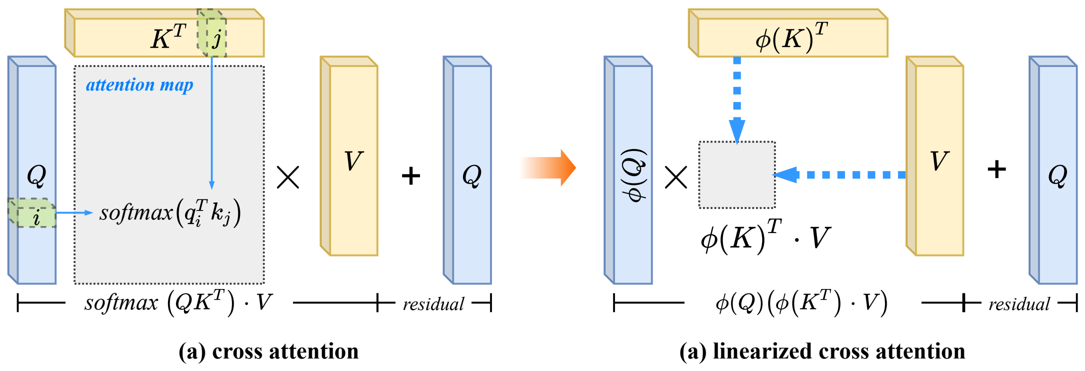


Figure 3.5: Linear cross attention evaluates similarity matrix

Suppose the encoding of graph nodes $x \in \mathcal{G}_s$ and $y \in \mathcal{G}_t$ of two different graphs in terminology of Transformer: $Q_i = W_q y, K_j = W_k x, V_j = W_v x$. The similarity matrix $\mathcal{S}$ of the two graphs can be estimated through the cross attention of the two graphs' encoding. We derive the kernel function of linear cross attention. Without considering the normalization factor, the main operation of *softmax* is through $exp(Q_i, K_j)$. Furthermore, by introducing a random feature map ω with a normal distribution $P(\omega - \mathbf{c}) = (2\pi)^{-d/2} exp\left(-\|\omega - \mathbf{c}\|^2/2\right)$, we evaluate an auxiliary term $exp\left(\frac{\|Q_i+K_j\|^2}{2}\right)$,

$$\begin{aligned} exp\left(\frac{\|Q_i + K_j\|^2}{2}\right) &= exp\left(\frac{\|Q_i + K_j\|^2}{2}\right) \int P\left(! - (Q_i + K_j)\right) d! \\ &= \int P\left(!\right) exp\left(\omega^T \left(Q_i + K_j\right)\right) d! \\ &= \mathbb{E}_{\omega \sim \mathcal{N}(0,1)} exp\left(\omega^T \left(Q_i + K_j\right)\right) \end{aligned} \tag{3.6}$$

with help of the above Equation 3.6 for $exp\left(\frac{\|Q_i+K_j\|^2}{2}\right)$, we further derive

$$\begin{aligned} exp\left(Q_i^T K_j\right) &= exp\left(\frac{\|Q_i + K_j\|^2}{2}\right) exp\left(-\frac{\|Q_i\|^2}{2}\right) exp\left(-\frac{\|K_j\|^2}{2}\right) \\ &= \mathbb{E}_{\omega \sim \mathcal{N}(\mathbf{0},\mathbf{1})} exp\left(\omega^T Q_i - \frac{\|Q_i^2\|}{2}\right) exp\left(\omega^T K_j - \frac{\|K_j^2\|}{2}\right) \\ &\approx \sum_{n=0}^{N-1} \left(e^{\omega_n^T Q_i}\right)\left(e^{\omega_n^T K_i}\right) = \phi(Q_i)^T \cdot \phi(K_j) \end{aligned} \tag{3.7}$$

where $\phi(Q_i) = [e^{\omega_0 Q_i}, e^{\omega_1 Q_i}, \cdots, e^{\omega_{N-1} Q_i}]^T$ and $\phi(K_j) = [e^{\omega_0 K_j}, e^{\omega_1 K_j}, \cdots, e^{\omega_{N-1} K_j}]^T$ are obtained from Monte-Carlo sampling. Taking $\Lambda = diag(\{\omega_i\}_{n=0}^{N-1})$, we finally get $\phi(Q_i) = e^{\Lambda Q_i}$ and $\phi(K_j) = e^{\Lambda K_j}$. With the help of Equation 3.7, we further compute the cross attention representation for the embedding of graph node y in one graph using all nodes' embedding of graph node x in another graph to compare the similarity, as expressed in Equation 3.8. This attention weighed representation is added to the skip connection of its original node embedding (residual), as shown in Figure 3.5

$$Q' = \phi\left(W_q y\right)\left(\phi\left(W_k x\right)^T \left(W_v x\right)\right); \quad Q'' = Q' + W_q y \tag{3.8}$$

3.4.4 Graph Heat Kernel imitates Gaussian Kernel

Heat kernel is defined as $f(\lambda_i) = e^{-s\lambda_i}$, where $s \geq 0$ is a scaling hyper-parameter. Denote $\Lambda_s = diag\left(\{e^{-s\lambda_i}\}_{i=1}^n\right)$. By applying heat kernel, the convolution kernel is now becoming $g_\theta = \sum_{k=0}^{K-1} \theta_k \left(e^{-sk\Lambda_s}\right)$, then the graph convolution applied to signal x is derived as

$$\begin{aligned} y = Ug_\theta U^T &= U\left(\sum_{k=0}^{K-1}\theta_k\left(e^{-sk\Lambda_s}\right)\right)U^T x \\ &= \Big[\theta_0 I + \theta_1\left(e^{-s\lambda_1}u_1u_1^T + \cdots + e^{-s\lambda_n}u_nu_n^T\right) + \cdots \\ &\quad + \theta_{K-1}\left(e^{-(K-1)s\lambda_1}u_1u_1^T + \cdots e^{-(K-1)s\lambda_n}u_nu_n^T\right)\Big]x \\ &= \Bigg(\theta_0 I + \theta_1 e^{-sL} + \underbrace{\theta_2 e^{-2sL} + \cdots + \theta_{K-1}e^{-(K-1)sL}}_{\text{negligible terms}}\Bigg)x \approx \left(\theta_0 I + \theta_1 e^{-skL}\right)x \end{aligned} \tag{3.9}$$

with the guarantee that eigenvectors $\{u_i\}_{n=1}^N$ are orthonormal and Laplacian matrix L is positive definite, we substitute the kernel function $\phi(x) = e^{\Lambda x}$ of Equation 3.3 and 3.7 with the first two terms of Equation 3.9: $\phi(x) = \left(\theta_0 I + \theta_1 e^{-skL}\right)x$, which is also the graph convolution operation of our GCN layer in the representation stage.

3.4.5 Analysis of complexity

The computation of node-level embedding stage. This stage comprises of GNN layers which performs the node-level embedding. Each layer of this stage has the computational complexity of $\mathcal{O}(|\mathcal{E}|CHF)$, where C, H, F are the dimensions of input, hidden, and output, respectively; $|\mathcal{E}|$ is the number of edges. The edges number is normally much larger than other dimensions, therefore, the computational complexity of this stage is towards $\mathcal{O}(|\mathcal{E}|)$. Furthermore, when heat kernel e^{-sL} is applied to the Laplacian matrix L, which is symmetric, and the exponential operation on edges is element-wise; therefore, the complexity is also $\mathcal{O}(|\mathcal{E}|)$.
The complexity of the linear attention. The matrix multiplication $V' = \phi(K)^T \cdot V$ has the time complexity of $\mathcal{O}(Ndr)$ for $K \in \mathbb{R}^{N\times d}$ and $V \in \mathbb{R}^{N\times r}$, and the space complexity for this step is $\mathcal{O}(Ld + Lr + rd)$. The second matrix multiplication $V'' = \phi(Q)\cdot V'$, for $Q \in \mathbb{R}^{N\times d}$, has the time complexity of $\mathcal{O}(Ndr)$ and space complexity $\mathcal{O}(Nd + rd)$. Normally, r and d are much smaller than N, then the complexity of both time and space is approaching $\mathcal{O}(N)$.

3.5 Experiments

3.5.1 Datasets

We use well-established benchmark datasets from literature [14, 11] - *AIDS, LINUX*, and *IMDBMulti* [91], and their statistics are summarized in Table 3.1. The graph nodes in these datasets are annotated with discrete features. We split the datasets into training, validation, and testing sets with ratios of 60%, 20%, and 20% . We use *Mean Squared Error (MSE)* to measures the average squared difference between the predicted similarities and the ground-truth similarities. To evaluate the global ranking results, we employ *Spearman's Rank Correlation Coefficient* (ρ) and *Kendall's Rank Correlation Coefficient* (τ), both measuring the discrepancies between the predicted and the actual ranking results. Compared with ρ and τ, precision at k ($p@k$) focuses on the top k ranking results, computed as the intersection of the predicted and ground-truth top k results divided by k.

Dataset	Graph Definition	graphs	pairs
LINUX	Program Dependency Graphs	1000	1M
IMDB	Actor/Actress Ego-Networks	1500	2.25M
AIDS	Chemical Compounds	700	490K

Table 3.1: Statistics of datasets [14]

3.5.2 Baselines and Experiment Configurations

We choose *SimGNN* [14], *GMN* [105], and *GraphSim* [12] as the baseline models, since *TRGSim* accomplishes the same tasks in real-world application as these architectures. *SimGNN* generates node features using *GCN*, aggregates node embeddings using an attention mechanism, and learns graph-level embeddings through a *Neural Tensor Network*. *GMN* learns the node embeddings through a message passing mechanism in its GNN layers. It computes the similarity score between an input graph pair by jointly reasoning on the pair through a new attention-based matching mechanism. *GraphSim* utilizes a multi-scale convolutional set architecture and it constructs a very deep pipeline with a large amount of parameters. We implemented *TRGSim* using *PyTorch* [118] and an efficient deep graph learning library *PyTorch Geometric Library* (PyG) [53] based on *PyTorch*. The first stage of model consists of two graph convolution layers with output dimensions of 64, 32 for node feature embeddings. Two variants of *graph convolution layer* have been adopted and tested, including *GCN* [96] and *GIN*[178]. Each graph convolution layer yields a two-dimensional graph feature vector for both $\mathcal{G}_s$ and $\mathcal{G}_t$. In the second stage, the cross-graph similarity matrices are computed as the dot products between feature vectors of $\mathcal{G}_s$ and those of $\mathcal{G}_t$. The last stage comprises a pipeline of two 2D convolution layers and two fully-connected layers. For all experiments, we fixed the learning rate at 10^{-3} for the optimizer, and use *ADAM* [92] as the optimizer. For both baseline models and our model, we set the batch size to 128 and the number of training epochs to 10,000.

	LINUX				AIDS700				IMDBMulti			
Model	MSE	ρ	τ	p@10	MSE	ρ	τ	p@10	MSE	ρ	τ	p@10
SimGNN	1.459	0.931	0.675	0.713	1.279	0.824	0.575	0.401	1.754	0.861	0.755	0.759
GMN	1.135	0.892	0.669	0.684	2.041	0.708	0.569	0.374	4.636	0.717	0.564	0.604
GraphSim	0.783	0.945	0.724	0.730	1.084	0.833	0.674	0.491	**1.531**	0.856	**0.787**	0.782
TRGSim	**0.370**	**0.983**	**0.892**	**0.779**	**1.011**	**0.857**	**0.715**	**0.814**	1.594	**0.878**	0.691	**0.804**

Table 3.2: Experiment results on GED metric (bold is the best)

	LINUX				AIDS700				IMDBMulti			
Model	MSE	ρ	τ	p@10	MSE	ρ	τ	p@10	MSE	ρ	τ	p@10
SimGNN	0.751	0.849	0.779	0.834	3.513	0.790	0.655	0.374	1.315	0.924	0.752	0.681
GMN	0.608	0.893	0.756	0.825	4.785	0.673	0.478	0.591	**0.619**	0.903	0.657	0.783
GraphSim	0.182	0.929	0.814	0.832	3.143	0.821	0.667	0.505	1.250	0.941	**0.791**	0.802
TRGSim	**0.073**	**0.953**	**0.820**	**0.858**	**2.947**	**0.826**	**0.674**	**0.610**	0.807	**0.932**	0.773	**0.821**

Table 3.3: Experiment results on MCS metric (bold is the best)

	LINUX				AIDS700				IMDBMulti			
Metric	MSE	ρ	τ	p@10	MSE	ρ	τ	p@10	MSE	ρ	τ	p@10
GED	0.501	0.987	0.884	0.783	2.412	0.813	0.663	0.875	2.748	0.806	0.720	0.741
MCS	0.245	0.923	0.797	0.604	3.640	0.672	0.537	0.648	1.531	0.765	0.545	0.622

Table 3.4: Experiment results on synthetic data with *TRGSim*

3.5.3 Results

For both *GED* and *MCS*, *TRGSim* outperforms all baseline models in 10 out of 12 measurements. This demonstrates that *TRGSim*'s strategy of exploring graph topology through spectral graph wavelets outweighs the conventional approach of only investigating graphs' local connectivity. Note that *TRGSim* has slightly suboptimal results than the baselines on MSE and τ in IMDBMulti, since these graphs have denser incidence relationships. This is because *TRGSim* manipulates wavelet coefficients well with graphs of higher sparsity, which is due to the selection of the scaling parameter. A small scaling parameter constrains the ability to diffuse energy of the graph signals to farther nodes. We further observe that more layers of transformation implies less stability, making the model subject to perturbations. So we fix the number of convolution layers to 2. In addition, we experimented with synthesized examples to verify the feasibility of our model. We generate these synthetic graphs with a certain noise rate. These experiments simulate the real-world environment of deployment of *TRGSim*. The results are shown in Table 3.4 and it shows that *TRGSim* keeps a stable performance in various situations.

3.5.4 Computation runtime comparison

We evaluate the performance of speed on different models. We take 78400 graph pairs from the synthesized LINUX/AIDS700 dataset as samples to perform the matching and then take the average runtime for the comparison.

Model	$\overline{time}$ (ms)
SimGNN	8.90 ± 0.36
GraphSim	33.57 ± 0.64
GMN	11.57 ± 0.58
Ours	4.89 ± 0.73

Table 3.5: Comparison of runtime

3.5.5 Ablation Study

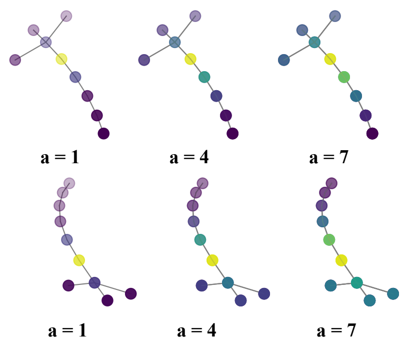
We conducted the ablation study of our model and analyze the result in this section. Below is part of the ablation experiment with *GIN* alternative. *GCN* used in our model has less parameters and competitive performance with *GIN*. Both GNN alternatives used in our model achieve very similar results, this demonstrates that the performance is viable with node-level embedding. The performance with GIN alternative in Table 3.6 shows the robustness of our model with different dataset.

	LINUX				AIDS700			
Metric	MSE	ρ	τ	p@10	MSE	ρ	τ	p@10
GED	0.425	0.980	0.886	0.775	3.311	0.888	0.621	0.798
MCS	0.073	0.952	0.818	0.331	5.946	0.645	0.473	0.109

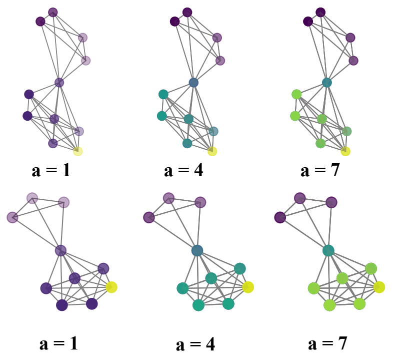
Table 3.6: Ablation study with replacing GCN [96] by GIN [178]

3.5.6 Graph Signal Diffusion Analysis

In this work, we also conduct the analysis of the heat diffusion on graphs, which is a good measure to display the graph topology involving the energy dissipation among connected nodes. The experiment first picks a pair of corresponding nodes from each of the two graphs, then a pulse signal $\delta(v_i)$ is propagated from each graph node v_i separately. Figure 3.6 gives a concrete illustration of how energy diffuses on the two graph pairs as probability distribution



(a) The *GED* is 3 between G_s (top) and G_t (bottom);



(b) The *GED* is 5 between G_s (top) and G_t (bottom);

Figure 3.6: Heat diffusion as probability distribution (with heat kernel $e^{-a\lambda}$) of energy diffusion on graph pairs from (a) *LINUX* dataset and (b) *IMDBMulti* dataset

(with heat kernel $e^{-a\lambda}$). The graph pairs are taken from *Linux* and *IMDBMulti* datasets, as shown in Figure 3.6(a) and (b), respectively. The spectral graph wavelets are calculated and plotted with the assistance of the graph signal processing library *PyGSP* [37]. The diffusion of heat is able to delineate the graph topology. In Figure 3.6, the yellow nodes represent the source nodes of energy, and the brightness of the node depicts the amount of energy received from the source. The scaling factor a sets up how far the energy will dissipate. In each pair of graphs, the yellow coloring nodes are also the nodes with highest similarity between two graphs, so that we can see the relationship between the energy diffusion and the graph topology.

3.6 Conclusion

In this work, we proposed a novel approach to graph similarity learning using heat kernel and linear cross attention. We introduced *TRGSim*, a novel, light-weighted, three-staged neural architecture for learning structural embedding and graph similarities in a supervised fashion. The heat kernel is used to simulate a probability distribution of energy diffusion.

TRGSim captures the graph's structural (topological) information by tracking how the heat diffuses over the network while simple graph convolution does not possess such property. We treat the cross attention as the similarity between two graphs and the linear cross attention dramatically reduces both the time and space complexity. Various experiments on both real-world and synthetic graphs have shown that our model can achieve superior results on approximating graph similarity scores, i.e. *Graph Edit Distance* and *Maximum Common Subgraph*, over the state-of-the-art graph matching models.

CHAPTER 4

Transformer-based Machine Learning for Fast SAT Solvers and Logic Synthesis

This chapter focuses on the design of an AI-based approach for SAT solvers. Due to the availability of several solvers and benchmarks targeted at logic synthesis for SAT solvers, logic synthesis is used to elaborate the effectiveness of our work. However, the contribution directly benefits the constraint solving problems involved in our video analytics pipeline.

The CSP/SAT solver discussed in this chapter is based on a deep learning approach. Constraint Satisfaction Problem (CSP) and Boolean Satisfiability Problem (SAT) are ubiquitous and adopted in many fields. Such kinds of problems are also NP-complete. In decades, people have conceived many heuristic approaches; however, these algorithms primarily concentrate on a backtracking mechanism from an analytical approach. These methods do not consider the overall topological relation between nodes of a bipartite graph constructed by the variables and clauses. They neither can summarize the hidden features embedded inside the topology and those derived from nodes' connectivity. We also propose the concept ”meta-paths” into our model to derive the correlated feature crossing the link between clauses and variables of the bipartite graph of SAT instance.

Figure 4.2 depicts a scenario of using canonical normal form and related SAT solver to process the scene understanding problem [84]. The platform at the left hand side of the Figure 4.2 constructs the And-Or graph to describe the interactions among objects or their components, the relationship shows the decomposition, combination, related position, or even the logic reasoning process. The constructed graph can further be transformed into

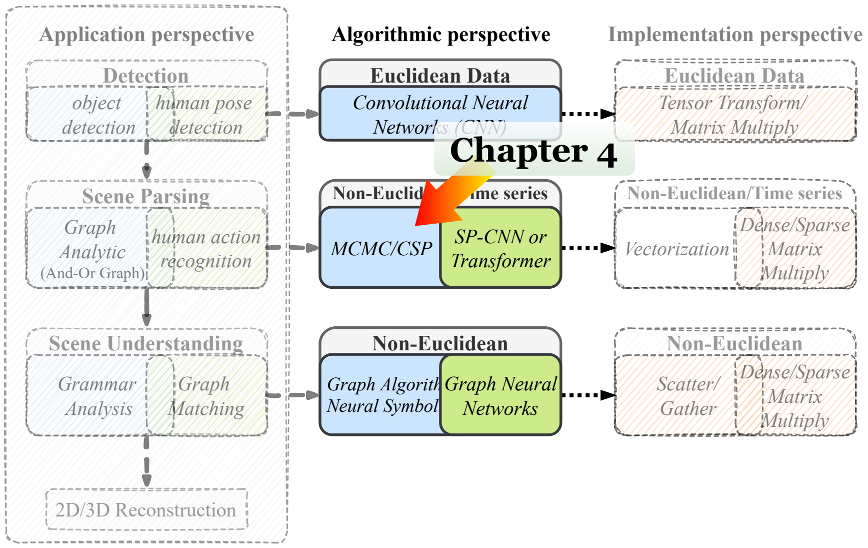
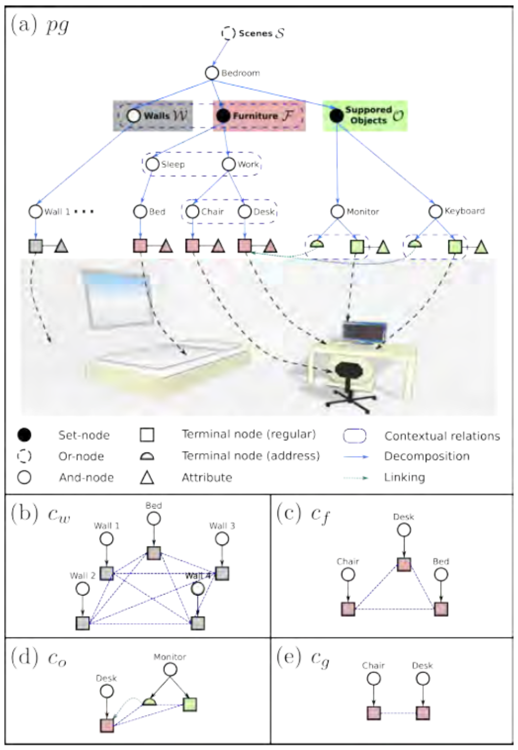


Figure 4.1: The position of this chapter in the platform



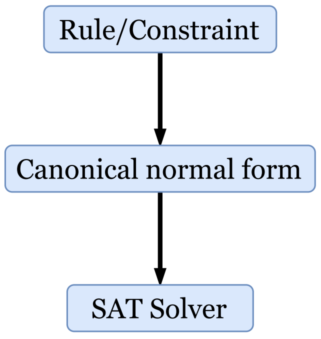


Figure 4.2: The rule-based scene parsing and understanding using can be processed with canonical normal form and the SAT solver [84]

canonical normal form, which is widely used in logic reasoning. This form is easily solved with deliberated solvers.

4.1 Introduction

Logic synthesis is a crucial step in design automation systems where abstract logic is transformed to physical gate-level implementation. There has been significant improvement in hardware performance and cost by optimizing logic at the synthesis level. The task to synthesize and minimize digital circuits is often translated to the Constraint Satisfaction Problem (CSP). CSP aims at finding a consistent assignment of values to variables such that all constraints, which are typically defined over a finite domain, are satisfied. The Boolean Satisfiability (SAT) and Maximum Satisfiability (MaxSAT) solvers have been the core of the Constraint Satisfaction methods to seek a minimal satisfiable representation of logic. Extensive studies have been conducted on MaxSAT problem for logic synthesis [102, 76, 111].

Previous SAT solvers are based on well-engineered heuristics to search for satisfying assignments. These algorithms focus on solving CSP via backtracking or local search for conflict analysis. David-Putnam-Logemann-Loveland (DPLL) algorithm exploits unit propagation and pure literal elimination to optimize backtracking Conjunctive Normal Form (CNF) [33]. Derived from DPLL, conflict-driven clause learning (CDCL) algorithms such as Chaff, GRASP, and MiniSat have been proposed [113, 145, 44]. Since SAT algorithms can take exponential runtime in the worst case, the search for additional speed up has continued. SAT Sweeping is a method to merge equivalent gates by running simulation and SAT solver in synergy [3, 122]. MajorSAT proposed efficient SAT solver for solving the instances containing majority functions [30]. Another method used directed acyclic graph topology for the Boolean chain to restrict on the search space and reduce runtime [66]. The heuristic models improved computational efficiency but are bounded by the greedy strategy, which is sub-optimal in general.

Recently, the machine learning community has seen an increasing interest in applications and optimizations related to neural symbolic, including solving CSP and SAT. With the fast advances in deep neural networks (DNN), various frameworks utilizing diverse methodologies have been proposed, offering new insights into developing CSP/SAT solvers and classifiers. NeuroSAT is a graph neural network model that aims at solving SAT without leveraging the greedy search paradigm [135, 134]. It approaches SAT as a binary classification problem and finds an SAT assignment from the latent representations during inference. The model is able to search for solutions to problems of various difficulties despite training for relatively small number of iterations. As an extension to this line of work, PDP-solver [4] proposes a deep neural network framework that facilitates satisfiability solution searching within high-performance SAT solvers on real-life problems. However, most of these works, such as neural approaches utilizing RNN or Reinforcement Learning, are still restricted to sequential algorithms, while clauses are parallelizable even though they are strongly correlated through shared variables.

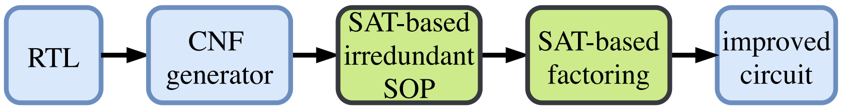


Figure 4.3: CNF-based SAT solver in logic synthesis flow.

In this work, we propose a hybrid model of the *Transformer* architecture [157] and the Graph Neural Network for solving CSP/SAT.

Our main contributions in this work are:

- We leverage meta-paths, a concept introduced in [150], to formulate the message passing mechanism between homogeneous nodes. This enables our model to perform self-attention and pass messages through either variables sharing the same clauses, or clauses that include the same variables. We apply the cross-attention mechanism to perform message exchanges between heterogeneous nodes (i.e., clause to variable, or variable to clause). This enhances the latent features, resulting in better accuracy in terms of the completion rate compared to other state-of-the-art machine learning methods in solving MaxSAT problems.

- In addition to using a combination of *self-attention* and *cross-attention* mechanism on the bipartite graph structure, we combine the Transformer with Neural Symbolic methods to resolve combinatorial optimization on graphs. Consequently, our model shows a significant speedup in CNF-based logic synthesis compared to heuristic SAT solvers as well as machine learning methods.

- We propose *Transformer-based SAT Solver* (TRSAT), a general framework for graphs with heterogeneous nodes. In this work, we trained the *TRSAT* framework to approximate the solutions of CSP/SAT. Our model is able to achieve competitive completion rate, parallelism, and generality on CSP/SAT problems with arbitrary sizes. Our approach provides solutions with completion rate of 97% in general SAT problem and 88% for circuit problem with significant speed up over prior techniques.

Gate	CNF equation
$z = a \cdot b$	$\phi = (a + \neg z) \cdot (b + \neg z) \cdot (\neg a + \neg b + z)$
$z = a + b$	$\phi = (\neg a + z) \cdot (\neg b + z) \cdot (a + b + \neg z)$
$z = \neg a$	$\phi = (a + z) \cdot (\neg a + \neg z)$
$z = a \oplus b$	$\phi = (a + b + \neg z) \cdot (a + \neg b + z) \qquad (\neg a + \neg b + \neg z) \cdot (\neg a + b + z)$

Table 4.1: CNF equations for the basic logic gates [166]

4.2 Background

In this section, we introduce the preliminaries for CNF-based logic synthesis and the advanced machine learning models, i.e., Transformers and Graph Neural Networks.

4.2.1 CNF equations for logic gates

For a logic gate with function $z = f(a, b, \dots)$, it equals to logic expression $(z \Rightarrow f(a, b, \dots)) \cdot (f(a, b, \dots) \Rightarrow z)$, which then derives $(\neg z + f(a, b, \dots)) \cdot (\neg f(a, b, \dots) + z)$. We further expand the above equation in product of sum (POS) form to obtain the CNF for the gate. Table 4.1 summarize the CNF equations for basic logic gates,

4.2.2 Transformers and relation to GNNs

To combine the advantages from both CNNs and RNNs, [157] presents a novel architecture, called Transformer, using only the attention mechanism. This architecture achieves parallelization by capturing recurrence sequence with attention and at the same time encodes each item's position in the sequence. As a result, Transformer leads to a compatible model with significantly shorter training time. The self-attention mechanism of each Transformer layer is depicted as a function $T : \mathbb{R}^{N \times F} \to \mathbb{R}^{N \times F}$; given $x \in \mathbb{R}^{N \times F}$, the lth layer T_l computes,

$$Q = xW_Q, K = xW_K, V = xW_V \tag{4.1}$$

$$A_l(x) = V' = softmax(\frac{QK^T}{\sqrt{D}})V \tag{4.2}$$

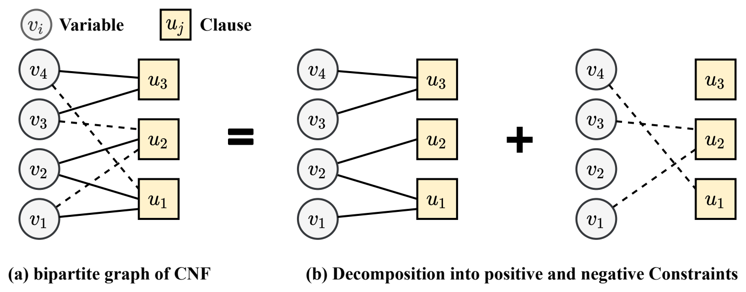
$$T_l(x) = f_l(A_l(x) + x) \tag{4.3}$$

where $W_Q, W_K \in \mathbb{R}^{F \times D}$ and $W_V \in \mathbb{R}^{F \times M}$ are projection matrices for evaluating queries Q, keys K, and values V, respectively. $A_l(x)$ are self-attention matrices which describe the similarities between vector entries of x. The self-attention matrix A_l is a complete graph which represents the connectivity between queries and keys. When queries and keys are loosely related, the attention map becomes a sparse matrix, similar to the aggregation phase of the Graph Neural Network (GNN). Another difference between the self-attention mechanism used in Transformer and the Graph Attention Network (GAT) [160] is that Transformer's attention mechanism is multiplicative, which is accomplished by dot product, while GAT employs additive attention.

4.2.3 How Attention Works for Solving CNF?

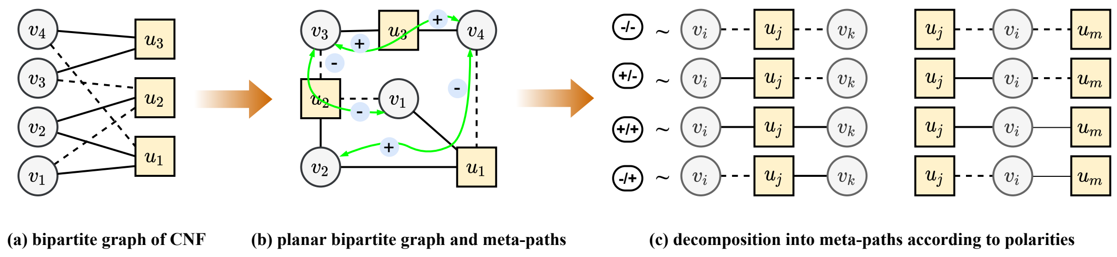
Given an example CNF: $\phi = (v_1 \lor v_2 \lor \neg v_4) \land (\neg v_1 \lor v_2 \lor \neg v_3) \land (v_3 \lor v_4)$, when v_2 is set to 1, two clauses are solved; while setting other variables, e.g., v_1 or v_4 and their complements, cannot immediately conduct the solution. Consequently, v_2 attracts more attention (or higher probability of flipping or set value) than other variables.

4.3 Methodology



v_i Variable
u_j Clause
(a) bipartite graph of CNF
(b) Decomposition into positive and negative Constraints

Figure 4.4: (a) bipartite graph for the *CNF* with measure $\phi = (v_1 \lor v_2 \lor \neg v_4) \land (\neg v_1 \lor v_2 \lor \neg v_3) \land (v_3 \lor v_4)$, where solid lines are the positive incidences of v_i in u_j, and dashed lines are the negative incidences of $\neg v_i$ in u_j; (b) the decomposition of the bipartite graph according to the positive and negative relations.



(a) bipartite graph of CNF
(b) planar bipartite graph and meta-paths
(c) decomposition into meta-paths according to polarities

Figure 4.5: left: bipartite graph from Fig.4.4(a); right: planar topology of bipartite graph and the meta-paths marked with $\{+, -\}$

In this section, we present the methodologies applied in this work. Specially, we discuss the graph representation of CNF in Section 4.3.1, a flexible sparse attention for both self- and cross-attention in Section 4.3.2, and the overall architecture of framework in Section 4.3.4.

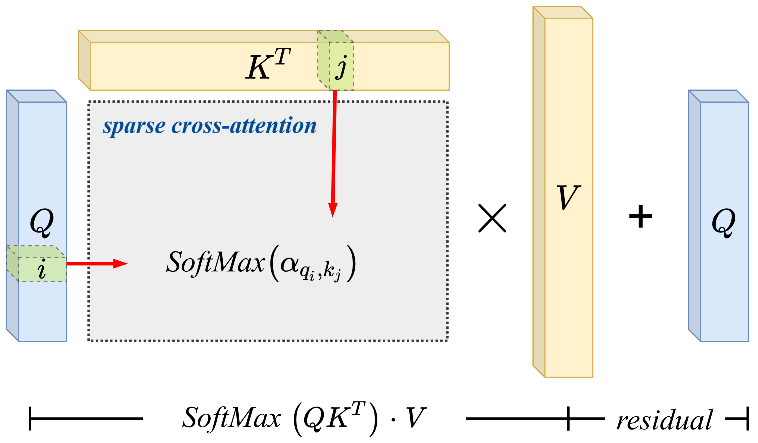


Figure 4.6: Sparse attention mechanism

4.3.1 CNF as bipartite graph and the concept of meta-paths

Each CNF equation can be formulated as,

$$\phi(V,U) = \prod_{j=1}^{M}\sum_{i\in u_j}\{v_i \text{ or } \neg v_i\} \text{ for } v_i \in V \text{ and } u_j \in U \tag{4.4}$$

where V and U are the sets of variables and clauses, respectively. Either variable v_i or $\neg v_i$ appears in the clause u_j, but not both at the same time. The expression can be properly presented as an undirected bipartite graph, as shown in Fig.4.4(a). We then construct such a bipartite graph $G((V,U),\mathcal{E})$ by defining the set of variables $V = \{v_1,\ldots,v_n\}$, the set of clauses $U = \{u_1,\ldots,u_m\}$, and edges $\mathcal{E}$ by: $e_{i,j} \in \mathcal{E}$ iff variable v_i is involved in constraint u_j either in positive or negative relation. To assist the message passing mechanism used in graph neural network, we further separate the bipartite graph in two sub-graphs, one for positive constraints and another for negative constraints, e.g., $\phi_+ = (v_1 \vee v_2)_{u_1}\cdot(v_2)_{u_2}\cdot(v_3 \vee v_4)$ and $\phi_- = (\neg v_2 \vee \neg v_4)\cdot(\neg v_1)$. Moreover, given the adjacency matrix A of the bipartite graph, each edge is assigned with a type depending on the polarity of the variable to which it connects. The positive occurrence of a variable v_i in a clause (or factor) u_j is represented with the positive sign $(+)$, whereas its negative occurrence $\neg v_i$ in u_j gets the negative sign $(-)$. Hence, a pair of $n \times m$ bi-adjacency matrix $\mathbf{A} = (A_+, A_-)$, which correspond to the pair (ϕ_+, ϕ_-), is used to store two types of edges such that $A_+(i,j) = 1 \Leftrightarrow v_i \in u_j$ and $A_-(i,j) = 1 \Leftrightarrow \neg v_i \in u_j$, the example of decomposed sub-graphs are shown in Fig.4.4(b). Here $v_i \in u_j$ implies that v_i instead of its negation $\neg v_i$ is directly involved in u_j. Each edge $e_{i,j} \in \mathcal{E}$ is then assigned a value equal to 1 for edges in A_+ and -1 for edges in A_-. With the graph representation, graph neural network can be applied to solve symbolic reasoning problem, e.g., CSP/SAT-solver [135, 184]. These two sub-graphs are then applied with the self-attention on positive and negative links, i.e., the positive and negative constraints in CNFs, respectively, as explained in Section 4.3.2.

Due to bipartite properties, variables are only connected to clauses, and vice versa, as shown in Figure 4.4.

Consequently, every node must traverse a node with different type to reach a node with same type. Furthermore, traditional *GNN* can only transfer messages between nodes with the same attributes. In this work, we propose to pass message through 2-hop **meta-paths** [150] in addition to existing edges, which enables variables (clauses) to incorporate the information from variables (clauses) that share the same clauses (variables) during the update of their states. In a *CSP/SAT* factor graph, we define that a *meta-path* $m_{i,j} = (v_i, u_k, v_j)$ between nodes v_i and v_j exists if there exists some $u_k \in U$ s.t. $\exists e_{i,k} \in \mathcal{E}$ and $\exists e_{k,j} \in \mathcal{E}$. Since self-attention mechanism is not symmetric, our meta-path is directed. As a result, we get four types of meta-paths in total, i.e., $\{(+,+), (+,-), (-,+), (-,-)\}$, as illustrated in right-hand side of Fig.4.5. The adjacent matrix of such a meta-path can be easily computed by matrix multiplication of A_+ and A_- or their transposes. Take $A_{(+,+)}$, $A_{(+,-)}$ as examples, the adjacency matrix $A_{(+,+)} = A_+ A_+^T$ stores all $(+,+)$ meta-paths, and $A_{(+,-)} = A_+ A_-^T$ stores all $(+,-)$ meta-paths. A diagonal entry $A_{(+,+)}[i,i]$ indicates the number of positive edges that v_i has, and an off-diagonal entry $A_{(+,+)}[i,j]$ indicates the existence of $(+,+)$ meta-path from v_i to v_j.

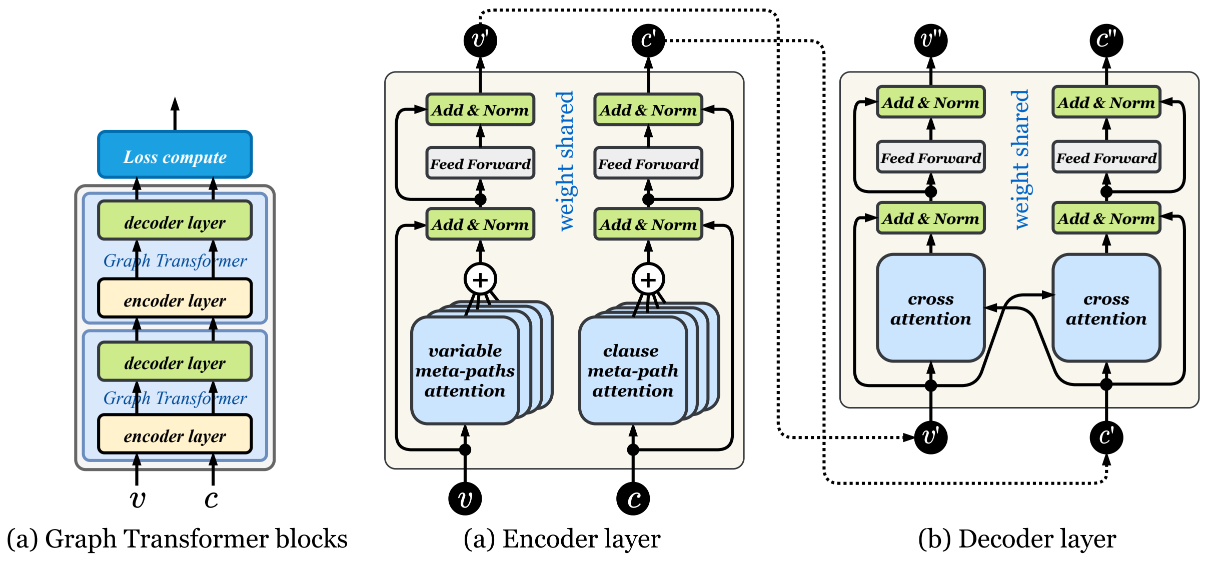


Figure 4.7: Our *Transformer-based SAT* (TRSAT) solver architecture consists of a set of encoders and decoders connected sequentially, as in (a). Its encoder and decoder architectures are shown in (b) and (c), respectively.

4.3.2 Sparse attention and graph Transformer

This work employs sparse attention coefficients for both the self-attention of meta-paths and the cross-attention between variables and clauses, as explained in section 4.3.4. The sparse attention coefficient is calculated according to the connectivity between graph nodes. As described in below equations: after the embedding in Equation 4.5, where $X = Y$ for self-attention and $X \neq Y$ for cross-attention, as shown in Figure 4.6. The node-to-node attention between a pair of connected nodes is computed first by an exponential score of the dot product of the feature vectors of these two nodes (Equation 4.6). Then the score is normalized by the sum of exponential scores of all neighboring nodes as described in Equation

4.7.

$$Q = XW_q, K = YW_k, V = YW_v \tag{4.5}$$

$$\alpha_{i,j} = \frac{\langle q_i, k_j \rangle}{\sum_{n \in N(i)} \langle q_i, k_n \rangle} \text{ or } SM(\langle Q[rows], K[cols] \rangle) \tag{4.6}$$

$$v'_i = \sum_{j \in N(i)} \alpha_{i,j} v_j, \text{ or } V' = \underbrace{\mathcal{A}}_{sparse} \times V \tag{4.7}$$

where Q, K, and V are queries, keys, and values in Transformer's terminology, respectively; and $q_i = Q(i)$, $k_j = K(j)$, $v_j = V(j)$, $\langle q, k \rangle = exp\left(\frac{q^T k}{\sqrt{d}}\right)$, and $SM(\cdot)$ is the *SoftMax* operation. Finally, we obtain attention maps $\mathcal{A}$ as multi-dimension (multi-head) sparse matrices sharing the identical topology described by a single adjacency matrix, where attention coefficients are $\mathcal{A}(i,j) = \alpha_{i,j}$. The sparse matrix multiplications can be efficiently implemented in high parallelism with the tensorization of node feature gathering and scattering operations through indexation.

4.3.3 Loss Evaluation

For a given SAT(V, U), each combination of variable assignments corresponds to a probability. The original measure $\phi(V, U)$ is a non-differentiable staircase function defined on a discrete domain. $\phi(V, U)$ evaluates to 0 if any $u_j \in U$ is unsatisfied, which disguises all other information including the number of satisfied clauses. For training purpose, a differentiable approximate function is desirable. Therefore, the proposed model generates a continuous scalar output $x_i \in [0, 1]$ for each variable, and the assignment of each v_i can be acquired through:

$$v_i = \lfloor \frac{x_i}{0.5 + \epsilon} \rfloor \tag{4.8}$$

where ϵ is a small value to keep the generated v_i in $\{0, 1\}$. With continuous $x_i, i = 1, ..., N$, we can approximate disjunction with $max(\cdot)$ function and define $\phi(\cdot)$ as

$$\phi(x_1, ..., x_N) = \prod_{j=1}^{M} \max(\{l(x_i) : v_i \in u_j\}) \tag{4.9}$$

Here, the literal function $l(x_i, e_{ia}) = \frac{1-e_{ia}}{2} + e_{ia}x_i$ is applied to specify the polarity of each variable. We replace the *max* function with a differentiable *smoothmax*, $S_r(\cdot)$:

$$S_\tau(x_1, ..., x_N) = \frac{\sum_{i=1}^{n} x_i e^{\tau x_i}}{\sum_{i=1}^{n} e^{\tau x_i}} \tag{4.10}$$

Mathematically, $S_\tau(x_1, ..., x_N)$ converges to $\max(x_1, ..., x_N)$ as $\tau \to \infty$.

We note that $\tau = 5$ is enough for our model in practice. By maximizing the modified ϕ, the proposed model is trained to find the satisfiable assignment for each CSP problem. For numerical stability and computational efficiency, we train our model by minimizing the *negative log-loss*

$$\mathcal{L}(x_i, ..., x_N) = -\sum_{j=1}^{M} log(S_\tau(\{l(x_i) : v_i \in u_j\})) \tag{4.11}$$

4.3.4 Heterogeneous Graph Transformer Architecture

We further propose the Heterogeneous Graph Transformer (TRSAT) which adopts an encoder-decoder structure, as illustrated in Figure 4.7(a). It is a flexible architecture allowing the number of encoder- and decoder-layers to be adjustable.

Encoder. Within each encoder-layer, every graph node first aggregates the message (or information) from nodes of its kind through *meta-paths*. Note that a node (variable or clause) of bipartite graph has no direct connection within homogeneous nodes. Messages can only pass among homogeneous nodes through *meta-paths*. We emphasize such type of communication between nodes of the same kind as *self-attention*, which is implemented with *homogeneous attention* mechanism regarding the polarity of variables. The attention are then connected to the residual block and *layer normalization* [9], as shown in Figure 4.7 (b).

Decoder. Inside each decoder-layer, the weighted messages are passed between variables and clauses through the *cross-attention* mechanism, implemented as the *heterogeneous attention* regarding nature of graph nodes (either variables or clauses), followed by residual connection and *layer normalization*, as in Figure 4.7 (c). The attention-weighted node features are then fed into the feed-forward network (FFN) for enhancing the node feature embedding.

4.3.5 Analysis of the Complexity

We initiate the discussion of the complexity from computing single attention head, multi-head follows the same analysis. Both the self-attention of meta-paths and the cross-attention between variables and clauses described in previous sections rely on the connectivity (or topology) of the relevant bipartite graphs, so the time complexity of computing these attention coefficients is $\mathcal{O}(|\mathcal{E}|\times|F|)$, where $|\mathcal{E}|$ is the number of edges in a graph and $|F|$ is the number of features of graph node. The node encoding module, which is a linear layer in the model, and the feed-forward network (FFN) module possess the time complexity of $\mathcal{O}(|V|\times F \times F')$, for $|V|$ the number of graph nodes. As $|\mathcal{E}|\gg |F|$ and $|V|\gg |F|$, total complexity of a single attention head is proportional to the number of nodes and edges. Furthermore, space complexity of the memory footprint for sparse attention is also linear in terms of nodes and edges.

4.3.6 MaxSAT approximates Exact SAT

Depends on the application's requirement, the SAT problem can be further categorized as the *maximum satisfiability problem (MAX-SAT)* and *exact SAT*. MaxSAT determines the maximum number of clauses of a given Boolean formula in Conjunctive Normal Form (CNF), which can be made true by an assignment of truth values to the formula's variables [170]. It is a generalization of the Boolean satisfiability problem (exact SAT), asking whether a truth assignment makes all clauses valid. Machine learning-based algorithms explore the solution space by minimizing the loss to ground truth and updating their models' weights through gradient descent during the training phase. This constraint has naturally drawn the machine learning-based approaches to focus on MaxSAT problems by performing probabilistic decision-making. Rather than obtaining the deterministic and complete solution, they approximate variable assignments. To remedy this drawback, iterative algorithms can be applied. Different from the **decimation** strategy employed in PDP, which selectively fixes the values of variables of the solved clauses, we deliver **Algorithm 1** which conditionally removes the solved clauses and their related variables from current problem. As the decimation approach of PDP does not reduce the problem's scale by fixing values of variables, our model can generate a faster and more efficient solution by decreasing the size of the problem.

With high accuracy in solving MaxSAT, we intend to approximate the solution of SAT by further replacing the *unit propagation* module of CDCL algorithm with TRSAT, as shown in Algorithm 1. To accomplish this task, we also need to perform two auxiliary procedures *conflict search* and *conflict select.*

4.4 Experiments Evaluation

Class	**Distribution**	**Variables (n)**	**Clauses** (m) / **Edges** (p)
Random 3-SAT	$rand_3(n, m)$	$n = \{100, 150, 200\}$	$m = \{430, 645, 860\}$
k**-coloring**	$color_k(N, p)$	$n = k \times N$ for N=$\{5, 10\}$	$p = 50\%$
k**-cover**	$cover_k(N, p)$	n=$(k + 1) \times N$ for N=$\{5, 7\}$	$p = 50\%$
k**-clique**	$clique_k(N, p)$	$n = k \times N$ for N=$\{5, 10\}$	$p = \{20\%, 10\%\}$

Table 4.2: The summary of our chosen dataset. For random k-SAT problems, n and m refer to the number of variables and clauses. For graph problems, N is the number of vertices, k is the problem-specific parameter, and p is the probability that an edge exists.

4.4.1 Dataset

To learn a CSP/SAT solver that can be applied to diverse classes of satisfiability problems, we selected our training set from four classes of problems with distinct distributions: random

3-SAT, graph coloring (k-coloring), vertex cover (k-cover), and clique detection (k-clique). For the random 3-SAT problems, we used 1200 synthetic SAT formulas in total from the *SATLIB* benchmark library [78]. These graphs, consisting of variables and clauses of various sizes, should reflect a wide range of difficulties. For the latter three graph-specific problems, we sampled 4000 instances from each of the distributions that are generated according to the scheme proposed in [184].

For evaluating our model's performance on CNF-based logic synthesis, we collected several circuit datasets [85, 86], including various Data Encryption Standard (DES) circuits and arithmetic circuits, from real-life hardware designs, and translated them into their corresponding *CNF* formats. Each dataset consists of 100 to 200 samples. Each benchmark subfamily of the DES circuit models, denoted as des_r_b, is parameterized by the number of rounds (r) and the number of plain-text blocks (b). The selected arithmetic circuits consist classical adder-tree (atree) as well as Braun multipliers (braun), and are denoted by their names. The largest instances from these circuit dataset contain 14K variables and 42K clauses on average, which is comparable to medium-sized SAT competition instances [77].

4.4.2 Baselines

Baseline models. To fully assess the validity and performance of our model in both CSP/SAT solving and CNF-based logic synthesis, we compared our framework against three main categories of baselines: (a) the classic stochastic local search algorithms for SAT solving - **WalkSAT** [133] and **Glucose** [6] (a variant of **MiniSAT** [44]), (b) the reinforcement learning-based SAT solver with graph neural network used for embedding phase - **RLSAT** [184], (c) the generic but innovative graph neural framework for learning CSP solvers - **PDP** [4]. Among these baselines, **PDP** falls into the hybrid of recurrent neural network and graph neural network based one-shot algorithm.

	$color_3$ (5, 0.5)	$color_3$ (10, 0.5)	$cover_2$ (5, 0.5)	$cover_3$ (7, 0.5)	$clique_3$ (5, 0.2)	$clique_3$ (10, 0.1)
RLSAT	99.01% ±9.93%	71.49% ±20.92%	93.78% ±12.12%	92.04% ±13.76%	95.72% ±15.40%	97.96% ±12.35%
Ours	**87.51%±1.45%**		**97.77%±0.11%**		**97.35%±0.37%**	

Table 4.3: Our model TRSAT's solver performance compared to that of the baseline model RLSAT. We present the metric of percentage completion in the format of: [avg.completion rate]±[std. deviation]%.

4.4.3 Experimental Configuration

Hardware. Every experiment is performed on the system with AMD Ryzen 7 3700X 8 core 16 threads CPU equipped with GeForce RTX 2080 8 GB of memory GPU. Since RLSAT, PDP, and our model consists of paralizable operations, we fully deployed on the GPU. Glucose and WalkSAT, which are sequential algorithms using backtracking, are unable to exploit the GPU.

Software. Our model is implemented with PyTorch deep learning framework and employs PyTorch Geometric [54] for graph representation learning, and is able to achieve high efficiency in both training and testing by taking full advantage of GPU computation resources via parallelism.

	$rand_3(100, 430)$		$rand_3(150, 645)$		$rand_3(200, 860)$	
	Time (s)	Acc (%)	Time (s)	Acc (%)	Time (s)	Acc (%)
PDP	0.0743	96.51±0.69	0.0413	95.50±0.23	0.0915	93.65±0.62
Ours	**0.00368**	**97.06±0.28**	**0.00361**	**96.80±1.31**	**0.0128**	**96.19±1.57**

Table 4.4: Our model TRSAT's performance compared to that of the baseline model PDP [4]. We present the validation accuracy (completion rate) in the format of: [avg.accuracy]±[std. deviation] %.

General setup. Our model for the experiments discussed in this section is configured as follows. Structures are implemented according to the architecture presented in Figure 4.7. For the encoder, we adopted four layers for both the *encoder-layers* and *decoder-layers* with the number of channels setting to 64 for all of them. Optimizer Adam [93] with $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-9}$ was applied to train the model. Our learning rate varies with each step taken, and follows a pattern that is similar to the one adopted by Noam [157].

SAT Dataset			**Ours**			**Glucose**	**WalkSAT**
Data	$\overline{\#V}$	$\overline{\#C}$	$\overline{p\%}$	$\bar{t}(CPU)$	$\bar{t}(GPU)$	$\bar{t}$	$\bar{t}$
des-3-1	5181	15455	90.7	1.56	0.113	0.73	1.26
des-4-1	7984	23944	89.5	6.77	1.102	6.36	6.14
des-4-2	14027	42232	87.4	7.45	1.396	6.17	7.33
atree	13031	41335	88.8	8.61	2.396	7.14	8.50
braun	4116	13311	92.7	6.37	1.166	5.06	6.68

Table 4.5: Average test completion rate ($\overline{p}\% = \frac{solved\, samples}{\#SAT\, samples}$) and average solving time ($\bar{t}$ seconds) between our model and other approaches for CNF-based logic synthesis.

4.4.4 Results and Evaluation

4.4.4.1 General CSP/SAT solving

We first compare the accuracy metric with RL-based deep model. The accuracy metric represents the average percentage of clauses solved by the models with the generated assignments to variables. Due to the sequential nature of RL, the runtime performance compared to our model is not insightful. Table 4.3 summarizes the performance of our model and that of RLSAT, after training for 500 epochs, on the chosen datasets. Since our model adopts a semi-supervised training strategy, and is capable of processing graphs of arbitrary size, we were able to combine numerous distributions of the same problem class into one single

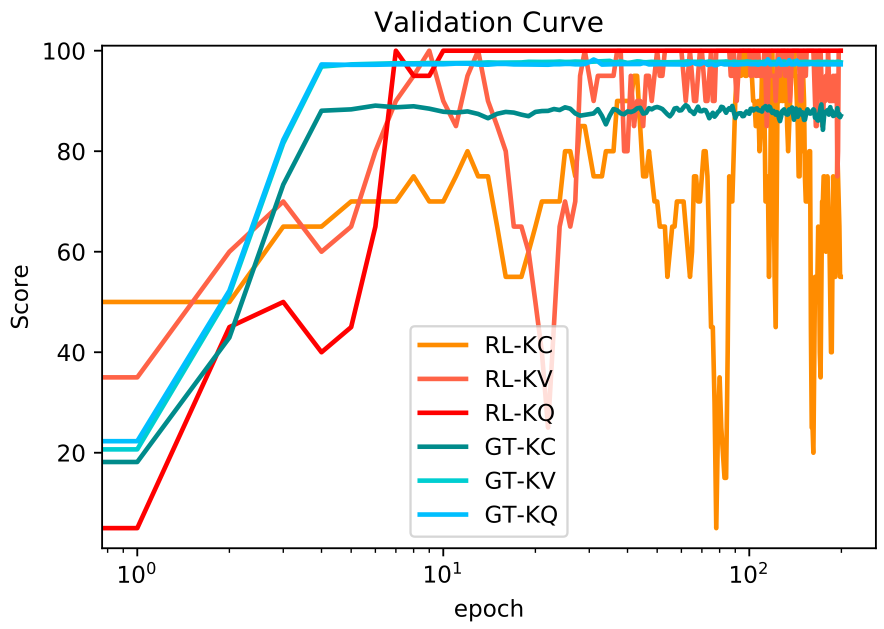


Figure 4.8: The learning curve of Our model (TRSAT) and that of RLSAT (RL).

dataset during training, regardless of the problem-specific parameters. Our model achieves higher completion rate than RLSAT.

For further analysis, we present the holistic learning curves in Figure 4.8. In this figure, both models are trained on $\{$KC: $color_3(10, 0.5)$, KV: $cover_3(7, 0.5)$, KQ: $clique_3(10, 0.1)\}$, and the shaded areas visualize the standard deviations of each model's validation scores. From the figure, we noticed that for the latter 100 epochs, *RL-KC* and *RL-KV*'s validation performance oscillate significantly. Investigating the characteristics of *Reinforcement Learning*, we discovered that RLSAT, upon encountering graphs with new scales, performs a whole new process of exploration. Therefore, RLSAT fails to generalize its learnt experience to subsequent larger graphs, which results in an unstable validation score during training. In contrast to RL-based model, our model adopts a highly parallel message-passing mechanism, which updates all nodes of all graphs simultaneously at each epoch.

In addition to testing on a diversified distribution of graph problems, we also experimented

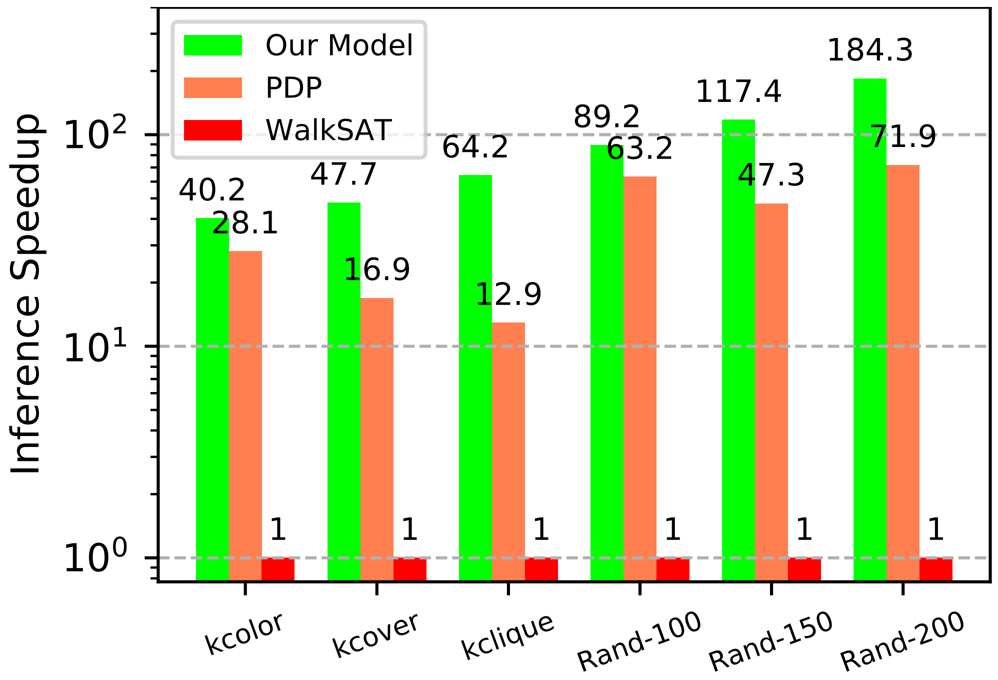


Figure 4.9: The inference speedup for CSP/SAT solving.

on the classic random 3-SAT dataset, and compared our results with that of PDP, which is recent work following NeuroSAT with a hybrid of GNN and RNN. As seen in Table 4.4, our model retains the ability to achieve a high clause assignment completion rate. In comparison, PDP takes a significantly longer time for inference, while reaching an average completion rate that does not exceed ours. To further analyze these speed discrepancies, we present in Figure 4.9 the average speedup of our model against that of PDP, with the performance of WalkSAT as the metric. As demonstrated in the figure, our model is capable of achieving higher average test speeds regardless of the graph structure. This observation can be explained by the fact that our model allows communication within homogeneous nodes, which provide all nodes with abundant semantic information when updating their states. Therefore, our model requires fewer iterations of message passing, and achieves greater efficiency.

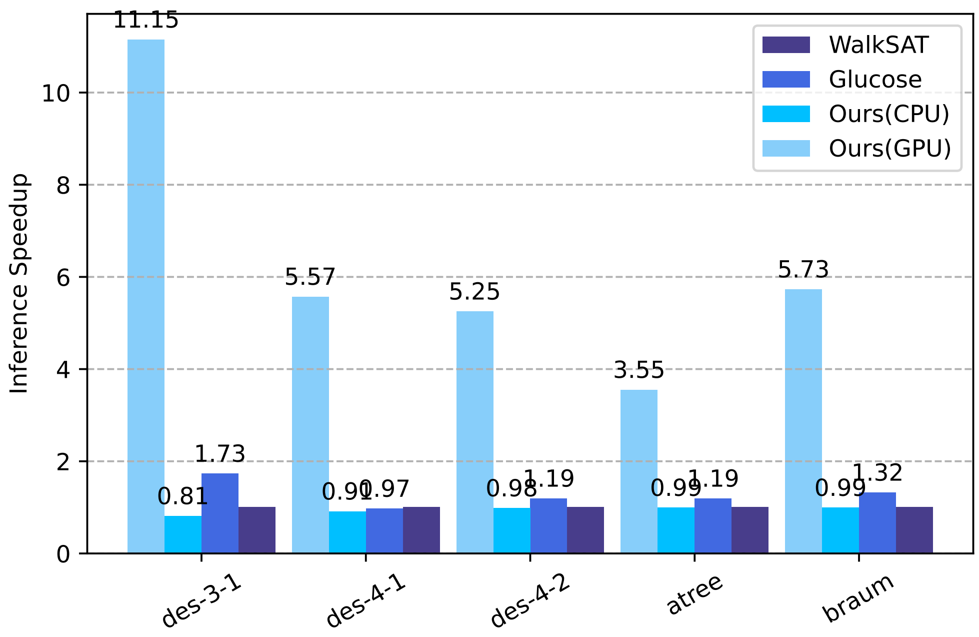


Figure 4.10: The inference speedup for CNF-based logic synthesis.

4.4.4.2 CNF-based logic synthesis

Apart from the general CSP/SAT evaluations, we also assessed our model's performance on solving CNF-based logic synthesis problems. Our proposed model is highly parallel and one-shot model based on neural symbolic learning for solving the CNF-based logic synthesis problems. Hence, we selected the classic stochastic algorithms WalkSAT and Glucose as authoritative baselines for comparison. We summarized the test results in Table 4.5. After training our model on the selected dataset for 500 epochs, our model achieves an average completion rate up to 88.7% for circuit of DES datasets, and 89.3% for arithmetic circuit datasets. We did not compare the completion rate of our model to those of the heuristic solvers, since they eventually solve all the problems without time limitation. Rather we focused on our model's latency to pursue acceleration, which could potentially help discovering early partial assignments to the heuristic solvers. The average solving time in Table 4.5 was calculated based on all the solved (including unsatisfied cases) cases in demanded time period.

Some difficult test cases causes the heuristic solver trapped into infinite loop and took hours to stop even by force; while learning-based approach with trainable weights has fixed pipeline (fixed execution time), for those hard problem, it quits quickly with wrong answer (false assignment or wrong counter example for unsatisfied cases) compared to ground truth.

Consequently, our solver paired with a more guaranteed but slower deterministic solver, provides substantial overall speedup, while ensuring a solution. Visualized in Figure 4.10 are our model's average solve speeds compared against that of Glucose, with the performance of WalkSAT as the metric. Once again, our model significantly outperforms the baseline models, regardless of the circuit structures being analyzed. Furthermore, it is worth noting that our test set contains instances with very different distributions regarding variable and clause numbers, which reflects our model's scalability to work on wide range of tasks (as shown in Table4.5) for logic synthesis problems of diverse difficulties without changing the main architecture.

Hybrid mode comparison. We also perform the experiment in which our solver is paired with a deterministic solver (CDCL), provides substantial overall speedup, while ensuring a solution, as shown in Table 4.6.

Table 4.6: Hybrid mode (TRSAT + CDCL) v.s Glucose

SAT Dataset	**Glucose**		**Ours**	
	$\overline{\#\text{ of run}}$	$\bar{t}/run(sec)$	$\overline{\#\text{ of run}}$	$\bar{t}/run(sec)$
$rand_3(20,*)$ [78]	10	3.11e-5	4	1.05e-5
des-4-2 [85]	733	2.82e-5	156	1.86e-5
MaxSAT2016 (v70c700) [**maxsat2016**]	47	6.97e-5	11	2.17e-5
MaxSAT2016 (v110c800) [**maxsat2016**]	289	4.43e-5	75	1.91e-5

Our model dramatically reduces the number of backtracking iterations ($\overline{\#\text{ of run}}$) by 60% to 78% compared to the heuristic-based solver, and each iteration benefits higher parallelism and costs much less time ($\bar{t}/run(sec)$) giving 57% to 69% improvement. The hybrid model improves the overall unit propagation latency by 86% to 93% from that of Glucose solver. In a case where TRSAT is solving the instance on the very first try, for example, the overall unit propagation latency for solving one of the instances of $rand_3(20, *)$ is only $0.00047s$ which is roughly 10 times less than that of Glucose, which is $0.0045s$.

4.5 Conclusion

In this work, we proposed *Transformer-based SAT Solver (TRSAT)*, a one-shot model derived from the eminent Transformer architecture for bipartite graph structures, to solve the MaxSAT problem. We then extended this framework for logic synthesis task. We defined the homogeneous attention mechanism based on meta-paths for the self-attention between literals or clauses, as well as the heterogeneous cross-attention based on the bipartite graph links from literals to clauses, vice versa. Our model achieved exceptional parallelism and completion rate on the bipartite graph of MaxSAT with arbitrary sizes. The experimental results have demonstrated the competitive performance and generality of *TRSAT* in several aspects. For future work, we want to analyze our initial results to check how the predicted partial solutions could contribute to the heuristic solvers to reduce the number of backtracking iteration in finding exact SAT solutions.

Algorithm 1 MaxSAT algorithm approximates exact-SAT in hybrid mode

```
function ExactSAT($V, C$)
▷ $V = \{v_1, v_2, \cdots, v_N\}$, $C = \{c_1, c_2, \cdots, c_M\}$, and
▷ $VAR(c_i)$: set of all variable in the clause $c_i$; $A$: bipartite graph of $C$ and $V$
    $V_{sat} \leftarrow \emptyset; U \leftarrow C$;
    for $U \neq \emptyset$ do
▷ our model replaces unit-propagation and fused with CDCL
        $(V, C) = TRSAT(V, C)$;
        $U \leftarrow \{c_i = 0 : c_i \in C\}$                                  ▷ get unsolved clauses
        if $U = \emptyset$ then
            return $V_{sat} \cup \{v_i \in V\}$
        else
            $V_u \leftarrow \cup\{VAR(c_i) : c_i \in U\}$
            if $V - V_u = \emptyset$ then
                return $solvable \leftarrow$ false
            else
                $C \leftarrow C - \{c_i = 1 : \exists v_j \in c_i \text{ and } v_j \notin V_u\}$    ▷ conflict search
                $V_u \leftarrow V_u \cup \{VAR(c_i) : c_i \in C\}$
                $c_{conflict} \leftarrow max(A \cdot A^T \cdot \mathbf{1}^T)$                    ▷ conflict select
                $(V, C) \leftarrow$ CDCL($V_u, c_{conflict}$)
    return $V_{sat}$
```

Part II

Unified Hardware Acceleration – Chip for AI

CHAPTER 5

Sparse Winograd Convolutional neural networks on small-scale systolic arrays

This chapter intends to accelerate the convolution operations in convolutional neural networks, as shown in Figure 5.1 — the bone architecture used in most modern computer vision tasks. We implement the Winograd transform on the input tensor and the trained weights. Winograd transform can dramatically reduce the number of operations compared with the direct convolution operations. With the rearrangement of data access pattern traversing dimensions, we utilize the matricization to transform tensors and then perform matrix multiplication between input tensors and weight tensors. Therefore the hardware design part is simplified to optimize the acceleration of the matrix multiplications. The complexity of hardware design resides manageable and easy to handle.

5.1 Introduction

Convolutional neural network (CNN) is a class of deep learning algorithms which has become dominant in various computer vision tasks [1, 109], so it is attracting research on acceleration for computational and power efficiencies. The core computations in the algorithm are convolution operations with multi-dimensional data, e.g. 3-D feature maps (FM) and 4-D filters, which require a high density of memory accesses and high throughput of the computation engine. One research topic emerging in recent years is to deploy the convolution operations onto FPGAs [190, 149, 189, 41], since FPGAs consist of massive compute units,

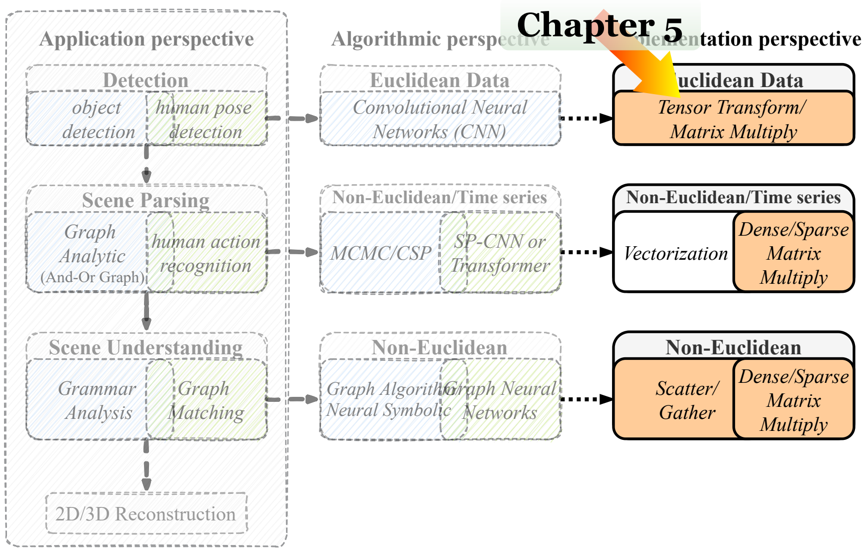


Figure 5.1: The position of this chapter in the platform

e.g. DSP blocks, and storage elements interconnected by reconfigurable switch blocks. The most recent works on systolic array-based FPGA accelerators [169, 31] deliver significant performance improvement on the automation of high-level synthesis (HLS) design flow. Unlike the works [46, 169], which first construct 2-D mesh architecture for systolic array then let the loops of codes to fit on these arrays (bitstream generated once), we recursively break the memory layout down to small blocks then map these blocks onto small-scale systolic arrays to perform multiplications of submatrices, and share these submatrices among working arrays to reduce required memory bandwidth. Another performance improvement can be achieved from algorithmic perspective by applying the Winograd transform. This approach attracts more and more attention from researchers since its first GPU implmentation [100]. Winograd CNN accelerators on FPGAs are also well studied recently [41, 8]; however, the greater volume after the Winograd transformation is stressing on FPGAs. To handle this issue we adopt an efficient memory layout, adopt the pruned Winograd weights [28] and their elaborate hardware, and extend the computation into 3-D. Pruning neural networks has been proven to greatly decrease both latency and energy consumption for all range of devices [70]. The major contributions are summarized in the following:

- **Unified small-scale systolic arrays for both Winograd transform and matrix multiplications**. We maximize the reusability of the existing design, e.g. RTL, for multiple modules. These modules share common characteristics, like matrix multiplication alike arithmetic operations.
- **Efficient memory access layout.** We employ a recursive memory access pattern to increase locality of buffers. This pattern significantly impacts the overall performance.
- **Block-based sparse matrix compression.** We employ this compression technique to adopt the above mentioned recursive memory layout.
- **A comprehensive model analysis of Winograd convolution.** We propose an analytical model to investigate the performance and energy consumption, and based on the analysis we use the conclusion as our design guidance.

5.2 Background

5.2.1 Spatial Convolution

The convolution layer in a feedforward pass takes C channels of $H \times W$ feature maps D as input, and convolve each of K filters of dimension $C \times r \times r$ with the input feature maps to produce K output featre maps, Y, of dimension $(H - r + 1) \times (W - r + 1)$. Let s be the stride and assume that the width and height of the filters are the same, then the mathematical description of the convolution is

$$Y_{k,i,j} = \sum_{t=1}^{C} \sum_{p=1}^{r} \sum_{q=1}^{r} G_{k,t,p,q} \times D_{t,i*s+p,j*s+q} \tag{5.1}$$

5.2.2 Winograd Algorithm

Winograd proposed an efficient algorithm for short convolutions [173] in computing of finite impulse response (FIR) filters in the signal processing field. [100] extends the Winograd algorithm to convolutional neural networks on GPU and CPU.

By applying Winograd transform to an r-tap FIR filter denoted as $F(m, r)$, which computes m outputs with the filter size of r, the number of multiplications is reduced from $m \times r$, if through the spatial convolution, to $m + r + 1$.

5.2.2.1 1-D Winograd Convolution

Taking $F(2, 3)$ as an example, Winograd algorithm first transforms an input vector $d = (d_0, d_1, d_2, d_3)$ and filter $g = (g_0, g_1, g_2)$ into $j = (j_0, j_1, j_2, j_3)$ and $h = (h_0, h_1, h_2, h_3)$ respectively through

$$\begin{aligned} j_0 &= d_0 - d_2, \quad h_0 = g_0 \\ j_1 &= d_1 + d_2, \quad h_1 = \frac{g_0 + g_1 + g_2}{2} \\ j_2 &= d_2 - d_1, \quad h_2 = \frac{g_0 - g_1 + g_2}{2} \\ j_3 &= d_1 - d_3, \quad h_3 = g_2 \end{aligned}$$

Next, element-wise multiplications are performed:

$$c_0 = j_0 \times h_0,\ c_1 = j_1 \times h_1,\ c_2 = j_2 \times h_2,\ c_3 = j_3 \times h_3 \tag{5.2}$$

Finally, the output $y = (y_0, y_1)$ can be generated via:

$$y_0 = c_0 + c_1 + c_2, \quad y_1 = c_1 - c_2 - c_3 \tag{5.3}$$

The matrix form of the above procedure can be written as $y = A^T \left[(Gg) \odot \left(B^T d\right)\right]$, where $\odot$ represents element-wise multiplication and

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

The element-wise product in (5.2) requires $m + r - 1 = 4$ multiplications, whereas the direct method does $m \times r = 2 \times 3 = 6$ multiplications.

5.2.2.2 2-D Winograd Convolution

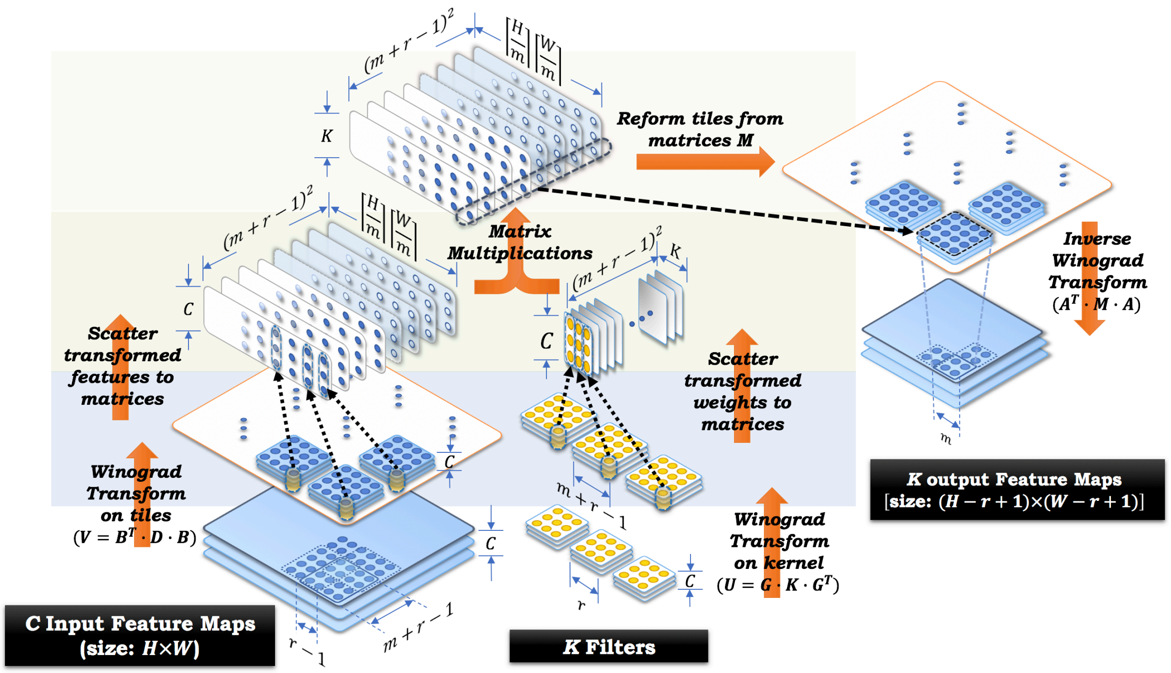
The 1-D Winograd algorithm can be easily extended to 2-D or higher dimensional convolutions by being nested with itself. 2-D Winograd algorithm $F(m \times m, r \times r)$ can be formulated as follows,

$$Y = A^T \left[\left(GgG^T\right) \odot \left(B^T dB\right)\right] A \tag{5.4}$$

where d and g are tiles of input and the filter, having size of $l \times l$ $(l = m + r - 1)$ and $r \times r$, respectively. The size of the output tile Y is $m \times m$. For larger input images, the Winograd transform is performed with the overlapping of tiles, with overlapping size $r - 1$, along each dimension. When applying Winograd algorithm to a convolution layer of CNNs, the tiles along the channel dimension of this layer can be fetched simultaneously and each of them is applied with (5.4).

5.3 Algorithm and Optimizations

This section gives an overview of our algorithm and presents several optimization methods. Fig. 5.2 shows the overview of our algorithm which consists of three stages of the Winograd-based convolution: input feature map and kernel transformations, matrix multiplications,



Reform tiles from matrices M
Matrix Multiplications
Inverse Winograd Transform ($A^T \cdot M \cdot A$)
Scatter transformed features to matrices
Scatter transformed weights to matrices
Winograd Transform on tiles ($V = B^T \cdot D \cdot B$)
Winograd Transform on kernel ($U = G \cdot K \cdot G^T$)
C Input Feature Maps (size: $H \times W$)
K Filters
K output Feature Maps [size: $(H - r + 1) \times (W - r + 1)$]

Figure 5.2: An overview of Winograd convolution layer.

and the inverse transformation of the output feature maps. These three stages form the pipeline of the data flow of our system design.

5.3.1 Reduction to Matrix multiplication

By reformulating (5.4) with the augmentation on the channel dimension, filter k, tile coordinates $(\tilde{x}, \tilde{y})$, and substitution of $U = GgG^T$ and $V = B^T dB$, we get

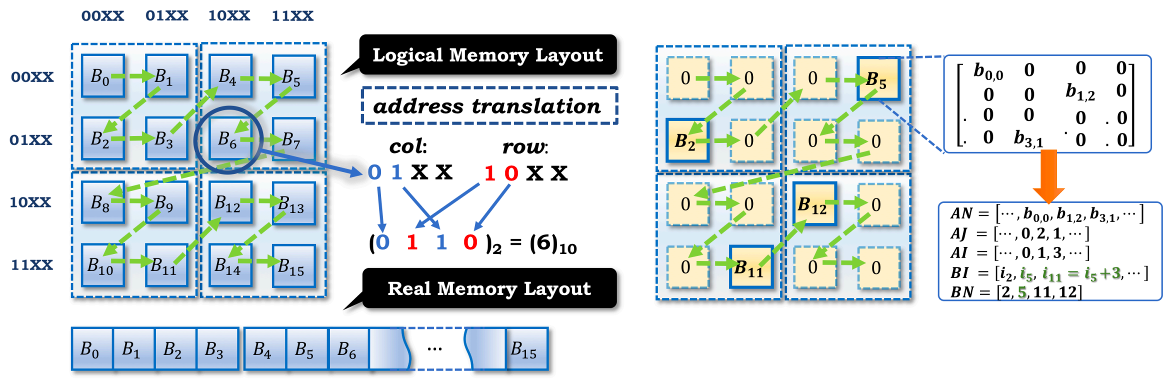
$$Y_{k,\tilde{x},\tilde{y}} = A^T \left[\sum_{c=1}^{C} U_{k,c} \odot V_{c,\tilde{x},\tilde{y}} \right] A \tag{5.5}$$

The summation part inside the parenthesis of (5.5) can be disentangled into $(m+r-1)^2$ individual multiplication of a matrix of size $(C \times K)$ with another of size $(C \times \lceil H/m \rceil \lceil W/m \rceil)$.

$$\mathcal{M}_{k,\tilde{x},\tilde{y}} = \sum_{c=1}^{C} U_{k,c} \odot V_{c,\tilde{x},\tilde{y}} \quad \xrightarrow[(\tilde{i},\tilde{j}) \text{ of tile}]{\text{collapsing } (\tilde{x},\tilde{y}) \text{ to } b} \quad \mathcal{M}_{(k,b)}^{(\tilde{i},\tilde{j})} = \sum_{c=1}^{C} U_{k,c}^{(\tilde{i},\tilde{j})} V_{c,b}^{(\tilde{i},\tilde{j})}$$

Another benefit of this reformation into matrix multiplications is that the number of inverse transforms has also been reduced over C channels [100], since the factorization of inverse transform along channels amortizes the cost. With this reformation, the matrix multiplications are then efficiently implemented on FPGAs.

5.3.2 Matrix multiplications and memory access patterns



00XX 01XX 10XX 11XX
Logical Memory Layout
address translation
col:
row:
0 1 X X
1 0 X X
(0 1 1 0)₂ = (6)₁₀
Real Memory Layout
B_0 B_1 B_2 B_3 B_4 B_5 B_6 … B_{15}
$AN = [\cdots, b_{0,0}, b_{1,2}, b_{3,1}, \cdots]$
$AJ = [\cdots, 0, 2, 1, \cdots]$
$AI = [\cdots, 0, 1, 3, \cdots]$
$BI = [i_2, i_5, i_{11} = i_5+3, \cdots]$
$BN = [2, 5, 11, 12]$

a) block-based logical to physical memory layout translation

b) block-based sparse coordinates (BCOO)

Figure 5.3: Z-Morton memory layout for both dense and sparse matrix [58, 34]: (a) the translation from logical layout to physical layout, (b) the block-based compressed coordinates (BCOO, $l \times l$ block and $l = 4$ for our design) for pruned Winograd weights

As described in section 5.3.1, Winograd convolution can be computed efficiently with matrix multiplications on GPUs or FPGA platforms. To optimze the performance of matrix multiplication, we employ the Z-Morton memory layout [58], which has been widely studied for the Cache oblivious algorithms on multithreaded CPUs [58, 2] and image processing on

FPGAs [34]. This memory layout increases both spatial and temporal locality of memory accesses of matrix multiplication and arithmetic operations [58].

Algorithm 2 Divide and Conquer Matrix Multiplication

```
function RECURSIVE_MATMULT($A, B$)            ▷ given $l$ is the smallest tiling size
    $n = A.rows$
    if $n = l$ then
        $C = A \times B$                      ▷ matrix multiply of $l \times l$ tiles
    else
        partition $A$, $B$, and $C$ into tiles of size $\frac{n}{2} \times \frac{n}{2}$
        $C_{1,1}$ = RECURSIVE_MATMULT($A_{1,1}, B_{1,1}$) [1]
            +RECURSIVE_MATMULT($A_{1,2}, B_{2,1}$)
        $C_{1,2}$ = RECURSIVE_MATMULT($A_{1,1}, B_{1,2}$)
            +RECURSIVE_MATMULT($A_{1,2}, B_{2,2}$)
        $C_{2,1}$ = RECURSIVE_MATMULT($A_{2,1}, B_{1,1}$)
            +RECURSIVE_MATMULT($A_{2,2}, B_{2,1}$)
        $C_{2,2}$ = RECURSIVE_MATMULT($A_{2,1}, B_{1,2}$)
            +RECURSIVE_MATMULT($A_{2,2}, B_{2,2}$)
    return $C$
```

Z-Morton uses a *divide and conquer* approach to access the memory as in Fig. 5.3 (a). It is actually derived from the recursive matrix multiplication described in *Algorithm* 2. Compared with Strassen's algorithm, the latter is not cache-friendly in real situations, whereas the former can provide notable improvement in performance [2]. Note, instead of implementing the algorithm exactly, we unrolled memory access order to reorganize the memory layout.

The physical memory layout in FPGAs is essentially linear, Fig 5.3 (a) also provides an example of translating the logical block address to physical block address. As shown in Fig. 5.3 (a), the address translation is easily implemented with LUTs in FPGAs by interleaving the bits of the logical column and row addresses to generate the physical address of a block.

5.3.3 Pruned Winograd weights and memory access patterns

After pruning the Winograd weights, we store them in a block-based sparse coordinates format (BCOO)–only those 4×4 blocks containing nonzeros will be compressed and stored. Fig. 5.3 (b) shows an example where the block B_5 is a 4×4 tile, and it has 3 nonzeros. The information of these nonzeros are stored into vectors BN, BI, AI, AJ, and AN. BN contains the block number for each block in memory layout, e.g. 5 for B_5. BI is the list of starting indices of each block within the other three arrays, e.g. i_5 of BI refers to the starting index in AI, AJ, and AN of information corresponding to B_5. Elements in AI and AJ represent the row and column number of the nonzeros in its own block, respectively, and AN stores the value of the corresponding nonzero. For B_5, the values of nonzeros are $b_{0,0}$, $b_{1,2}$, and $b_{3,1}$, the corresponding column numbers are 0, 2, 1 and row numbers are 0, 1, 3 in AJ and AI, respectively. The compressed blocks are still fetched following the order determined by Z-Morton layout.

5.4 Architecture Design

This section discusses our implementation of accelerator for Winograd convolution. The most time-consuming parts in the computation pipeline are the Winograd transform for feature maps and matrix multiplications. In our design, we propose using unified small-scale systolic arrays, of size $l \times l$ ($l = m + r - 1$), for both these arithmetic operations.

5.4.1 Winograd transform by Systolic Arrays

Recall the 2-D Winograd transform nesting 2 transform matrices, $B^T \cdot D \cdot B$. Instead of directly computing $B^T \cdot D \cdot B$, we change it into $\left(D^T \cdot B\right)^T \cdot B$. Thus, we let transform matrix B be stationary inside the systolic arrays. In the first iteration $\textcircled{1}$ of the Fig. 5.4 D^T passes through systolic arrays to operate with B and the output is $P = \left(C + D^T \cdot B\right)^T$ (no

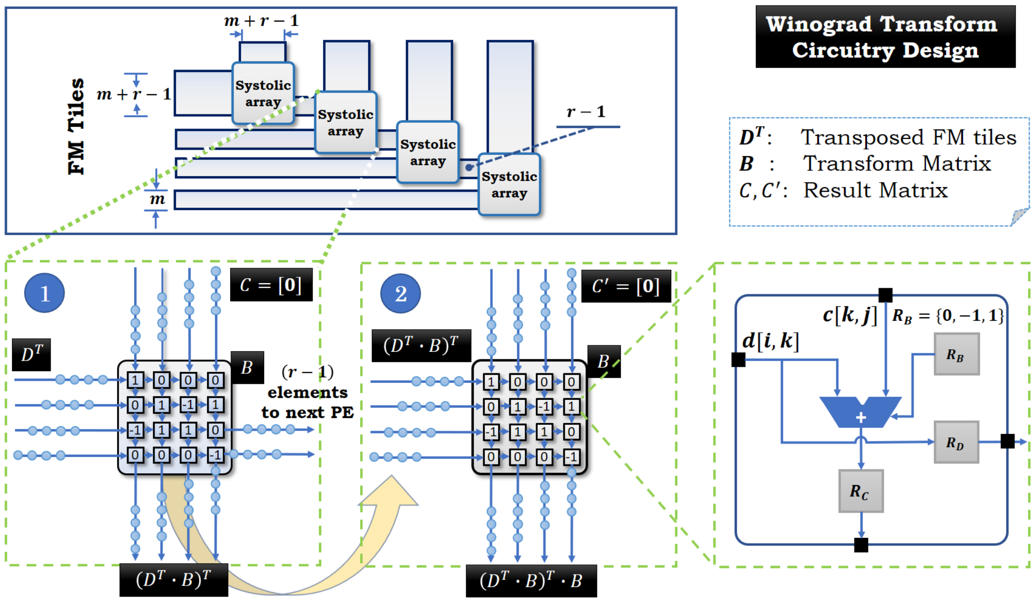


Figure 5.4: Small-scale Systolic Arrays for Winograd Transform

additional transpose needed). This intermediate result $(D^T \cdot B)^T$ feeds back to systolic arrays as "new D^T" in the second iteration ②. Then $P' = C' + P \cdot B = (D^T \cdot B)^T \cdot B = B^T \cdot D \cdot B$ is the final resutl. Note that C and C' are zero-matrices and there is no multiplication occured inside these systolic arrays–the value of elements of B is just used to control the adder–such as, "1" for addition, "-1" for subtraction, and "0" for passing by the data to next processing element (PE) inside its systolic array.

The data sharing is through the overlapping of tiles, which has been described in section 5.2.2.2. Fig. 5.4 illustrates that $(m + r - 1)$ wide data stream into each systolic array, and among these data, $(r - 1)$ of them travel through the current systolic array and are forwarded to the next systolic array at the same direction. The output is streamed out in the orthogonal direction after two iterations as stated previously, and is transfered into shift-registers for scattering into matrices.

5.4.2 Matrix Multiplication by Systolic Arrays

To perform the recursive matrix multiplication *Algorithm 2* with hardware, we conceive the cluster of small-scale systolic arrays. Each cluster consists of 4 $l \times l$ systolic arrays ($l = 4$ for our case) and a set of shared circular FIFO built by shift-registers, shown in Fig. 5.5. To understand how this cluster works, let us examine the example from Fig. 5.3. By unrolling the recursive code given by *Algorithm 2* and using the tiles of matrices organized by Z-Morton layout, we calculate sub-matrix C_0 by summing up the products of submatrices $A_0 \times B_0$ and $A_1 \times B_2$, C_4 by sum of $A_0 \times B_4$ and $A_1 \times B_6$, and so on.

$$\begin{aligned}
C_0 &\mathrel{+}= A_0 \times B_0 + A_1 \times B_2;\\
C_4 &\mathrel{+}= A_0 \times B_4 + A_1 \times B_6;\\
C_8 &\mathrel{+}= A_8 \times B_0 + A_9 \times B_2;\\
C_{12} &\mathrel{+}= A_8 \times B_4 + A_9 \times B_6;\\
&\cdots\\
C_0 &\mathrel{+}= A_4 \times B_8 + A_5 \times B_{10};\\
C_4 &\mathrel{+}= A_4 \times B_{12} + A_5 \times B_{14};\\
&\cdots
\end{aligned}$$

As shown in Fig.5.5 (a), A_0 is shared by northwest and southwest systolic arrays, A_8 is shared by northeast and southeast systolic arrays, and so on. After the first iteration, the partial results of C_0, C_4, C_8, and C_{12} are produced and stored inside the corresponding systolic arrays. In the second iteration, the blocks A_1, A_9, B_4, and B_9 get into their corresponding systolic arrays and perform the matrix multiplications, and their products are accumulated to the partial results, which still stay in their systolic arrays from iteration 1. At iteration 3 the results of C_0, C_4, C_8, and C_{12} are spilled out, and systolic arrays continue to work on the partial results of C_1, C_5, C_9, and C_{13}. This procedure continues until all the submatrices are calculated. Also the sharing of circular FIFOs reduces the memory bandwidth requirement by

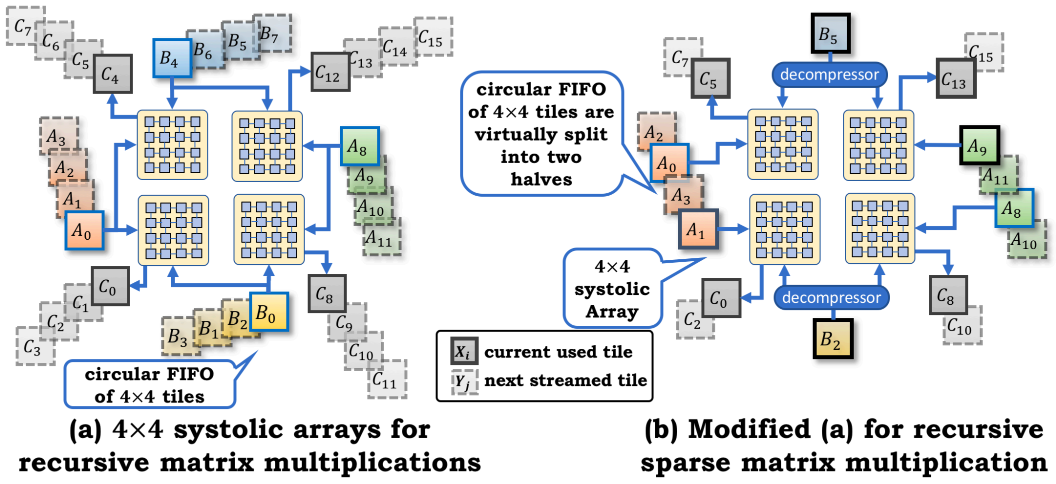

circular FIFO of 4×4 tiles
circular FIFO of 4×4 tiles are virtually split into two halves
4×4 systolic Array
decompressor
current used tile
next streamed tile
(a) 4×4 systolic arrays for recursive matrix multiplications
(b) Modified (a) for recursive sparse matrix multiplication

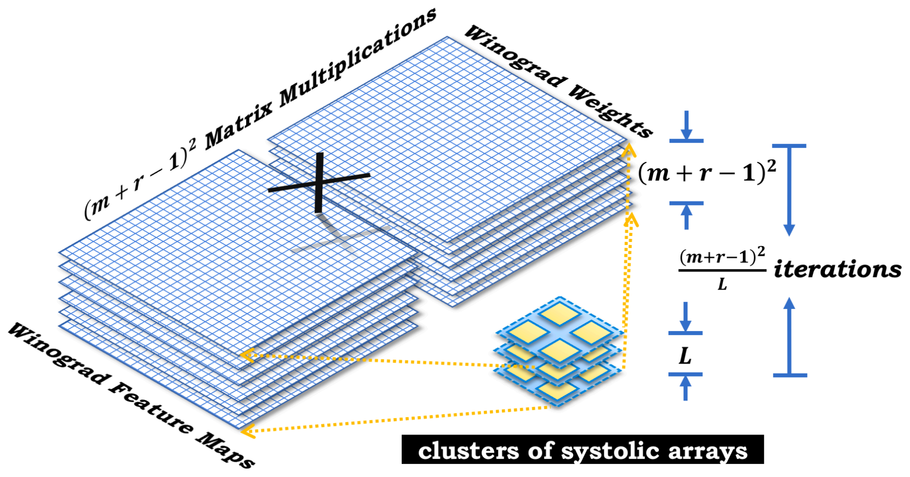
Figure 5.5: Systolic Arrays for *Algorithm 2*: (a) the original design for dense case, (b) modified architecture for sparse case

4 folds. Recent work of [165] proposes an automatic flow for generating systolic array-based accelerator, which is able to facilitate the design of our approach.

When the computation is comprised of sparse matrix multiplications, we need some modifications on the cluster of systolic arrays. First, each of the circular FIFOs which supply the compressed Winograd weight blocks need to be equipped with a decompressor. Second, the circular FIFOs for Winograd feature maps are virtually split into two halves since some Winograd feature maps blocks are no longer shared between the systolic arrays. The overall memory access pattern is now determined by how the sparse blocks distributed in the memory layout. Take the sparse blocks B_2 and B_5 from Fig. 5.3 for example; now we notice that the computation of C_0 becomes $A_1 \times B_2$ only, C_8 becomes $A_9 \times B_2$, block B_2 is still shared by the products of submatrices C_0 and C_8.

5.4.3 Extends the computation into third dimension

Whenever the computation resource is available, we can extend the computation into higher dimensions. As we have analyzed in section 5.3.1, there are $(m + r - 1)^2$ independent matrix



$(m+r-1)^2$ Matrix Multiplications
Winograd Weights
$(m+r-1)^2$
$\frac{(m+r-1)^2}{L}$ iterations
L
Winograd Feature Maps
clusters of systolic arrays

Figure 5.6: Extension of computation to 3-D dimension

multiplications, and they can be executed in parallel with several clusters of systolic arrays as demonstrated in Fig. 5.6. With this enhencement, the DSP utilization and throughput of the FPGA system are dramatically improved. In our design, we organize the DSPs into 8 clusters due to the limited amount of DSPs in our FPGA board.

5.4.4 Extension to other types of layers

In addition to convolution layers, fully-connected (FC) layers are essentially computed through matrix multiplications. Therefore, the techniques previously discussed can be also employed to FC layers. ReLU layers and Max Pooling layers are easily implemented by accompanying comparators to the output buffers.

5.4.5 Overall architecture

The overall architecture of our model is illustrated in Fig. 5.7. We use a cluster of small systolic arrays for Winograd transform, as described in Section 5.4.1. A larger cluster of small systolic arrays are grouped into sets of 4 to perform the recursive version of matrix multiplications as described in Section 5.4.2. Data buffers are utilized to synchronize data

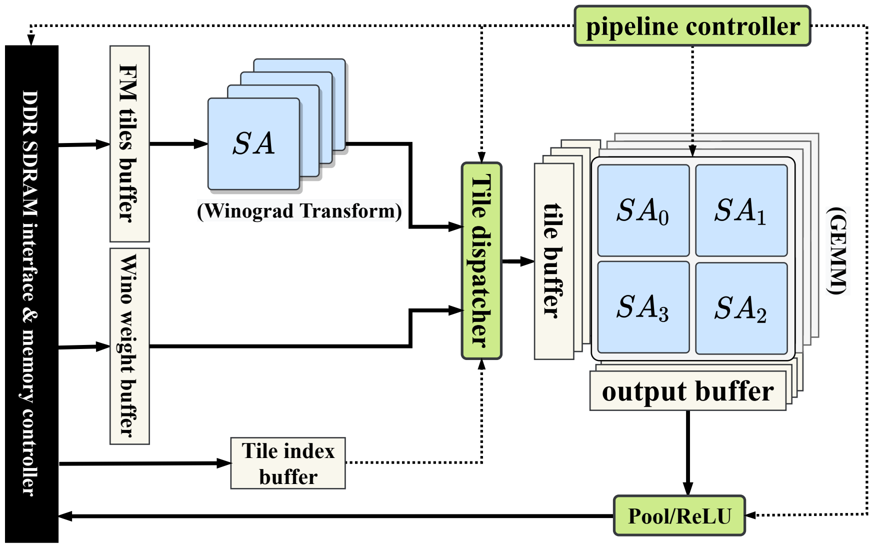


Figure 5.7: Architecture of our design

transfer between modules and keep the pipeline filled up. The tile indices are generated as by-product during the Winograd transform, and their buffers are served for two purpose: 1) when dense matrix multiplication is performed (e.g., FC layers), the tile index is used for selecting which cluster and systolic arrays to be dispatched; 2) when sparse mode is set, the tile index is also used to indicate whether current weight tile containing only zeros and assist to skip the matrix multiplication with corresponding Winograd-transformed feature map tiles. The weights are pre-processed through Winograd transform and directly stored in memory and fetched into Wino-weight buffers.

5.5 Design Space Exploration

5.5.1 Model Analysis

A detailed study of the complexity of Winograd convolution is conducted in the following subsections, it helps us to design an optimzed accelerator for both dense and sparse cases.

5.5.1.1 Data Layout of Winograd transform

As previously mentioned, the input feature maps are fed in system in real-time. It's not convenient to prune them during the inference, and it will increase the difficulty in system design. Moreover, the multiplication of a sparse matrix with a dense one does not necessarily produce another sparse matrix. In such case, our analysis keeps the same characteristics of feature maps for both dense and sparse cases. The volume of i^{th} Winograd convolution layer D^i_{wi}, the volume of corresponding Winograd weights D^i_{wk} (without pruning), and the volume of the results D^i_{wo} before the inverse Winograd transform can be computed as

$$D^i_{wi} = \left\lceil \frac{H}{m} \right\rceil \times \left\lceil \frac{W}{m} \right\rceil \times C \times l^2 \approx \left(\frac{l}{m} \right)^2 \times H \times W \times C \tag{5.6}$$

$$D^i_{wo} = \left\lceil \frac{H}{m} \right\rceil \times \left\lceil \frac{W}{m} \right\rceil \times K \times l^2 \approx \left(\frac{l}{m} \right)^2 \times H \times W \times K \tag{5.7}$$

$$D^i_{wk} = C \times K \times l^2 \tag{5.8}$$

The Winograd transform dilates both the input feature maps and weights by a scale factor of $\left(\frac{l}{m}\right)^2$, e.g. when m takes value of 2 and r of 3, the transformed feature maps and weights require roughly 1.78 times larger storage. The increased volume of the storage not only affects the latency of computations due to the drastically slow access speed, but also causes more energy consumption.

5.5.1.2 Arithmetic complexity

The arithmetic complexity greatly depends on the data layout since the volume of feature maps and weights decides how much data does the algorithm needs to process. The number of multiplications performed by Winograd convolution layer i is

$$M^i_W = \left\lceil \frac{H}{m} \right\rceil \cdot \left\lceil \frac{W}{m} \right\rceil \cdot C \cdot K \cdot l^2 \approx H \cdot W \cdot C \cdot K \cdot \left(\frac{l}{m} \right)^2$$

The number of additions involved in matrix multiplications is

$$S_W^i = \left\lceil \frac{H}{m} \right\rceil \cdot \left\lceil \frac{W}{m} \right\rceil \cdot (C-1) \cdot K \cdot l^2 \approx H \cdot W \cdot (C-1) \cdot K \cdot \left(\frac{l}{m} \right)^2$$

The number of additions required by Winograd transforms are S_B and S_A for $\left(B^T dB\right)$ and $\left(A^T \left[\mathcal{M}_{k,\tilde{x},\tilde{y}}\right] A\right)$ respectively. In most cases, Winograd transform matrices B and A are sparse, therefore, (5.9) and (5.10) utilize the operator $nnz\,(\cdot)$ (number of nonzeros).

$$S_B^i = 2 \times \left\lceil \frac{H}{m} \right\rceil \times \left\lceil \frac{W}{m} \right\rceil \times C \times K \times l \times [nnz\,(B) - l] \tag{5.9}$$

$$S_A^i = 2 \times \left\lceil \frac{H}{m} \right\rceil \times \left\lceil \frac{W}{m} \right\rceil \times C \times K \times l \times [nnz\,(A) - m] \tag{5.10}$$

The Winograd weights are pre-calculated and stored in memory, so the overhead of computing Winograd weights has not been taken into account.

5.5.1.3 Optimal Winograd transform and the corresponding "m"

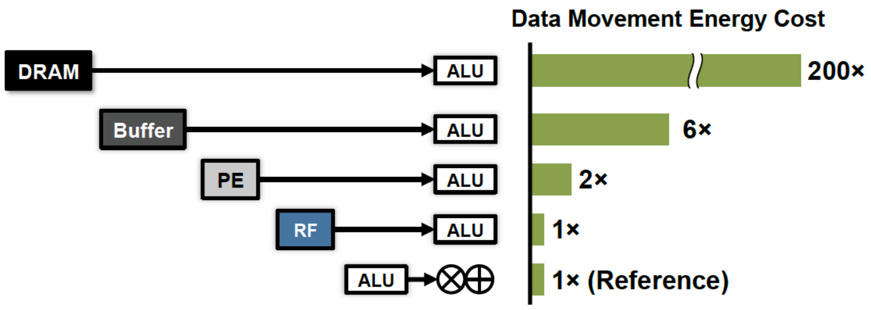


Figure 5.8: Data movement energy comparison among memory hierarchies [151]

When the value of r is specified, e.g. $r = 3$ for every layer of VGG, the value of m is crucial for determining both the power consumption and the arithmetic complexity. Furthermore, the calculation of the optimal power consumption is straightforward, whereas the optimal computation time is much more complicated to evaluate. Since the degree of parallelism and the memory access patterns are dynamic, these uncertain factors hinder accurate estimation of optimal computation time in an obvious mathematical analysis. Therefore, we focus on the analysis of achieving the optimal power consumption as the reference.

As shown in Fig. 5.8, the energy consumption for local (e.g. buffers, FIFOs) and external memory accesses are several times and orders of magnitude higher than arithmetic operations, respectively [151]. Let us assume for the sake of simplicity that every storage element in both local and external memory is accessed exactly once, transformed feature maps are stored in local memory after Winograd transform, and the Winograd weights are read from external memory.

Let E_{me} and E_{ml} be the unit energies consumed by an access to the external memory and an access to the local memory, respectively. Let E_{mul} and E_{add} be the unit energies consumed by a multiplication operation and an addition operation, respectively. Then the total energy consumption of layer i is

$$\begin{gathered} E^i_{tot} = E_{ml} \cdot \left(D^i_{wi} + D^i_{wo}\right) + E_{me} \cdot D^i_{wk} + \\ E_{mul} \cdot M^i_W + E_{add} \cdot \left(S^i_W + S^i_B + S^i_A\right) \end{gathered}$$

Another fact derived by eq. (5.6) and (5.8) is that greater m generates less elements of the transformed feature maps but more elements of the transformed weights. This fact indicates that the pruning of Winograd weights is more efficient with greater m.

After having given the above formulas and summarizations, we conduct the analysis and experiments in section 5.6.2.

5.6 Experimental Evaluation

VGG [146] is one of the most popular and mature deep learning models which has been widely used in research and industry. In this work, we use $VGG16$ for our analysis and experiments.

Table 5.1: number of parameters in each convolution layer of different stages in VGG [146] after Winograd transform (m=2)

Stage [146]	# of Winograd neurons	# of Winograd weights
Conv1 (×2)	12,845,056	65,536
Conv2 (×3)	6,422,528	262,144
Conv3 (×4)	3,211,264	1,048,576
Conv4 (×4)	1,605,632	4,194,304
Conv5 (×4)	401,408	4,194,304
Conv6	131,072	4,194,304

5.6.1 Experiment Setup

For the CNN model part, we set the input feature map size to $224 \times 224 \times 3$, which are standard input dimensions for VGG pipeline.

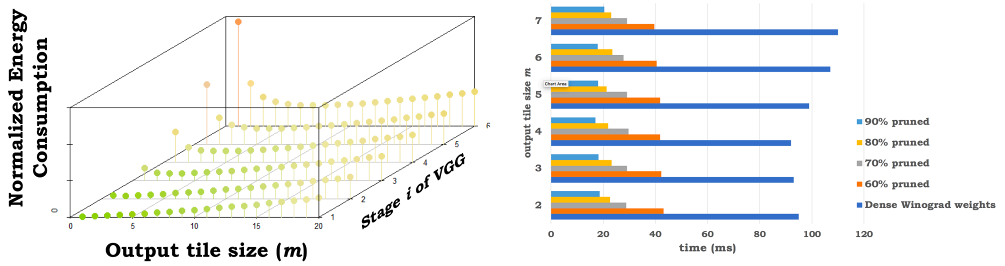
Table 5.1 shows the number of neurons and weights of each layer in different stages after the Winograd transform. For the hardware part, we implemented our design with RTL Verilog and Chisel [10]. We then evaluate our design on an FPGA board, Xilinx Virtex Ultrascale XCVU095. Although it is not fabricated with the lastest technologies, and equips only with a medium amount of DSPs (768 DSPs), this configuration reveals better the performance gain than the lastest FPGAs since optimizations for FPGAs with scarce computation power is more representative. Table 5.4 further demonstrates the resource consumption by each block.

5.6.2 Experiment on energy consumption analysis

Fig. 5.9 (a) plots the trend when different m is applied. The simulations run by synthesis tools show that the design with small values of m normally consume less energy. In order to simplify our design, we decide to use $m = 2$, which eventually affects the dimension of our systolic arrays, tiling size, memory access patterns of our accelerator design, and so

Table 5.2: Comparison with State-of-the-art implementations

Impl.	FPGA'15 [190]	FPGA'16 [189]	FPGA'16 [149]	DAC '17 [169]		our impl.	
FPGA	V7 VX485T	Xilinx VC709	Stratix-V GSD8	Arria10 GT1150		V-Ultra XCVU095	
Precision	32 bit float	16 bit fixed	8-16 bit fixed	32 bit float	8-16 bit fixed	8-16 bit fixed	
Frequency (MHz)	100	200	120	221.65	231.85	150	
Throughput (Gops/s)	61.6	354	47.5	460.5	1171.3	460.8/230.4 (8 bit/16 bit fixed)	921.6 (8 bit sparse fixed)
DSP utilization	1120/1400	2833/3632	727/1963	1340/1523	1500/3046	(512+128)/768	
Power efficiency (Gops/s/W)	3.31	14.22	1.84	25.78		55.9	



(a) The normalized energy consumption of each stage with different *m*

(b) The latency of inference with different *m* and sparsity

Figure 5.9: Energy consumption estimation and latency of Winograd convolution

on. Although the plot indicates that $m = 4$ might be the optimal value for the energy consumption, we are limited by other hardware resources in our FPGA system, but the situation might be different if designing with a different FPGA system. In Fig. 5.9 (b) we provide the latencies for the inference by VGG with different configuration of m and sparsity ranging from 60% to 90%. For the best case, we achieve almost 5× speedup.

Table 5.3: Resource usage

Resources	LUTs	FF	BRAM	DSP
Used	241,202	634,136	1,480	512 (arith.) + 128 (wino.)
Available[83]	537,600	1,057,200	1,728	640
Percentage	44.9%	60.8%	85.6%	67% + 16.7% = 83.7%

Table 5.4: Breakdown of resource

Module	Number	LUTs	FF	BRAM	DSP
SmartConnect.	2	2,973	3,913	0	0
rst_ps7_0_100M	1	38	80	0	0
processing_system7_0	1	224	0	0	0
axi_apb_bridge_0	1	178	288	0	0
index_dispatcher_0	1	384	576	0	0
wino_transform_wrapper_0	4	47,181	125,255	295	128
matmul_wrapper_0	8	188,724	501,023	1180	512

5.6.3 Results and analysis

With $m = 2$, we get the synthesized result with the resource usage as shown in Table 5.3. The end-to-end comparison with the state-of-art CNN FPPGA accelerators is listed in Table 5.2. We achieve the highest DSP usage and power efficiency. Since we are targeting the low-power FPGA for edge or smart devices, we only test our design on a medium scale FPGA. In current design, we use four 4×4 systolic arrays as one cluster for one matrix multiplication, and stack 8 such clusters for eight matrix multiplications in parallel. Meanwhile, 8 4×4 systolic arrays work on the Winograd transform, and forms the pipeline with the systolic arrays. In total, all 640 PEs are used.

5.7 Conclusion

In this work we propose a design with highly efficient recursive memory access layout for both dense and sparse Winograd convolution, unified systolic arrays for both Winograd transform and matrix multiplication, and three dimensional compute engine for Winograd convolution. We also provide a comprehensive algorithmic level analysis for the performance model of Winograd convolution. We achieve high computation power utilization and high power efficiency in our design. There are several aspects that we can investigate further in the future. In particular, the automation design flow will help a lot to reduce the burden of development. And the processing in memory technology is also a promising solution, as more and more new FPGA architecture incorporate such kind of brilliant concept.

CHAPTER 6

HUGE^2: a Highly Untangled Generative-model Engine for Edge-computing

This chapter proposes an approach to tackle the complexity of the deconvolution operations, as shown in Figure 6.1. Deconvolution, possessing particular characteristics compared to standard convolution, is a crucial component of the generative model, especially for generative adversarial networks (GAN), variational autoencoder (VAE). We observe the regularity of the data access pattern to the input data and kernels of the deconvolution operations, and we decompose the input tensor and weight tensor into certain regular repeated combinations. The standard convolution operator then operates on each of those combinations. This method can be easily deployed on generic hardware platforms, e.g., CPU, GPU, FPGA, even novel accelerators.

6.1 Introduction

Recently, the deep generative models and semantic segmentation algorithms have shown their stunning abilities in various fields, such as creating realistic images from the learned distribution of a given dataset, providing the robots with the ability to learn from environment without human input, generating the synthetic 3D objects for the scene parsing in a scenario, and so forth. These creative deep learning models attract great interests in research by both scholar and industry. The representative works include the Generative Neural Networks (GAN) [64], the Variational Auto-encoder (VAE) [94], and the semantic image segmentation

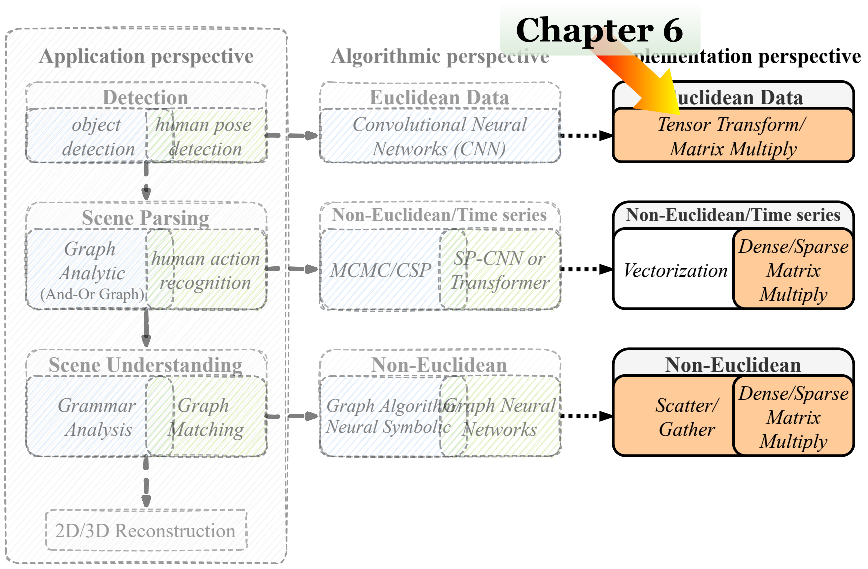


Figure 6.1: The position of this chapter in the platform

algorithms [137, 22].

However, the generative models and semantic segmentation algorithms rely heavily on the *deconvolution* which is an inefficient, and both computation- and memory-intensive operation. The inefficiency comes either from the zero insertions in either input tensor or kernels or from repeatedly accesses to the overlapped regions. Zero insertions cause wasteful computations, hence high latency. The non-consecutive memory access manner in deconvolutions also hurts system performance drastically. The overlapped region in outputs hinders the concurrent processing because the chained memory-writings happen to the same location.

In this work, we conceive a set of solutions from an algorithmic perspective to improve the performance of deconvolution for embedded systems. And the experiments show that we achieve the speedup nearly 10× on GPUs and 5× on CPU, the memory storage and their accesses are reduced by more than 50 percent.

6.2 Background

6.2.1 Deconvolution

The so-called **deconvolution** used in the deconvolution layers of the *generative adversarial networks* and the *semantic image segmentation* is actually not as exact as the reverse operation of the *convolution.* Actually, the deconvolution layers are learnable up-sampling layers. Two categories of special convolution operations can fulfill such kind of task, they are the *transposed convolution* and the *dilated convolution*, respectively. The following subsections give details about how these operations work.

6.2.1.1 Transposed Convolution

Transposed convolution, also called *Fractionally-Strided Convolution*, is used not only to upsample an initial layer but also to create new features in enlarged output feature maps. Theoretically, transposed convolution works as a process of swapping the Forward and backward passes of a convolution, and this is where its name comes from. Algorithm 3 describes how this kind of convolution works. As it shows, when s_m and s_n are bigger than 1, the kernels slide on the feature maps with fractional steps. Figure 6.2 shows the implementations of the transposed convolution and its counterpart.

It is always possible to emulate a transposed convolution with a direct convolution. Such process first spreads the input feature map by inserting zeros (or blank lines) between each pair of rows and columns. The original input tensor I now becomes $\hat{I}$. It then applies a standard convolution, with strides of 1 (equivalent to sliding step of $\frac{1}{stride}$ on the original input [43]), on the resulting input representation, as shown in Algorithm 3. Let us take left hand side of Figure 6.3 as an example. It illustrates a $2D$ transposed convolution of a zero-inserted feature map of size 6×6 and a 3×3 transposed kernel.

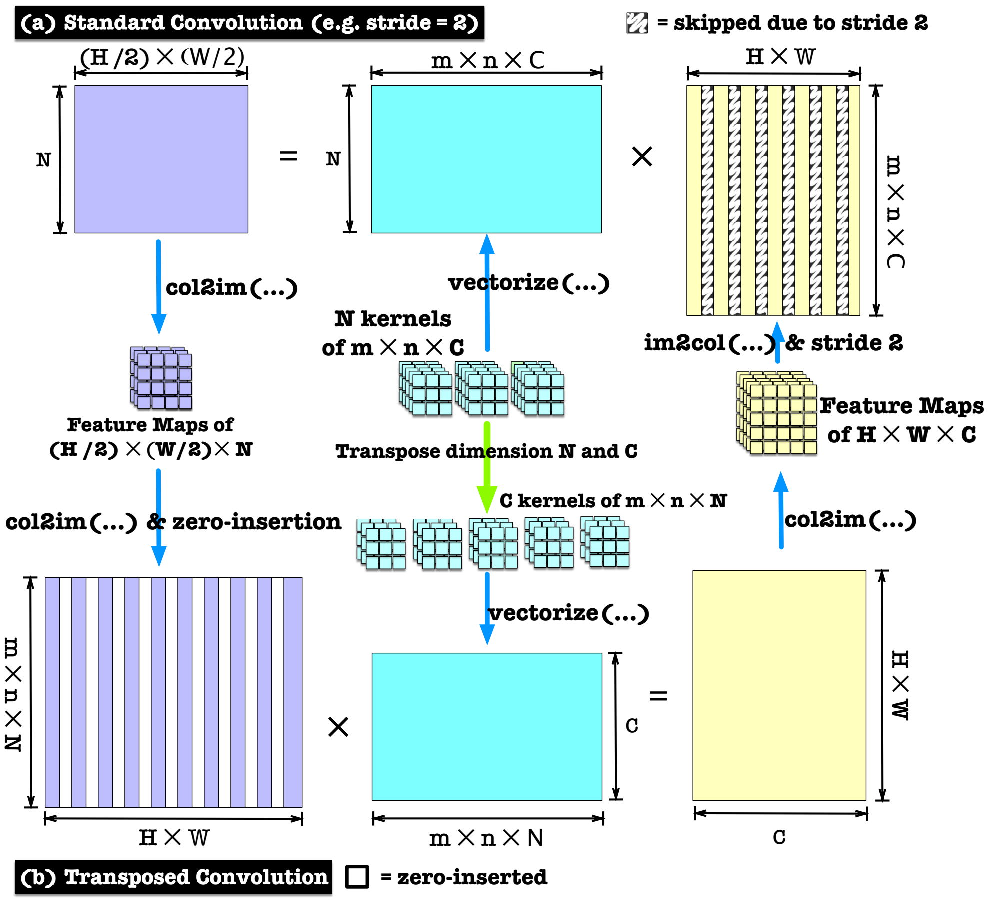

(a) Standard Convolution (e.g. stride = 2)
▨ = skipped due to stride 2
(H/2) × (W/2)
N
=
N
m × n × C
×
H × W
m × n × C
col2im(...)
vectorize(...)
N kernels of m × n × C
im2col(...) & stride 2
Feature Maps of (H/2) × (W/2) × N
Transpose dimension N and C
Feature Maps of H × W × C
C kernels of m × n × N
col2im(...) & zero-insertion
col2im(...)
vectorize(...)
m × n × N
×
C
=
H × W
H × W
m × n × N
C
(b) Transposed Convolution
□ = zero-inserted

Figure 6.2: The upper is the implementation of a strided convolution, and the bottom is its related transposed convolution.

6.2.1.2 Dilated Convolution

Strictly speaking, despite of that dilated convolution, also known as *atrous convolution*, is not acknowledged as a kind of deconvolution, it has been widely explored to upsample input tensors in the semantic image segmentation algorithms. Moreover, it shares some characteristics on which we can apply our acceleration algorithm as it does for the transposed convolution. On the contrary to transposed convolution, dilated convolution inserts zeros into kernels but not input tensors. The kernels are dilated so as to enlarge their corresponding receptive fields. One thing needs to be noticed is that only stride bigger than 1 has the effect

Algorithm 3 Transposed Convolution

```
function CONV2DTRANSPOSE($I, K, O, s_m, s_n$)    ▷ $s_m$, $s_n$ are fraction factor also zero-insertion stride on input tensors
    for $0 \leq k \leq N - 1$ do
        for $0 \leq h \leq H - R + 1$ do
            for $0 \leq w \leq W - S + 1$ do
                for $0 \leq c \leq C - 1$ do
                    for $0 \leq m \leq R - 1, m = m + s_m$ do
                        for $0 \leq n \leq S - 1, n = n + s_n$ do
                            $O[h, w, k] += I[h + \frac{m}{s_m}, w + \frac{n}{s_n}, c] \times K[h \bmod s_m + m, w \bmod s_n + k_n, c, k]$
```

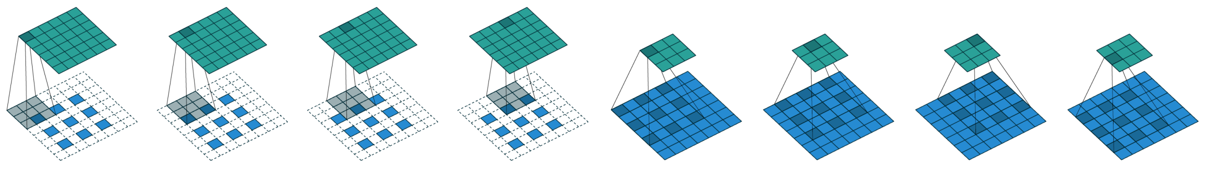


Figure 6.3: left: the transposed convolution of a 3×3 kernel over a 6×6 input padded with a 1×1 border of zeros using 2×2 strides; right: dilated convolution of a 3×3 kernel over a 7×7 input with a kernel of the dilation factor of 2 [43].

of upsampling on input tensors. Details are demonstrated in Algorithm 4 and right side of Figure 6.3.

6.2.2 Previous Work

To our best knowledge, up to this work, most of optimized solutions have been proposed are from the research of the hardware accelerator realm. These designated designs achieve much higher throughput compared with non-optimized generic hardware. Hence, our goal is to conceive an easily-accessible and cost-efficient solution for the generic hardware.

Algorithm 4 Dilated Convolution

```
function Conv2DDilated($I, K, O, s_m, s_n$)    ▷ $s_m$, $s_n$ are dilation factors also zero-insertion stride on kernels
    for $0 \leq k \leq N - 1$ do
        for $0 \leq h \leq H - R + 1$ do
            for $0 \leq w \leq W - S + 1$ do
                for $0 \leq c \leq C - 1$ do
                    for $0 \leq m \leq R - 1$ do
                        for $0 \leq n \leq S - 1$ do
                            $O[h, w, k] \mathrel{+}= I[h + s_m \times m, w + s_n \times n, c] \times K[m, n, c, k]$
```

1. **Zero-Skipping**: [182, 181] present a set of designs by swapping zero rows and columns with non-zeros ones, and then rearranging non-zero rows and columns into effective working groups. However, this design doesn't thoroughly solve the unbalanced working load problem among effective computation groups. [148] discovers the delicate mathematical relation of indices among input tensors, kernels, and output tensors. These relations help rearrange the computations to skip zeros. But this method lacks memory access coalescing. Therefore, input tensors and kernels are accessed in a non-consecutive fashion with degradation in the overall performance of the system.

2. **Reverse Looping and Overlapping**: *Reverse Looping* is introduced by both [193] and [176]. This technique avoids accessing the output tensors in an overlapped manner with more operations, especially the accumulations and memory writings. Reverse looping, on the contrary, uses the output space to determine corresponding input blocks, and thus eliminating the need for the additional accumulations and memory accesses. However, the overlapped regions are not evenly distributed, hence the work load unbalancing issue among processing elements is still not well solved by such kind of solution.

6.3 Algorithm

In this section, we introduce our algorithm for accelerating the deconvolution operations. Our algorithm consists of three steps: 1) kernel decomposition, 2) untangling of kernels and matrix multiplications, and 3) dispatching and combining the results to the output tensor.

The following subsections provide the explanation for each of them.

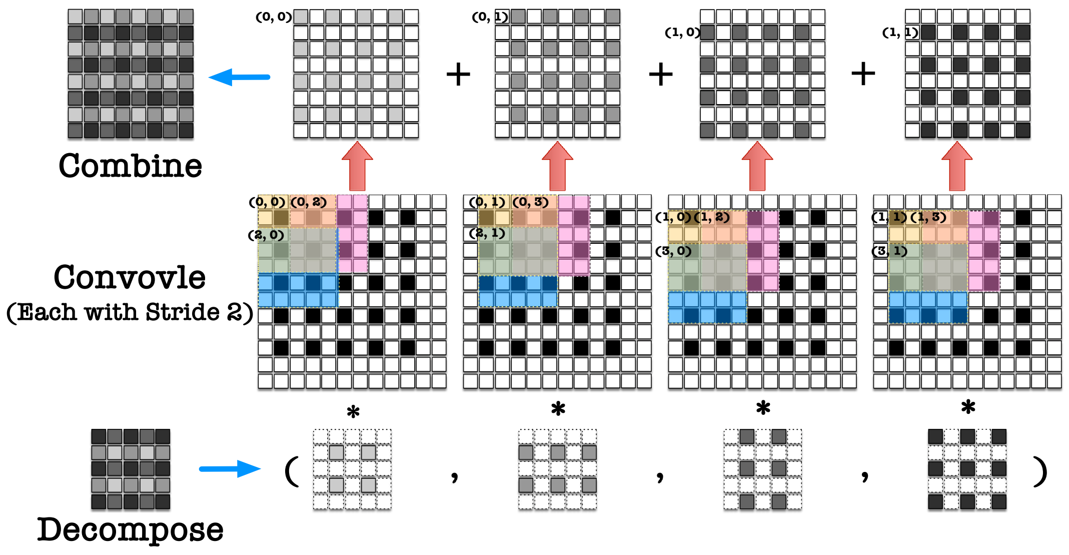


Figure 6.4: the kernel of the transpose convolution is decomposed into 4 patterns, each convolves with zero-inserted feature maps with stride 2, the final result is obtained by combining the 4 partial feature maps; the yellow, pink, and blue patches correspond to sliding windows at different positions.

6.3.1 Decompose Deconvolution

Given an input tensor with stride 2 zero-inserted as example, and let us take a transposed kernel to slide on it. We discover that there exists 4 kinds of patterns as shown at the bottom of Figure 6.4 where the nonzero elements in the kernel meet the nonzero elements in the zero-inserted input tensor $\hat{I}$, and thus generate non-overlapped effective outputs as shown on the top of Figure 6.4. Mathematical description of these 4 patterns is given below:

Pattern 1: *odd columns* and *odd rows* of kernel convolve with stride 2 on input tensor $\hat{I}$ and generate *even columns* and *even rows* of output tensor O.

$$O[2h, 2w, k] = \sum_{c=0}^{C} \sum_{m=0}^{R} \sum_{n=0}^{S} K[2m+1, 2n+1, k, c] \times \hat{I}[2h+2m, 2w+2n, c] \tag{6.1}$$

Pattern 2: *even columns* and *odd rows* of kernel convolve with input tensor $\hat{I}$ and generate *odd columns* and *even rows* of output tensor O.

$$O[2h, 2w+1, k] = \sum_{c=0}^{C} \sum_{m=0}^{R} \sum_{n=0}^{S} K[2m+1, 2n, k, c] \times \hat{I}[2h+2m, 2w+2n+1, c] \tag{6.2}$$

Pattern 3: *odd columns* and *even rows* of kernel convolve with input tensor $\hat{I}$ and generate *even columns* and *odd rows* of output tensor.

$$O[2h+1, 2w, k] = \sum_{c=0}^{C} \sum_{m=0}^{R} \sum_{n=0}^{S} K[2m, 2n+1, k, c] \times \hat{I}[2h+2m+1, 2w+2n, c] \tag{6.3}$$

Pattern 4: *even columns* and *even rows* of kernel convolve with input tensor $\hat{I}$ and generate *odd columns* and *odd rows* of output tensor.

$$O[2h+1, 2w+1, k] = \sum_{c=0}^{C} \sum_{m=0}^{R} \sum_{n=0}^{S} K[2m, 2n, k, c] \times \hat{I}[2h+2m+1, 2w+2n+1, c] \tag{6.4}$$

One benefit of such decomposition is that the non-overlapped sparse regions on the output tensor do not cause any race conditions for writing in memory, since it will not be blocked by consecutive memory writings.

After having investigated the relation between indices of the nonzeros in input tensor and the decomposed kernels, we draw the conclusion that we can safely remove all the zero inserted in both input tensor and the decomposed kernels as shown in Figure 6.5. The flow demonstrates the zero-removal for all patterns where they become 4 smaller standard convolutions on input tensors without zero-insertion. Then we scatter and combine their results. The scattering of the results from each pattern follows the corresponding indices used in the zero-inserted version.

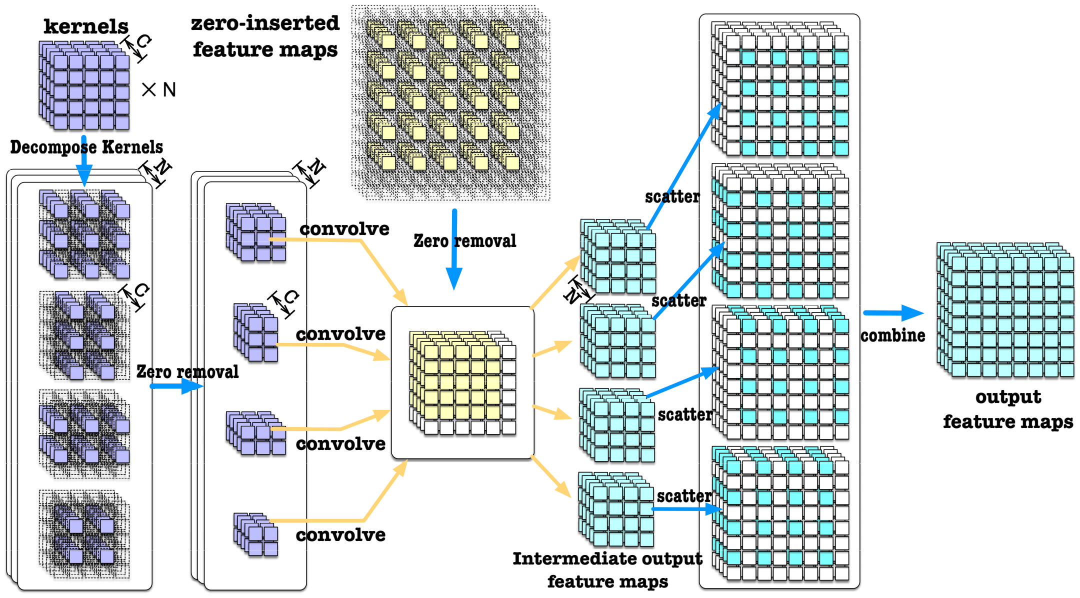


Figure 6.5: The corresponding flow of the example; bottom:

And the simplified flow in the Figure 6.6 resembles the Inception module of GoogLeNet [152] except that the last step here is scattering and combination instead of stacking.

6.3.2 Untangling

To further improve the parallelism of arithmetic computations, we propose an algorithm to untangle every decomposed pattern of the transposed convolution into a set of 1×1 convolutions.

6.3.2.1 Untangle standard convolution

As shown in the right side of Figure 6.5 each pattern convolves with input tensor as a standard convolution. To better understand our algorithm, let us take pattern 4 (the 3×3 one) as an example. The process is shown in Figure 6.7. Given N decomposed kernels of pattern 4 (with zero removed) from previous subsection and the original input tensor, each kernel has dimension of $m \times n \times C$ and the input tensor has dimension $H \times W \times C$. We regroup the

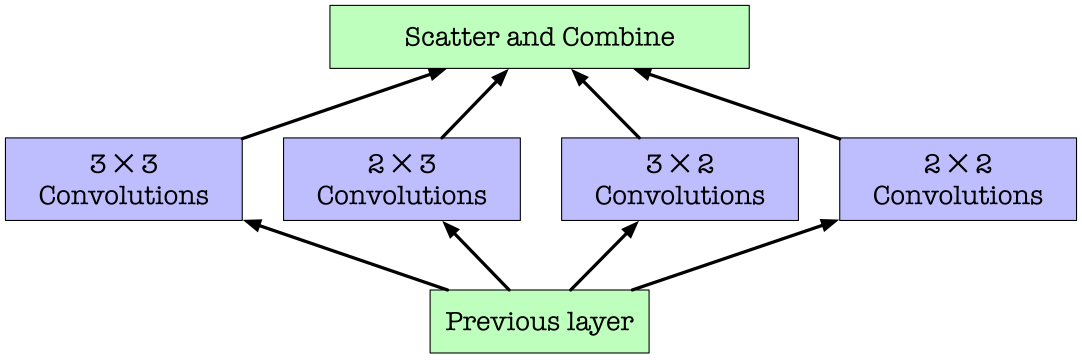


Figure 6.6: A simplified view

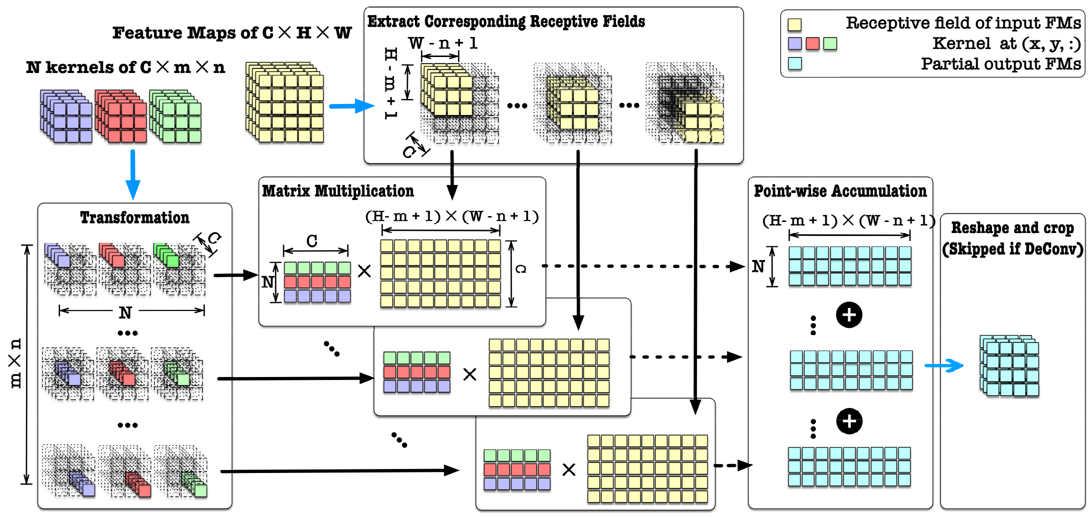


Figure 6.7: untangle a standard convolution into a set of 1×1 convolutions

elements of the kernels by gathering N columns along the dimension C from every kernel at position (x, y) (e.g. $(0, 0)$ at top left case, $(0, 2)$ for the bottom right case in the Figure 6.7) of the (m, n) plane. These columns form a matrix of dimension $N \times C$. Then, their corresponding receptive fields on the input tensor can be fetched for input tensor to form another matrix of dimension $(H - m + 1)(W - n + 1) \times C$. Such configuration can be regarded as a 1×1 convolution with N 1×1 kernels working with a cropped tensor. The products of the $m \times n$ matrix multiplications are then accumulated together. The elements of the resultant matrix are then dispatched to the corresponding position in the output tensor.

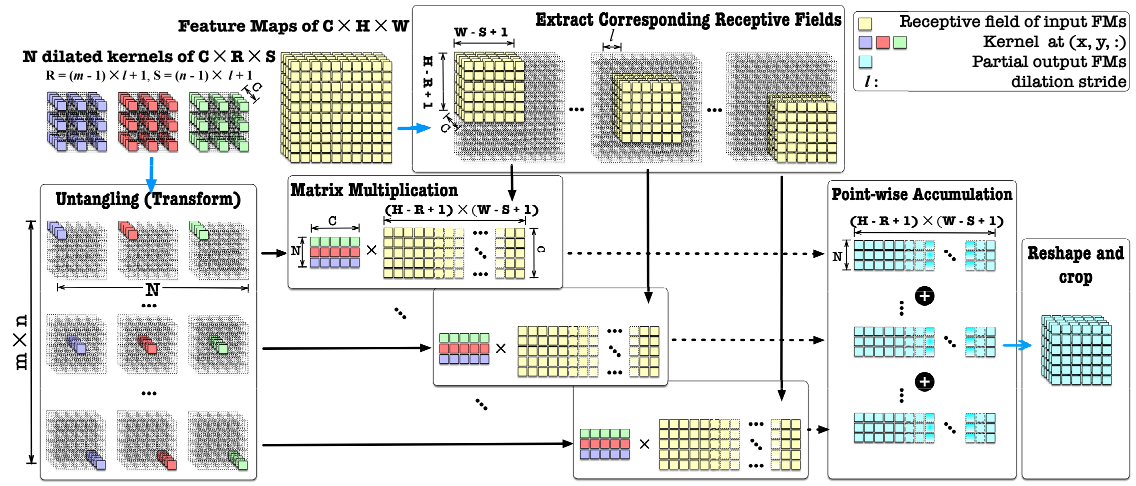

Feature Maps of C × H × W
N dilated kernels of C × R × S
R = (m - 1) × l + 1, S = (n - 1) × l + 1
Extract Corresponding Receptive Fields
W - S + 1
H - R + 1
Receptive field of input FMs
Kernel at (x, y, :)
Partial output FMs
l: dilation stride
Untangling (Transform)
m × n
N
Matrix Multiplication
(H - R + 1) × (W - S + 1)
Point-wise Accumulation
Reshape and crop

Figure 6.8: Untangle a dilated convolution

6.3.2.2 Untangle dilated convolution

Dilated convolution can also take the advantage of untangling. As it shows in left side of Figure 6.8 untangling technique is also applicable. The sliding step on input tensor is larger, and the receptive field shrink with multiple of stride.

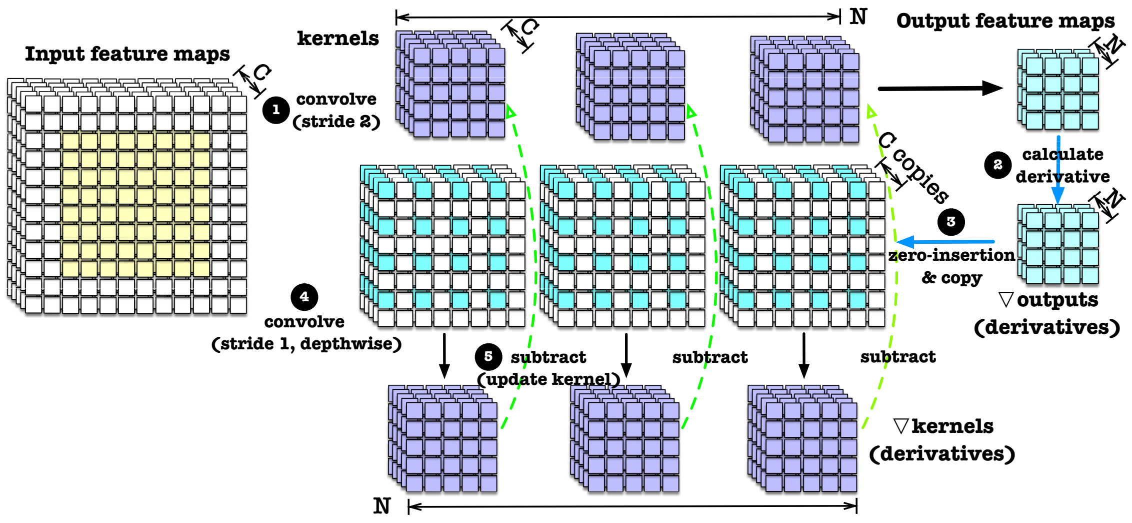


Figure 6.9: Training of discriminator with dilated convolutions

6.3.2.3 Training of GAN

The back-propagation of the discriminator of GAN can be seen as a special case of dilated convolution. Step 3 of right side of Figure 6.9 depicts that, in order to propagate the derivatives of the output errors, there are $C \times N$ convolutions by C input feature maps and N derivative maps. Each derivative map is dilated (since discriminator uses the strided convolution) and convolves with each input feature map.

Therefore, we can make C copies of each of N derivative maps from output errors to form N new dilated kernels of C channels. Then the dilated kernels convolve with input tensor to form the derivates of kernels and the results are subtracted from corresponding kernels.

The dilated convolution in step 4 of right side of Figure 6.8 is actually a depth-wise version. Hence, it corresponds to $C = 1$ in left side of Figure 6.8 which is seen as a outer-product of two vectors.

The back-propagation of generator of GAN can be seen as a strided convolution of derivative maps of output errors and input tensor (not shown in this paper).

6.4 Experimental Results

This section provides the evaluation of our algorithms. We use the deconvolution layers of DCGAN [124] and cGAN [112] as case study. Their configurations are shown in Table 6.1. In this paper, we mainly focus on the inference phase of deconvolution layers, and all models are pretrained with CIFAR100 [97] dataset. The experiments for GAN's training are only investigated at several typical layers.

The baseline of library we pick up is DarkNet [129] since it is open-sourced, and the commercial library such as cuDNN [25] only delivers with binary code. The system used in our experiments equips with embedded CPU, 4-core ARM Cortex-A57, and a Nvidia's GPU (256-core NVIDIA Pascal™ Embedded GPU). The experiments are run on both embedded

Table 6.1: Configuration of deconvolution layers

GAN	Layer	Input	Kernel	Stride
DCGAN	DC1	$4 \times 4 \times 1024$	$5 \times 5 \times 1024, 512$	2×2
	DC2	$8 \times 8 \times 512$	$5 \times 5 \times 512, 256$	2×2
	DC3	$16 \times 16 \times 256$	$5 \times 5 \times 256, 128$	2×2
	DC4	$32 \times 32 \times 128$	$5 \times 5 \times 128, 3$	2×2
CGAN	DC1	$8 \times 8 \times 256$	$4 \times 4 \times 256, 128$	2×2
	DC2	$16 \times 16 \times 128$	$4 \times 4 \times 128, 3$	2×2

CPU and embedded GPU. The metrics used for performance comparison includes the speedup and memory access reduction. We compared our implementations of transposed convolution and dilated convolution with the baseline, which is the naive implementations from DarkNet for both CPU and GPU. Most 2D standard and transpose convolution implementation in modern deep learning library are based on **im2col**.

6.4.1 Speedup of Computation

The speedup is obtained by the comparison of the computational runtime with the baseline. Figure 6.10 demonstrates the speedup gained by applying kernel decomposition and untangling for DCGAN and cGAN, respectively.

From the results, we can see that the shallower deconvolution layer are more compute-bounded since they have more kernels which deconvolve with input tensor and require much more computational operations. Untangling transposed kernels can efficiently improve the parallelism by taking advantage of larger C and N.

6.4.2 Reduction of Memory Access

The results of experiments for the memory access reduction by decomposition and untangling are provided in Figure 6.11.

One more thing we want to mention is that untangling technique we applied favors the $C \times N \times R \times S$ memory layout for the transposed kernels and $C \times H \times W$ for the input tensor. This is because elements along C and N dimensions are stored consecutively in these layouts, and this helps with the data fetching in coalescing memory access pattern.

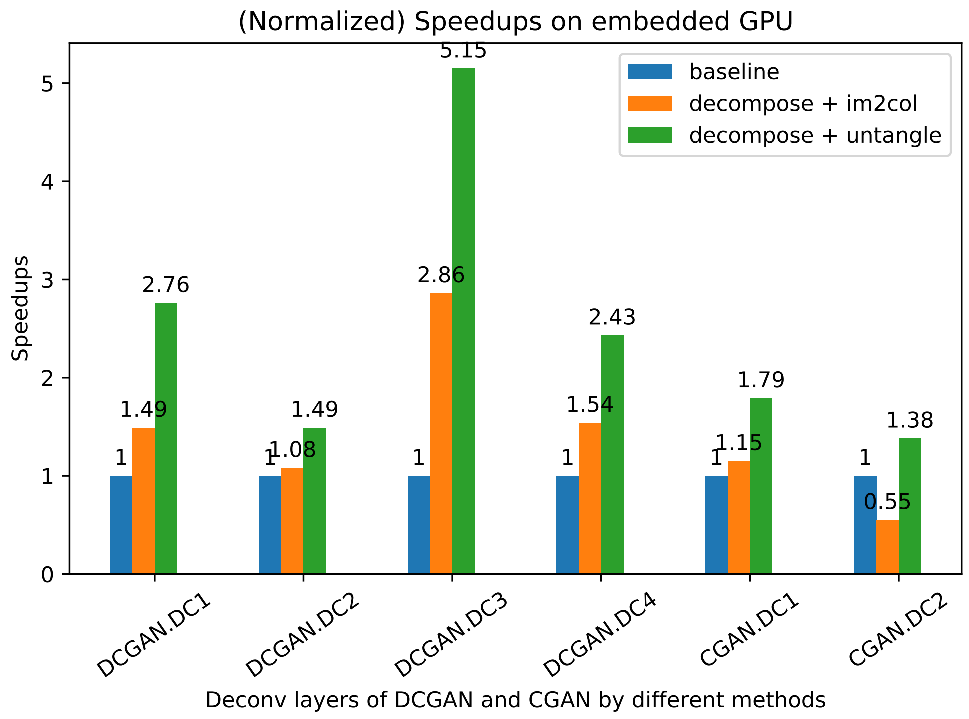
As shown in Figure 6.11, it is obvious that the deeper deconvolution layers are data-bounded, the reduction can be obtained more on the deeper layers since the output tensor becomes larger by the unsampling effects. We achieve a memory access reduction around 30% to 70% by only applying untangling technique.

6.4.2.1 Speedup in GAN training

The right side of Figure 6.11 plots the speedup of training of GANs. We select several typical layers for the experiments, we want to cover both the cases for dilated derivative maps convolving input tensor and derivative maps stridedly convolving input tensors.

6.5 Conclusion

In this paper we presented a set of efficient algorithms and optimizations for deconvolutions, these algorithms are the core components in our deep generative model engine "*HUGE*". We devised them as pervasive as possible to fit on most hardware platforms. *HUGE* really accomplishes the outstanding results for our applications. It shows great improvements in two crucial aspects, computation loads and memory access, respectively.



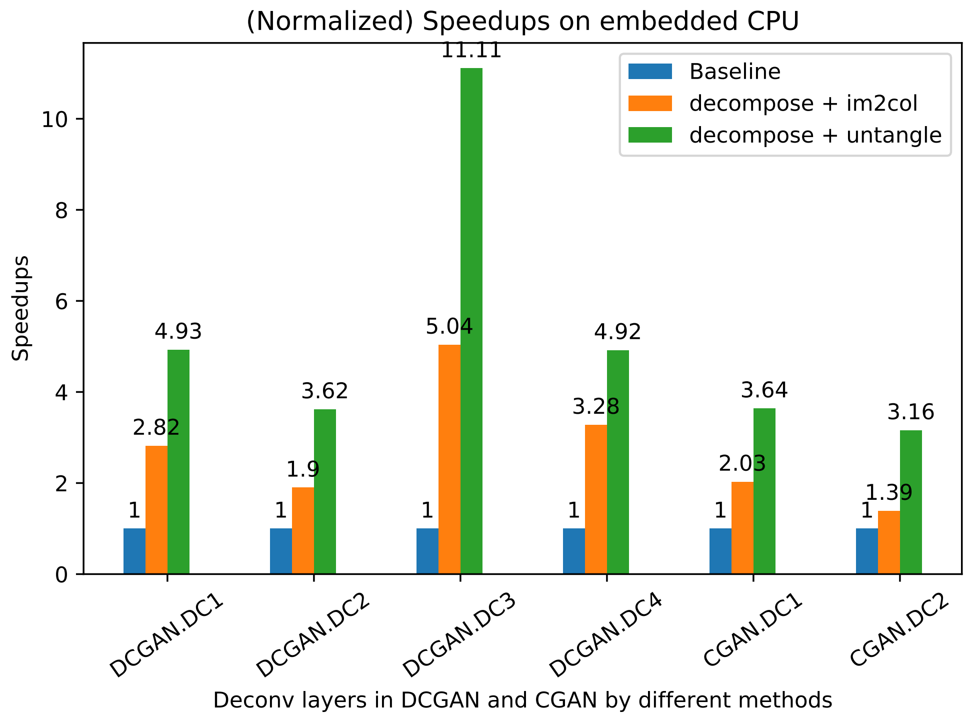
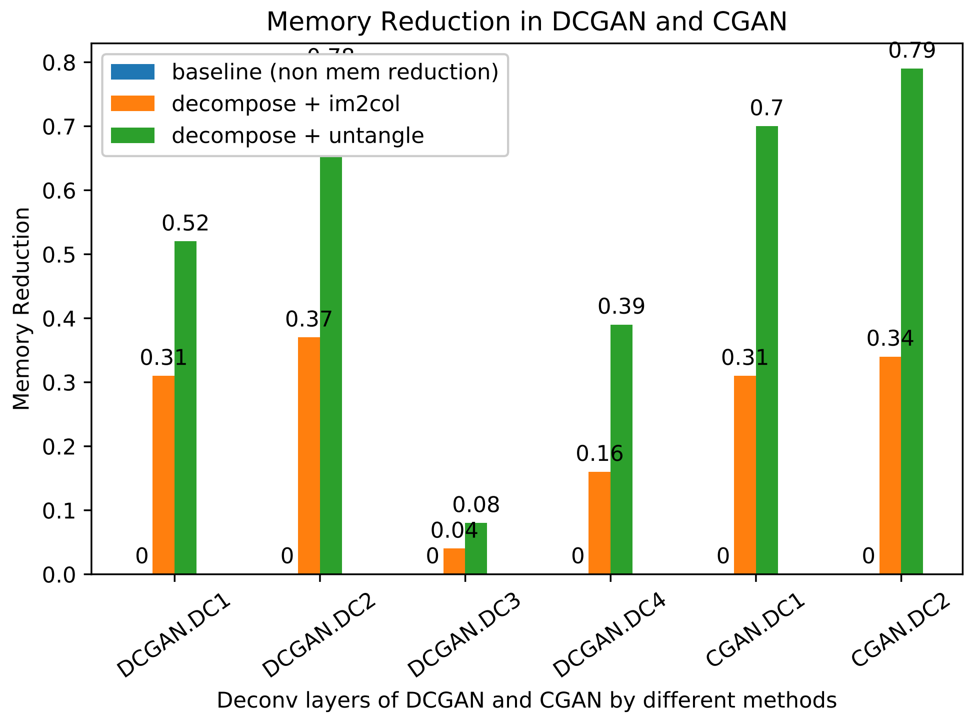


Figure 6.10: The speedups of the inference of GANs on embedded system. Top: on CPU (4core Cortex-A57) of the same board; bottom: on embedded GPU of Jetson TX2



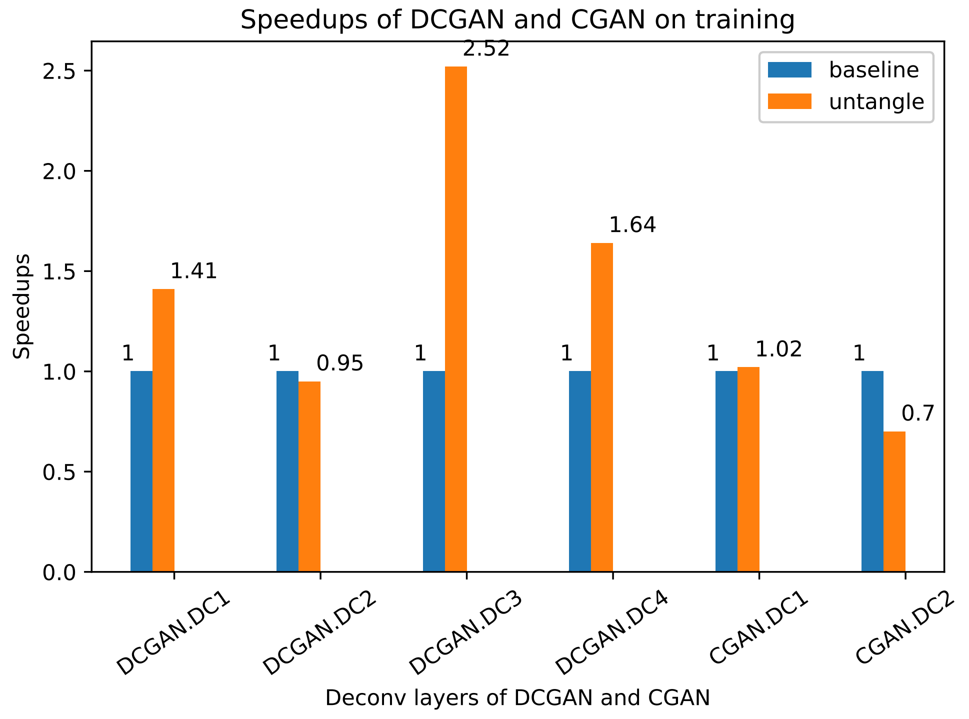


Figure 6.11: Top: the memory access reduction for GANs; bottom: the speedup of training GANs

CHAPTER 7

VersaGNN: a Versatile accelerator for Graph Neural Networks

This chapter focuses on designing a versatile accelerator for both sparse and dense matrix multiplications. This module is for the acceleration of graph analytic, as shown Figure 7.1. The Graph Neural Network (GNNs) is a very suitable use case for this accelerator since the two phases, feature transform and feature aggregation, of a graph convolution layer are dense and sparse matrix multiplications, respectively. The hardware algorithm employed in this accelerator design can be regarded as an extended version of the one in Chapter 5. Based on the recursive matrix multiplication fashion used in Chapter 5, we further decrease the number of complex arithmetic operations, i.e., multiplications, through Strassen's algorithm for the dense case; for the sparse case, we propose novel FIFOs design for storing the index of nonzero during the processing. Moreover, a software approach algorithm for regrouping the sparse matrices reduces the idled time slots during index alignment.

7.1 Introduction

Graph Neural Networks (GNNs) have achieved state-of-the-art performances in node classification [96, 69], link prediction [95, 191], graph classification [183], graph generation [95, 163], and clustering [162, 183] on arbitrarily structured graphs. The power of representation learning on graphs comes from feature embedding, which includes extracting structured, low dimensional features from unstructured, high dimensional graphs.

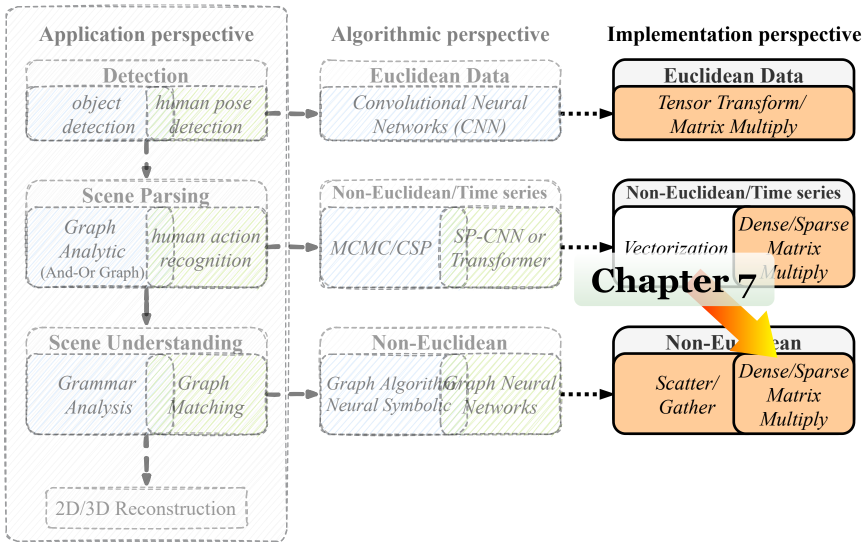


Figure 7.1: The position of this chapter in the platform

On one hand, traditional *Convolutional Neural Networks* (CNNs) [101] that operate on Euclidean data are characterized by local connections and shared weights, and are able to extract multi-scale localized spatial features. Euclidean data, such as images, can be represented as regular grids in the Euclidean space. Thus, a CNN is able to exploit the shift-invariance and local connectivity of Euclidean data. On the other hand, GNNs inherit the irregular computing patterns and processing dataflow of graph analytics, resulting in the inefficient use of CPUs and GPUs.

Meanwhile, the majority of real-word graphs for GNNs follow the Power-Law distribution - the number of nodes with degree N is proportional to N^{α} for some constant α [60]. Thus, a minority of nodes share high degrees, leading to remarkably unbalanced adjacency matrices. Therefore, hardware architectures must adapt to the varying shapes of the high-dimensional convolutions in GNN. Generally, two primary execution phases, **Aggregation** and **Transformation**, occupy the most execution time [52, 69, 174, 179].

Aggregation (Message Passing) Phase: GNN follows a neighborhood aggregation strategy in which each node's representation is updated by aggregating features of its adjacencies. Common aggregation strategies include **Sum**, **Average**, **Mean**, and **Max**. After N iterations of aggregation, a node's representation captures the structural information within its N-hop network neighborhood. The main computation kernels are data loading, which collects feature vectors that are indexed by the neighboring node's addresses, and the execution of aggregation. Due to their sparsity, the feature vectors can be efficiently fetched by coalescing their elements. Within the feature matrix, the feature vectors of adjacent nodes can be separated by strides. Poor use of these inter-vertex data parallelism can incur significant cache misses and address calculations.

Transformation (Node Encoding) Phase: This phase is usually expressed as a Multi-layer Perceptron (MLP) that transforms node feature vectors to lower dimensional embeddings using Matrix-Vector Multiplication (MVM). The matrix multiplication can be either dense or sparse, depending on the sparsity of feature matrices at different convolutional layers. A non-linear activation function is applied to each vertex to yield the outputs. This phase is characterized by regular computational graph and homogeneous data access patterns.

To accelerate GNN-based applications that harness these two distinct stages and process highly variable real-world graphs, we propose several optimization approaches for accelerating GNN with a software/hardware co-design paradigm. Our main contributions are:

- We unify the processing of Aggregation and Transformation stages using a single processor that is competent in handling the irregular data access patterns and the hybrid computing mode of GNNs.
- We apply the classic Strassen's algorithm to accelerating dense matrix multiplication, significantly reducing the number of costly multiplications.
- We exploit an ultra-efficient, greedy-based load-balancing approach that achieves considerable speedup in sparse-dense multiplication.
- We propose *VersaGNN*, a high-throughput and memory-efficient Graph Neural Network accelerator based on the well-known systolic array design.

- We implement our architecture design using Chisel HDL [10], and test our *VersaGNN* using four well-known GNN models on six benchmark graph datasets. Compared to the state-of-the-art software frameworks, our work achieves on average 3712× speedup with 1301.25× energy reduction on CPU, and 35.4× speedup with 17.66× energy reduction on GPU, respectively.

7.2 Background

In this section, we review the core concepts of Graph Neural Networks. Table 7.1 lists the notations of GNNs used throughout the paper.

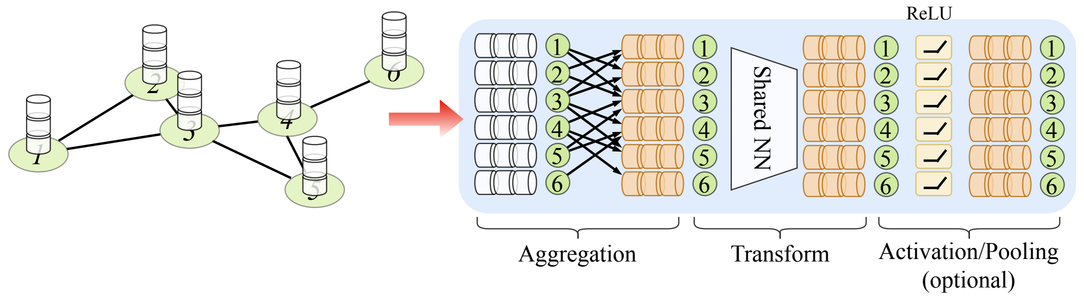
Table 7.1: Notations of GNN

Notation	***Description***	***Notation***	***Description***
$\mathbf{G}$	graph $\mathbf{G} = (V, E, H_0)$	V	nodes/vertices of G
E	edges of G	deg(v)	degree of node v
$e_{i,j}$	edge between node i and j	*N(v)*	neighbors of node v
$\Theta(\cdot)$	Neural Networks	$a_v^{(i)}$	aggregated feature vector of v at layer i
A, $A_{i,j}$	adjacency matrix, element at (i, j)	$W^{(i)}$	weight matrix of i^{th} layer
$h_v^{(i)}$ r	hidden feature vector of node v	$b^{(i)}$	bias of i^{th} layer
H_0	initial state of *feature matrix*	$\sigma(.)$	activation function

7.2.1 Graph Convolutions

Graph Neural Networks utilize the *message passing* mechanism [196, 174] for graph node embedding, usually generated by *graph convolutions*. A classic GNN is constructed with a stack of two or three graph convolution layers, whose structure is illustrated in Fig.7.2. A graph convolution layer takes the feature matrix as the input, arranged by graph node signals, and performs convolution on the feature matrix, followed by one or two optional nonlinear operators, such as nonlinear activation (e.g ReLU, LeakyReLU) and pooling.

Convolutional GNNs can be categorized as spectral-based GNNs or spatial-based GNNs [174]. Spectral-based approaches such as [96, 75, 36] define graph convolutions by adopting filters from the perspective of graph signal processing [144]. The graph convolution operations are interpreted as removing noises from graph signals. Spatial-based approaches, including [5, 61, 114], exploits the information propagation paradigm and aims at collecting features of each node from its K-hop neighbors.



(a) A graph with stack of signals on nodes (b) A convolution layer of a toy Graph Neural Network

Figure 7.2: For the graph in (a), (b) shows how the node signals are passing through the graph convolution operator

The *convolution operator* in a graph convolution layer consists of two consecutive phases, the **Aggregation** (or **Message Passing**) phase, and the **Transformation** (or **Node Encoding**) phase. Equation (7.1) gives a mathematical formulation of the message-passing network [52],

$$\mathbf{h}_i^{(k)} = \Theta_i^{(k)} \left(\mathbf{h}_i^{(k-1)}, \Xi_{j \in N(i)} \, \Theta_j^{(k)} \left(\mathbf{h}_i^{(k-1)}, \mathbf{h}_j^{(k-1)}, \mathbf{e}_{j,i} \right) \right) \tag{7.1}$$

, where Ξ embodies a differentiable and permutation-invariant aggregation function, e.g., **Sum**, **Mean**, or **Max**. Θ_i and Θ_j indicate neural networks or linear transformations. In the following subsections, we discuss some state-of-the-art models and their aggregation strategies.

7.2.2 Direct Aggregation

GraphSAGE [69] exploits vertex features such as text attributes and node degrees to learn an embedding function, formulated as:

$$\begin{aligned} \hat{\mathbf{h}}_i &= \Theta\left(\boxed{\mathbf{mean}_{j \in N(i) \cup \{i\}}} \, (\mathbf{h}_j) \right) \\ \mathbf{h}'_i &= \frac{\hat{\mathbf{h}}_i}{\| \hat{\mathbf{h}}_i \|_2} \end{aligned} \tag{7.2}$$

where the feature aggregation is the *mean* operator marked in Equation (7.2). Instead of training individual node-wise embedding vectors, **GraphSAGE** generates embeddings through uniform sampling and feature aggregation from the node's neighbors, thus taking into account both the node's own features as well as its neighboring features, and balance execution overhead with accuracy.

GIN [177] utilizes the Graph Isomorphism Operator to enhance the representation power of GNNs. For each graph node, **GIN** recursively aggregates and transforms representation vectors of its adjacent nodes. With its high expressive power, **GIN** is able to capture the structural information of both large and small graphs.

$$\mathbf{h}'_i = \Theta\left((1 + \epsilon) \cdot \mathbf{h}_i + \boxed{\sum_{j \in N(i)}} \mathbf{h}_j \right) \tag{7.3}$$

ϵ is a learnable parameter that improves the node's self-confidence, and Θ represents a Multi-Layer Perceptron (MLP). **GIN** uses summation as its aggregation operator, which can represent universal functions over multisets, as marked in Equation (7.3).

7.2.3 Weighted Aggregation

GCN [96] is composed of two or three convolutional layers with residual connections.

$$\mathbf{H}' = \hat{\mathbf{D}}^{-1/2} \hat{\mathbf{A}} \hat{\mathbf{D}}^{-1/2} \mathbf{H} \Theta \tag{7.4}$$

$\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ denotes the adjacency matrix with self-loops, and $\hat{D}_{i,i} = \sum \hat{A}_{i,j}$ represents its diagonal degree matrix. Expanding Equation (7.4), we get

$$\mathbf{h}_i^{(k)} = \Theta \left(\sum_{j \in N(i) \cup \{i\}} \boxed{\frac{1}{\sqrt{\deg(i)} \cdot \sqrt{deg(j)}}} \cdot \mathbf{h}_j^{(k-1)} \right) \tag{7.5}$$

We observe from Equation (7.5) that the feature aggregation phase performs a weighted summation (aggregation) of neighbor features. The coefficients within the marked region are derived from the degree matrix and perform a degree-based normalization.

GAT [159] leverages masked self-attention layers in feature aggregation to assign computation importance to each nodes. It does not require costly matrix operations as well as pre-knowledge of graph structural information, and is able to achieve computation efficiency including intra-graph parallelization.

$$\begin{aligned} \mathbf{h}_i' &= \boxed{\alpha_{i,i}}\, \Theta \mathbf{h}_i + \sum_{j \in N(i)} \boxed{\alpha_{i,j}}\, \Theta \mathbf{h}_j \\ &= \Theta \left(\sum_{j \in N(i) \cup \{i\}} \boxed{\alpha_{i,j}} \cdot \mathbf{h}_j \right) \end{aligned} \tag{7.6}$$

The attention coefficient $\alpha_{i,j}$ denoting the importance of node j's features to node i is computed as

$$\alpha_{i,j} = \mathit{SoftMax}\left(\mathit{LeakyReLU}\left(\mathbf{a}^\top [\Theta \mathbf{h}_i \,\|\, \Theta \mathbf{h}_j] \right) \right) \tag{7.7}$$

Equation (7.6) uses attention coefficients for the weighted aggregation of neighbor node embeddings. The denominator marked in Equation (7.7) also has a form similar to aggregation, where *SoftMax* is a normalization performed after the node-level exponential operation $exp(\cdot)$.

7.3 Motivation

GNNs can be considered as an extension of classic deep neural networks with irregular topology that support graph-structured inputs and outputs. Specialized architecture designs for GNNs are required because existing machine learning accelerators are not suited to GNNs for the following reasons.

Compounded Execution Mode Most GNN models are shallow with approximately three layers. The weight matrices W are generally dense as the result of layer-wise node aggregation. Current sparse matrix accelerators are specialized solely in processing sparse data formats, but will suffer significant overhead with dense matrix operations. However, the execution flow of GNNs follows a hybrid mode. Despite the sparsity of the input node feature matrices, the following aggregation operations will gradually populate the intermediary node embeddings with non-zero values. On the other hand, node feature transformation involves dense matrix multiplications. Present GNN accelerators are designed either to undertake the sparsity in **Aggregation**, or to employ the regularity in dense matrix multiplications that constitute **Transformation**, but lack the generality to handle both cases.

Workload Imbalance The irregular access and computation patterns of the **Aggregation** phase, which involves graph traversals that require tremendous memory access relative to only small amounts of calculation, make the mainstream computation platforms unsuitable for GNNs. CPUs and GPUs are not capable of irregular data movements and computations that constitute GNN operations. Their inefficiency in memory access causes the waste of off-chip memory bandwidths. Although modern GPUs such as NVIDIA A100 support sparse matrix multiplication with pruning techniques; for many real-world graph datasets, their feature matrices cannot be pruned, as the features are represented using binary values. Meanwhile, the adjacency matrices are subject to significant loss of graph structural information, making compression unrealistic despite their sparsity.

For traditional computation platforms including GPUs, their performance bottleneck on GNNs originates from their inability to settle the irregularity in Aggregation phase. Their relatively high performance on GNNs is mostly attributed to the high-bandwidth memory, which incurs considerable energy consumption. Furthermore, although they leverage the regularity in **Transformation** phase, the data copying and synchronization between threads for parameter reusing are expensive. Meanwhile, graph analytics accelerators are only optimized to alleviate irregularity or exploit regularity.

Usecase Generality Aggregation can be direct aggregation or weighted aggregation. The weighted one is not just performing summation or other aggregation operation. Neighbor node's feature vector is scaled with a factor before applying the aggregation operator, e.g., Equation 7.6 of *GAT*. Current GNN accelerators such as *EnGN* [106] and *HyGCN* [179] are only optimized for the directed aggregation case. This scaling issue can be solved by sparse-dense matrix multiplication and is more efficient than other designs.

HyGCN [179] is a GNN accelerator with a hybrid architecture. To harness the hybrid execution patterns of GNNs, *HyGCN* separates the modules for the regular neural network processing and irregular graph processing. The *Aggregation Engine* and *Combination Engine*, used for node aggregation and transformation respectively, each require separate on-chip buffers which consume considerable chip area. Despite their potentials for pipelining, the distinct computing patterns of these two processing stages make it difficult to harness both modules for efficient processing of general GNN structures.

EnGN [106] adopts the Ring-Edge-Reduce(RER) dataflow to tames the poor locality of sparsely connected vertices, and manipulates the ring-edge-reduce (RER) PE-array to practice RER dataflow. However, RER's storage of neighbor node indices consumes enormous local registers. Moreover, the index comparisons incurred by RER data movements may induce considerable latency and soaring computation cycles.

The characteristics of **Aggregation** phase reveals that it can adopt sparse matrix multiplication, then it provides an option to unify the *Transformation* phase and *Aggregation* phase into one single arithmetic operation, i.e., matrix multiplication, but aggregation phase tends to be sparse-dense case. Fig.7.3 demonstrates the possibility of unification.

Our Solution Therefore, we propose to accelerate the two phases with one versatile accelerator. We propose a unified hardware design that can reuse the limited on-chip buffers

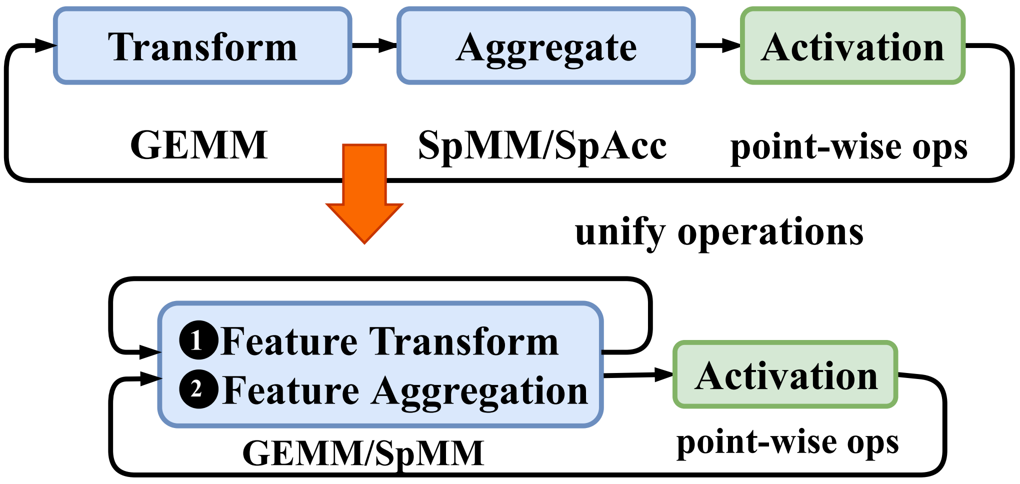


Figure 7.3: Unify transformation and aggregation phases

among the different processing stages. We can turn the aggregation stage into pseudo sparse matrix multiplication. Our design is a scalable and efficient parallel processing engine and support large-scale GNNs.

7.4 Microarchitecture

In this section, we discuss the hardware design of *VersaGNN*, with its system-level overview outlined in Fig.7.4. *VersaGNN* consists of a general-purpose processor, a memory management interface, a multi-purpose bus, and an accelerator. The processor is used to control the whole system, send instructions to both the accelerator and the memory system, and collect the status of the accelerator. The accelerator comprises an instruction queue, a DMA engine, a scratchpad memory with several banks, and the systolic array tiles which are interconnected side-by-side in a chain/ring fashion. An additional *Block Address Mapping* module is equipped for address matching when tiling is applied, and a Result Reordering module is used for the write-back phase of *sparse-dense matrix multiplications* (SpMM). Since the bulk of GNN calculations are matrix computations, our accelerator targets the acceleration of both dense and sparse matrix addition and multiplication.

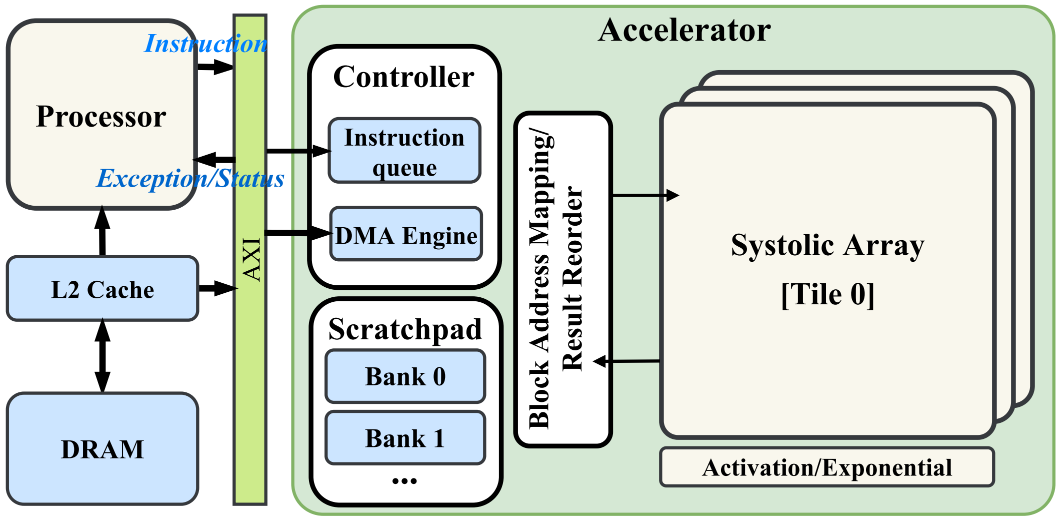


Figure 7.4: Overview of the hardware system design

7.4.1 Systolic Style Matrix Multiplier

The *Transformation* phase of the Graph Neural Network, formed by Multi-Layer Perceptrons (MLP), is essentially a multiplication of node feature matrix with layer weight matrix [196, 174] that maps the high dimensional node features to lower dimensional spaces. The input feature vector of a graph node is usually very sparse [174], implying that a node does not hold all its defined features, with absent features holding zero values. The intermediary node embeddings are gradually populated with non-zero values as feature aggregation proceeds, whereas most weight matrices for these layers are dense. To fuse matrix operations among these disparate structures, GNN requires a versatile accelerator architecture to tackle both dense and sparse matrix calculations.

7.4.1.1 Revisiting Strassen's Algorithm

Before diving into systolic array tile design, we revisit *Strassen's algorithm* [172], and expand it to block-based dense matrix multiplications to reduce the volume of expensive arithmetic multiplications. Strassen's algorithm has been studied and deployed on FPGA and GPU in the work of [32] and [81], respectively, which deliver eminent performance gains.

Given two matrices (or matrix tiles) A and B as input, we partition each of them into 4 sub-blocks: $A_0, \ldots, A_3$ for A, and $B_0, \ldots, B_3$ for B. Similarly, the result matrix C can be divided into $C_0, \ldots, C_3$.

$$
\begin{array}{ll}
 & \boxed{\begin{array}{ll} S_2 = A_2 + A_3 & \\ S_8 = A_1 - A_3 & S_9 = B_2 + B_3 \\ S_5 = A_1 + A_0 & S_3 = B_1 - B_3 \end{array}} \\
\boxed{\begin{array}{ll} S_4 = B_2 - B_0 & \\ S_0 = A_3 + A_0 & S_1 = B_0 + B_3 \\ S_6 = A_2 - A_0 & S_7 = B_0 + B_1 \end{array}} & \boxed{\begin{array}{ll} M_1 = S_2 \times B_0 & C_2 \mathrel{+}= M_1 \\ M_2 = A_0 \times S_3 & C_3 \mathrel{+}= M_2 \\ M_4 = S_5 \times B_3 & C_1 \mathrel{+}= M_4 \\ M_6 = S_8 \times S_9 & C_0 \mathrel{+}= M_6 \end{array}} \\
\boxed{\begin{array}{ll} M_0 = S_0 \times S_1 & C_0 \mathrel{+}= M_0 \\ M_3 = A_3 \times S_4 & C_2 \mathrel{+}= M_3 \\ M_5 = S_6 \times S_7 & C_3 \mathrel{+}= M_5 \end{array}} & \\
\boxed{\begin{array}{l} C_0 \mathrel{+}= M_3 \\ C_3 \mathrel{+}= M_0 \end{array}} & \boxed{\begin{array}{l} C_3 \mathrel{-}= M_1 \\ C_1 \mathrel{+}= M_2 \\ C_0 \mathrel{-}= M_4 \end{array}}
\end{array}
\tag{7.8}
$$

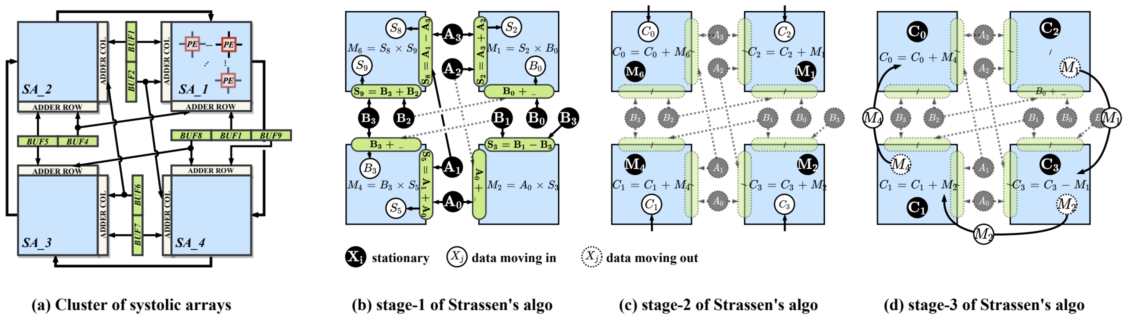
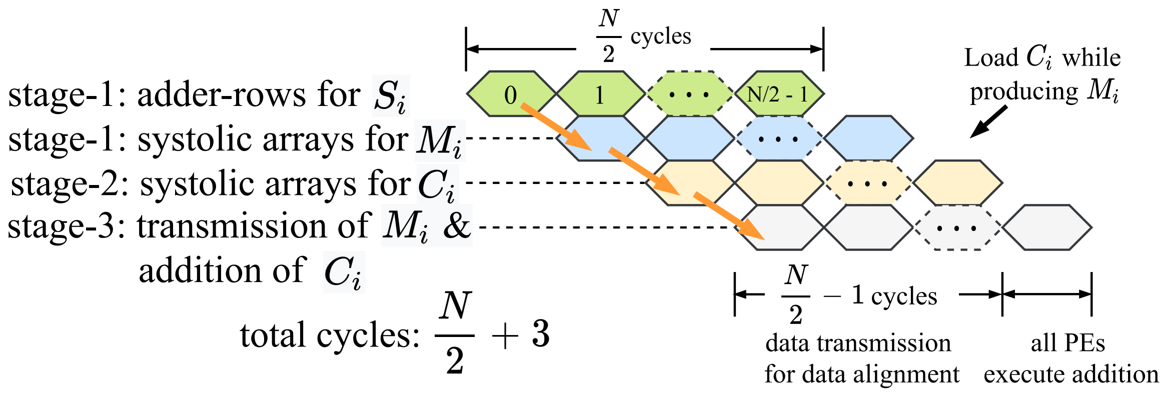


Figure 7.5: Scenario of **Strassen's Algorithm**: *dataflow* and *processing*

We rearrange the equations of *Strassen's algorithm* into 6 groups, as shown in Equation (7.8). Equations inside each group are performed in parallel, and the right-hand side of some equations in identical groups also share their operands, e.g., A_0, B_0 in the right-hand side of



$\frac{N}{2}$ cycles
Load C_i while producing M_i
stage-1: adder-rows for S_i
stage-1: systolic arrays for M_i
stage-2: systolic arrays for C_i
stage-3: transmission of M_i & addition of C_i
0
1
• • •
N/2 - 1
total cycles: $\frac{N}{2} + 3$
$\frac{N}{2} - 1$ cycles
data transmission for data alignment
all PEs execute addition

Figure 7.6: A simulated cycle-level illustration of the execution in a systolic tile of the Strassen's Algorithm. All 4 systolic tiles execute in parallel

the upper-left equation. To harness this element-wise parallelism, the three groups in the left column can be executed in pipeline mode, the same for the three groups in the right column. For example, after calculating M_3, we can directly apply it to computing C_2, and simultaneously forward M_3 to the next group and calculate C_0 without saving or fetching the intermediate result.

With the above analysis, we group the tiles of the systolic array into clusters (or meshes). Fig.7.5(a) illustrates the layout of a systolic array cluster. Unlike the mesh architecture of *Gemmini* [59], data across rows and columns are also shared in diagonal directions. The shared buffers act as interfaces to the scratchpad memory. Our systolic method minimizes the I/O cost by allowing each row or column of matrix operands to enter the processing element array only once for all its associated matrix computations [98]. Since matrix additions are element-wise operations, addition along rows and columns are independent. Further, the matrix additions can overlap with matrix multiplications as the pipeline proceeds. In our case, every systolic array receives the result of row or column addition from a pair of 1-D adder arrays. Each 1-D adder array is either a column or a row of adders located at one side of a systolic array, as shown in Fig.7.5 (a). The systolic arrays can forward the data through interconnections between neighboring systolic arrays within the ring structure.

Algorithm 5 One cycle in each PE of the systolic array for hybrid mode matrix multiplication.

```
function MAC( $a_{col}, a_{val}, b_{row}, b_{val}, a\_sparse, a\_dense, direct\_aggr, \mathbf{FA}$ )
    ▷ $direct\_aggr$: direct or weighted aggregation
    ▷ FA: circular $FIFO\_CAM$ (increasing order)
    $found \leftarrow$ false
    if $a\_dense$ or ($a\_sparse$ and $a_{col} = b_{row}$) then
        $a_{buf} \leftarrow a_{val}$; $b_{buf} \leftarrow b_{val}$
        $found \leftarrow$ true
        if $a\_sparse$ then
            $purge(\mathbf{FA})$
    else
        if $a_{col} > b_{row}$ then
            $(found, a_{buf}) \leftarrow \boxed{Find\&Skip\ (\mathbf{FA}, b_{row})}$
            $b_{buf} \leftarrow b_{val}$
            $push(\mathbf{FA}, a_{col}, a_{val})$
    if $found$ then
        if $direct\_aggr$ then          ▷ reduction operation can be add, min, max, or mean
            $c_{val} \leftarrow reduction\_operation(c_{val}, b_{buf})$
        else
            $c_{val} \leftarrow c_{val} + a_{buf} \times b_{buf}$
        if $a\_sparse$ then
            $(c_{h\_idx}, c_{v\_idx}) \leftarrow (a_{row}, b_{col})$     ▷ indices used for write-out phase when sparse
        $b_{row} \leftarrow b_{row} + 1$;
        $found \leftarrow$ false
    else
        $c_{val} \leftarrow c_{val}$
    $a_{out} \leftarrow (a_{val}, a_{col})$
    $b_{out} \leftarrow b_{val}$
```

Algorithm 6 **Find&Skip**

```
function Find&Skip($\mathbf{FI}, \mathbf{FV}, idx$)
▷ $\boldsymbol{FI}$: circular FIFO_CAM of nonzero indices;
▷ $\boldsymbol{FV}$: circular FIFO of nonzero values;
▷ $idx$: an index value to be found;
▷ $val$: the value of nonzero found;
▷ $found$: indicates existence of $idx$;
    $mask[:] \leftarrow 1$
    for all $i < \mathbf{FI}.size$ do
        if $\mathbf{FI}[i].idx < idx$ or $i < \mathbf{FI}.head$ then
        else
            $mask[i] \leftarrow 0$
▷ use the leading zero detector to find the position of first bit-one. And set the new
head of FIFO
    $\mathbf{FI}.head \leftarrow LeadingZeros(mask)$
    $val \leftarrow \mathbf{FV}[\mathbf{FI}.head]$
    $found \leftarrow equal(idx, \mathbf{FI}.head)$
    return $val, found$
```

Fig.7.5 (b-d) demonstrates the execution of the right three groups in Equation (7.8). The left three groups can be performed similarly and pipelined with the right three groups. Within each cycle, every 1-D adder array imports data from shared buffers to produce a row or column, and feed them to systolic array for matrix multiplication. 1-D adder arrays and systolic arrays orchestrate in a pipeline mode. Once the marginal PEs of systolic array receives the resultant row and column, they perform the MAC (Multiply-Accumulate) and forward the inputs to their neighboring PEs to perform MAC of M_is, as shown in Fig.7.5 (b). The MAC starts execution one cycle after the adder array as the row of Fig.7.6. As the results of matrix multiplication M_i are stored in the local register of PEs (in output

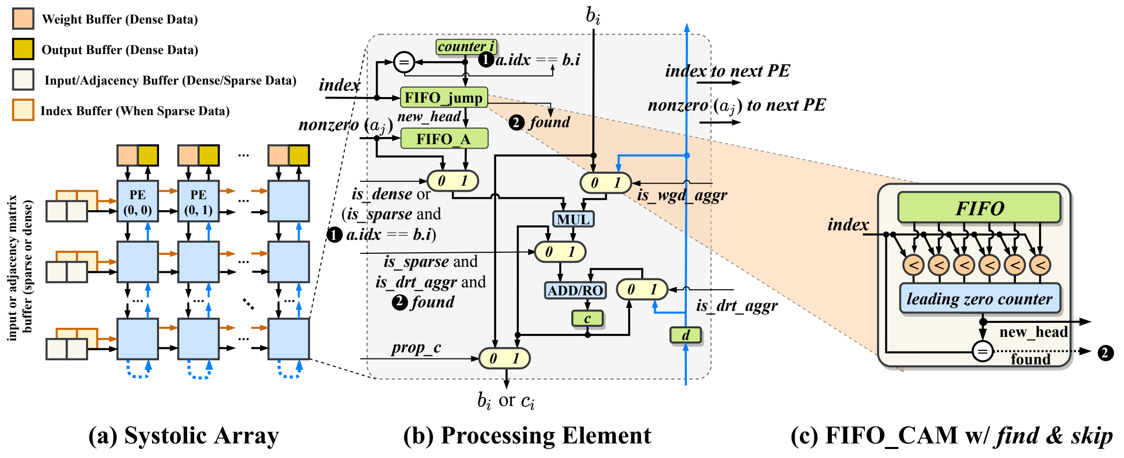


Figure 7.7: Layout and Logic view

stationary mode), we want to reuse them for the following matrix additions, then they are transferred to the neighboring systolic array in the ring, as shown in Fig.7.5 (c). As the two $C_i = C_i + M_j$ operations in Fig.7.5 (b) and (c) are independent, they can be pipelined along with data forwarding of M_is, but the one in stage 3 need to transmit M_i to be align with those at the same coordinates in the systolic arrays prior to performing addition. All PEs perform addition simultaneously during the last cycle, as shown in the 4th row of Fig.7.6.

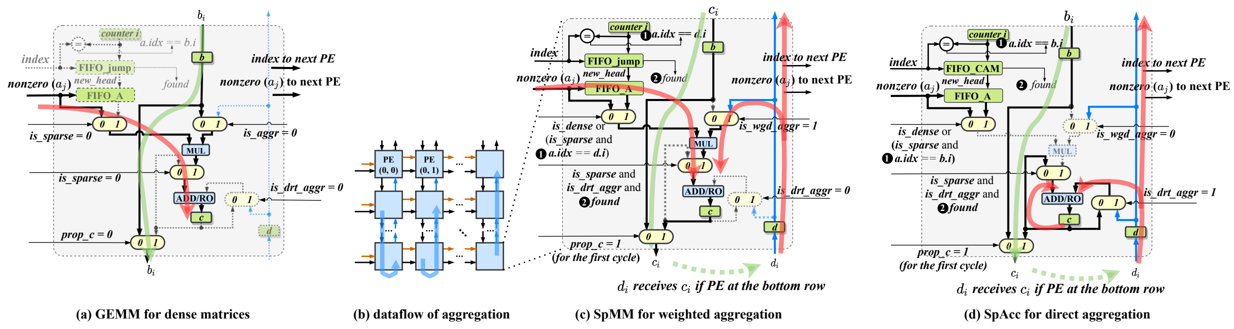


Figure 7.8: Three operation modes of PE: (a) dataflow of dense-dense matrix multiplication for *Transformation* phase; (b) dataflow for the SpMM of weighted *Aggregation*; (c) dataflow of the direct *Aggregation*; where *RO* is *reduction operation*, e.g., add, min, max, etc.

With this hardware-level implementation of *Strassen's algorithm*, we decrease the volume of multiplication and data transfer of operands from memory by profiting the internal bandwidth among PEs, as well as reading and writing of intermediate results from and to the scratchpad memory. *Strassen's algorithm* takes an asymptotic complexity of $O(N^{2.8074})$ when applied in recursive manner, compared with $O(N^3)$ of standard matrix multiplication. As shown in Fig.7.5 (b), the four tiles of systolic array execute in parallel and form a cluster, with skid buffers serving as bridges for inter-tile communication and the interface to local memory. Compared with the case in which four tiles are consolidated into one large systolic array, our design adopts a data source at the geometric center of the tile cluster, which effectively halved the data transmission path. Meanwhile, The hardware approach in *VersaGNN* is in 1-level Strassen's, and the software support of 2-level Strassen's can be combined with this design for further performance gain. [80].

7.4.1.2 Hybrid Mode Processing Elements

Fig.7.8(b) shows the internal structure of the processing element. To enable high-throughput operations on both dense and sparse matrix data structures, we devise an efficient hybrid mode *processing element* (PE) of the systolic array, it equips with dedicated searchable FIFOs, with which we call it *FIFO_CAM*. The *FIFO_CAM* can search a target element within its elements and skip unused elements as depicted in Algorithm 6 and Fig.7.8 (c). Instead of using a shared storage structure, we leverage distributed *FIFO_CAM*s design. Our design accepts the COO (Coordinate list) and CSR/CSC (Compressed Sparse Row/Column) formats [171]. The row is interpreted as a graph node and the column indices in such row are the node's neighbors. As the column indices moving in and out of the *FIFO_CAM*, it only needs to keep a relatively small sliding window for the indices under processing. Thus, it is not necessary to store a whole list of neighbors for a graph node in the *FIFO_CAM* of a PE. Empirical result shows that 4 entries of *FIFO_CAM* is big enough to accommodate ongoing data.

As described in previous section, *Transformation* phase of intermediate layers of GNNs is

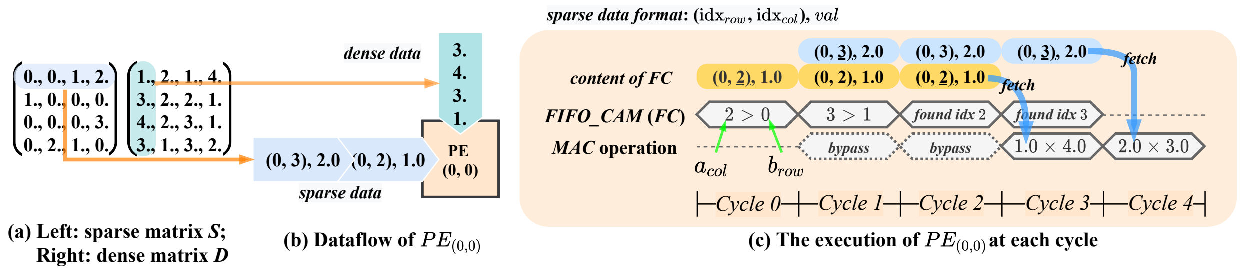


Figure 7.9: An internal view on the sparse MAC of Processing Element

simply a dense-dense matrix multiplication. In Algorithm 5, the PE conduct directly the MAC operation for the dense matrix multiplication, the dataflow is shown as the light green and red curve lines in Fig.7.9 (a), the FIFOs for operating sparse data are bypassed and concealed.

Recall the general mathematical expression of the convolution layer of GNN:

$$\underbrace{X^{'} = XW}_{\text{Transform}}; \quad \underbrace{X^{''} = AX^{'}}_{Aggregate} \tag{7.9}$$

In order to reuse $X^{'}$, the result of dense matrix multiplication of *Transformation* phase, and to avoid transferring $X^{'}$ back to scratchpad memory, we introduce an additional dataflow path from bottom to up in PE and stored locally into register d of PE, as shown in Fig.7.7 (b). At the initial cycle of *Aggregation*, the PE selects the value of c register by setting signal *prop_c* to 1 and passes it down to the south neighbor PE. After this cycle, each PE sets *prop_c* back to 0 and transfer value of c register as b of dense matrix. At bottom row of systolic array, PEs feed back the c to d register of PEs, the matrix $X^{'}$ turns back from bottom to up inside the systolic array, as shown in Fig.7.8 (b) and (c). This manner avoids the long distance data transfer leaping across all rows used in the ring structure of *RER* in *EnGN* [106] which leads to imbalance of data transfer rate between rows of processing array.

For the *weighed aggregations*, we treat them as *SpMM* $(AX^{'})$. Instead of using b register of PE, now the dataflow uses d register of PE, which flows into PE in opposite direction, as dense matrix element for MAC operation. The sparse data (from matrix A) are fed into

PEs, in horizontal direction, from west to east. As described in Algorithm 5 and shown in Fig.7.7(b), PE possesses a counter, b_{row}, for the row number of dense matrix, it augments itself at each cycle. When condition $a_{col} = b_{row}$ satisfies, the PE performs the MAC directly as the dense-dense case. However, if condition $a_{col} > b_{row}$ meets, the **PE** checks whether *FIFO_CAM* of A contains indices smaller than or equal to b_{row}, if there exists $a_{col} = b_{row}$, it fetches the corresponding nonzero value a_{val} from *FIFO* of nonzero, performs the MAC operation, and puts a_{col} and the corresponding nonzero into *FIFO_CAM* and *FIFO*, respectively. All i$a_{col} < b_{row}$ and their corresponding nonzeros are expelled from *FIFO*'s for sparse A. Note that indices are enqueued into the *FIFO_CAM*s in increasing order, thus are already in sorted order in *FIFO_CAM*s. Algorithm 6 describes the logic that performs searching and skipping mechanism, and we can integrate *FIFO* with such logic into our *FIFO_CAM*, as shown in Fig.7.8(c). The detection in *FIFO_CAM* is performed in parallel with all elements as Fig.7.8(c). With the help of these *FIFO_CAM*s, the **PE** can produce one result per cycle without any inter-cycle stalls.

Fig.7.9 delivers a concrete example. Suppose that in Fig.7.9(a), S and D are a sparse matrix and a dense matrix, respectively. The dataflow traversing $PE_{(0,0)}$ are the first row of S in compressed format and first column of D that enter the PE from left and above in Fig.7.9(b), respectively. Fig.7.9(c) demonstrates the cycle-by-cycle execution. In the first two cycles, the comparison operator does not find the matching index of current element. Thus, the *FIFO_CAM* stores them, and the *MAC* bypasses the data of dense matrix to neighboring *PE*s. At *Cycle*2, the *FIFO_CAM* performs parallel comparisons of row indices of S it stored with the incoming column index of D, and fetch the nonzero value of matched entry. Since the *FIFO_CAM* and *MAC* are fully pipelined, at *Cycle 3*, *MAC* performs the multiplication and accumulation on the data fetched from previous cycle, and *FIFO_CAM* performs the comparison of new incoming index in parallel. The *MAC* utilization rate relates to the number of matched indices. This issue can be solved with the algorithm introduced in Section 7.5.2.

The direct *Aggregation* can be seen as a special case of weighted *Aggregation*, where every nonzero is value one. Therefore, it is useless to perform multiplication with value one. As the light-red dataflow shown in Fig.7.8 (c), when indices a_{col} and b_{row} matches each other or it found a column index in $FIFO\_CAM$ equal to b_{row}, PE directly performs the addition of c and d register of PE without touching multiplier.

7.5 Software Approach

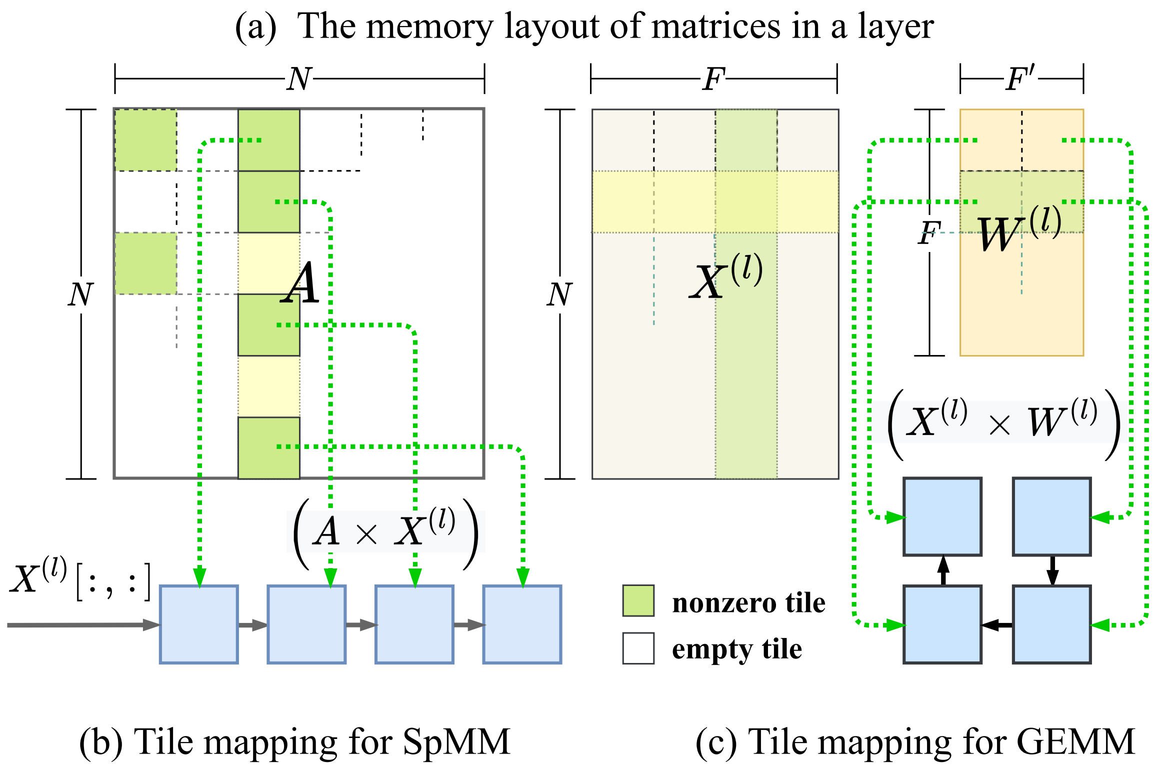

(a) The memory layout of matrices in a layer
N
F
F′
N
A
N
X^(l)
F
W^(l)
(X^(l) × W^(l))
(A × X^(l))
X^(l)[:, :]
nonzero tile
empty tile
(b) Tile mapping for SpMM
(c) Tile mapping for GEMM

Figure 7.10: Memory layout and tiling strategy

7.5.1 Tile Traversal strategy

The vast majority of real-world graphs that GNNs operate on cannot be fitted to the limited on-chip memory of accelerators. Thus, the algorithms often divide these large-scale graphs into tiles using grid partition approaches before applying arithmetic operations, and then merge individual tiles into the full result layout.

There exist several tiling traversal strategies, including row/column-major, Z-Morton, U-Morton, and Hilbert layouts [57, 20], that can be harnessed by GNN accelerators. However, the shape of feature matrix X and weight matrix are constantly changing through different layers. The feature matrix tends to become narrower and taller as the layers going deeper, whereas the weight matrix becomes smaller in size. As modern deep learning libraries support batched matrix multiplication with which the feature matrix can be seen as batched independent smaller matrices. In such way, small matrices are streamed into accelerator consecutively. In this work, we utilize two tile mapping strategies. For *Transformation* phase, we directly map the 4 tiles in a bigger square onto the ring of systolic arrays from both input and weight matrices, as shown in Fig.7.10 (c). In this way, the four systolic arrays in a ring perform the dense Strassen's algorithm as described in Section7.8. While, for the SpMM of *Aggregation* phase, we adopt an alternative strategy. After tiling, only tiles with nonzeros will take into account, those empty tiles are eliminated directly; the non-consecutive nonzero tiles in the same column are mapped onto the four systolic arrays, which is now in a chain, the dense tile from input matrix X is then traversing the four tiles. with such manner, the systolic arrays perform the batched SpMM, as shown in Fig.7.10 (b).

7.5.2 Greedy Workload Balancing

As the sparse matrix elements are irregularly distributed, some graph nodes may have relatively more neighbors. For sparse matrices in compressed formats, e.g., *CSR* or *COO*, each row of systolic array PEs processes features of a single node and aggregates its neighbor nodes' information. PEs in different rows will be assigned different workloads. The imbalanced workload can cause significant idling of PEs, with modules having less assigned workload finishing earlier and kept idle while waiting for those with heavier workload before the advent of next data stream, which will lead to degradation of the overall system-level performance.

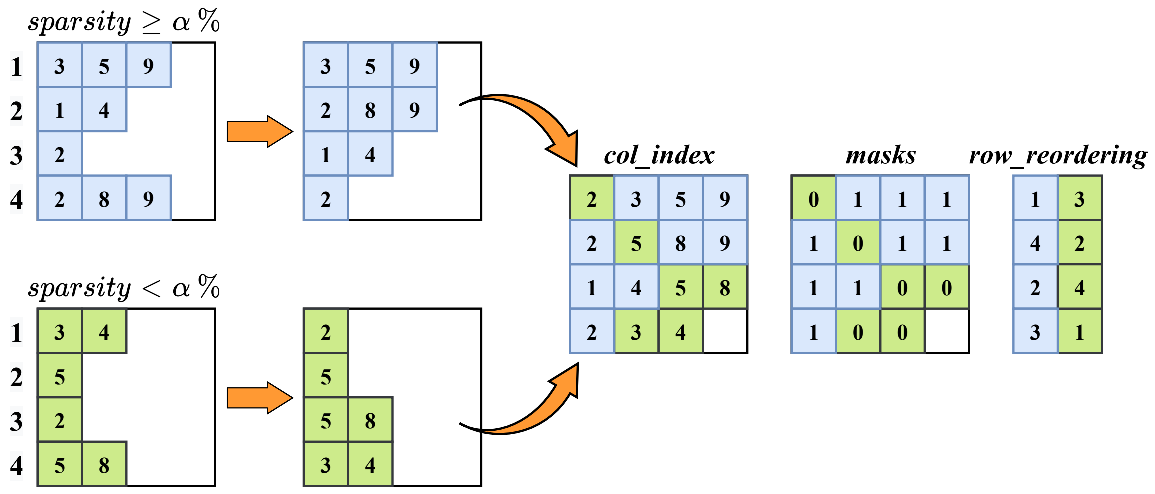


Figure 7.11: The greedy algorithm used for workload balancing: Given two tiles in CSR (or COO) format, each row represents one graph node, and the number in each row represent the column indices *col_index* (or neighboring nodes) belonging to that row (or node).

To remedy these issues, We introduce an effective greedy algorithm for workload balancing, which is an offline software scheme that groups tiles of sparse matrices into condensed ones. Our algorithm first sets a sparsity threshold $\alpha\%$, e.g. $40\% < \alpha \leq 50\%$. Assume the sparse adjacency matrix is split into tiles, and each tile is stored in *CSR* or *COO* format. For each tile with sparsity greater than $\alpha\%$, the algorithm searches for a complementary tile with sparsity less than $\alpha\%$ to combine with, as shown in Fig.7.11. Note that the number in each cell represents the column index of sparse matrix but not the value of non-zeros. Rows of the two tiles are sorted in reverse orders according to their number of elements and then combined (or packed) into one single tile. As shown in Fig.7.11, the column indices are arranged in increasing order. Duplicated elements are eliminated in each row, which prevents overlapping summation of feature vectors that belong to the same neighboring node when multiple vertices share a common set of neighbors. Our approach exploits an element mask as an identifier to trace the affiliation relationship between cells and their corresponding tiles. For combination of two tiles, each entry will have a single bit. Meanwhile, we adopt two 1-D arrays, i.e. two *reorder vectors*, to record the original row ordering of tile elements. When this combined tile is fed into the systolic array, almost every PE is utilized to its full capacity during each cycle. The mask and reorder vectors are utilized by the *Reordering Module* in the accelerator to direct the final results back to the scratchpad memory. Each entry of the combined tile, in form of $((row, col), val)$, is fetched with:

$$\begin{aligned} row &\leftarrow row\_reordering[i, masks[i, j]] \\ col &\leftarrow b_{col} \\ val &\leftarrow values[i, j] \end{aligned} \tag{7.10}$$

The execution of *Reordering Module* can be coordinated with that systolic arrays whenever the output is ready, causing execution overhead that is generally negligible. Further, to facilitate packed sparse tiles, both *CSR* and *COO* formats are treated internally as *COO* within systolic arrays. This approach is extensible to combine 3 or more tiles.

7.6 Evaluation

In this section, we begin with the experimental datasets and hardware configurations. Then, we deliver the detailed analysis of our optimizations.

Table 7.2: Configurations of system

	PyG-CPU	**PyG-GPU**	**HyGCN**	**EnGN**	**VersaGNN**
Compute Unit	3.0GHz @ 65 cores	1.25GHz @ 5120 cores	1GHz @ 32 SIMD 16 cores and 32×128 arrays	1GHz @ 128×16 arrays 32 PE units in VPU	1GHz @ 8 tiles of 32×32 arrays
On-Chip Memory	60MiB	34MiB	22MiB + 128KiB	1600KiB	4MiB + 256KiB
Peak Performance (GOP/s)	-	-	8704	6144	8192
Area (mm^2)	-	-	7.8 (12nm)	4.54 (14nm)	4.78 (16nm)
Power (W)	150	120	6.7	2.56	3.58
Energy Efficiency (GOPS/W)	-	-	1.30	2.4	1.71
Area Efficiency (GOPS/mm^2)	-	-	1.16	1.35	1.65

7.6.1 Experiment Configurations

Methodology We implemented our accelerator, *VersaGNN*, along with the baselines using Chisel3 Hardware Design Language (HDL) [10] Our design is also inspired by *Gemmini* [59] and *HardFloat* [128]. The systolic array adopts the output-stationary fashion with 16-bit floating point input and 32-bit floating point output. We evaluated the performance of the entire system and individual modules of *VersaGNN* with *FireSim* [87], a highly efficient, open-source simulator that simulates ASIC RTL designs with timing-accurate system components, which is of several magnitudes faster than software-based RTL simulation. We used *FireSim* to facilitate the full-system simulation by enabling integration of the simulated SoC with accurate peripheral and system-level interface models such as DDR3 memory or High Bandwidth Memory (HBM) and a last-level-cache (LLC). We synthesized *VersaGNN* using open-source *Yosys* and the TSMC 16nm process technology. Power and area are evaluated using a Cadence VLSI flow with TSMC 16 nm FinFET technology libraries. The placement and routing of the physical design were performed using Innovus, and power estimation using Voltus. The accelerators aim at achieving frequency of 1 GHz. To afford the high-throughput request volume, we equip the accelerator with HBM 2.0 interface with 256GB/s bandwidth, and a 256 KiB L2 Cache and a 4MiB last level cache (LLC). The energy of HBM 2.0 is estimated with 3.9 pJ/bit as in [115]. The configuration of *VersaGNN* and the baselines are described in Table 7.2.

Baselines We choose three distinct types of baseline architectures for performance and energy efficiency comparison, including the general-purpose processors (GPP), i.e. CPU and GPU, and two state-of-the-art GNN accelerators including *HyGCN* and *EnGN*. We selected the server processor, an Intel Xeon (Skylake) 6151@3.0GHz processor with 512GiB DRAM, as the CPU platform. The GPU platform is equipped with NVIDIA Tesla V100 and 32GiB HBM2. The software environment for the two platforms is PyTorch [120] and PyTorch Geometric (PyG) [52]. PyG is the state-of-the-art library for geometric deep learning that provides the majority of mainstream GNN models. We denote CPU and GPU platforms

Table 7.3: Dataset Statistics

Dataset	**Nodes**	**Edges**	**Features**	**Classes**	**Storage**	**Sparsity**	**Ave. Degree**
Cora (CA)	2,708	10,556	1,433	7	1.5MB	1.44E-03	4
Citeseer(CR)	3,327	9,104	3,703	6	47MB	8.22E-04	5
Pubmed (PB)	19,717	88,648	500	3	38MB	2.28E-04	6
IMDB-BIN (IB)	2,647	28,624	136	2	1.5MB	4.09E-03	39
Reddit (RD)	23,296	114,615,892	602	41	972MB	2.11E-03	9
Amazon (AM)	8.6M	231.6M	86	22	30.4MB	3.13E-06	2
COLLAB (CL)	12,087	1,446,010	492	3	28MB	9.90E-03	263

running PyG as PyG-CPU and PyG-GPU, respectively. The configuration of HyGCN and EnGN are listed in Table 7.2.

Benchmark Graph Datasets. Table 7.3 shows the statistics of benchmark graph datasets. The *Feature* column specifies the length of initial feature vector (H_0 from Table 7.1). The *Class* column marks the number of labels. The graphs in all listed datasets do not contain edge attributes. The sparsity of adjacency matrix is determined by the ratio of the number of graph edges to the square of the number of graph nodes. As we have stated in previous section, the *Aggregation* phase, i.e. $A \times X'$, is essentially **SpMM**, so we focus on how the sparsity affects the efficiency of the accelerator in executing SpMM. Lengths of node feature vectors determine tiling sizes and strategies of the dense matrix multiplication in the *Transformation* phase.

GNN models We benchmark the performance of *VersaGNN* using 4 GNN models, including *GCN* [96], *GraphSAGE (GSA)* [69], *GIN* [177], and *GAT* [159]. The first three models are mainly used for semi-supervised classification, while *GAT* can also be applied to inductive tasks, such as graph edge prediction and node feature prediction. To make the *GAT* profiting the sparse matrix multiplication and addition, we reformulate the calculation of the attention coefficients of Equation 7.6 and 7.7. Equation 7.6 can be expressed in matrix form,

$$H^{'} = \mathcal{A}HW \tag{7.11}$$

where $\mathcal{A}$ is the attention matrix and $\mathcal{A}[i,j]$ corresponds to $\alpha_{i,j}$ of Equation 7.6. Suppose trainable vector **a** of Equation 7.7 can be split into two sections:

$$\mathbf{a}^T[h_i^{'}||h_j^{'}] = (\mathbf{a}_1||\mathbf{a}_2)^T[h_i^{'}||h_j^{'}] = \mathbf{a}_1^T \cdot h_i^{'} + \mathbf{a}_2^T \cdot h_j^{'} \tag{7.12}$$

where $h_i^{'} = \Theta h_i$ and $h_j^{'} = \Theta h_j$. Turning it into matrix form, we get $H_1^{'} = H \cdot \mathbf{a}_1, H_2^{'} = H \cdot \mathbf{a}_2$. And the calculation of attention coefficient matrix $\mathcal{A}$ becomes:

$$\mathcal{A} = Softmax\left(\sigma\left(Diag(H_1^{'}) \cdot A + A \cdot Diag(H_2^{'})\right)\right) \tag{7.13}$$

where A is the adjacency matrix and σ is the activation function *LeakyReLU*. We then implement our customized code and replace the one in PyG. Most parts of Equation 7.11, and 7.13 can be executed by SpMM, thus *GAT* can also benefit from our highly efficient SpMM engine.

Evaluation Metrics We conduct our experiment with several metrics. We estimate 1) performance through the end-to-end inference time of GNN models; 2) throughput by billion operations per second (GOP/s); and 3) energy-efficiency by billion operations per second per Watt (GOPS/W).

7.6.2 Results of experiment

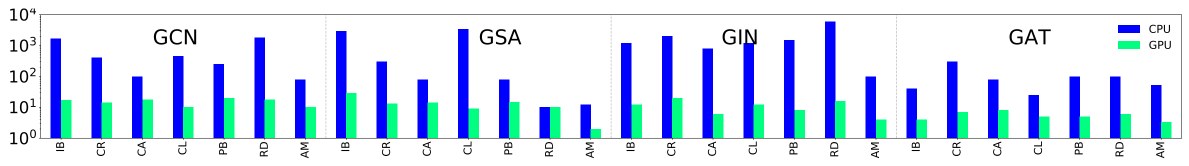


Figure 7.12: Performance speedup of *VersaGNN* over PyG-CPU/GPU

Power & Area We summarize the power and area of *HyGCN*, *EnGN*, and *VersaGNN* in Table 7.2. As reported by the CAD tool, the 4 banked 512 KiB scratchpad memory

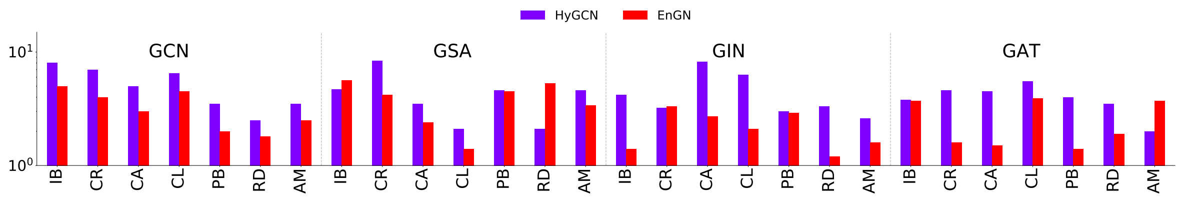


Figure 7.13: Performance speedup of *VersaGNN* over *HyGCN* and *EnGN*

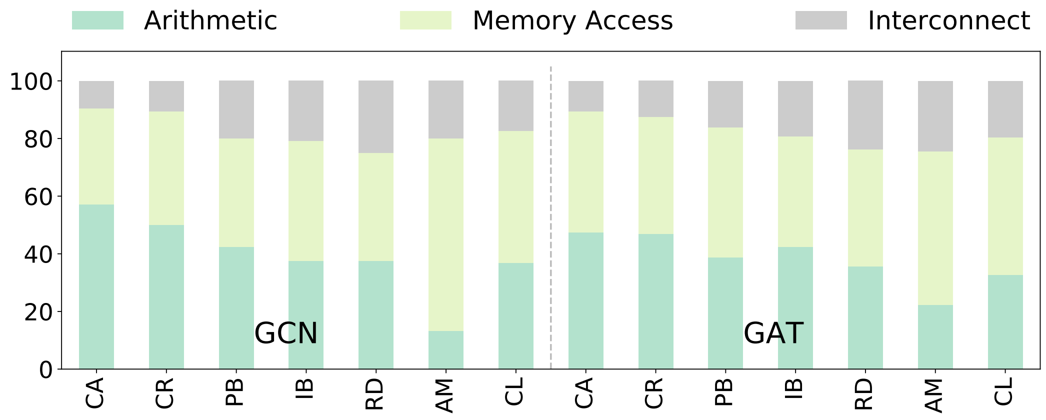


Figure 7.14: Energy breakdown of *VersaGNN*, left: GCN; right: GAT

and 128 KiB L2 Cache are the biggest part in our place-and-routed design. In the floor plan we organize the SRAMs of the accelerator in a semi-ring around the computational tiles. The second contributor of the area is the wires of the interconnections between tiles of systolic array. Due to the limit of process technology at 16nm, the power of *VersaGNN* is higher than *EnGN* but still achieves a higher energy efficiency than both *EnGN* and *HyGCN*. Static timing analysis at net-list level shows that there is still some positive slack, signifying potential for further frequency increases of our design. Fig.7.14 provides the breakdown of the energy consumed by arithmetic operations, memory accesses, and interconnect between tiles for the model of *GCN* and *GAT*. As the figures illustrates that the sparser dataset tends to be compute-bound and denser dataset tends to be memory-bound. It is crucial to improve the cache utilization for the denser dataset; while sparser graph's neighbors having larger stride crossing several tiles cause cache to evict more frequently.

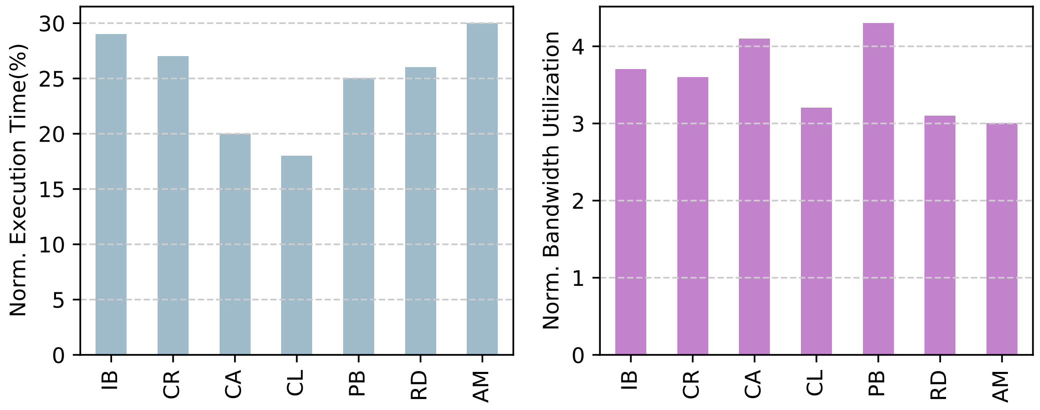


Figure 7.15: Effects of Strassen's algorithm for GEMM (dense) on 4 tiles of systolic array compared with a single large systolic array used by the baseline *Gemmini*; Left: normalized execution time; Right: bandwidth utilization rate

Performance The performance of *VersaGNN* is compared with baseline platforms including PyG-CPU and PyG-GPU, *HyGCN*, and *EnGN*. The original implementation of PyG adopts the *Pytorch Scatter Library* [49] for the *Aggregation* phase. Our experimental results show that the average performance speedup of all models on all datasets compared with PyG-CPU is 3712×, as shown in Fig.7.12. For the case of GPU, we rewrite the *Aggregation* function as SpMM by using *PyTorch Sparse library* [50] and solved the memory leakage problem for the version that we used in this experiment. As the writing of this work, PyG has announced the re-implementation of the *Aggregation* phase as SpMM in its future release. We obtained an average 35.4× speedup compared to PyG-GPU over all models on all datasets, as shown in Fig.7.12. Compared to *HyGCN* and *EnGN*, *VersaGNN* achieves higher performance speedup on both small and big datasets, especially for the model with weighted *Aggregation* since prior to perform the summation the neighbor node's feature vector, it needs to scale up with the coefficient, e.g., the fraction of degree term in *GCN*, and the attention coefficient in *GAT*. Finally, *VersaGNN* is 6.32× faster than *HyGCN* and 2.73× faster than *EnGN*.

Throughput Through our experiments, we observe that datasets with higher densities of graph connections (the number of edges) or longer node feature vector tend to yield higher throughput. This is because longer feature vectors are well-suited to dense-dense matrix multiplication due to their superior memory coalescing mechanisms, whereas their high graph connectivity facilitates the memory access pattern since connected nodes are more likely to share neighbors, which increases the spatial locality of memory access. By applying greedy workload balancing, *VersaGNN* is able to maintain a steady throughput even on datasets with massive irregular memory access, thus leveraging the computation and irregular memory access of the SpMM.

7.6.3 Analysis of Optimization of VersaGNN

In this section, we evaluate the performance improvement of each optimization for *Transformation* and *Aggregation*, respectively.

Strassen's Algorithm This optimization is applied only to the *Transformation* phase which consists primarily of dense-dense matrix multiplication. The normalized execution time and the bandwidth utilization in Fig.7.15 show that *VersaGNN* achieves 1.7$\sim$3.1$\times$ speedup when Strassen's algorithm is applied at hardware level. The performance gain is due to the parallelism from simultaneous matrix multiplications by 4 tiles of systolic array instead of one large tile, and the data transmission traversing the boundaries of neighboring tiles is achieved by utilizing the internal bandwidth of systolic arrays, which facilitates data reuse and reduces the data write-backs. This demonstrates that through collaboration, smaller spatial accelerators are able to achieve superior results than a single large module.

Greedy Workload Balancing The greedy algorithm introduced in Section 7.5.2 can be utilized by both direct and weighted *Aggregation* phases. The efficiency of this greedy approach is also affected by the sparsity of datasets, with sparser graphs bearing more tiles to be coalesced. Graphs with higher average degree tend to produce denser tiles, and better sparse tile packing strategy delivers more parallelism and utilizes less operation cycles to improve the utilization of PEs. To further evaluate the improvement in SpMM of *Aggregation*

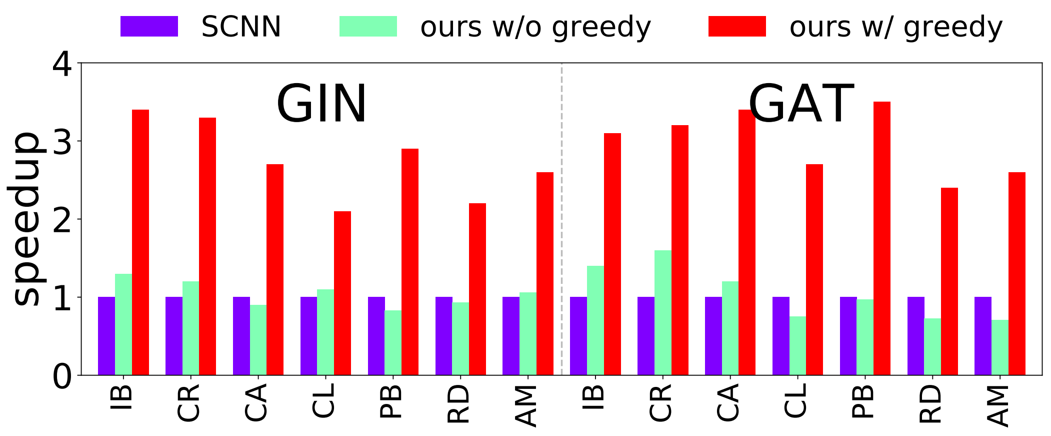


Figure 7.16: the speedup of SpMM of *Aggregation* phase compared with SCNN [117]

phase, we use the *SCNN*, specialized for sparse model, as baseline. Experiments show that *SCNN* is less efficient for *GNN*, since *Cartesian Product*-based SpMM consumes most of time in processing massive irregular reduction of intermediate results, and the out-of-order scattering operation causes stall when multiple intermediate results are written into same memory location; while VersaGNN's greedy algorithm feed the operands according to the increasing order of indices in pipeline, which guarantees free of stalling, as shown in Fig.7.16. This greedy algorithm is also affected by the tiling strategy described in Section 7.5.1, since the sparsity varies according to the tile size and the distribution of node neighbors. Generally, the greedy workload balancing algorithm afford great improvement in performance of speed and energy saving. Since it is a static data pre-processing method prior to the execution of model, greedy workload balancing improves the utilization of PEs and packs the sparse tiles to increase more useful workload, as shown in Fig.7.17. Our greedy algorithm is more flexible than the tile packing algorithm used in *EnGN*, which requires two tiles with fully compatible empty slots. Moreover, *HyGCN*'s window sliding strategy cannot remove zero-entries inside the tiles.

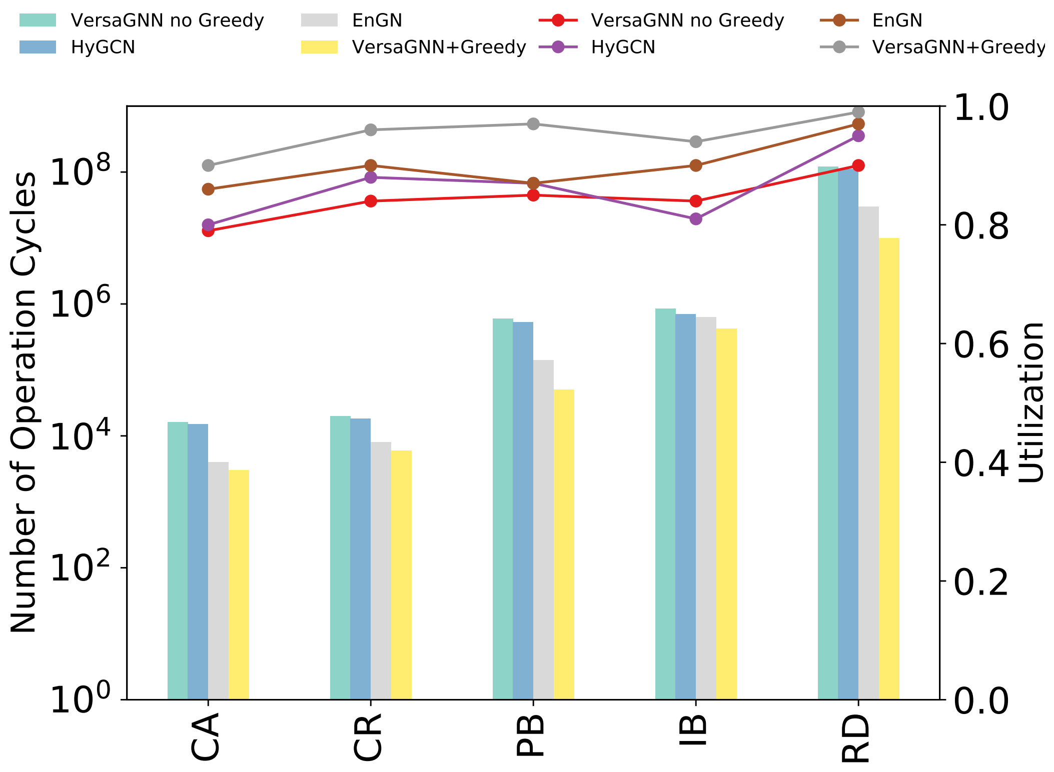


Figure 7.17: The utilization of PEs and overall cycles used to complete *Aggregation* phase.

7.7 Related Work

There have been ongoing researches on tackling the hybrid computing pattern of Graph Neural Networks [187, 179, 38, 7, 60]. *GraphACT* [187] devises an algorithm to exploit redundant operations by looking for neighbor pairs. *HyGCN* [179] and *EnGN* [106] design high-performance ASIC accelerators with two individual computing components for Aggregation and Transformation phases, respectively. *GraphZoom* [38] proposes an efficient clustering algorithm to condense the graph, reducing considerable inference latency.

Deep Learning Acceleration on Sparse Structures The latest machine learning models, especially those for embedded and mobile systems[21], reduce their weight volumes by driving smaller weight parameters towards zeros in the feature maps and filters, leading to highly sparse models. Several works have been proposed for accelerating SparseNN and sparse matrix computing [71, 123, 73, 74, 99]. [99] describes a novel approach of packing sparse networks into denser formats for efficient implementation using systolic arrays. [21] proposes *Eyeriss v2* for execution of both compact and sparse DNNs. Architectures such as [117] and [63] are committed to the sparse model design. Although the Cartesian-product in [117] avoids the index matching in producing products, the calculation of destination addresses corresponding to indices of products are still required while doing the partial sums, which can be seen as a postponed index matching. *SparTen* [63] provides an effective inner join mechanism, but their vector-vector multiplier cannot share broadcast inputs internally as the highly-efficient systolic array. The bandwidth is wasteful, and the broadcasting demands an intricate protocol for data synchronization.

Graph Analytics Systems solve graph-related problems from different dimensions. Generally, they optimize conventional graph algorithms using a deterministic approach. Node in these graph structures typically do not possess high-dimension attributes. Therefore, the software and hardware solutions targeting these types of graphs [67, 19] are ineffective for GNN models.

7.8 Conclusion

The hybrid computation mode of Graph Neural Networks impose huge obstacles in acceleration of GNN architectures. In this paper, we tackled GNN acceleration by first generalizing its computation pattern into two stages, *Aggregation* and *Transformation*. *Aggregation* phase is essentially formed by sparse matrix multiplication, whereas the *Transformation* phase is dominated by dense matrix calculation. Then, we propose *VersaGNN*, a high-throughput and memory-efficient GNN accelerator based on the systolic array paradigm. To offer the flexibility towards both dense and sparse matrix multiplication, we re-factor the processing element of systolic arrays. We further design the architecture of multiple tiles that form a computing cluster, which supports efficient execution of Strassen's algorithm. This hardware-level Strassen's algorithm dramatically reduces the computation and memory access. At the same time, We also designed a greedy workload balancing algorithm from software perspective to improve the efficiency of sparse matrix multiplication. Our vast experiments have demonstrated that, under the state-of-the-art GNN software frameworks, *VersaGNN* achieves on average 3712× performance gain with 1301.25× energy reduction on CPU, and 35.4× speedup with 17.66× energy reduction on GPU.

CHAPTER 8

Conclusion

This dissertation contributes to building a full-stack platform for video analytic. The video analytic platform introduces machine intelligence to assist humans to fulfill various tasks and interactions between machines and humans. The whole system integrates several well-studied modules as front-end, including object detection, object classification, human action detection, etc., and these front-end modules produce information observed from the environment. This information needs to be further processed by the backend modules to obtain the underlying relationship between objects and humans; These backend modules synthesize new or hidden information, which plays an essential role in scene parsing and understanding of the videos. In such cases, the backend modules involve more sophisticated algorithms, such as graph analytics and logic reasoning, which are not limited to deep learning.

8.1 Dissertation Summary and Contributions

Most of the works in this dissertation are focusing on the backend part of the video analytic platform. This dissertation first presents a set of artificial intelligence algorithms utilized for various scene parsing and scene understanding modules of the video analytic platform. Then, according to the essential operation performed in these modules, this dissertation demonstrated the design and implementation of the customized hardware accelerator and optimization from an algorithmic angle by accelerating the underlying arithmetic operations, such as tensor or matrix operations.

Chapters 2, 3, and 4 are targeting efficient algorithm design for various modules in video analytic platform. The combination of logic reasoning, graph algorithms with deep learning approach are actually categorized as neural symbolic learning. Especially for traditional NP-complete/hard problem, neural symbolic learning dramatically reduces the complexity of computation while still keeps very close accuracy compared to traditional heuristic based approaches. The works in this dissertation also target efficiency for low volume storage usage, as the data format introduced for human action recognition and the dynamically generated sparse attention coefficients during the computation.

Chapters 5, 6, and 7 are targeting design efficient hardware acceleration solution for the aforementioned algorithms. Since we adopt the hardware/software co-design paradigm, the hardware designs can keep their architecture as simple as possible while the software approach facilitates the data transfer and reduces the number of complicated operations.

8.2 Future Work

As the video analytic platform becomes more and more complex, more functionalities are required to fulfill more advanced tasks. The design complexity gradually exceeds the manipulation ability of human beings, where it requires more involvement of machine intelligence to assist. Meanwhile, the artificial intelligence community and the VLSI design community are active and vital. The advancement in one community also improves another. It is evident in demand for faster accelerators for AI algorithms. Recently, the AI community is welcoming its new era for helping the VLSI community; especially, the deep learning approach found its role in assisting electronic design automation (EDA), neural architecture search, and high-level synthesis (HLS).

As the trial of the deep learning approach used for logic synthesis, an essential step in VLSI design, in Chapter 4 has shown a promising solution; we want to extend this approach to other graph algorithms utilized in EDA. The stochastic approach, such as deep learning, has notable characteristics other than the traditional deterministic approach, such as faster convergence, better scalability, higher robustness.

The popularity of the Transformer model and its variants also drew many researchers' attention. Moreover, people found that the Transformer model could be an omnipotent model surpassing the convolutional neural network. The transformer model can be seen as a particular case of the graph neural networks as they share some key characters and basic operations. Therefore, in future work, we want to transfer the design of graph neural network accelerator in Chapter 7 for the Transformer model.

Bibliography

[1] Q. Zhang et al. “Interpreting CNN knowledge via an Explanatory Graph”. In: *AAAI '18* (2018).

[2] S. Chatterjee et al. “Recursive array layouts and fast matrix multiplication”. In: *IPDS* 13.11 (Nov. 2002), pp. 1105–1123. ISSN: 1045-9219.

[3] L. Amarú et al. “SAT-Sweeping Enhanced for Logic Synthesis”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: `10.1109/DAC18072.2020.9218691`.

[4] Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. “PDP: A general neural framework for learning constraint satisfaction solvers”. In: *arXiv preprint arXiv:1903.01969* (2019).

[5] James Atwood and Don Towsley. “Diffusion-convolutional neural networks”. In: *Advances in neural information processing systems*. 2016, pp. 1993–2001.

[6] Gilles Audemard and Laurent Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. IJCAI'09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404.

[7] Adam Auten, Matthew Tomei, and Rakesh Kumar. “Hardware Acceleration of Graph Neural Networks”. In: *2020 IEEE International Symposium on Design Automation Conference (DAC)*. 2020, pp. 1–6.

[8] Utku Aydonat et al. “An OpenCL™Deep Learning Accelerator on Arria 10”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey, California, USA: ACM, 2017, pp. 55–64. ISBN: 978-1-4503-4354-1.

[9] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. “Layer normalization”. In: *arXiv preprint arXiv:1607.06450* (2016).

[10] Jonathan Bachrach et al. “Chisel: constructing hardware in a scala embedded language”. In: *DAC Design Automation Conference 2012*. IEEE. 2012, pp. 1212–1221.

[11] Yunsheng Bai et al. “Convolutional set matching for graph similarity”. In: *arXiv preprint arXiv:1810.10866* (2018).

[12] Yunsheng Bai et al. “Learning-based Efficient Graph Similarity Computation via Multi-Scale Convolutional Set Matching”. In: AAAI. 2020.

[13] Yunsheng Bai et al. “SimGNN: A Neural Network Approach to Fast Graph Similarity Computation”. In: WSDM ’19. Melbourne VIC, Australia: Association for Computing Machinery, 2019, pp. 384–392. ISBN: 9781450359405. DOI: 10.1145/3289600.3290967. URL: https://doi.org/10.1145/3289600.3290967.

[14] Yunsheng Bai et al. “SimGNN: A Neural Network Approach to Fast Graph Similarity Computation”. In: (2019).

[15] Michael M. Bronstein et al. “Geometric Deep Learning: Going beyond Euclidean data”. In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42. DOI: 10.1109/MSP.2017.2693418.

[16] Joan Bruna et al. “Spectral networks and locally connected networks on graphs”. In: *arXiv preprint arXiv:1312.6203* (2013).

[17] Horst Bunke. “On a relation between graph edit distance and maximum common subgraph”. In: *Pattern Recognition Letters* 18.8 (1997), pp. 689–694.

[18] Horst Bunke and Kim Shearer. “A graph distance metric based on the maximal common subgraph”. In: *Pattern recognition letters* 19.3-4 (1998), pp. 255–259.

[19] Nagadastagiri Challapalle et al. "GaaS-X: Graph Analytics Accelerator Supporting Sparse Data Representation using Crossbar Architectures". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 433–445.

[20] S. Chatterjee et al. "Recursive array layouts and fast matrix multiplication". In: *IEEE Transactions on Parallel and Distributed Systems* 13.11 (2002), pp. 1105–1123.

[21] Yu-Hsin Chen et al. "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308.

[22] Liang-Chieh Chen et al. "Rethinking Atrous Convolution for Semantic Image Segmentation". In: *CoRR* abs/1706.05587 (2017). arXiv: `1706.05587`. URL: `http://arxiv.org/abs/1706.05587`.

[23] Xinlei Chen et al. "Iterative visual reasoning beyond convolutions". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7239–7248.

[24] Ke Cheng et al. "Skeleton-Based Action Recognition with Shift Graph Convolutional Network". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020.

[25] Sharan Chetlur et al. "cuDNN: Efficient Primitives for Deep Learning". In: *CoRR* abs/1410.0759 (2014). arXiv: `1410.0759`. URL: `http://arxiv.org/abs/1410.0759`.

[26] Minsu Cho, Karteek Alahari, and Jean Ponce. "Learning Graphs to Match". In: *Proceedings of the IEEE Interational Conference on Computer Vision*. 2013.

[27] Jin Choi et al. "A View-Based Real-Time Human Action Recognition System as an Interface for Human Computer Interaction". In: *Virtual Systems and Multimedia*. Ed. by Theodor G. Wyeld, Sarah Kenderdine, and Michael Docherty. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 112–120. ISBN: 978-3-540-78566-8.

[28] Y. Choi, M. El-Khamy, and J. Lee. “Compression of Deep Convolutional Neural Networks under Joint Sparsity Constraints”. In: *ArXiv e-prints* (May 2018). arXiv: `1805.08303 [cs.CV]`.

[29] Krzysztof Marcin Choromanski et al. “Rethinking Attention with Performers”. In: *International Conference on Learning Representations*. 2021. URL: `https://openreview.net/forum?id=Ua6zuk0WRH`.

[30] Y. Chou et al. “MajorSat: A SAT solver to majority logic”. In: *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2016, pp. 480–485. DOI: `10.1109/ASPDAC.2016.7428058`.

[31] Jason Cong and Jie Wang. “PolySA: Polyhedral-Based Systolic Array Auto-Compilation”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. San Diago, CA, USA: ACM, Nov. 2018.

[32] Jason Cong and Bingjun Xiao. “Minimizing Computation in Convolutional Neural Networks”. In: *Artificial Neural Networks and Machine Learning – ICANN 2014*. Ed. by Stefan Wermter et al. Cham: Springer International Publishing, 2014, pp. 281–290. ISBN: 978-3-319-11179-7.

[33] Martin Davis and Hilary Putnam. “A Computing Procedure for Quantification Theory”. In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411. DOI: `10.1145/321033.321034`.

[34] P. Deepa and C. Vasanthanayaki. “FPGA based efficient on-chip memory for image processing algorithms”. In: *Microelectronics Journal* 43.11 (2012), pp. 916–928. ISSN: 0026-2692.

[35] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. “Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering”. In: *Advances in Neural Information Processing Systems*. Ed. by D. Lee et al. Vol. 29. Curran As-

sociates, Inc., 2016. URL: `https://proceedings.neurips.cc/paper/2016/file/04df4d434d481c5bb723be1b6df1ee65-Paper.pdf`.

[36] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. "Convolutional neural networks on graphs with fast localized spectral filtering". In: *Advances in neural information processing systems*. 2016, pp. 3844–3852.

[37] Michaël Defferrard et al. *PyGSP: Graph Signal Processing in Python*. DOI: `10.5281/zenodo.1003157`. URL: `https://github.com/epfl-lts2/pygsp/`.

[38] Chenhui Deng et al. "GraphZoom: A Multi-level Spectral Approach for Accurate and Scalable Graph Embedding". In: *International Conference on Learning Representations*. 2020. URL: `https://openreview.net/forum?id=r1lGO0EKDH`.

[39] Tyler Derr et al. "Deep adversarial network alignment". In: *arXiv preprint arXiv:1902.10307* (2019).

[40] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 4171–4186. DOI: `10.18653/v1/N19-1423`. URL: `https://www.aclweb.org/anthology/N19-1423`.

[41] R. DiCecco et al. "Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks". In: *2016 International Conference on Field-Programmable Technology (FPT)*. Dec. 2016, pp. 265–268.

[42] Alexey Dosovitskiy et al. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929* (2020).

[43] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *arXiv e-prints*, arXiv:1603.07285 (Mar. 2016). arXiv: `1603.07285 [stat.ML]`.

[44] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In: *SAT*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, 2003, pp. 502–518.

[45] Stefan Elfwing, Eiji Uchibe, and Kenji Doya. "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning". In: *Neural Networks* 107 (2018), pp. 3–11.

[46] C. Farabet et al. "NeuFlow: A runtime reconfigurable dataflow processor for vision". In: *CVPR 2011 WORKSHOPS*. June 2011, pp. 109–116.

[47] Matthias Fey. *PyTorch Scatter*. 2021. URL: `https://github.com/rusty1s/pytorch_scatter` (visited on 09/30/2020).

[48] Matthias Fey. *PyTorch Sparse*. 2021. URL: `https://github.com/rusty1s/pytorch_sparse` (visited on 09/30/2020).

[49] Matthias Fey. *The PyTorch Scatter Library*. Online. An optional note. July 2019.

[50] Matthias Fey. *The PyTorch Sparse Library*. Online. An optional note. Apr. 2020.

[51] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[52] Matthias Fey and Jan E. Lenssen. "Fast Graph Representation Learning with PyTorch Geometric". In: *ICLR Workshop on Representation Learning on Graphs and Manifolds*. 2019.

[53] Matthias Fey and Jan Eric Lenssen. "Fast graph representation learning with PyTorch Geometric". In: *arXiv preprint arXiv:1903.02428* (2019).

[54] Matthias Fey and Jan Eric Lenssen. "Fast graph representation learning with PyTorch Geometric". In: *arXiv preprint arXiv:1903.02428* (2019).

[55] Matthias Fey et al. "Deep graph matching consensus". In: *arXiv preprint arXiv:2001.09621* (2020).

[56] Alex Fout et al. "Protein interface prediction using graph convolutional networks". In: *Advances in neural information processing systems*. 2017, pp. 6530–6539.

[57] Matteo Frigo et al. "Cache-Oblivious Algorithms". In: *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. FOCS '99. USA: IEEE Computer Society, 1999, p. 285. ISBN: 0769504094.

[58] Matteo Frigo et al. "Cache-Oblivious Algorithms". In: *ACM Trans. Algorithms* 8.1 (Jan. 2012), 4:1–4:22. ISSN: 1549-6325.

[59] Hasan Genc et al. "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures". In: *arXiv preprint arXiv:1911.09925* (2019).

[60] Tong Geng et al. *AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing*. 2020. arXiv: `1908.10834 [cs.DC]`.

[61] Justin Gilmer et al. "Neural message passing for quantum chemistry". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1263–1272.

[62] S. Gold and A. Rangarajan. "A graduated assignment algorithm for graph matching". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 18.4 (1996), pp. 377–388.

[63] Ashish Gondimalla et al. "SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 151–165. ISBN: 9781450369381. DOI: `10.1145/3352460.3358291`. URL: `https://doi.org/10.1145/3352460.3358291`.

[64] Ian J. Goodfellow et al. "Generative Adversarial Nets". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 2672–2680. URL: `http://dl.acm.org/citation.cfm?id=2969033.2969125`.

[65] K. Gouda and M. Hassaan. "CSI_GED: An efficient approach for graph edit similarity computation". In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 2016, pp. 265–276.

[66] W. Haaswijk et al. "SAT Based Exact Synthesis using DAG Topology Families". In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 2018, pp. 1–6. DOI: `10.1109/DAC.2018.8465888`.

[67] Tae Jun Ham et al. "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics". In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13.

[68] William Hamilton et al. "Querying Complex Networks in Vector Space". In: June 2018.

[69] William L. Hamilton, Rex Ying, and Jure Leskovec. "Inductive Representation Learning on Large Graphs". In: *NIPS*. 2017.

[70] Song Han, Huizi Mao, and William J Dally. "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *International Conference on Learning Representations (ICLR)* (2016).

[71] Song Han et al. "EIE: efficient inference engine on compressed deep neural network". In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 243–254.

[72] Fei Hao et al. "Similarity Evalution between Graphs: A Formal Concept Analysis Approach." In: *JIPS* 13.5 (2017), pp. 1158–1167.

[73] Xin He et al. "Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices". In: *International Conference on Supercomputing (ICS'20)*. 2020.

[74] Kartik Hegde et al. “Extensor: An accelerator for sparse tensor algebra”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 319–333.

[75] Mikael Henaff, Joan Bruna, and Yann LeCun. “Deep convolutional networks on graph-structured data”. In: *arXiv preprint arXiv:1506.05163* (2015).

[76] Federico Heras, Javier Larrosa, and Albert Oliveras. “MiniMaxSat: A New Weighted Max-SAT Solver”. In: *Theory and Applications of Satisfiability Testing – SAT 2007*. Ed. by João Marques-Silva and Karem A. Sakallah. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 41–55. ISBN: 978-3-540-72788-0.

[77] Marijn Heule, Matti Järvisalo, and Martin Suda. “SAT Competition 2018”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 11 (Sept. 2019), pp. 133–154. DOI: 10.3233/SAT190120.

[78] Holger H Hoos and Thomas Stützle. “SATLIB: An online resource for research on SAT”. In: *Sat* 2000 (2000), pp. 283–292.

[79] K. K. Htike et al. “Human activity recognition for video surveillance using sequences of postures”. In: *The Third International Conference on e-Technologies and Networks for Development (ICeND2014)*. 2014, pp. 79–82. DOI: 10.1109/ICeND.2014.6991357.

[80] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. “Implementing Strassen’s Algorithm with CUTLASS on NVIDIA Volta GPUs”. In: *CoRR* abs/1808.07984 (2018). arXiv: 1808.07984. URL: http://arxiv.org/abs/1808.07984.

[81] Jianyu Huang, Chenhan D. Yu, and Robert A. van de Geijn. “Strassen’s Algorithm Reloaded on GPUs”. In: *ACM Trans. Math. Softw.* 46.1 (Mar. 2020). ISSN: 0098-3500. DOI: 10.1145/3372419. URL: https://doi.org/10.1145/3372419.

[82] Zilong Huang et al. “Ccnet: Criss-cross attention for semantic segmentation”. In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 603–612.

[83] Xilinx Inc. *UltraScale Architecture*. June 2015. URL: https://www.xilinx.com/products/technology/ultrascale.html.

[84] Chenfanfu Jiang et al. "Configurable 3D Scene Synthesis and 2D Image Rendering with Per-Pixel Ground Truth Using Stochastic Grammars". In: *Int. J. Comput. Vision* 126.9 (Sept. 2018), pp. 920–941. ISSN: 0920-5691. DOI: 10.1007/s11263-018-1103-5. URL: https://doi.org/10.1007/s11263-018-1103-5.

[85] Tommi Junttila. *Tools for Constrained Boolean Circuits*. 2000. URL: http://users.ics.aalto.fi/tjunttil/circuits/index.html.

[86] Tommi A Junttila and Ilkka Niemelä. "Towards an efficient tableau method for Boolean circuit satisfiability checking". In: *International Conference on Computational Logic*. Springer. 2000, pp. 553–567.

[87] Sagar Karandikar et al. "FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud". In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA '18. Los Angeles, California: IEEE Press, 2018, pp. 29–42. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00014. URL: https://doi.org/10.1109/ISCA.2018.00014.

[88] Angelos Katharopoulos et al. "Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 5156–5165. URL: http://proceedings.mlr.press/v119/katharopoulos20a.html.

[89] Angelos Katharopoulos et al. "Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 13–18 Jul 2020, pp. 5156–5165. URL: http://proceedings.mlr.press/v119/katharopoulos20a.html.

[90] Q. Ke et al. “A New Representation of Skeleton Sequences for 3D Action Recognition”. In: (2017), pp. 4570–4579. DOI: 10.1109/CVPR.2017.486.

[91] Kristian Kersting et al. “Benchmark data sets for graph kernels, 2016”. In: *URL http://graphkernels. cs. tu-dortmund. de* (2016).

[92] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[93] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2017. arXiv: 1412.6980 [cs.LG].

[94] Diederik P. Kingma and Max Welling. “Auto-Encoding Variational Bayes”. In: *CoRR* abs/1312.6114 (2013).

[95] Thomas N Kipf and Max Welling. “Variational Graph Auto-Encoders”. In: *NIPS Workshop on Bayesian Deep Learning* (2016).

[96] Thomas N. Kipf and Max Welling. “Semi-Supervised Classification with Graph Convolutional Networks”. In: *Proceedings of the 5th International Conference on Learning Representations* (ICLR). ICLR ’17. Palais des Congrès Neptune, Toulon, France, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[97] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. “CIFAR-10 (Canadian Institute for Advanced Research)”. In: (). URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[98] HT Kung, Bradley McDanel, and Sai Qian Zhang. “Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks”. In: *2018 24th International Conference on Pattern Recognition (ICPR).* IEEE. 2018, pp. 1006–1011.

[99] HT Kung, Bradley McDanel, and Sai Qian Zhang. “Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* 2019, pp. 821–834.

[100] Andrew Lavin. “Fast Algorithms for Convolutional Neural Networks”. In: CVPR ’16 (2016), pp. 4013–4021.

[101] Yann LeCun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[102] Zhendong Lei, Shaowei Cai, and Chuan Luo. “Extended Conjunctive Normal Form and An Efficient Algorithm for Cardinality Constraints”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020.* Ed. by Christian Bessiere. ijcai.org, 2020, pp. 1141–1147. DOI: `10.24963/ijcai.2020/159`. URL: `https://doi.org/10.24963/ijcai.2020/159`.

[103] Maosen Li et al. “Actional-Structural Graph Convolutional Networks for Skeleton-Based Action Recognition”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* June 2019.

[104] Yujia Li et al. “Gated graph sequence neural networks”. In: *arXiv preprint arXiv:1511.05493* (2015).

[105] Yujia Li et al. “Graph matching networks for learning the similarity of graph structured objects”. In: *arXiv preprint arXiv:1904.12787* (2019).

[106] Shengwen Liang et al. “EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks”. In: *IEEE Transactions on Computers* (2020).

[107] Hong Liu, Juanhui Tu, and Mengyuan Liu. “Two-Stream 3D Convolutional Neural Network for Skeleton-Based Action Recognition”. In: (May 2017).

[108] Jun Liu et al. “Ntu rgb+ d 120: A large-scale benchmark for 3d human activity understanding”. In: *IEEE transactions on pattern analysis and machine intelligence* 42.10 (2019), pp. 2684–2701.

[109] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: *ECCV ’16.* 2016.

[110] Ziyu Liu et al. "Disentangling and Unifying Graph Convolutions for Skeleton-Based Action Recognition". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 143–152.

[111] Joao Marques-Silva and Vasco Manquinho. "Towards More Effective Unsatisfiability-Based Maximum Satisfiability Algorithms". In: *Theory and Applications of Satisfiability Testing – SAT 2008*. Ed. by Hans Kleine Büning and Xishun Zhao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 225–230. ISBN: 978-3-540-79719-7.

[112] Mehdi Mirza and Simon Osindero. "Conditional Generative Adversarial Nets". In: *CoRR* abs/1411.1784 (2014).

[113] Matthew W Moskewicz et al. "Chaff: Engineering an efficient SAT solver". In: *Proceedings of the 38th annual Design Automation Conference*. 2001, pp. 530–535.

[114] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. "Learning convolutional neural networks for graphs". In: *International conference on machine learning*. 2016, pp. 2014–2023.

[115] M. O'Connor et al. "Fine-Grained DRAM: Energy-Efficient DRAM for Extreme Bandwidth Systems". In: *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2017, pp. 41–54.

[116] Yutaro Omote, Akihiro Tamura, and Takashi Ninomiya. "Dependency-Based Relative Positional Encoding for Transformer NMT". In: *Proceedings of the International Conference on Recent Advances in Natural Language Processing (RANLP 2019)*. Varna, Bulgaria: INCOMA Ltd., Sept. 2019, pp. 854–861. DOI: `10.26615/978-954-452-056-4_099`. URL: `https://www.aclweb.org/anthology/R19-1099`.

[117] Angshuman Parashar et al. "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017,

pp. 27–40. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080254. URL: http://doi.acm.org/10.1145/3079856.3080254.

[118] Adam Paszke et al. “Automatic differentiation in pytorch”. In: (2017).

[119] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[120] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems*. 2019, pp. 8026–8037.

[121] Chiara Plizzari, Marco Cannici, and Matteo Matteucci. “Spatial temporal transformer network for skeleton-based action recognition”. In: *arXiv preprint arXiv:2008.07404* (2020).

[122] Qi Zhu et al. “SAT sweeping with local observability don’t-cares”. In: *2006 43rd ACM/IEEE Design Automation Conference*. 2006, pp. 229–234. DOI: 10.1109/DAC.2006.229206.

[123] Eric Qin et al. “Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 58–70.

[124] Alec Radford, Luke Metz, and Soumith Chintala. “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”. In: *CoRR* abs/1511.06434 (2015).

[125] Alec Radford et al. “Improving language understanding by generative pre-training”. In: (2018).

[126] Prajit Ramachandran, Barret Zoph, and Quoc V Le. “Searching for activation functions”. In: *arXiv preprint arXiv:1710.05941* (2017).

[127] Jeff Rasley et al. “DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters”. In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. KDD ’20. Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 3505–3506. ISBN: 9781450379984. DOI: 10.1145/3394486.3406703. URL: https://doi.org/10.1145/3394486.3406703.

[128] Ivan Ratković et al. “A Fully Parameterizable Low Power Design of Vector Fused Multiply-Add Using Active Clock-Gating Techniques”. In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. ISLPED ’16. San Francisco Airport, CA, USA: Association for Computing Machinery, 2016, pp. 362–367. ISBN: 9781450341851. DOI: 10.1145/2934583.2934587. URL: https://doi.org/10.1145/2934583.2934587.

[129] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. http://pjreddie.com/darknet/. 2013–2016.

[130] Adam Santoro et al. “A simple neural network module for relational reasoning”. In: *arXiv preprint arXiv:1706.01427* (2017).

[131] Mona Mona Mona Najafi Sarpiri et al. “A Hybrid Method for Spammer Detection in Social Networks by Analyzing Graph and User Behavior.” In: *JCP* 13.7 (2018), pp. 823–829.

[132] Franco Scarselli et al. “Computational Capabilities of Graph Neural Networks”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 81–102. DOI: 10.1109/TNN.2008.2005141.

[133] Bart Selman, Henry A Kautz, Bram Cohen, et al. “Local search strategies for satisfiability testing.” In: ().

[134] Daniel Selsam and Nikolaj Bjørner. “Guiding high-performance SAT solvers with unsat-core predictions”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2019, pp. 336–353.

[135] Daniel Selsam et al. “Learning a SAT solver from single-bit supervision”. In: *arXiv preprint arXiv:1802.03685* (2018).

[136] Amir Shahroudy et al. “Ntu rgb+ d: A large scale dataset for 3d human activity analysis”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 1010–1019.

[137] E. Shelhamer, J. Long, and T. Darrell. “Fully Convolutional Networks for Semantic Segmentation”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.4 (Apr. 2017), pp. 640–651. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2016.2572683.

[138] Zhuoran Shen et al. *Efficient Attention: Attention with Linear Complexities*. 2020. arXiv: 1812.01243 [cs.CV].

[139] Zhuoran Shen et al. “Efficient Attention: Attention with Linear Complexities”. In: *WACV*. IEEE, 2021.

[140] L. Shi et al. “Skeleton-Based Action Recognition With Directed Graph Neural Networks”. In: *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 7904–7913. DOI: 10.1109/CVPR.2019.00810.

[141] Lei Shi et al. “Skeleton-based action recognition with directed graph neural networks”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 7912–7921.

[142] Lei Shi et al. “Two-Stream Adaptive Graph Convolutional Networks for Skeleton-Based Action Recognition”. In: *CVPR*. 2019.

[143] Vighnesh Shiv and Chris Quirk. “Novel positional encodings to enable tree-based transformers”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: `https://proceedings.neurips.cc/paper/2019/file/6e0917469214d8fbd8c517dcdc6b8dcf-Paper.pdf`.

[144] David I Shuman et al. “The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains”. In: *IEEE signal processing magazine* 30.3 (2013), pp. 83–98.

[145] João P. Marques Silva and Karem A. Sakallah. “GRASP—a New Search Algorithm for Satisfiability”. In: ICCAD ’96. San Jose, California, USA: IEEE Computer Society, 1997, pp. 220–227. ISBN: 0818675977.

[146] K. Simonyan and A. Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *CoRR* abs/1409.1556 (2014).

[147] Yi-Fan Song et al. “Stronger, Faster and More Explainable: A Graph Convolutional Baseline for Skeleton-Based Action Recognition”. In: *Proceedings of the 28th ACM International Conference on Multimedia (ACMMM)*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1625–1633. ISBN: 9781450379885. DOI: `10.1145/3394171.3413802`. URL: `https://doi.org/10.1145/3394171.3413802`.

[148] M. Song et al. “Towards Efficient Microarchitectural Design for Accelerating Unsupervised GAN-Based Deep Learning”. In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 66–77. DOI: `10.1109/HPCA.2018.00016`.

[149] Naveen Suda et al. “Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey, California, USA: ACM, 2016, pp. 16–25. ISBN: 978-1-4503-3856-1.

[150] Yizhou Sun et al. "Pathsim: Meta path-based top-k similarity search in heterogeneous information networks". In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 992–1003.

[151] Vivienne Sze et al. "Hardware for machine learning: Challenges and opportunities". In: *2018 IEEE Custom Integrated Circuits Conference (CICC)* (2017), pp. 1–8.

[152] Christian Szegedy et al. "Going Deeper with Convolutions". In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842.

[153] Yi Tay et al. *Efficient Transformers: A Survey*. 2020. arXiv: 2009.06732 [cs.LG].

[154] Yao-Hung Hubert Tsai et al. "Transformer Dissection: An Unified Understanding for Transformer's Attention via the Lens of Kernel". In: *EMNLP*. ACL, 2019.

[155] Ashish Vaswani et al. "Attention is All You Need". In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS'17. Long Beach, California, USA: Curran Associates Inc., 2017, pp. 6000–6010. ISBN: 9781510860964.

[156] Ashish Vaswani et al. "Attention is all you need". In: *arXiv preprint arXiv:1706.03762* (2017).

[157] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[158] Petar Veličković et al. "Graph Attention Networks". In: *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=rJXMpikCZ.

[159] Petar Veličković et al. "Graph Attention Networks". In: *International Conference on Learning Representations* (2018). URL: https://openreview.net/forum?id=rJXMpikCZ.

[160] Petar Veličković et al. "Graph attention networks". In: *arXiv preprint arXiv:1710.10903* (2017).

[161] R. Vemulapalli, F. Arrate, and R. Chellappa. “Human Action Recognition by Representing 3D Skeletons as Points in a Lie Group”. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 588–595. DOI: 10.1109/CVPR.2014.82.

[162] Chun Wang et al. “MGAE: marginalized graph autoencoder for graph clustering”. English. In: *CIKM’17 - Proceedings of the 2017 ACM Conference on Information and Knowledge Management*. Ed. by Mark Sanderson, Ada Fu, and Jimeng Sun. United States of America: Association for Computing Machinery (ACM), 2017, pp. 889–898. DOI: 10.1145/3132847.3132967.

[163] Hongwei Wang et al. “GraphGAN: Graph Representation Learning with Generative Adversarial Nets”. In: *AAAI*. 2017.

[164] J. Wang et al. “Mining actionlet ensemble for action recognition with depth cameras”. In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 1290–1297. DOI: 10.1109/CVPR.2012.6247813.

[165] Jie Wang, Licheng Guo, and Jason Cong. “AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA”. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 93–104. ISBN: 9781450382182. DOI: 10.1145/3431920.3439292. URL: https://doi.org/10.1145/3431920.3439292.

[166] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting Tim Cheng. *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[167] Runzhong Wang, Junchi Yan, and Xiaokang Yang. “Learning combinatorial embedding networks for deep graph matching”. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 3056–3065.

[168] Xiaolong Wang et al. “Non-local neural networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 7794–7803.

[169] Xuechao Wei et al. “Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. Austin, TX, USA: ACM, 2017, 29:1–29:6. ISBN: 978-1-4503-4927-7.

[170] Wikipedia contributors. *Maximum satisfiability problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-May-2021]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Maximum_satisfiability_problem&oldid=990994650`.

[171] Wikipedia contributors. *Sparse matrix — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-April-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Sparse_matrix&oldid=950481632`.

[172] Wikipedia contributors. *Strassen algorithm — Wikipedia, The Free Encyclopedia*. [Online; accessed 17-April-2020]. 2020. URL: `https://en.wikipedia.org/w/index.php?title=Strassen_algorithm&oldid=944214439`.

[173] Shmuel Winograd. *Arithmetic complexity of computations*. Vol. 33. CBMS-NSF Regional Conference Series in Applied Mathematics. Philadelphia: Society for Industrial and Applied Mathematics, 1980. ISBN: 0–89871–163–0.

[174] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: *CoRR* abs/1901.00596 (2019). arXiv: `1901.00596`. URL: `http://arxiv.org/abs/1901.00596`.

[175] Chunyu Xie et al. “Memory Attention Networks for Skeleton-based Action Recognition”. In: July 2018, pp. 1639–1645. DOI: `10.24963/ijcai.2018/227`.

[176] Dawen Xu et al. “FCN-engine: Accelerating Deconvolutional Layers in Classic CNN Processors”. In: *Proceedings of the International Conference on Computer-Aided Design*. ICCAD ’18. San Diego, California: ACM, 2018, 22:1–22:6. ISBN: 978-1-4503-5950-4. DOI: `10.1145/3240765.3240810`. URL: `http://doi.acm.org/10.1145/3240765.3240810`.

[177] Keyulu Xu et al. “How Powerful are Graph Neural Networks?” In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=ryGs6iA5Km.

[178] Keyulu Xu et al. “How powerful are graph neural networks?” In: *arXiv preprint arXiv:1810.00826* (2018).

[179] M. Yan et al. “HyGCN: A GCN Accelerator with Hybrid Architecture”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 15–29.

[180] Sijie Yan, Yuanjun Xiong, and Dahua Lin. “Spatial temporal graph convolutional networks for skeleton-based action recognition”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 2018.

[181] Amir Yazdanbakhsh et al. “FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (2018), pp. 65–72.

[182] Amir Yazdanbakhsh et al. “GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks”. In: *Proceedings of the 45th Annual International Symposium on Computer Architecture*. ISCA ’18. Los Angeles, California: IEEE Press, 2018, pp. 650–661. ISBN: 978-1-5386-5984-7. DOI: 10.1109/ISCA.2018.00060. URL: https://doi.org/10.1109/ISCA.2018.00060.

[183] Rex Ying et al. “Hierarchical Graph Representation Learning with Differentiable Pooling”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS’18. Montréal, Canada: Curran Associates Inc., 2018, pp. 4805–4815.

[184] Emre Yolcu and Barnabás Póczos. "Learning local search heuristics for boolean satisfiability". In: *Advances in Neural Information Processing Systems*. 2019, pp. 7992–8003.

[185] Yong Du, W. Wang, and L. Wang. "Hierarchical recurrent neural network for skeleton based action recognition". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1110–1118. DOI: 10.1109/CVPR.2015.7298714.

[186] Andrei Zanfir and Cristian Sminchisescu. "Deep learning of graph matching". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 2684–2693.

[187] Hanqing Zeng and Viktor Prasanna. "GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms". In: *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '20. Seaside, CA, USA: Association for Computing Machinery, 2020, pp. 255–265. ISBN: 9781450370998. DOI: 10.1145/3373087.3375312. URL: https://doi.org/10.1145/3373087.3375312.

[188] Zhiping Zeng et al. "Comparing stars: On approximating graph edit distance". In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 25–36.

[189] C. Zhang et al. "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks". In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2016, pp. 1–8. DOI: 10.1145/2966986.2967011.

[190] Chen Zhang et al. "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey, California, USA: ACM, 2015, pp. 161–170. ISBN: 978-1-4503-3315-3.

[191] Muhan Zhang and Yixin Chen. "Link Prediction Based on Graph Neural Networks". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran

Associates, Inc., 2018, pp. 5165–5175. URL: http://papers.nips.cc/paper/7763-link-prediction-based-on-graph-neural-networks.pdf.

[192] Pengfei Zhang et al. "View adaptive recurrent neural networks for high performance human action recognition from skeleton data". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2017, pp. 2117–2126.

[193] Xinyu Zhang et al. "A Design Methodology for Efficient Implementation of Deconvolutional Neural Networks on an FPGA". In: *CoRR* abs/1705.02583 (2017).

[194] Zhen Zhang and Wee Sun Lee. "Deep Graphical Feature Learning for the Feature Matching Problem". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2019, pp. 5087–5096.

[195] Zhen Zhang et al. "KerGM: Kernelized Graph Matching". In: *Advances in Neural Information Processing Systems*. 2019, pp. 3330–3341.

[196] Jie Zhou et al. "Graph Neural Networks: A Review of Methods and Applications". In: *CoRR* abs/1812.08434 (2018). arXiv: 1812.08434. URL: http://arxiv.org/abs/1812.08434.