Optimizing Multiple Object Tracking with Graph Neural Networks on a Graphcore IPU

by

Mustafa Orkun Acar

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

January 17, 2024

Optimizing Multiple Object Tracking with Graph Neural Networks on a Graphcore IPU

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Mustafa Orkun Acar

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Dr. Didem Unat (Advisor)

_____

Prof. Dr. Yücel Yemez

_____

Asst. Prof. Dr. Ayşe Yılmazer

Date: _____

ABSTRACT

Optimizing Multiple Object Tracking with Graph Neural Networks on a Graphcore IPU

Mustafa Orkun Acar

Master of Science in Computer Science and Engineering

January 17, 2024

This thesis presents a comprehensive study focused on enhancing the efficiency of MOT using GNNs, specifically by leveraging the capabilities of Graphcore's IPUs. In the realm of real-time applications such as autonomous driving, robotics, and surveillance, the ability of GNNs to effectively model complex interactions between objects is crucial. However, the computational intensity of GNNs, particularly in key message passing operations, poses significant performance bottlenecks.

Initially, I discuss the subtleties of adapting an existing PyTorch model to TensorFlow and tailoring it for IPU execution. Then, a comparative analysis was conducted between IPU and GPU by running the model on both platforms. This phase focused on evaluating the baseline performance of the model on these two computing architectures, using metrics such as average training and inference time per epoch. The findings from this phase provided a foundational understanding of the strengths and limitations inherent to each platform in handling the model training.

Subsequently, the study advanced to the implementation of optimizations specific to the IPU, focusing on enhancing the model's message passing operations that are vital for the efficiency of GNNs. The effects of these targeted IPU-centric optimizations, along with adjustments made to IPU-specific configurations, were evaluated.

ÖZETÇE

Graphcore IPU üzerinde Grafik Sinir Ağları ile Çoklu Nesne Takibini Optimize Etme

Mustafa Orkun Acar

Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans

17 Ocak 2024

Bu tez, Çoklu Nesne Takibi'nin verimliliğini Grafiksel Sinir Ağları kullanarak, özellikle Graphcore'un IPU'sunun yeteneklerinden yararlanarak artırmaya odaklanan kapsamlı bir çalışmayı sunmaktadır. Otonom sürüş, robotik ve gözetleme gibi gerçek zamanlı uygulamalar alanında, Grafiksel Sinir Ağları'nın nesneler arasındaki karmaşık etkileşimleri etkili bir şekilde modelleme yeteneği önem taşımaktadır. Ancak, özellikle mesaj iletim operasyonlarında, Grafiksel Sinir Ağları'nın hesaplama yükü, önemli performans darboğazlarına yol açmaktadır.

Başlangıçta, PyTorch ile geliştirilmiş mevcut bir Çoklu Nesne Takibi modelini TensorFlow'a uyarlamak ve IPU üzerinde çalıştırmak için yapılması gereken özelleştirmeler incelenmektedir. Ardından, model her iki platformda çalıştırılarak IPU ve GPU arasında karşılaştırmalı bir analiz yapılmıştır. Bu aşama, ortalama eğitim ve tahminleme süresi gibi metrikler kullanarak bu iki mimari üzerinde modelin temel performansını değerlendirmeye odaklanmıştır. Bu aşamada elde edilen bulgular, her iki platformun model eğitimi için güçlü yönlerini ve sınırlamalarını anlamamızı sağlamıştır.

Sonrasında, çalışma, Grafiksel Sinir Ağları'nın verimliliği için hayati öneme sahip olan mesaj iletim operasyonlarını geliştirmeye odaklanarak, IPU'ya özgü optimizasyonların uygulanmasına ilerlemiştir. Bu amaç doğrultusunda IPU tabanlı optimizasyonların etkileri, IPU'ya özgü yapılandırmalarla birlikte değerlendirilmiştir.

ACKNOWLEDGMENTS

I extend my deepest gratitude to my advisors, Prof. Didem Unat and Prof. Fatma Güney, for their invaluable guidance, support, and enriching mentorship throughout this research journey. I am truly grateful for the privilege of being under your mentorship.

To my wife, your encouragement, understanding, and patience have been my pillar of strength. Your support has made this academic pursuit possible.

Heartfelt thanks to my family for their constant encouragement. Your support has been a source of motivation and strength.

Grateful to Simula Research Laboratory, for granting access to the eX3 HPC cluster for IPU and GPU access. Special thanks to Graphcore for their technical support and assistance throughout the course of my research.

I acknowledge with appreciation the funding that this project has received from the European High-Performance Computing Joint Undertaking under grant agreement No 956213. Additionally, the project has received support from the Turkish Science and Technology Research Centre under Grant No 120N003.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

GNN	Graph Neural Networks
MOT	Multiple Object Tracking
IPU	Intelligence Processing Unit
DL	Deep Learning
ML	Machine Learning
CPU	Central Processing Unit
GPU	Graphics Processing Unit
TPU	Tensor Processing Units
IPC	Inter-Process Communication
CNNs	Convolutional Neural Networks
MIMD	Multiple Instruction Multiple Data
AI	Artificial Intelligence
MPI	Message Passing Interface
MPNs	Message Passing Networks
GCNs	Graph Convolutional Networks
RGNNs	Recurrent Graph Neural Networks
MPNNs	Message Passing Neural Networks
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Threads
TOPS	Trillion Operations Per Second
BSP	Bulk Synchronous Parallel
SOT	Single Object Tracking
TGNs	Temporal Graph Networks
STGCN	Spatio-Temporal Graph Neural Network
MLP	Multilayer Perceptron
FPS	Frames Per Second

Chapter 1

INTRODUCTION

1.1 Motivation and Contributions

The ability to track multiple objects simultaneously in real-time is crucial for various applications, including autonomous driving, robotics, and surveillance. Graph Neural Networks (GNN) have shown promising results in Multiple Object Tracking (MOT) applications due to their ability to model complex relationships between objects. However, the high computational complexity of GNNs and their dependence on message-passing operations can lead to performance issues when applied to large-scale MOT systems. This is especially true for scatter and gather operations, which are essential components of GNN-based MOT algorithms but can be computationally expensive and challenging to optimize due to inefficient memory access, significant communication overhead, irregular data patterns, and load imbalance among processing units.

The primary aim of my research is twofold: ❶ to conduct a comprehensive comparative analysis between current-generation GPUs and IPUs with respect to their training and inference performance for MOT applications and ❷ to enhance the efficiency of message-passing operations when executing a GNN-based MOT application on a Graphcore Intelligence Processing Unit (IPU), with a specific emphasis on optimizing scatter and gather operations.

The selection of IPU as the hardware for this research was based on its inherent advantages for machine learning (ML) workloads. Computation graphs are commonly used to represent the structure and flow of computations of deep learning (DL) models. These graphs enable efficient computation by organizing the computation and its order, allowing for parallelism and reducing the memory footprint.

However, the traditional design of GPUs, optimized primarily for 2D matrix operations, may pose limitations for certain tasks, necessitating the use of large data batches to attain peak performance, which is a practice that may not always be suitable and could potentially contribute to overfitting, while it's worth noting that small batches can offer a regularizing effect [Wilson and Martinez, 2003]. In contrast to the Central Processing Unit (CPU) and Graphics Processing Unit (GPU), IPUs were specifically designed to accelerate machine intelligence workloads. They offer a distinctive architectural design that facilitates efficient massive compute parallelism, working cohesively with a large memory bandwidth, which is crucial for the processing of machine intelligence tasks.

1.1.1 The Growing Popularity of GNNs and their significance in Multiple Object Tracking Applications

The limitations of existing models such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) in handling graph data have led to the proposal of GNNs [Zhou et al., 2020a, Wu et al., 2020]. GNNs have gained increasing attention in recent years [Dwivedi et al., 2023] due to their ability to model and analyze complex relationships between entities in a graph [Xu et al., 2018]. This has led to numerous applications in various domains including but not limited to traffic state prediction, user-item interaction, neural machine translation, social relationship understanding, and object tracking [Li et al., 2020, Zhou et al., 2020a, Fan et al., 2019].

An instance of the practical applications of GNNs in the domain of MOT involves the capability to track multiple objects over time throughout a video sequence. In this context, GNNs can be utilized to represent the connections between objects in the spatial domain and how they develop or change over time in the temporal domain [Li et al., 2020, Wang et al., 2021].

In addition to modeling complex relationships, one of the other key strengths of GNNs is their ability to perform discriminative feature learning, which is the process of learning features that are informative and relevant to a particular task

[Weng et al., 2020]. In the context of GNNs, discriminative feature learning involves learning node embeddings that capture relevant information about each node in the graph and their interrelationships.

The capacity of GNNs to model complex relationships and extract informative features from graphs has positioned them as a desirable choice for tracking applications. By leveraging the ability to predict future movements of entities based on their past relationships, GNNs enable efficient and effective tracking across various domains, supporting real-time decision-making. This capability presents a significant advantage in tracking applications, where timely and accurate predictions are crucial for successful tracking outcomes.

In addition to their strengths in modeling complex relationships and discriminative feature learning, GNNs also offer flexibility in their architecture design. GNNs can be designed with various types of message-passing operations, activation functions, and aggregation schemes, allowing for customization and optimization for specific tasks and datasets. This flexibility in design also enables the integration of additional information sources, such as temporal or spatial features, to enhance the performance of the model.

1.1.2 The Advantages of using an IPU for GNNs

IPU [Jia et al., 2019] is a completely new processor known for its MIMD (Multiple Instruction Multiple Data) architecture, which allows it to process multiple instructions and data concurrently. By leveraging its MIMD architecture, the IPU exhibits superior proficiency in handling sparse data structures, which are common in numerous deep learning applications [Mohan et al., 2020].

This architecture is particularly suitable for the demands of deep learning workloads which involves the simultaneous processing of multiple instructions and data points. In addition, it supports lower precision floating point operations [Louw and McIntosh-Smith, 2021], which can accelerate deep learning models while consuming less energy. These Artificial Intelligence (AI) specific features are built directly into the hardware of IPUs, allowing them to efficiently execute deep learning work-

loads and achieve high levels of performance with energy efficiency [Bilbrey et al., 2022]. The result is a powerful hardware platform that is ideal for accelerating the development and deployment of advanced AI and machine learning models.

IPU is recognized as the ideal solution for processing GNNs due to its excellent parallel processing capabilities, high memory bandwidth, scalability, and ability to efficiently process sparse data, resulting in a reduced memory usage [Helal et al., 2022]. These features make IPUs especially well-suited for handling the large and complex graphs that are common in GNNs, with billions of nodes and edges [Moe et al., 2022].

Real-time performance is a crucial requirement for various AI applications, including MOT for autonomous driving. By utilizing the IPU's parallel processing capabilities and high memory bandwidths, GNN computations can be executed with high efficiency and speed, leading to faster training and inference. This, in turn, can enable more responsive applications and faster decision-making, which are key factors for achieving high performance in AI systems.

Furthermore, the scalability of IPUs is an additional advantage that makes them suitable for processing GNNs on large-scale graphs. Graphcore's IPU-POD systems, such as the IPU-POD16 or IPU-POD64, allow for the easy scaling of IPUs by connecting multiple IPU chips together using high-speed interconnects. These systems provide an efficient way to increase the processing power of IPUs and can be used to accelerate GNN computations on graphs of varying sizes. Additionally, the Graphcore Poplar software framework provides tools for the efficient distribution of computations across multiple IPUs, enabling GNN computations to be scaled effectively on large graphs. However, in this research, my focus is on using a single IPU to process GNN computations.

1.1.3 Summary of related work

The paper [Brasó and Leal-Taixé, 2020] proposes a novel approach to solving the MOT problem using Message Passing Networks (MPNs), a type of neural network that operates on graphs. The authors argue that existing learning-based methods

for MOT have mainly focused on improving feature extraction, rather than directly learning the data association step, which is a critical challenge in the MOT problem.

The authors propose a novel approach to improve the accuracy of MOT systems by combining two tasks: learning features for MOT and providing a solution by reasoning globally over a set of detections. To achieve this, they introduce a differentiable framework based on MPNs that can directly predict the final partitions of the graph into trajectories, without relying on existing solvers or pairwise costs. By combining deep features into high-order information across the graph, their method can account for global interactions among detections and learn in the natural MOT domain. The proposed approach achieves significant improvements in both MOTA and IDF1 on three publicly available benchmarks, demonstrating the potential for learning in MOT to be applied to the data association step and not just feature extraction. Additionally, the authors make their code available as open source, which could be beneficial to other researchers and practitioners in the field.

1.1.4 Solution and contributions

The purpose of Graphcore's IPU is to construct specialized hardware and associated software tools to improve the performance of AI workloads. This research seeks to assess the capabilities of the IPU by conducting a benchmark of a particular MOT application. This evaluation aims to investigate various aspects of the IPU, such as its advantages, limitations, adaptability, and potential to deliver promised performance gains as compared to GPU technology, by investigating ways to achieve efficient execution.

The Poplar SDK, which supports both TensorFlow and PyTorch, is used to facilitate the implementation of deep learning models on IPUs, thereby eliminating the need for users to learn a new API. This higher-level development approach has proven sufficient for most cases, as Graphcore provides optimized implementations of existing methods in the supported libraries. However, certain library-provided functions may not be implemented in the Graphcore system, or the existing implementation may not meet the required efficiency standards. Custom kernels may

need to be developed for such scenarios to optimize the performance. Thus, while it may be theoretically feasible to transfer a TensorFlow or PyTorch project directly onto the IPU platform, this may not always result in optimal performance in practice. Therefore, this study focuses on investigating the performance optimization of a GNN-based MOT system, which represent a significant class of real-world AI applications. Moreover, while previous works have investigated GNN-based MOT systems on conventional hardware, no studies have been conducted on the IPU platform, as far as the authors are aware. As such, this study presents the first attempt to train GNN-based MOT systems on IPUs, thereby expanding the body of knowledge in the field of AI hardware acceleration.

In this study, I opted to implement the model designed by the authors of "Learning a Neural Solver for Multiple Object Tracking" [Brasó and Leal-Taixé, 2020] using TensorFlow. While PyTorch was used by the authors in their original implementation, I chose TensorFlow as I was able to find more practical implementation examples for IPU at the time of implementation. This implementation serves as a benchmark to evaluate the performance of the IPU platform for the GNN-based MOT application.

In summary, I conducted a comparison of IPUs with GPUs for training a GNN for MOT problem. I migrated an existing PyTorch implementation to TensorFlow for IPU execution. My findings demonstrate the superior performance of IPUs over GPUs, particularly in scenarios with smaller batch sizes. Additionally, I identified significant performance improvements achieved through device-optimized functions.

1.2 Thesis Organization

The thesis is organized as follows: In Chapter 2, the technical background of GNNs, MOT and Graphcore IPU, along with the challenges in the MOT domain and also challenges associated with using GNNs. Chapter 3 presents the general overview of previous studies related to our research on MOT (3.1), GNN methods (3.2), GNN methods for object tracking (3.2.1), and their applications on GPU (3.2.2) and IPU (3.2.3). Chapter 4 reveals the proposed methodology in detail. Chapter 5 presents

the evaluation results of the conducted experiments along with the dataset that we use in our experiments and finally, Chapter 6 is devoted to concluding remarks and possible future research directions.

Chapter 2

BACKGROUND

2.1 Graph Neural Networks (GNNs)

GNN [Scarselli et al., 2008] is a type of deep learning architecture specifically designed to operate on data structured as graphs. A graph, in this context, is a mathematical structure consisting of a set of nodes and edges connecting them, which represent relationships between the entities represented by the nodes. Going forward, a node's feature vector will be denoted as h_i, where i is the node's index. Likewise, an edge's feature vector will be denoted as e_{ij}, where i and j are the nodes that the edge connects.

The purpose of GNN is to understand the topology and data patterns present in a graph by encoding the features of nodes and updating the representation vector through the aggregation of neighboring nodes on the graph [Ma et al., 2019]. The core principle of GNNs is to propagate information between nodes through message passing, allowing them to capture complex dependencies and relationships within the graph [Zhou et al., 2020a]. Their ability to represent a wide range of real-world data as graphs makes them a powerful tool for a variety of applications, such as recommendation systems [Ying et al., 2018], social network analysis, image, and video understanding [Pradhyumna et al., 2021], and object detection [Wang et al., 2021].

Each node in the graph is associated with a feature vector, and edges between nodes represent relationships or connections. At each iteration, nodes receive messages from their neighbors and update their feature vectors based on those messages. This process continues until the feature vectors of all nodes have been updated to reflect the collective information of the graph. There exist various GNN variations, each with its own specific characteristics and applications.

There exist various types of graph neural networks that include Graph Convolutional Networks (GCNs), Recurrent Graph Neural Networks(RGNNs), Graph Auto-Encoder Networks, and Spatial-Temporal GNNs.

Each variant of GNNs has its unique characteristics and is suitable for diverse applications. For instance, RGNNs are well-suited for modeling dynamic graphs over time, while Spatial Convolutional Networks and Spectral Convolutional Networks are two types of GCNs that excel in processing graph data with a grid-like structure and graph signals in the frequency domain, respectively. Graph Auto-Encoder Networks are optimal for unsupervised learning and dimensionality reduction of graph data, and Spatial-Temporal GNNs are adept at capturing spatiotemporal patterns in data.

2.1.1 The Structure of GNNs

A GNN is a category of neural network architecture designed to process and analyze data structured as graphs or networks. GNNs are particularly well-suited for tasks involving interconnected data points, where relationships between data elements can significantly impact the analysis. GNNs are particularly adept at capturing intricate relationships and dependencies within these graphs, providing a powerful framework for understanding the underlying structure of interconnected data points. They exhibit resilience in handling irregular and diverse data structures, accommodating varying graph sizes, and gracefully managing noisy or incomplete information.

The structure of a GNN is determined by the graph it is operating on and the task it is trying to perform. Generally, a GNN consists of multiple layers, each of which performs message passing on the graph. The input to the GNN is the initial feature vectors of the nodes in the graph, and the output is the updated feature vectors after a certain number of iterations.

2.1.2 Common Operations in GNNs

GNNs utilize both the graph structure and the features of the nodes to infer a set of representations that encapsulate the intricate relationships among the nodes

and edges. By exploiting the topology of the graph and iteratively updating the node features using information from neighboring nodes, GNNs can capture the high-order structural and relational information present in graphs. To perform the iterative updates, GNNs employ a range of message-passing operations:

- **Convolution operator:** The convolution operator is one of the most commonly used operators in GNNs. It applies a learnable filter to the feature representations of a node and its neighbors and then aggregates the results to obtain a new feature representation for the node. This operator is inspired by CNNs and has been adapted for use with graph-structured data.

- **Scatter and gather operators:** The scatter and gather operators are used for message passing in GNNs. In a scatter operation, each node aggregates information from its neighboring nodes and computes a new feature representation for itself based on this information. This new feature representation is then passed on to the neighboring nodes in a gather operation, where each node collects the updated feature representations from its neighbors and uses them to compute its own new feature representation. Together, the scatter and gather operations enable information to be passed between nodes in a graph and allow the nodes to update their feature representations based on the information received from their neighbors.

- **Attention operator:** The attention operator is another commonly used operator in GNNs. It allows nodes to selectively attend to the most important and informative neighbors by weighting the messages received from neighboring nodes. The attention weights are computed using a learnable function that takes into account the features of the nodes and the edges connecting them. The weighted messages are then aggregated to obtain a new feature representation for the node. The attention mechanism has been shown to be effective in capturing the complex relationships between nodes in a graph and achieving state-of-the-art performance on a variety of tasks.

In addition to these operators, there are others used in GNNs, such as gated convolution, graph pooling, and edge convolution, which modify the way messages are propagated and aggregated through the graph.

2.1.3 Message Passing Neural Networks (MPNN)

MPNN [Gilmer et al., 2017] are a class of GNNs that utilize the concept of message passing to model graph-structured data. In MPNNs, a message m_{ij} is exchanged between nodes, i and j, through the edge between them, and each message is computed using a message function, f_e.

More formally, let h_i denote the feature vector of node i, and the feature vector of the edge connecting nodes i and j as h_{ij}. Then, the message m_{ij} sent from node i to node j can be computed using an MLP (Multilayer Perceptron) based message function f_e as follows:

$$m_{ij} = f_e(h_i, h_j, e_{ij}) \tag{2.1}$$

Here, the message function f_e is typically a small MLP that takes as input the feature vectors of the two nodes and the edge connecting them and outputs a new message vector that is sent to the receiving node.

After computing the messages, the MPNN aggregates them using a message passing function to update the node features. The message passing function is often defined as a permutation-invariant function, such as a sum or a max function, that aggregates the messages received by each node from its neighbors.

2.1.4 Challenges of Utilizing GNNs

Despite the remarkable achievements of GNNs in a range of tasks related to graphs, such as graph and node classification, clustering, and link prediction, they have been identified to face several obstacles. One of the challenges encountered by GNNs is the phenomenon of **over-smoothing** [Rusch et al., 2023], characterized by the gradual convergence of node representations as additional layers are added, resulting in the loss of crucial information pertaining to the initial nodes. Moreover, GNNs

may encounter the issue of **over-squashing** [Topping et al., 2021], a phenomenon whereby the non-linear activation functions within the network excessively compress node representations, resulting in the loss of significant information. Lastly, GNNs could potentially confront the challenge of **under-reaching**, in which the model is unable to sufficiently capture essential information about nodes or edges within the graph, leading to suboptimal performance in downstream tasks.

One of the fundamental aspects of GNN architectures is the message-passing paradigm. In this paradigm, information is transmitted between nodes along the edges of the graph, which serves both as the input data and the computational structure. This represents a distinct approach in contrast to conventional neural networks such as CNNs, which solely employ the input data as an input rather than as the computational structure for information transmission. However, recent studies have shown that this paradigm may not consistently yield optimal results in particular graphs and scenarios. Some graphs do not support effective message-passing, calling into question the viability of this paradigm.

The issue of over-squashing, where an excessive amount of information is condensed into a single node representation, may arise due to bottlenecks in the graph's structural characteristics. Some graphs present a less than optimal platform for effective information propagation due to inherent structural bottlenecks.

To mitigate this issue, current GNN implementations employ a technique termed "graph rewiring". This approach involves disentangling the input graph from the computational structure or refining it for computational purposes. To rewire the input graph, GNNs such as GraphSAGE, GAT, and latent graph learning techniques typically utilize neighborhood sampling, virtual nodes, connectivity diffusion, and node and edge-dropout mechanisms. Meanwhile, transformers and attention-based GNNs learn a new graph by assigning distinct weights to each edge, which can be considered as a form of rewiring. It's worth noting that in modern GNN models, information propagation often extends beyond the initial input graph structure.

2.2 Multiple Object Tracking

Multi-object tracking (MOT) is a critical task in computer vision that involves detecting, localizing, and tracking multiple objects in a video sequence. The primary objective of MOT is to estimate the trajectories of objects across frames while maintaining their identities. This requires resolving complex data association problems, where objects may undergo occlusions, interactions, and variable motion patterns.

MOT can be classified into two categories based on their approach to utilizing frames: online tracking and offline tracking. The fundamental distinction lies in the incorporation of information from future frames when processing the current frame [Xiang et al., 2015]. In the context of an online solution, trackers operate by sequentially processing frames, without access to future frames but with access to past frames for reference. This characteristic makes online tracking well-suited for time-sensitive scenarios where real-time tracking is essential. On the other hand, offline tracking allows for the utilization of a batch of frames as input, enabling the incorporation of future information from subsequent frames to predict the outcome of the current frame. This approach provides offline tracking with the advantage of leveraging a more comprehensive set of information from both past and future frames, thereby typically resulting in higher accuracy compared to online tracking.

Additionally, in the field of single-camera multi-object tracking, these methods can be classified into two main approaches based on their model structure: tracking by detection and joint detection and tracking, which are also commonly referred to as detection by tracking. The tracking-by-detection approach typically involves utilizing an object detector to locate objects in each frame of the input video. Subsequently, these detected objects are associated across frames to form tracklets based on their similarity scores. To this end, many existing multiple object tracking methods rely on separate object detection modules for initializing new tracks and updating existing tracks like SORT [Bewley et al., 2016], DeepSORT [Wojke et al., 2017], FairMOT [Zhang et al., 2021], and ByteTrack [Zhang et al., 2022]. However, tracking and detection are strongly interconnected and can benefit from each other. For example, the affinity model from the tracking method can reuse appearance

features already calculated by the detector, and the detector can leverage object information from the past to avoid missed detections [Kieritz et al., 2018]. To address this, the joint-detection-and-tracking approach came up. It tackles detection and tracking tasks simultaneously within a single framework allowing for the boosting of detection performance by leveraging tracking information. This has emerged as a recent trend in MOT research, with notable methods like JDE [Wang et al., 2020b], CenterTrack [Zhou et al., 2020b], and TrackFormer [Meinhardt et al., 2022].

2.2.1 Importance of Multiple Object Tracking

It is clear that multi-object tracking plays a significant role in computer vision-related applications, and it has received considerable attention from both academia and industry. Multi object tracking is a fundamental technology that facilitates the detection, identification, and real-time tracking of multiple objects from video or image sequences. The significance of multi-object tracking lies in its diverse range of applications including, surveillance [Elhoseny, 2020], autonomous vehicles [Rangesh and Trivedi, 2019], human-computer interaction [Kamkar et al., 2020], crowd management [Poiesi et al., 2013], and more.

One of the primary reasons for the significance of multi-object tracking is its pivotal role in surveillance. In today's world, as security becomes increasingly important, surveillance systems are essential for monitoring and detecting threats at public and secured places, such as terrorist attacks, requiring efficient surveillance systems that include embedded object tracking components [Chandrajit et al., 2016]. MOT plays a pivotal role in overcoming the limitations of object detection algorithms, while object detectors may fail when objects are occluded or overlapped by obstacles, a robust multi-object tracker can still accurately predict and track the objects in such scenarios. MOT allows for real-time and accurate tracking of objects, enabling the detection of suspicious behavior, and potential security cases. This capability enhances the reliability, effectiveness, and efficiency of surveillance systems in addressing security challenges, making multi-object tracking an indispensable technology in the field of computer vision.

Additionally, MOT is crucial for developing artificial intelligence interfaces for applications like augmented reality and virtual reality by using computer vision. As a result of accurate tracking of movements and gestures, users can interact seamlessly with virtual objects, enhancing the user experience and usability in several different domains, including gaming, education, and training. [Islam et al., 2020]

Multi-object tracking also holds significant importance in the field of autonomous driving, where it is employed to detect and predict the behavior of pedestrians, other vehicles, and obstacles. For autonomous driving to be safe, the vehicle perception system must be able to perceive the environment accurately. MOT enables autonomous vehicles to accurately track and predict the movement of objects in their environment, ensuring safe and efficient navigation. By providing real-time information about the position, velocity, and trajectory of surrounding objects, multi object tracking enables autonomous vehicles to make informed decisions and take appropriate actions. Hence, the development of an advanced object tracking algorithm is crucial to ensure accurate and efficient perception in autonomous driving scenarios [Guo et al., 2022].

Furthermore, multi-object tracking has applications in sports analytics [Cui et al., 2023], crowd management [Kumaran and Reddy, 2017], and retail analytics. In sports analytics, it is used to track players in real-time, gathering their statistics and analyzing their performance, providing in-game tactics, and developing strategies accordingly. The use of multi-object tracking for crowd management is crucial in public events, transportation hubs, and urban areas. It aids in crowd control, detects suspicious behavior, and enhances public safety in various real-world scenarios. In the context of retail analytics, it leverages MOT to monitor customer movements, behaviors, and interactions with products, as well as optimize store layouts. In addition to gaining valuable information about how customers interact with products, retailers can gain insight into how long they spend in various areas of the store by utilizing MOT. By leveraging MOT in retail analytics, retailers can provide better customer experiences, boost sales, and build loyalty among customers over the long term.

2.2.2 Challenges in Multiple Object Tracking

Compared to tracking a particular object, MOT is a more intricate process. The development of efficient and effective MOT methods is a challenging problem due to the diverse and complex nature of real-world environments.

MOT involves creating new tracked objects using detection results, re-identifying lost objects when they reappear, and terminating objects when they leave the camera's field of view. Additionally, MOT faces additional challenges such as dealing with object occlusions, id switching, changes in appearance, motion patterns, and pose changes, which are more complex compared to tracking a single object [Luo et al., 2021]. Occlusion and ID switch are two common challenges in MOT [Park et al., 2021]. When an occlusion occurs, it becomes difficult to accurately predict the current position of an object using a simple tracking algorithm, as the object may be partially or fully obscured by other objects. Additionally, when a tracker determines that an object is no longer visible in the frame due to occlusion or other factors, it may mistakenly assign a new ID to the same object when it reappears, leading to an ID switch. Both occlusion and ID switch can result in tracking errors and can significantly impact the accuracy and reliability of MOT algorithms. Overcoming these challenges requires the use of robust methods that can effectively handle occlusion and ID switch scenarios in MOT. This can be achieved through various techniques, such as incorporating contextual information, utilizing sophisticated motion models, or leveraging deep learning techniques for object representation and tracking.

One of the other primary challenges is dealing with intra-class variations in surveillance video event detection. In some events, the visual appearance of objects can exhibit significant variations, posing problems in tracking and detecting them over time due to their visual appearance [Kumaran and Reddy, 2017]. In addition to these, there are a few key challenges researchers need to address when analyzing sports footage especially, due to their fast speeds and motion blur in complex background scenes. In order to meet these challenges, researchers must focus on improving the performance of associations, particularly in these areas [Cui et al., 2023].

The primary challenge of creating an online MOT framework are developing robust associating metrics that link detections with tracks effectively, accurately identifying the optimal timing for creating new tracks, distinguishing true detections from false positives, and making informed decisions about terminating lost tracks. While in the offline MOT framework, the main challenges lie in constructing the graph and network, as well as optimizing the global labeling problem associated with them [Xu et al., 2019].

Additionally, the need to achieve fast running speeds remains a constant priority in tracking scenarios. To achieve smooth performance and responsiveness in dynamic environments where objects change appearances and move quickly, it is extremely important to enhance tracking speed, especially in real-time object tracking models [Cobos et al., 2019].

2.3 Graphcore IPU

Graphcore, a UK-based startup, unveiled the Intelligence Processing Unit (IPU) in 2017 and has since introduced the second generation devices. The Graphcore IPU stands out as an innovative and highly scalable parallel processor explicitly crafted for the acceleration of ML and AI workloads. Tailored for a spectrum of tasks including Computer Vision, Natural Language Processing (NLP), GNNs, and other advanced applications, the IPU showcases versatility in handling diverse AI challenges. Notably, its capability to effectively manage small batch sizes enhances its applicability for real-time applications, emphasizing low latency—a critical aspect for a wide range of practical use cases [Sumeet et al., 2022].

2.3.1 Structure of IPU

Graphcore's IPU diverges from the conventional SIMD (Single Instruction Multiple Data) / SIMT (Single Instruction Multiple Threads) architecture commonly employed by GPUs. Instead, it adopts a massively parallel approach known as MIMD (Multiple Instructions Multiple Data). This architecture features ultra-high bandwidth memory strategically positioned in close proximity to the processor cores. The

IPU's design proves exceptionally efficient for applications demanding irregular and sparse data access. It excels in executing individual processing threads on modest data blocks, capitalizing on its MIMD architecture [Mohan et al., 2020]. This architectural choice is particularly advantageous for graph algorithms, which inherently exhibit unpredictable and irregular memory access patterns. Such characteristics typically result in performance bottlenecks in traditional processors that rely on pre-caching mechanisms.

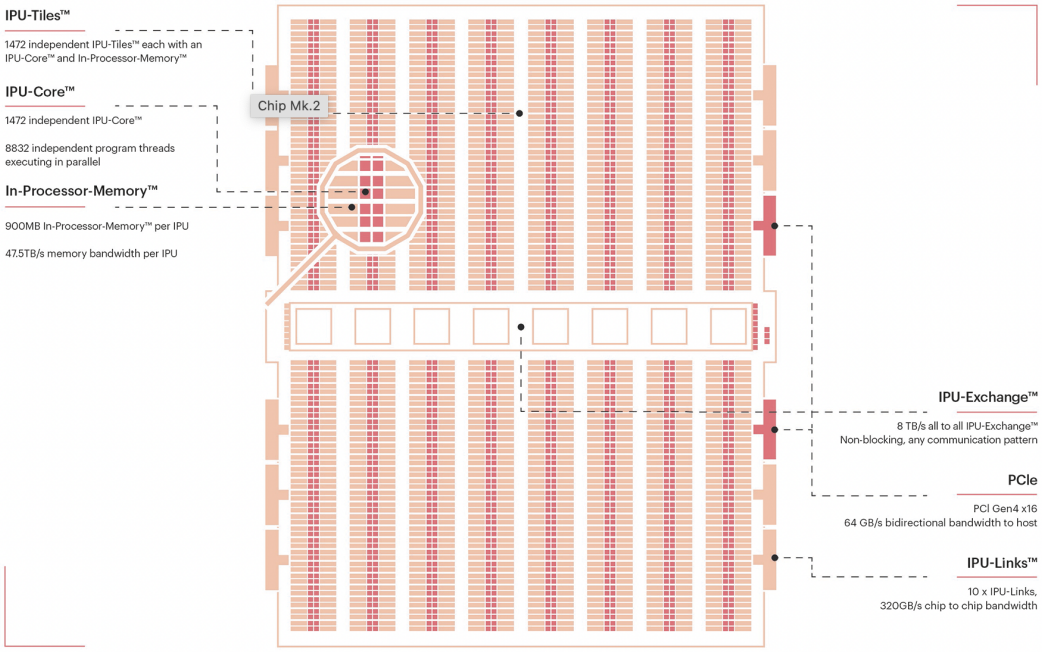


Figure 2.1: The Graphcore Colossus™ MK2 GC200 IPU [Graphcore, 2020]

The IPU comprises individual tiles, each housing a multi-threaded core and a limited amount of private SRAM. In the MK2 version, the IPU is equipped with 1472 potent processor cores, capable of concurrently executing nearly 9000 independent program threads. Notably, the original MK1 iteration, situated on the M1000 platform, featured a dual-chip design. In contrast, the more recent M2000 model represents a substantial advancement, adopting a quadruple-chip configuration. This architectural progression directly correlates with significant performance enhancements. Specifically, in the context of neural network training, the M2000 machine exhibits an impressive 7 to 9 times acceleration compared to the first-generation

Graphcore IPU, MK1. Moreover, the M2000 machine achieves an eight-fold acceleration in inference processing [Freund and Moorhead, 2020].

The utilization of TSMC's advanced 7nm manufacturing process has yielded a remarkable transistor count of 59.4 billion in the production of a high-performance computing system. This substantial transistor count empowers the system to deliver an impressive computational capability of approximately 250 trillion operations per second (TOPS) through its 1,472 cores. Complementing this computational power, the system boasts a noteworthy memory capacity, housing 900MB of SRAM. The interconnection of these components is facilitated by a high-speed, low-latency fabric, resulting in a substantial bandwidth of 2.8TB/s [Freund and Moorhead, 2020]. Moreover, the intelligent distribution of memory across the system's tiles enhances data processing efficiency. This strategic allocation ensures the optimized utilization of memory resources, enabling the IPU to navigate diverse computational tasks with precision and effectiveness.

The analysis revealed that [Mohan et al., 2020] the IPU exhibited superior training performance for GANs in contrast to the GPU. Moreover, in scenarios involving small batch sizes, the IPU outperformed the GPU by a substantial margin, exhibiting a speedup factor ranging from 4 to 5.

2.3.2 Memory Architecture

The memory architecture of the IPU is a crucial facet of its design, showcasing distinctive choices made by Graphcore to optimize performance. In particular, two key types of memory exist on IPU: on-tile memory in the form of SRAM and Streaming Memory with DDR4 technology in IPU-Machines. These memory types contribute significantly to the IPU's efficiency, especially in multi-IPU configurations.

On-Tile Memory (SRAM):

An essential design choice by Graphcore involves the utilization of SRAM as the on-tile memory, akin to cache memory. In the second generation of IPUs, each tile is equipped with 624KB of SRAM, resulting in a substantial total of 897MB across 1472 tiles. While modest on a per-tile basis, this on-tile SRAM proves significant,

potentially accommodating entire models for smaller-scale tasks. However, in the context of multi-IPU setups, such as configurations involving 16, 64, or 256 IPUs, the collective on-tile memory offers substantial capacity for handling larger models.

Streaming Memory (DDR4):

Complementing on-tile SRAM, IPU-Machines, composed of multiple IPU devices, introduce another critical memory type known as Streaming Memory. Presently sized at 448 GBs, Streaming Memory plays a crucial role in facilitating data exchange among IPUs. Utilizing DDR4 technology, this memory is directly accessible by IPUs on the machine. During the exchange stage, IPUs can efficiently access and copy data from Streaming Memory to their respective on-tile memories, enhancing the overall computational capabilities of the IPU-Machine.

2.3.3 Computing Model

At the hardware level, the IPU adopts the BSP (Bulk Synchronous Parallel) model, a robust computing paradigm designed for crafting parallel algorithms and effectively addressing issues like deadlocks in parallel software. The BSP model is structured into several stages, collectively forming a superstep:

1. Local Compute Stage: Each process autonomously executes its designated task utilizing its local memory.

2. Communication Stage: Processes engage in all-to-all communication to exchange data. In the IPU, two types of communication mechanisms are employed: DoExchange (facilitating data exchange between tiles within an IPU) and GlobalExchange (enabling data exchange between different IPUs).

3. Synchronization Barrier Stage: A global synchronization barrier ensures that all processes conclude their local computations and communication stages before transitioning to the subsequent superstep. In the IPU, two types of synchronization occur: Internal Sync (taking place between tiles of an IPU) and External Sync (occurring between different IPUs).

Graphcore strategically employs this computing model to ensure the seamless execution of highly parallel programs, minimizing the risk of deadlocks or race conditions. This model provides a structured and efficient framework for parallel algorithm design and execution on the IPU hardware.

2.3.4 IPU Machine

A single IPU is constrained by limited device memory, measuring approximately 900 MB SRAM, rendering it impractical for larger models. Instead, the design principle encourages the collaborative use of multiple IPUs to establish a collective memory pool, enhancing efficiency. The IPU-link technology, boasting a communication speed of up to 320 GB/s, facilitates seamless interaction between IPUs. In response to this collaborative paradigm, Graphcore engineered a compute platform known as the IPU-Machine. This platform serves as a fundamental building block for creating more expansive compute systems, exemplified by "the Bow Pod_{16}." The latter comprises 4 Bow-2000 IPU-Machines, each consisting of 4 Colossus Mk2 IPU devices, resulting in a total of 16 devices. Larger setups are also available, such as Pod_{256}.

2.3.5 IPU Programming

Graphcore presents Poplar, an extensive software stack meticulously crafted for the compilation and execution of programs on IPUs, complemented by seamless integration with popular deep learning frameworks like TensorFlow and PyTorch. Comprising a compiler, a host run-time, and a library collection [Vaswani et al., 2022], Poplar empowers C++ applications to construct, compile, and execute programs on the IPU, leveraging the capabilities of the Poplar graph library (libpoplar). The inclusion of the PopLibs library further enhances efficiency by providing pre-defined optimized functions, allowing developers to fully exploit IPU hardware resources and streamline their development workflows.

In tandem with Poplar, Graphcore introduces PopVision, a suite of tools designed for program analysis and performance optimization. The PopVision Graph Anal-

yser offers valuable insights into memory usage and execution details of programs executing on IPUs. Simultaneously, the PopVision System Analyser generates an event trace, capturing execution and communication events between the host and IPUs. These robust analysis tools play a pivotal role in assisting developers, providing a nuanced understanding of program behavior, and facilitating the fine-tuning of programs for optimal performance on Graphcore IPUs.

By combining novel floating-point technologies with IPU processor cores, Graphcore's AI-Float enhances IPU processor performance. It supports sparse operations with floating-point arithmetic and includes library support for a variety of sparse operations. In this way, training and inference can be performed more efficiently on sparse data, resulting in fewer parameters, faster training times, and lower energy consumption [Graphcore, 2020].

2.3.5.1 Looping Utilities

In TensorFlow for IPU, akin to the standard TensorFlow version, there are specialized constructs designed to optimize the efficiency of training or inference loops. To optimize training on the IPU with TensorFlow, it's essential to encapsulate the training operations within a loop. This approach is necessary because executing the training code separately for each batch of the training dataset would incur the overhead of frequent control transfers between the host and the IPU. The training loop can be created using `tensorflow.python.ipu.loops` loop construct. Conventionally, one iteration of the loop involves utilizing `session.run()`. However, the use of the `tensorflow.python.ipu.loops` loop construct introduces an enhancement by enabling a single invocation of `session.run()`, thereby mitigating the overhead associated with multiple calls. Therefore, by employing a training loop, we can execute training operations iteratively without the need to relinquish control back to the host.

In the typical TensorFlow execution flow, operations outside of loops expect a fixed number of tensors as input. However, when a training operation is embedded within a loop, the input data must be structured as streams of values. Traditional

TensorFlow Python feed dictionaries are not designed for delivering data in this format. Therefore, when conducting training within a loop, it becomes necessary to feed data from a TensorFlow dataset. Likewise, the outputs should be formatted as streams. TensorFlow for IPU offers two mechanisms, namely infeed queues and outfeed queues, to facilitate this process when utilizing the IPU. Serving as a wrapper around the `tf.Dataset` object, the `IPUInfeedQueue` seamlessly integrates infeed operations, creating a queue from the host device to the IPU. This mechanism empowers the IPU to retrieve dataset elements as needed. Programmatically, the implementation is simplified by passing the `IPUInfeedQueue` instance to a loop generated using `tensorflow.python.ipu.loops`. This streamlined setup effortlessly handles the iterative fetching of input data without the computational burden of calling `session.run` at each step.

2.3.6 Host to Device Communication in IPU

To harness the full potential of any hardware accelerator, it is crucial to optimize input/output (I/O) performance. In TensorFlow for IPU, two primary constructs facilitate device-to-host communication: the *IPUInfeedQueue* and *IPUOutfeedQueue*. These classes encapsulate the functionality needed to add infeed and outfeed enqueue/dequeue operations into the computational graph, enabling the communication between the device and the host. They serve the purpose of fetching data for training loop input and transmitting outputs back to the host device.

IPU provides 3 mechanisms to elevate I/O performance further:

1. **Prefetching Elements:** Prefetching elements involves strategically positioning data in logical proximity to the IPU before it is required by the running program, utilizing resources like Streaming Memory. This optimization enables the IPU to access the data swiftly when needed, surpassing the speed of accessing data currently residing on the host. The prefetch depth, a parameter configurable when creating the IPUInfeedQueue, governs the quantity of dataset elements moved to the Streaming Memory in each step. The PopVision tool facilitates the analysis of cycle allocation to various operations in BSP

steps, such as OnTileExecute and DoExchange, as well as I/O operations like StreamCopyBegin. Elevated percentages of cycles spent on StreamCopyBegin operations indicate potential I/O-related program execution delays, particularly in the data availability to the running program. In such scenarios, adjusting the prefetch_depth to a value higher than 1 can be useful.

2. **Utilizing I/O Tiles:** An I/O tile is an IPU tile exclusively dedicated to executing I/O operations without engaging in the actual computations required by the running program. Adjusting this parameter involves striking a balance between reducing the number of tiles participating in the computation when set to a higher value and ensuring sufficient memory capacity to accommodate the transferred tensors into the tile memory when set too low. This parametrization plays a critical role in optimizing the efficiency of I/O operations while managing computational resources effectively.

2.3.7 Using DL Frameworks with IPU

Developers can leverage high-level support for TensorFlow and PyTorch when working with IPUs. To further streamline development, the PopLibs library is available within the Poplar SDK and on GitHub. This library incorporates a wide range of pre-defined functions, including linear algebra operations, element-wise tensor operations, non-linearities, and reductions, all optimized for efficient execution on the IPU. A high-level description of the preferred AI framework, such as TensorFlow or PyTorch, is used to construct the complete compute graph, including computation, data, and communication elements. This compute graph is then compiled and runtime programs are generated that effectively manage compute operations, memory, and networking. This approach ensures the optimal utilization of IPU hardware resources, harnessing their full potential.

2.3.8 Profiling IPU programs

Graphcore provides comprehensive profiling capabilities for IPU applications. To leverage these, users must initiate the capture of profiling reports during program

execution. This is achieved by setting the `POPLAR_ENGINE_OPTIONS` environment variable to a JSON string. For example, executing a program with `POPLAR_ENGINE_OPTIONS='{"autoReport.all":"true", "autoReport.directory":"./desired_report_path/", "debug.allowOutOfMemory":"true"}' python3 application.py` enables the capture of profiling data. This variable allows users to specify the directory for the profiling report and to set specific parameters, such as `allowOutOfMemory`. This particular parameter instructs the IPU to proceed with the compilation process for profiling purposes, even if it determines that the available memory is insufficient for executing the program.

Upon completion of execution, users can retrieve the report from the specified directory. The report typically comprises several files, including a large profile.pop file containing the profiling data. For a medium-sized model profiled over 100 epochs, the report file can be several tens of gigabytes in size.

There are two primary methods for analyzing these reports:

PopVision Graph Analyser GUI Tool: This user-friendly tool allows for the opening of local and remote report files (via SSH). It displays various reports like Memory Report, Liveness Report, Program Tree, Operations Summary, Operations Graph, and Execution Trace. Of these, the Execution Trace is the most resource-intensive, often requiring significant system memory to load. This report can be prohibitively large for models beyond a basic complexity, with file sizes reaching tens of gigabytes. In practice, opening such extensive reports can consume more memory than the file's disk size. In some instances, attempting to load an Execution Trace report can lead to excessively high memory usage without successful loading, possibly indicating a memory leak. This issue typically occurs during the "Creating BSP stats, processing data" stage. I observed this issue in reports generated using Poplar SDK version 3.0 and later. Opening smaller reports tends to be more stable.

PopVision Analysis Library (libpva): As part of the Poplar SDK, libpva provides a programmatic approach to analyze profiling reports. As an example, it enables users to iterate over execution steps and gather detailed information on the cycles spent by IPU tiles. This Python library serves as a robust alternative to the GUI tool, facilitating the development of reusable profiling scripts.

In summary, the advanced profiling capabilities offered by Graphcore are indispensable for identifying and addressing performance and memory issues in IPU applications. The detailed insights provided by the profiling reports are invaluable for optimizing computational and memory usage.

Chapter 3

LITERATURE REVIEW

In this section we provide a comprehensive review of deep learning-based methods for MOT. Firstly, we discuss existing MOT methods that utilize deep learning techniques. Then, we explore how GNNs have enhanced the performance of MOT. Finally, we examine the current methods that take into account hardware considerations for GPU and IPU to enhance the tracking performance even further.

3.1 Multiple Object Tracking Methods

The development of deep learning-based methods, which are capable of automatically extracting high-level features from input frames using CNNs, has significantly advanced the field of MOT.

In the field of tracking algorithms, there has been a greater focus on investigating the tracking of individual objects rather than multi-object tracking. Li et al.[Li et al., 2018] provide a comprehensive overview and comparative analysis of the most advanced deep learning-based methods for tracking a single object. A tracking benchmark [Wu et al., 2013] has been created to enable fair comparisons and evaluations of tracking performance in diverse environments, with accompanying experimental evaluations conducted [Smeulders et al., 2013] to assess the effectiveness of various tracking techniques. By combining the discriminative power of Single Object Tracking (SOT) effectively and efficiently, Zheng et al. [Zheng et al., 2021] proposed a novel end-to-end trainable MOT architecture. Their approach extends the CenterNet [Zhou et al., 2019] detector by incorporating a SOT branch in parallel with the object detection branch. Notably, their SOT branch trains a separate model for each target online, enabling specific discrimination. These trained SOT models then perform object association in the subsequent frame, leading to more

efficient online learning and tracking compared to using multiple SOT models directly in MOT. Later on, Zhang et al. [Zhang et al., 2021] introduced FairMOT, a MOT system built on top of CenterNet, which addresses the fairness issue with new discoveries that are novel and valuable for the field of MOT. They showed that FairMOT achieves high levels of detection and tracking accuracy, outperforming previous state-of-the-art methods by a significant margin on multiple datasets, including 2DMOT15, MOT16, MOT17, and MOT20.

The survey article [Xu et al., 2019] examines the use of deep neural networks for multi-object tracking and discusses three main approaches: (i) extracting semantic features using pre-trained deep networks, (ii) designing the core of the tracking framework with a deep neural network, and (iii) designing deep networks for end-to-end tracking. The article highlights the effectiveness of deep networks for tracking when used in different ways.

Apart from the accuracy of the tracking results, achieving fast running speeds remains a high priority for tracking. To this end, Wang et al. [Wang et al., 2020b] presented JDE, a novel MOT system that integrates target detection and appearance feature learning in a shared model. Their design effectively reduces the runtime of the MOT system, enabling it to operate at (near) real-time speeds. Despite the accelerated processing, their proposed system achieves comparable tracking accuracy with state-of-the-art online MOT methods.

In recent years, tracking using Siamese networks has demonstrated promising performance. Liu et al. [Liu et al., 2019] proposed a Siamese network with an auto-encoding constraint to extract features for objects on the scene, and introduced a composite feature called PAN to better describe the sequential features of tracklets. Following that, Shuai et al. [Shuai et al., 2020] introduced an integrated network architecture, Siamese Track-RCNN, that combines detection and association in a single forward pass, achieving efficiency and accuracy. Their study showed that sharing a CNN backbone among three branches - detection, tracking, and re-identification - results in low computational cost, memory footprint, and increased accuracy through complementary functions. Xu et al. [Xu et al., 2020] proposed DeepMOT which trains Siamese trackers along with other components under the MOT training frame-

work by mainly focusing on improving structured loss in MOT and they achieved a new state-of-the-art result on MOTChallenge benchmark datasets. By leveraging their study, Shuai et al. [Shuai et al., 2021] presented an integrated end-to-end multi-object tracking network, SiamMOT, using Siamese trackers. SiamMOT unifies detector and tracker in a single network, unlike its closest rival, DeepMOT and they outperformed the state-of-the-art.

Despite their encouraging results, Siamese methods have limitations in fully leveraging spatial-temporal target appearance modeling when applied to complex situations where objects are occluded or remain invisible for multiple frames, resulting in unsatisfactory tracking outcomes [Yu et al., 2022]. In addition to that, CNNs are not well-suited for processing graph data due to their arbitrary and complex topology, which results in a lack of spatial locality. Moreover, the lack of a fixed node ordering further complicates the use of CNNs for graph data processing. These factors make it difficult to apply CNNs directly to graph data, and as an alternative approach, GNNs have been developed to address these challenges.

3.2 GNNs

GNNs were first introduced and named in a 2009 paper [Scarselli et al., 2008]. They proposed a graph neural network that takes a directed graph as input, where nodes and edges have associated static feature vectors. Each node has a state vector that is recursively updated using information from neighboring nodes and edges, and a parametric output function computes the final output for a node. Their potential was later demonstrated in 2017 with the introduction of a variant called the Graph Convolutional Network (GCN) [Kipf and Welling, 2016], where they have applied the convolution filters directly on the graph nodes and their neighbors, has since become one of the most widely used GNNs. The concepts and the application of GNN are summarized by [Battaglia et al., 2018]. Furthermore, the advantages, disadvantages, and potential improvements of GNNs is conducted in [Gao and Hao, 2021].

Message Passing Neural Networks (MPNNs), as proposed in [Gilmer et al., 2017],

involve each node aggregating feature vectors from its neighbors to compute its new feature vector. The MPNN is characterized by the authors as a versatile and generalized framework for message-based GNN computation. The explanation emphasizes that MPNNs have the capacity to articulate numerous GNN architectures proposed in the existing literature, positioning themselves as a comprehensive and overarching architectural paradigm. This recognition underscores the MPNN's efficacy in encapsulating a broad spectrum of graph-based learning methodologies, thereby establishing it as a foundational and unifying structure in the field.

3.2.1 GNN Methods for Object Tracking

GNNs are a type of machine learning approach that has exhibited promise in addressing challenging problems involving graph structures, especially in the domain of computer vision [Tang et al., 2022]. Chen et al. [Chen et al., 2022] presented a comprehensive survey of GNNs in computer vision from a task-oriented perspective. They categorized the algorithms into five groups based on the modality of input data. They provided discussions on key innovations, limitations, and potential research directions for helping to obtain new insights towards human-like visual understanding. There has been a growing interest in applying GNNs to various subfields of computer vision, including human action recognition [Li et al., 2021, Feng and Meunier, 2022], visual question answering [Sharma and Jalal, 2021], object detection [Shi and Rajkumar, 2020], person re-identification [Shen et al., 2018], single object tracking [Gao et al., 2019], and multiple object tracking [Li et al., 2020, Wang et al., 2021]. Among these, GNN-based approaches have demonstrated remarkable success in the task of MOT [Wang et al., 2020a].

GNN-based tracking techniques in MOT use graph representations of objects and incorporate temporal information to capture motion patterns. This approach yields more precise and resilient tracking performance, enhancing the efficiency and accuracy of MOT algorithms. In pursuit of this goal, researchers have endeavored to reformulate the MOT problem within a differentiable GNN framework. This task involves overcoming several challenges, such as developing effective graph construc-

tion techniques and establishing appropriate loss functions to ensure the model's differentiability. The ultimate goal is to exploit the potential of GNNs to enhance the precision and efficiency of object tracking, facilitating broader applications in this domain.

Ma et al. [Ma et al., 2019] treated bounding boxes as nodes, and CNN and Motion Encoder are used to extract motion and appearance features. In order to learn the relationships between the nodes, an adjacency matrix is constructed using the cosine similarity of node embeddings, and the concatenated features are fed into a GNN. Those within the same range are considered the same person and linked sequentially. Gao et al. [Gao et al., 2019] introduced GCT, an end-to-end graph convolutional tracking framework for visual tracking, which outperformed state-of-the-art trackers on five benchmarks and runs in real time. They utilized GCNs in a Siamese network for spatial-temporal target appearance modeling and context-guided adaptive learning. Later on, a MOT framework is proposed by Wang et al. [Wang et al., 2020a] that utilizes GNNs for modeling interaction in object detection, and RNNs for modeling motion dynamics. Motion and appearance features are concatenated and fed into GNNs. A new object detector, data association network, and joint MOT framework are included in the framework, which achieves state-of-the-art results on MOT challenge datasets. Ma et al. [Ma et al., 2021] proposed an end-to-end Deep Association Network (DAN) for multiple object tracking, based on GNNs, combined with a Human-Interaction Model (HIM), which extracts inter-relation details, and is effective for occluding targets. Apart from GNNs, both models also include CNN and RNN networks. GNNs were leveraged in two parts of their Deep Association Network: feature extraction and graph optimization. In this study, DAN provides an unprecedented model structure for MOT and achieves superior performance compared to state-of-the-art methods.

GCNN is also utilized by Papakis et al. [Papakis et al., 2020] to introduce an online tracking method, GCNNMatch, for MOT benchmarks. Instead of using Siamese architectures to learn the appearance features of each object individually, the proposed method uses context information such as location and object size. Additionally, the proposed method uses Sinkhorn normalization for bipartite matching

constraints, and geometric information when constructing graph edges and computing affinity. Compared to other GNN-based MOT methods, the approach achieved superior accuracy at that time. Rangesh et al. [Rangesh et al., 2021] presented a framework that leverages dynamic undirected graphs and a message-passing graph neural network, TrackMPNN, to tackle the data association problem across multiple timesteps. In the proposed approach, each detection in the MOT problems is represented as a node, and potential associations are represented as edges. At each timestep, new detections and associations are added, while inactive ones are removed. In this way, they managed to rectify errors and handle missed/false detections in real-time, and achieved promising results on popular benchmarks for autonomous driving.

Recently Braso et al. [Brasó et al., 2022] proposed a method which builds upon their previous CVPR paper that our study also leverages [Brasó and Leal-Taixé, 2020] to track and segment multiple objects using message passing networks for feature learning and solution prediction. A time-aware neural message passing update step is included in their approach, which is inspired by traditional MOT graph formulations. Furthermore, a unified framework that integrates tracking and segmentation is presented for improved association performance. They achieve state-of-the-art results in eight MOT and MOTS benchmarks.

3.2.2 GNN Methods on GPU

As GNNs become more widely adopted in scientific machine learning, the significance of optimizing their training and inference efficiency is gaining greater attention. As deep learning communities continue to adopt deeper networks and work with larger datasets, hardware limitations have become more prevalent. Fortunately, the advent of specialized hardware platforms, such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), offers a promising solution to address these challenges. Numerous techniques have been proposed to optimize GNN training on GPUs for efficient performance [Wang et al., 2022, Wu et al., 2023].

Wang et. al. [Wang et al., 2022] compared the performances of GPUs and TPUs

by training a GNN in the context of real-life pattern recognition. Their findings indicate that while the accuracy of the GNN model trained with TPUs and GPUs is comparable, there are notable differences in latencies between the two platforms. Specifically, a TPU with 32 TPU v2 cores performs on par with 4 GPU V100s in terms of speed. The authors observed that TPUs tend to dedicate a significant portion of their training time to message passing operations, in contrast to GPUs which primarily emphasize matrix multiplications. Furthermore, GPUs are comparatively more cost-effective and energy-efficient than TPUs. Their analysis reveals that the bottlenecks of training GNN on GPUs are the computing capability for the matrix operations and the memory bandwidth for the message passing operations.

Hosseini et al. [Hosseini et al., 2022] conducted a thorough examination of Graph Neural Networks (GNNs) in the context of scientific machine learning, utilizing the PyTorch Geometric software framework and NVIDIA A100 GPUs for profiling and benchmarking GNN operations. The study revealed that memory inefficiency emerges as a significant bottleneck, with native PyTorch operations often outperforming their PyTorch Geometric counterparts. Notably, several GNN operations lack optimization for sparsity, exemplified by torch.addmm, which does not benefit from sparse input optimizations compared to torch.sort. Consequently, this deficiency results in runtime and memory bottlenecks that hinder overall GNN performance. To address these challenges and enhance GNN efficiency, the authors propose the implementation of sparse versions for benchmarked native operations.

In the same year, Zhou et al. [Zhou et al., 2022] proposed TGL, a general framework for training Temporal Graph Networks (TGNs) on a large scale, with billions of nodes and edges. TGL employs distributed memory systems to enable parallel GNN training, making it suitable for large-scale offline settings. According to the authors, TGL is groundbreaking as the first work to propose a general framework for training TGNs on multiple GPUs. It also includes implementations of various existing temporal GNNs, making it a powerful tool for learning representations on dynamic graphs. TGL allows for efficient training of different TGN variants on both single GPU and multiple GPUs, facilitated by simple configuration files. TGL's performance was assessed by comparing it with five open-source methods on four

small-scale datasets using a single GPU, as well as two large datasets with multiple GPUs, for tasks such as link prediction and node classification. Overall, TGL demonstrated comparable prediction accuracy to baseline models, while significantly speeding up both training and evaluation, with an average speedup of 13x.

3.2.3 GNN Methods on IPU

When it comes to hardware acceleration for GNNs, IPUs and GPUs have distinct features. IPUs, with their high-bandwidth memory situated proximately to processor cores, are specifically designed for efficient handling of irregular and sparse data structures commonly encountered in graph-based tasks. These advantages translate into significantly improved throughput and performance for smaller batch sizes over GPUs. In general, IPU's advantage grows with smaller and more fragmented memory operations over GPU. In contrast, GPUs, while also equipped with high-bandwidth memory, excel in dense matrix computations prevalent in many deep learning tasks but may encounter challenges, including reduced data locality, when confronted with irregular data structures due to differences in memory hierarchy and access patterns.

Recently Moe et. al. implemented a Spatio-Temporal Graph Neural Network (STGCN) on the IPU and its performance was thoroughly compared with the conventional GPU implementations of STGCN. Their findings proved the claims that the IPU effectively delivers the promised performance improvements for STGCN. The key finding of this study is a substantial performance increase of approximately 4 times when using the IPU in comparison to the Nvidia V100 SCM3 for training of GNNs [Moe et al., 2022].

3.3 Comparing IPU and GPU

As the pioneering study on IPUs, Mohan et al. [Mohan et al., 2020] conducted a comprehensive comparison of the performance on IPUs, GPUs and CPUs across various neural network architectures and parameters. Their findings highlighted the significance of batch size as a critical variable. Their study revealed that IPU and

GPU, both outperformed CPU, with IPU demonstrated superior performance over the GPU for batch sizes that are accessible to both processors. Furthermore, despite GPUs being able to handle larger batch sizes compared to IPUs, IPUs exhibited superior performance in terms of event generation speed, even when utilizing smaller batch sizes.

In a subsequent study, Sumeet et al. [Sumeet et al., 2022] conducted a performance comparison between IPU and GPU for a compute-intensive text region detection application. Based on compute precision, number of IPUs used, and batch size, they evaluated the IPU's throughput, power consumption, and accuracy capabilities. Overall, the IPU demonstrated superior throughput than CPU and NVIDIA A100 GPU across all batch sizes, particularly with FP16 implementations. As compared to larger batches, the IPU demonstrated significant gains in throughput over other hardware, particularly with small batch sizes.

Chapter 4

METHODOLOGY

GNNs are extensively employed in the context of MOT tasks, with message passing serving as a crucial element within GNNs. This iterative process entails the exchange of messages between nodes within the graph, resulting in the updating of node and/or edge feature vectors.

"Learning a Neural Solver for Multiple Object Tracking" [Brasó and Leal-Taixé, 2020] presents a compelling case for evaluating the efficacy of GNN training. This real-world application represents a common scenario, characterized by a simple model architecture that solely relies on basic MLPs to construct the GNN, wherein a fundamental linear layer is predominantly employed. Such an application example facilitates the discernment of the impact of scatter/gather operations on both the overall training and inference performance of the model. Consequently, we have selected this project as the foundation for our IPU implementation and experimental investigations.

This implementation afforded me the opportunity to scrutinize the distinctions between utilizing a GPU versus an IPU when implementing a TensorFlow model. It encompassed an examination of the data loading process, diverse optimization considerations specific to IPU usage, and, significantly, the optimization of scatter-gather operations that hold paramount importance for enhancing the performance of GNNs on the IPU.

Optimizing scatter/gather operations holds significant importance when executing a MOT application on an IPU. These operations encompass the transmission of messages from a single node to multiple other nodes (scatter) and the collection of messages from multiple nodes into a single node (gather). In the context of GNNs, scatter/gather operations play a crucial role in facilitating information propagation across the graph and gathering relevant information from neighboring nodes.

As a result, this research endeavors to investigate the performance of GNN training and inference on IPU, with a specific emphasis on techniques aimed at optimizing scatter/gather operations within a MOT application running on the IPU.

4.1 Implementation of the MOT Neural Solver in TensorFlow for IPU

The Graphcore IPU is a dedicated hardware accelerator optimized for enhancing the training and inference of deep learning models. TensorFlow is a popular open-source machine learning framework that is widely used for training and deploying deep learning models. The TensorFlow IPU integration enables users to leverage the power of the IPU to accelerate their TensorFlow models. Accessing the IPU through TensorFlow or PyTorch, both of which are widely recognized as high-level deep learning frameworks, presents a convenient approach. This is advantageous as it caters to the familiarity of developers in the field, facilitating their engagement with the IPU for various tasks.

To utilize the TensorFlow IPU integration, it is required to install the Graphcore's tailored version of TensorFlow. This specialized version of TensorFlow is designed explicitly for the integration with IPUs. Despite maintaining compatibility with the conventional TensorFlow API, the implementations within this IPU-specific version are optimized to leverage the full potential of IPUs.

The integration of TensorFlow with IPU offers an attractive feature for developers, as it allows them to build TensorFlow models using the familiar API while benefiting from the superior performance of IPU. This means that users do not need to learn a new API or programming paradigm to leverage the power of the IPU. The TensorFlow runtime identifies IPU-specific operations in the computation graph and transforms them into an optimized low-level representation for execution on IPU. This automatic optimization process allows users to take full advantage of the IPU's capabilities without needing to make manual modifications to their code.

It is crucial to keep in mind that while the integration of TensorFlow with IPU offers notable performance advantages, not all standard TensorFlow functions are mapped to an efficient lower-level kernel in the Poplar runtime. In scenarios where

such inefficiencies are encountered, end-users can leverage the option of designing their custom operations at the Poplar level and integrate them into their TensorFlow application, similar to the regular custom operations in TensorFlow. By pursuing this approach, end-users can optimize the performance of their models further and take full advantage of the comprehensive capabilities of the IPU hardware.

Besides expediting TensorFlow models, the outcome of the compilation process yields an IPU program regardless of its construction method. Moreover, we gain the ability to leverage debugging and profiling tools available in Graphcore PopVision, facilitating the measurement of performance metrics and identification of potential issues.

The initial step in executing and evaluating the aforementioned MOT model on the IPU involved rewriting the model for the IPU platform. Given that the original implementation relied on several dependencies, including PyTorch Lightning, and considering that Lightning did not support the IPU backend at that time, I made the decision to develop the model from scratch. This approach not only allowed me to address compatibility issues but also provided an opportunity to deepen my understanding of the model through hands-on implementation.

During the project's inception, resources on constructing IPU models using widely adopted deep learning frameworks were limited. However, the Graphcore TensorFlow examples repository offered more practical insights and examples in this context. Consequently, I chose to re-implement the model in TensorFlow specifically for the IPU, drawing inspiration from sample projects developed by Graphcore. While the overall code structure closely aligns with conventional TensorFlow model development, there are certain distinctions, such as the incorporation of a strategy definition and modifications in data loading mechanisms.

I referenced a Graphcore implementation of TGNs optimized for the IPU architecture as a guide. This specialized version of TGN is tailored for enhanced performance on the IPU. TGN itself falls under the category of GNNs designed specifically for modeling and analyzing dynamic or temporal graphs. At the time of my research, Graphcore's custom implementation of TGN for TensorFlow 1 stood out as the most comprehensive example available, aligning well with the requirements

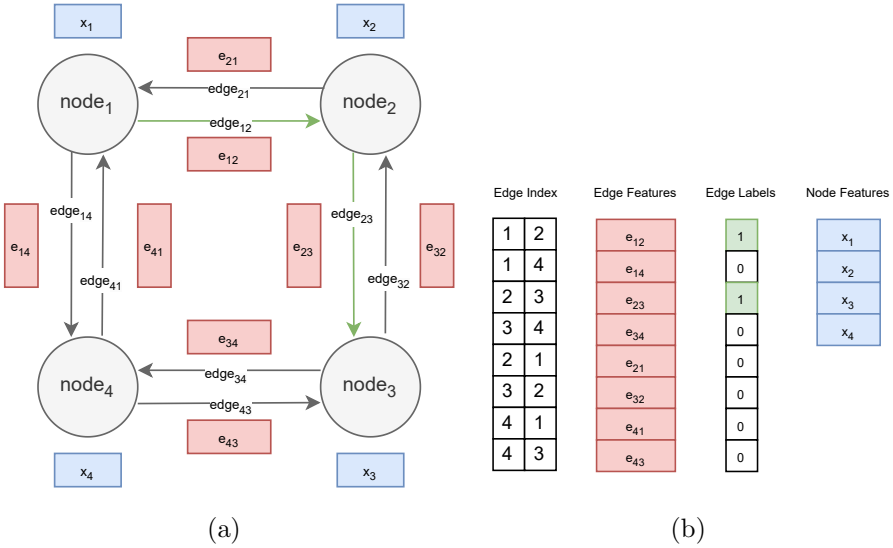
of a widely-used deep learning framework. Leveraging this project as a foundation for my code development proved advantageous due to its clarity, encompassing crucial components such as a train/evaluation loop, as well as efficient data input and output queues. These elements greatly facilitated the streamlined management of both training and evaluation processes.

4.1.1 Creation of graph inputs from MOT dataset

The input to the model consists of video segments, while the GNN requires graph objects as its input. To generate these graph objects, a tracking method is employed. During the tracking process, a frame window is selected to construct a single graph. For each frame in the chosen video segment, all detections are transformed into nodes, carrying the initial node embeddings as feature tensors. These nodes are then interconnected, creating edges between them. The edge features are derived by considering both the connected node features and the geometric data as edge features, which includes the relative positions of the respective data points. In my experiments, I used 15 frames per graph similar to the original implementation.

To mitigate unnecessary computational complexity, edges are pruned by removing those edges that connect nodes not present within their most similar 50 nodes list. By forming these graphs, the model can be trained using message passing, and subsequently, the edge features can be utilized in a fully connected layer for binary classification, distinguishing active edges from inactive edges. An active edge signifies that two detections belong to the same object. By leveraging these classification results, the final trajectory of a pedestrian across frames can be constructed.

In this study, I utilized the authors' code for creating graph datasets directly to generate training graphs for my experiments. The code incorporates a pre-trained R-CNN model, employed to generate initial node embedding features by processing the image within the bounding box. The computation of initial edge features involves considering the bounding box sizes of two detections (nodes in the graph) and the time difference between the two frames. Furthermore, for the evaluation of machine learning performance in the context of my IPU model, I leveraged the authors'



(a) (b)

Figure 4.1: **(a)** An example graph with 4 nodes **(b)** The same graph represented with 4 tensors

implemented evaluation framework.

MOT datasets consist of some video segments involving pedestrians from random city street views with people walking. The original model processes these segments to generate graph objects which are then fed to the MPNN for training. Each input graph consists of 4 tensors:

1. Edge List (The graph connectivity)

2. Node Features (An encoding of the node's data represented as 2048 floats)

3. Edge Features (An encoding of the edge's data represented as 6 floats)

4. Labels (Ground-truth binary values representing active/inactive edges)

The processing and graph creation are carried out using a window-based method, as previously described. The default window size for this approach is set to 15 frames, and the processing rate is at 6 frames per second (FPS).

4.1.2 Data Loading

In this section, we will first explain how data is typically loaded for GPU in TensorFlow, and then we will discuss the specific considerations and changes required for data loading when using TensorFlow for IPU.

Data Loading for GPU in TensorFlow

When loading data for the GPU in TensorFlow, the standard practice is to utilize the 'tf.data.Dataset' API, which provides a flexible and efficient way to build data pipelines. The following steps outline the process of loading data for GPU:

1. *Data Preparation:* The dataset is initially organized in a suitable format, such as TFRecord files, NumPy arrays, or other commonly used data formats. Among these options, TFRecord files are often preferred due to their exceptional I/O efficiency and streaming capabilities.

2. *Data Pipeline Creation:* The 'tf.data.Dataset' API is harnessed to build a well-structured data pipeline. This powerful API provides a range of methods for reading, preprocessing, and batching data. Common operations include data parsing, image decoding, resizing, normalization, shuffling, and batching.

3. *Data Augmentation (Optional):* To augment the diversity of the training dataset and strengthen the model's generalization capabilities, data augmentation techniques can be employed. TensorFlow offers a variety of built-in functions for augmenting data, including random cropping, flipping, rotation, and color jittering. These operations can be seamlessly integrated into the data pipeline.

4. *Model Training and Evaluation:* Once the data pipeline is constructed, it is integrated into the training or evaluation loop of the model. During the training process, batches of data are efficiently fed to the GPU, enabling the model to iteratively update its parameters and optimize performance.

A code snippet illustrating the data loading process for GPU in TensorFlow has been provided:

```
import tensorflow as tf

# Step 1: Prepare the data
train_tfrecord_path = "path/to/train.tfrecord"
validation_tfrecord_path = "path/to/validation.tfrecord"

# Step 2: Create a data pipeline
def parse_tfrecord_fn(serialized_example):
    # Parse and preprocess the data
    # ...

train_dataset = tf.data.TFRecordDataset(train_tfrecord_path)
train_dataset = train_dataset.map(parse_tfrecord_fn)
train_dataset = train_dataset.shuffle(buffer_size=1000)
train_dataset = train_dataset.batch(batch_size=32)
train_dataset = train_dataset.prefetch(tf.data.experimental.AUTOTUNE)

# Step 3: Data augmentation (optional)
# Apply data augmentation techniques if needed

# Step 4: Model training and evaluation
for images, labels in train_dataset:
    # Model training with the batch of data
    # ...
```

Listing 4.1: Pseudo-code of data loading process for GPU in TensorFlow

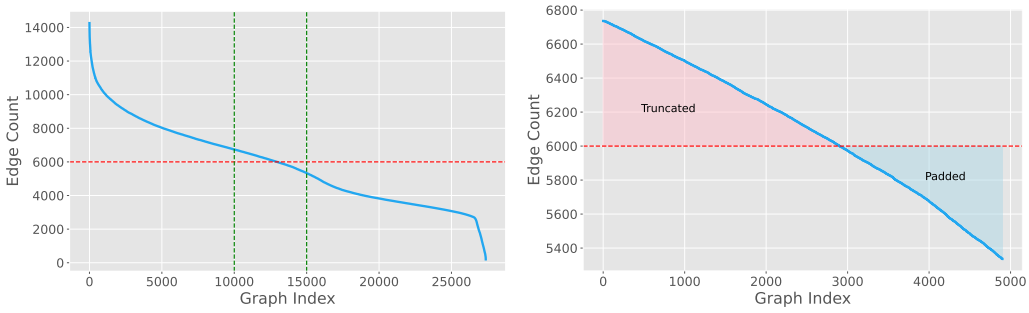
Data Loading for IPU in TensorFlow:

Data loading in TensorFlow for IPU involves a slightly different approach compared to loading data for GPU.

1. *Creating a TensorFlow Dataset as Input:*

   As explained in section 4.1.2.1, we represent an input graph with 4 tensors: edge list, node features, edge features, and binary labels.

   Tensor Size Standardization for IPU Efficiency: Before execution, Ten-



(a) (b)

Figure 4.2: **(a)** Number of edges per graph in the augmented dataset **(b)** Number of edges per graph in the selected training dataset before padding and truncation operations

sorFlow models are compiled into Poplar programs to be executed on the IPU. When working with the IPU, using fixed-size tensors is crucial; otherwise, it impedes static compilation, leading to multiple Poplar programs. For instance, a model performing both training and validation will generate two Poplar programs because the computations differ, as the validation omits the backward pass. Similarly, tensors of varying sizes necessitate distinct Poplar programs for each unique tensor size. This process is highly inefficient, as it requires compiling and executing different programs for each data point. Consequently, utilizing fixed-size tensors is required to allow static compilation of Poplar programs, enabling consistent program use throughout training and inference iterations. This constraint poses a challenge with GNNs, where varying numbers of nodes or edges result in tensors of different shapes in subsequent iterations. To address this, I engineered a solution to standardize the number of nodes and edges across the dataset. Inspired by PopTorch Geometric (PyTorch Geometric for IPU), which sets a fixed node and edge count and pads feature tensors accordingly, I replicated this methodology within the TensorFlow data loader. This ensures that all feature tensors across the dataset maintain a fixed size, optimizing computational efficiency.

For nodes, with counts typically below 500 even for the largest graphs in the

dataset, I chose to pad smaller node counts to match the maximum node count. In the case of edges, where the maximum edge count exceeds 10000, leading to significant computational costs as they dominate the matrix multiplications during training, and considering most experiments were conducted in a single IPU setting, I had to limit the edge count to 6000. This decision was made to optimize for training speed while maintaining a satisfactory level of accuracy. Consequently, some graphs have fewer than 6000 edges, requiring padding, while others have more, leading to edge cutting. Edge cutting involves the removal of nodes with the highest node IDs, corresponding to the latest detections in the time axis. This allows for the processing of smaller video segments, adjusting the length of the segment to be processed. This choice is made because setting the number of detections and, consequently, edges based on a fixed number of frames for input graph creation is impractical. Therefore, edge cutting is applied at this stage, ensuring a consistent number of edges across the dataset for all graphs. As illustrated in Figure 4.2, the original dataset exhibits a significant variance in edge counts across graphs. However, standardizing edge counts in the presence of such high variability would be inefficient due to excessive padding, leading to computationally meaningless operations. Moreover, for truncated edges, it would result in substantial data loss. To address this, I carefully selected a subset of the dataset with a consistent number of edges for the edge count standardization process. Specifically, I focused on a subset of 5000 graphs, each containing approximately 6000 edges, as depicted in Figure 4.2b. This standardization enables the IPU to determine the sizes of each computation during the training process at compile time, allowing for static compilation which results in a single compiled program.

Loss mask: In addition to these tensors, each graph is accompanied by a dedicated loss mask tensor. This loss mask serves as a crucial tool during loss calculation, allowing for effective masking in response to the fixed-sized tensor requirement. This requirement stems from the need to maintain a consistent edge count across all processed input graphs, as explained previously. To fulfill

this condition, padding is applied by introducing additional edges when the count falls below the chosen fixed edge size. During this edge-padding process, which involves the insertion of dummy edges, it is imperative to exclude these padded edges from the loss calculation. To seamlessly achieve this, an individual loss mask tensor is assigned to each graph with the size of edge count. This binary tensor is structured to assign a value of 1 to edges representing actual connections and 0 to padded edges, effectively segregating the essential structural elements from those introduced solely for padding purposes. I apply the loss mask in the last step of loss calculation, multiplying it with the loss tensor. This yields refined final losses, eliminating the impact of padded edges.

Creation of tf.data.Dataset: During the data loading phase, each input graph generates the mentioned five tensors, which are then appended to their respective lists. Subsequently, these tensors are utilized to construct a `tf.data.Dataset` object through the `tf.data.Dataset.from_generator()` method. The choice of this method is crucial, as alternatives like `tf.data.Dataset.from_tensor_slices` can encounter memory limits, particularly with large datasets. In my specific scenario, attempting to use `from_tensor_slices` led to tensor creation exceeding 2 GBs, resulting in a `"ValueError: Cannot create a tensor proto whose content is larger than 2GB."` error. This issue arises because the method stores the dataset as tf.constants, embedding them within the computational graph, which is both memory-inefficient and can surpass tensor size limits. Consequently, I switched to `tf.data.Dataset.from_generator`, which operates similarly to Python generators, allowing the dataset elements to be yielded on-the-fly. This approach enabled me to load all dataset elements without creating excessively large tensors. Moreover, as I leverage this in conjunction with `tensorflow.python.ipu.loops` and `IPUInfeedQueue`, it establishes an efficient data consumption model. The `from_generator` function produces a dataset whose elements are dynamically generated by a generator. Paired with IPUInfeedQueue, it facilitates IPU access to data, utilizing memory efficiently compared to embedding `tf.constants` in the graph.

Below code fragment shows the creation of tf.data.Dataset from input graphs:

```
def generate_graph_data():
    for index in range(graph_count):
        yield (
            (
                adjacency_indices_list[index], # Edge pairs in the graph
                node_features_list[index], # Node features
                edge_features_list[index], # Edge features
            ),
            binary_labels_list[index], # Binary labels
            binary_loss_mask_list[index], # Binary loss mask
        )
# Creating TensorFlow Dataset using from_generator
tf_dataset = tf.data.Dataset.from_generator(
    generate_graph_data,
    output_shapes=(
        (
            (edge_count, 2), # Shape of adjacency indices
            (node_count, 2048), # Shape of node features
            (edge_count, 6), # Shape of edge features
        ),
        (edge_count, 1), # Shape of binary labels
        (edge_count,), # Shape of binary loss mask
    ),
    output_types=(
        (
            tf.int32,
            tf.float32,
            tf.float32,
        ),
        tf.int32,
        tf.int32,
    )
)
```

Listing 4.2: Pseudo-code of data loading process for IPU in TensorFlow

2. *Setting Up an IPUInfeedQueue and IPUOutfeedQueue:*

   Specifically, as explained in section 2.3.6, the main I/O constructs in IPU are `IPUInfeedQueue` and `IPUOutfeedQueue`. The `IPUInfeedQueue` establishes a host-device queue for efficient input data transfer. In terms of computation, this results in the addition of I/O operations to the computation graph.

   As explained in the section 2.3.5.1 Looping Utilities, `IPUOutfeedQueue` objects can be passed to a loop of type `tensorflow.python.ipu.loops` as argument and the loop dequeues dataset elements automatically from the queue.

4.1.2.1 Optimized Data Loading Using Pickle Files

Without this optimization, the data loader underwent a lengthy process of loading thousands of graph input files, applying preprocessing such as paddings and edge cuttings, creating the loss mask tensor, and collecting these five tensors for each graph to generate the tf.data.Dataset object. In my development environment, this process extended beyond 2 hours. To expedite experiments, I implemented a strategy to create a pickled version of the dataset after its initial preprocess and loading using the pickle library. Subsequently, I loaded only this pickle file for each experiment. This adjustment significantly reduced the data loader's execution time to only a few seconds. Consequently, this optimization has accelerated the experimentation phase, allowing for faster and more efficient execution of experiments.

4.1.3 Model Structure

The reference study [Brasó and Leal-Taixé, 2020] introduces a MPNN framework comprising four distinct networks responsible for updating embeddings, including three for nodes and one for edges. Specifically, two of the node models ($\mathcal{N}_v^{fut}$ and $\mathcal{N}_v^{past}$) are leveraged to construct a time-aware node model, a significant aspect emphasized by the authors. By integrating an understanding of time, the model incorporates two separate MLPs. - one dedicated to the future ($\mathcal{N}_v^{fut}$) and the other to the past ($\mathcal{N}_v^{past}$). These MLPs effectively utilize the temporal information encoded in a node's neighbors, differentiating between the impact of future and past

interactions. This time-aware approach enhances the model's capacity to capture and analyze the temporal dynamics inherent in the graph data, thus contributing to the overall efficacy of the MPNN framework as presented in the paper. Using these two networks, the past and the future embeddings for nodes are computed. The final updated node embedding is calculated by the third MLP for nodes, namely, $\mathcal{N}_v$, using the computed future and past embeddings as inputs. The fourth MLP ($\mathcal{N}_e$) is used to calculate edge embeddings. It considers the node embeddings of the 2 nodes that it connects, as well as the current edge embedding as inputs to ($\mathcal{N}_e$) to calculate edge embedding at the next time step.

Table 4.1: Message Passing Network. The layer count and input/output dimensions are provided for each MLP. "FC" indicates a fully connected layer.

Past Update ($N_v^{\mathbf{past}}$)		
0	Input	80
1	FC+ReLU	56
2	FC+ReLU	32
Future Update ($N_v^{\mathbf{fut}}$)		
0	Input	80
1	FC+ReLU	56
2	FC+ReLU	32
Node Update (N_v)		
0	Input	64
1	FC+ReLU	32
Edge Update (N_e)		
0	Input	160
1	FC+ReLU	80
2	FC+ReLU	16

The model structure in this work consists of two parts: the Message Passing Network as shown in Table 4.1, and the Classifier as illustrated in Table 4.2. This architecture is akin to the one proposed in [Brasó and Leal-Taixé, 2020].

Table 4.2: Classifier MLP. Applies binary classification to edges based on edge features into active or inactive edges.

Edges ($N_e^{\mathbf{class}}$)		
0	Input	160
1	FC+ReLU	80
2	FC+Sigmoid	1

- 2 encoder MLPs: One MLP is dedicated to processing node information, while the other focuses on edge information. These MLPs provide the initial node and edge embeddings, respectively.

- 4 update MLPs: There are three update MLPs specifically designed to update node representations within the MPNN. Additionally, there is a single update MLP that handles edge embeddings used in the Message Passing Network.

- 1 edge classifier MLP: This MLP performs binary classification over the output of the Message Passing Network, allowing for prediction and decision-making based on the processed information.

- The edge classifier MLP conducts binary classification on the output of the MPNN. This enables the model to make predictions and decisions by leveraging the processed information from the network.

4.1.3.1 Linear Layer

In the implementation of the model based on the TGN-IPU codebase, a custom linear layer was utilized. This layer was incorporated to align with the model's architecture as described in the original paper, which includes MLPs with linear layers. The linear layer performs linear transformations on the input data using matrix multiplication with learnable weights and bias terms. The implementation of the linear layer is as follows:

```
@scoped_fn
```

```
def linear(input: tf.Tensor,
           n_output: int,
           use_bias: bool = True) -> tf.Tensor:
    """A standard linear layer 'W x + b'."""
    weight = tf.get_variable(
        "weight",
        dtype=input.dtype,
        shape=(input.shape[-1], n_output),
        initializer=tf.glorot_normal_initializer(),
    )
    output = input @ weight
    if use_bias:
        bias = tf.get_variable(
            "bias",
            dtype=input.dtype,
            shape=(n_output, ),
            initializer=tf.zeros_initializer(),
        )
        output += bias
    return output
```

Listing 4.3: Pseudo-code of implementation of the linear layer

4.1.4 Training Loop Setup

As outlined in the Section 2.3.5.1, we utilize tensorflow.python.ipu.loops.repeat method to prevent multiple calls to session.run. This method takes the infeed_queue as a parameter and handles fetching data to the IPU automatically.

```
def run_loop():

    infeed = ipu.ipu_infeed_queue.IPUInfeedQueue(dataset, prefetch_depth=4)
    outfeed = ipu.ipu_outfeed_queue.IPUOutfeedQueue()

    with ipu.scopes.ipu_scope("/device:IPU:0"):
        ipu.ipu_compiler.compile(
            lambda: ipu.loops.repeat(self.steps, loop_body, infeed_queue=infeed),
```

```
            {}
        )

    ...

session.run(run_loop)
```

Listing 4.4: Pseudo-code for the IPU training loop runner

4.2 Adapting Code for IPU: Implementation Changes and Optimization Strategies

While rewriting the model for using on the IPU in TensorFlow, I applied several optimizations to make the model run faster on the IPU. In this section, I list some examples of these optimizations.

MetaLayer:

In forward pass of MetaLayer in the original PyTorch implementation, there is a part where the model uses *edge_index* to index the tensor x by first row and then by column dimension to be passed to *edge_model*:

```
row, col = edge_index
edge_attr = self.edge_model(x[row], x[col], edge_attr)
```

Listing 4.5: Part of MetaLayer in the original implementation. Taken from [Brasó and Leal-Taixé, 2020]

I implemented this part as follows:

```
row, col = tf.split(edge_index, 2, axis=-1)
index = tf.concat([row, col], 1)
features = grouped_gather(params=x, indices=index) # Equivalent to calling
    tf.gather(x, index, axis=1, batch_dims=1)
x_row, x_col = tf.split(features, 2, axis=1)
edge_attr = edge_model(x_row, x_col, edge_attr)
```

Listing 4.6: Part of MetaLayer rewritten in TensorFlow

Here I employ two optimizations, first, making a single call to gather function instead of 2 separate calls. I achieve this by first concatenating splitted row and col indices and using it to index the x tensor. The result of this indexing is then splitted to get separate x_row and x_col tensors. This optimization improves the speed by reducing the number of calls to gather functions. Secondly, instead of using the default tf.gather function, I utilize the *grouped_gather* function which internally calls a custom operation developed by Graphcore as it is significantly faster.

TimeAwareNodeModel:

I incorporated similar optimizations during the implementation of my iteration of the $TimeAwareNodeModel$. This model is pivotal to the overall architecture, responsible for computing flows independently from past and future nodes at each node. The final node embedding is determined by concatenating these flows and feeding the result as input to the `node_mlp` (formally denoted as N_v). Below, I present both the original forward pass implementation and the TensorFlow version optimized for IPU, delving into the introduced optimizations.

```
def forward(self, x, edge_index, edge_attr):
    row, col = edge_index
    flow_out_mask = row < col
    flow_out_row, flow_out_col = row[flow_out_mask], col[flow_out_mask]
    flow_out_input = torch.cat([x[flow_out_col], edge_attr[flow_out_mask]],
        dim=1)
    flow_out = self.flow_out_mlp(flow_out_input)
    flow_out = self.node_agg_fn(flow_out, flow_out_row, x.size(0))

    flow_in_mask = row > col
    flow_in_row, flow_in_col = row[flow_in_mask], col[flow_in_mask]
    flow_in_input = torch.cat([x[flow_in_col], edge_attr[flow_in_mask]], dim=1)
    flow_in = self.flow_in_mlp(flow_in_input)

    flow_in = self.node_agg_fn(flow_in, flow_in_row, x.size(0))
    flow = torch.cat((flow_in, flow_out), dim=1)
```

```
    return self.node_mlp(flow)
```

Listing 4.7: Forward pass of TimeAwareNodeModel in the original implementation. Taken from [Brasó and Leal-Taixé, 2020]

In my implementation, it is implemented as a function, receiving different parameters.

```
@scoped_fn
def time_aware_node_model(
    x: tf.Tensor,
    row: tf.Tensor,
    col: tf.Tensor,
    edge_attr: tf.Tensor,
    x_col: tf.Tensor=None,
) -> tf.Tensor:

    x_out_col, x_in_col = tf.split(x_col, 2, axis=1)
    edge_attr_out_masked, edge_attr_in_masked = tf.split(edge_attr, 2, axis=1)
    flow_out_row, flow_in_row = tf.split(row, 2, axis=1)
    flow_out_row, flow_in_row = tf.expand_dims(flow_out_row, 2),
        tf.expand_dims(flow_in_row, 2)

    def get_flow(flow_mlp, x_col, edge_attr_masked, flow_row, out_shape):
        flow_input = tf.concat([x_col, edge_attr_masked], axis=2)
        flow = flow_mlp(flow_input)
        flow = grouped_scatter_sum(data=flow, indices=tf.squeeze(flow_row,
            axis=-1), table_size=out_shape)
        return flow

    flow_out = get_flow(flow_out_mlp, x_out_col, edge_attr_out_masked,
        flow_out_row, x.shape[1])
    flow_in = get_flow(flow_in_mlp, x_in_col, edge_attr_in_masked, flow_in_row,
        x.shape[1])

    flow = tf.concat([flow_in, flow_out], axis=2)
    return node_mlp(flow)
```

Listing 4.8: Optimized implementation in TensorFlow using grouped_scatter_sum for aggregating messages from incident edges

The first optimization is passing already computed parameters to the function instead of raw parameters. Specifically, instead of passing only x, $edge\_index$ and $edge\_attr$ tensors, I pass x, row, col, $edge\_attr$ and x_col. Here, row and col are splitted columns from the $edge\_index$. Since they are previously computed, I directly pass them as parameters. x_col is x tensor indexed by col and is also already computed previously. Passing these already computed parameters helps reduce computational cost.

Secondly, in $edge\_index$ tensors, upper-half indices correspond to "out flow", that is, the "row" column has a value smaller than the "column" column, and lower-half indices correspond to "in flow", that is, the "row" column has a value greater than the 'column' column. Using this information, we can use $tf.split$ operation to split the row tensor in half in vertical dimension to get $flow\_out\_row$ and $flow\_in\_row$ instead of using two costly gather operations such as $tf.gather(row, tf.where(row < col))$ and $tf.gather(row, tf.where(row > col))$. The same idea was used to compute x_out_col and x_in_col as well as $edge\_attr\_out\_masked$ and $edge\_attr\_in\_masked$.

Thirdly, I use $grouped\_scatter\_sum$ as the aggregation function which was implemented by Graphcore and works more efficiently than $tf.tensor\_scatter\_nd\_add$ on IPU.

These optimizations significantly accelerated the execution of this model on the IPU, resulting in improved performance.

4.2.1 Optimization of the GNN Message Passing Process

The process of message-passing is a fundamental component of MPNs, which serves as a central mechanism in various applications, including MOT. The message-passing approach involves propagating messages iteratively across the graph's nodes, thereby enabling the nodes' feature vectors to be updated based on the received messages. The computational aspect of this process is carried out through scatter and gather

operations. During my analysis of the training process using the PopVision tool, I discovered that the IPU cycles spent on gather and scatter operations dominated the computation. This observation aligns with expectations, considering that these operations are the primary computational workload for MPNs. Hence, it becomes imperative to execute these operations efficiently on the IPU for optimal performance.

4.2.1.1 Grouped Gather vs tf.gather

During the process of transitioning the model from PyTorch to TensorFlow, I replaced the PyTorch indexing operator with `tf.gather()` as a substitution. For example, the expression x[flow] (where x and flow are tensors) was translated as tf.gather(x, flow), assuming that tf.gather could be efficiently executed on the IPU. However, it should be noted that not all standard TensorFlow library functions have optimized kernels for the IPU. In my experimental analysis, using tf.gather() resulted in significantly longer execution times compared to efficient gather implementations, leading to higher inter-tile communication within the IPU.

For optimizing gather operations, various approaches were explored initially. These methods are detailed in the following section. Subsequently, a solution provided by Graphcore was adopted for enhanced efficiency. Graphcore devised a specialized implementation for grouped gather and scatter operations, designed to outperform conventional TensorFlow operations. The core functionality of this solution is implemented in C++, with a Python wrapper invoking the compiled code, as regular Tensorflow custom operations. Within the C++ implementation, custom methods from popops, Graphcore's tensor operation library, are utilized. To be specific, the `grouped_gather` operation invokes `popops::groupedMultiSlice`, while `grouped_scatter_max` calls the `groupedMultiUpdateMax` function. Comparative evaluations demonstrate that these custom operations significantly outperform their TensorFlow counterparts, specifically `tf.gather` and `tf.scatter_max` operations. This optimized solution has been integrated into the project code, providing a substantial boost in performance for gather and scatter operations [Graphcore,

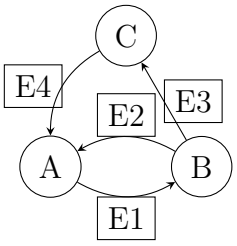
2022].

4.2.1.2 Investigating Other Optimization Strategies

For our application model, gather operation plays a pivotal role in the message passing of GNNs, enabling the collection and aggregation of messages from neighboring nodes for each node in the graph. Given the challenges mentioned in the previous section about using standard tf.gather operation in Tensorflow on IPU, I initially examined an alternative approach, inspired by the fused gather operation mentioned in [Zhang et al., 2020]. It's important to note that this exploration occurred before Graphcore's introduction of the optimized grouped gather operation. The idea involved constructing a vertex-to-edge matrix, which was then multiplied with the edge feature matrix. The vertex-to-edge matrix is a binary, 2-D representation with dimensions (node count, edge count), where a value of 1 indicates the presence of an edge connecting a vertex, and 0 indicates no connection. The result of this multiplication is equivalent to a gather operation with summation as the aggregation function. However, the vertex-to-edge matrix is highly sparse, with only $\frac{1}{\text{node\_count}}$ filled entries. To efficiently execute this methodology, a sparse-dense matrix multiplication that can accommodate an additional (batch) dimension was necessary. At that time, neither the standard TensorFlow operations nor the Graphcore SDK provided ready-made functions for such an operation. Additionally, even if such a support were available in TensorFlow, specialized IPU targeting would still be necessary for optimal performance. Consequently, substantial efforts were devoted to crafting a customized rank-3 sparse-dense matrix multiplication operation designed for optimal IPU utilization, serving as a custom operation for TensorFlow for IPU.

The example in Figure 4.3 shows the construction of a node-to-edge matrix. By performing matrix multiplication between this matrix and an edge feature matrix, we effectively compute the gather operation using summation as the aggregation function.

While this custom multiplication operation was more efficient than using tf.gather due to its internal use of Poplar library functions for multiplication, the subsequent



	E1	E2	E3	E4
Node A	0	1	0	1
Node B	1	0	0	0
Node C	0	0	1	0

Figure 4.3: Graph Representation with Adjacency Matrix. A, B and C are nodes; E1, E2, E3 and E4 are edges. For each directed edge, a 1 value is added to the corresponding location at the matrix.

introduction of grouped_gather proved even more efficient. This led me to shift from my initial method to adopting grouped_gather for message passing.

4.3 Training

For the training phase, the Adam optimizer was selected with the following hyper-parameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 3 \times 10^{-4}$.

I employ the identical loss configuration as the original model. The labels are highly imbalanced, with positive edges constituting less than 4% of the total. Consequently, Weighted Binary Cross-Entropy was used to account for imbalance. In each batch, the loss function dynamically computes the ratio of negative to positive edges. This ratio is then supplied as the `pos_weight` parameter to the `tf.nn.weighted_cross_entropy_with_logits` function. The resulting loss is further refined by multiplying it with a loss mask tensor, eliminating the influence of padded edges to the training process.

A comprehensive discussion pertaining to the training process has been provided in Chapter 5.

4.3.0.1 Parallelization

IPU offers ways to utilize model and data parallelization.

Data Parallelization

Using TensorFlow interface, it is possible to set number of IPU's to use by simply setting:

```
config = utils.ipu.config.IPUConfig()
config.auto_select_ipus = NUMBER_OF_IPUS
config.configure_ipu_system()
```

Listing 4.9: Pseudo-code of setting number of IPUs

This configuration's default behavior involves replicating the model across multiple IPUs for data parallelization. Initially, the expectation was that this setup would not only duplicate the model on each IPU but also efficiently distribute the training data among them, thereby accelerating the training process. However, through empirical observations in my experiments, it became evident that, despite the automatic allocation of multiple IPUs and model replication, the data distribution did not align with the anticipated optimization.

Contrary to the envisioned scenario, this setup unexpectedly resulted in an increase in overall training time proportional to the number of IPUs. This unexpected outcome can be attributed to the concentration of all processing on a single IPU, without explicit configuration for effective data distribution among the allocated nodes. It's likely that an incorrect configuration played a role in this suboptimal performance. I am currently in contact with Graphcore support to seek clarification and resolution for this issue.

Model Parallelization

For model parallelization, IPU supports sharding and pipelining. Since it was out of the scope of this work, I left them as future work.

Chapter 5

EXPERIMENTS AND RESULTS

5.1 Dataset and Testbed (Simula machines)

The original model employs sequences from MOT15 and MOT17 datasets for training. For simplicity, I utilized the graph builder from the original implementation and used those graphs for both training and evaluation.

I conducted my experiments on Simula's high-performance computing cluster, eX3, accessible at [ex3, 2024]. The experiments were run on GPUs, IPU, and CPU nodes to compare the performance of the models on four different hardware configurations. The specifications of the GPUs, IPU, and CPU resources are as follows:

Table 5.1: Processor specifications of Graphcore GC200, NVIDIA Tesla V100-SXM3 and A100 SXM gathered from [Shekofteh et al., 2023], [Graphcore, 2020]

.

	Name	**Cores**	**Memory**	**FP32 FLOPS**	**TDP**
GPU	A100 SXM	6912	80GB	19.5 TFLOPS	400 W
GPU	Tesla V100-SXM3	5120	32GB	16.35 TFLOPS	350 W
IPU	Colossus Mk2 GC200	1472	900MB SRAM	62.5 TFLOPS	150 W
CPU	Intel Xeon Platinum 8168	24	754GB (System)	-	205W

The comparison in 5.1 underscores distinct differences among four computing devices: A100 SXM GPU, Tesla V100-SXM3 GPU, Colossus Mk2 GC200 IPU, and Intel Xeon Platinum 8168 CPU. Notably, the A100 SXM GPU stands out for its expansive memory capacity and high compute performance, while the Tesla V100-SXM3 GPU offers a balance between performance and power efficiency. In contrast, the Colossus Mk2 GC200 IPU prioritizes exceptional compute performance and

lower power consumption. Meanwhile, the Intel Xeon Platinum 8168 CPU distinguishes itself with its expansive system memory capacity, offering an architecture tailored to accommodate a wide range of computational tasks. Each device represents a distinct trade-off in compute power, memory capacity, and power efficiency, catering to diverse computational requirements.

5.2 Metrics for Evaluation

5.2.1 ML Performance Comparison: IPU vs. GPU

1. **Precision:** Measure of the accuracy of positive predictions. High precision indicates a low false-positive rate.

2. **Recall:** Measure of the ability to capture all positive instances. High recall indicates a low false-negative rate.

5.2.2 Computing Performance Comparison: IPU vs. GPU

- **Total Training Time:** Measures the total time taken for the model to complete training. Lower training times are preferable.

- **Average Inference Time:** Calculates the average time taken for the model to perform inference. Lower average inference times are desirable.

- **Number of Epochs for Convergence:** Measures the number of training epochs required for the model to converge. Fewer epochs to convergence indicates faster training.

5.3 Performance Comparison: MOT Neural Solver vs Our Model

I used the same sequence to generate the dataset for both IPU and GPU experiments, namely MOT17-09-SDP. When using parameters max_detects = 500 (maximum number of detections allowed) and frames_per_graph = 15 it creates 5000 training and 100 validation graphs with 6000 edges for this sequence using the previously explained graph generation method.

Table 5.2: Model ML Performances Comparison after 100 epochs on validation set

Metric	**MOT Model on IPU**	**MOT Neural Solver**
Precision	0.58	0.698
Recall	0.51	0.655

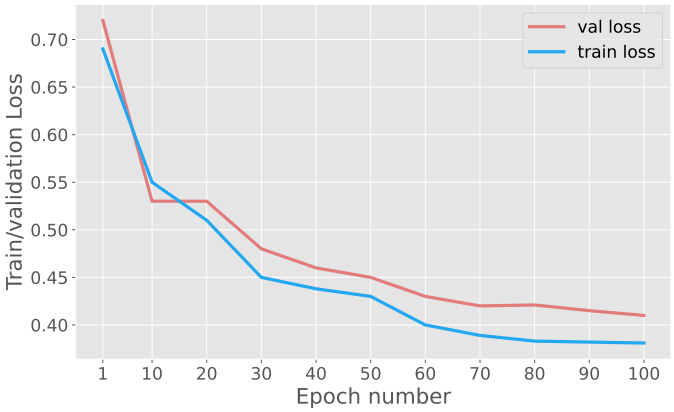
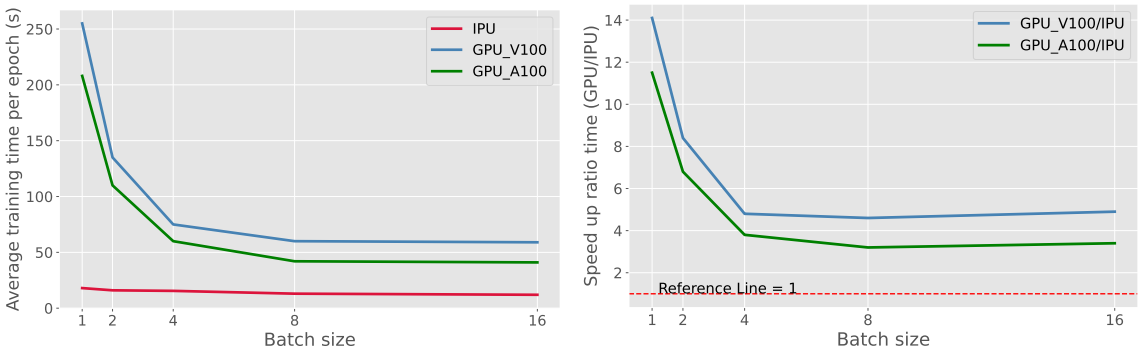


Figure 5.1: Train and validation loss graphs for IPU (batch size = 4)

In the comparison in Table 5.2, I evaluate the training precision and recall of the original model running on a GPU against our model on an IPU after 100 epochs. Notably, our model exhibits lower performance compared to the original model at the 100th epoch using the same learning rate 3×10^4. This observation aligns with our expectations, given that our implementation employs fixed-size graphs, requiring the removal of edges from certain input graphs. Despite the slower learning pace, our model demonstrates the ability to train and learn from the input data.

Figure 5.1 displays how the training and validation losses changed over 100 training epochs during training of the MOT model on the IPU. The training loss shows how well the model predicts outcomes within the data it learned from, while the validation loss represents the error on data not used for training.

Initially, we observe a significant improvement in both metrics, demonstrating that the model is rapidly learning from the data. As the epochs continue, the rate of improvement slows down, indicating that the model is getting closer to a stable state where it's not gaining much additional insight. This trend indicates that the



(a) (b)

Figure 5.2: **(a)** Comparison of average training times between IPU and GPUs as a function of batch size. **(b)** Speed up ratio of GPUs and IPU as a function of batch size. The red horizontal line represents a reference value of 1.

model is converging towards an optimal solution, where additional cycles may only lead to slight improvements.

The experimental setup was optimized to employ a batch size of 4, since this batch size was found to be the most efficient for the IPU during these experiments. Several considerations guided this decision: larger batch sizes exceeding 16 caused out-of-memory errors due to the memory constraints of single IPU, while batch sizes smaller than 4 slowed down the training process. Intermediate batch sizes of 8 and 16 were also evaluated but led to a decline in the overall model accuracy. Consequently, a batch size of 4 was the optimal trade-off between computational efficiency and model performance. This choice aims to minimize memory issues and prevent the reduction in accuracy observed with larger batch sizes.

In my experiments, I conducted a comparative analysis to evaluate the performance of GPUs, including an NVIDIA V100 and the NVIDIA A100, followed by the Graphcore IPU MK2. The training sessions involved varying batch sizes of 1, 2, 4, 8, and 16, spanning 100 epochs to calculate the average training time. To ensure accuracy and consistency in the results, the initial ten epochs were omitted from the calculation to mitigate the influence of the initial device warm-up period, following the methodology outlined in [Nasari et al., 2022].

In the experiment shown in Figure 5.2, I have conducted a comparative analysis

between my version of the model implemented using TensorFlow and run on an IPU, and the original model[Brasó and Leal-Taixé, 2020] developed with PyTorch and executed on GPU. The two implementations have identical layer architectures and an equivalent number of parameters. Furthermore, the same dataset is utilized across both experiments to ensure consistency in the comparison.

Figure 5.2a illustrates the average training time per epoch as a function of batch size for both IPUs and GPUs. It is evident from the graph that as the batch size increases, the average training time for both types of processors decreases. This trend is consistent with the parallel processing capabilities of both IPUs and GPUs, where larger batch sizes allow them to utilize the processor's resources more efficiently. Notably, the IPU consistently outperforms the GPU across all batch sizes, particularly notable in the context of smaller batch sizes.

Figure 5.2b illustrates the speed-up ratio of GPU to IPU training times across various batch sizes. Initially, the IPU exhibits a significant speed advantage, notably prominent with a batch size of 1, where it outperforms the A100 and V100 by around 11 and 14 times, respectively. However, as the batch size increases, this advantage diminishes, stabilizing at approximately 3 and 5 times faster for A100 and V100, respectively, for batch sizes exceeding 4. The convergence towards a ratio of 4 suggests that as the batch size grows, the performance gap between IPUs and GPUs narrows, consistent with the findings in [Arcelin,].

These findings underscore an important trend: GPU performance experiences a decline when operating with smaller batch sizes. This outcome aligns with the intrinsic design of GPUs, optimized for efficient handling of extensive parallelism, with larger batch sizes presenting increased opportunities for concurrent computation. The observed shift towards GPU competitiveness in larger batches can be attributed to several factors. The larger batch size significantly enables more parallel execution of instructions, which matches well with the GPU's architecture that is built to take advantage of parallel processing capabilities. Moreover, larger batch sizes help decrease kernel overhead. The overhead associated with launching kernels diminishes on average per data point as the batch size increases. Additionally, increased batch sizes facilitate a more effective utilization of the existing memory

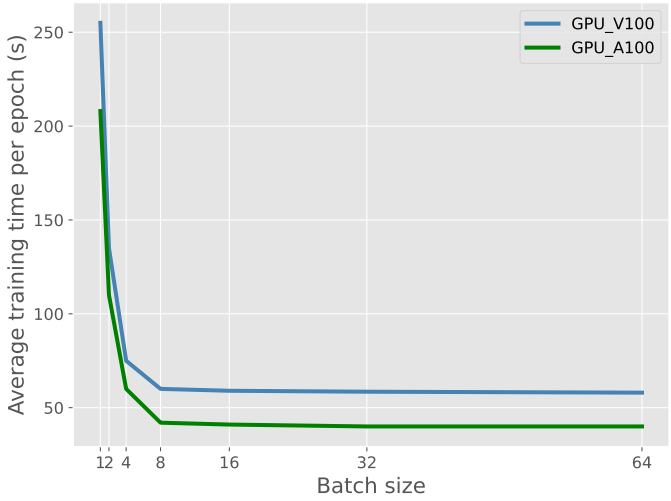


Figure 5.3: Average training time for V100 and A100 as a function of batch size.

bandwidth, thereby contributing to further enhancements in overall performance.

The analysis in Figure 5.3 focuses on how different batch sizes impact the average training time per epoch, specifically in experiments conducted using the original model using PyTorch running on V100 and A100 GPUs. It should be pointed out that the matching IPU program runs into an "Out of Memory" error when the batch size goes past 16 in the single IPU configuration. However, the NVIDIA GPUs that I utilized for these experiments have significantly larger memory capacity, which permits the use of larger batch sizes, as emphasized in [Nasari et al., 2022], necessitating the presentation of the GPU results exclusively in a separate graph. The horizontal axis of this graph scales batch sizes logarithmically, providing a comprehensive perspective of their impact across a wide range. Conversely, the vertical axis quantifies the average training time per epoch in seconds, offering a direct measure of computational efficiency.

Initially, when using small batch sizes, we see a sharp drop in the amount of time per epoch needed for training. This region shows that the GPU is underutilized, since the computational work required to set up each batch is not spread out over enough data points to be efficient. In practical terms, this implies that the parallel processing units of the GPU are in a state of awaiting data, resulting in idle cycles and, consequently, extended training durations.

As the batch size increases, the graph depicts a notable reduction in training time, demonstrating the advantages of increased parallel processing. Within this intermediate range, the GPU demonstrates its capability to concurrently process multiple training examples, leading to an enhanced and more efficient training process. The significant decrease in time is indicative of the reduced overhead per example and the efficient utilization of the GPU's computational cores.

The graph further indicates that beyond a certain point, the benefits of increasing the batch size begin to wane. This plateau area is especially informative, as it indicates the GPU's approaching computational limits. In this context, the bottleneck emerges in the form of memory bandwidth—the rate at which data can be read from or stored into memory. Furthermore, factors such as synchronization issues and the constraints of data transfer rates over the PCIe bus between the host and the GPU may contribute to the observed leveling off.

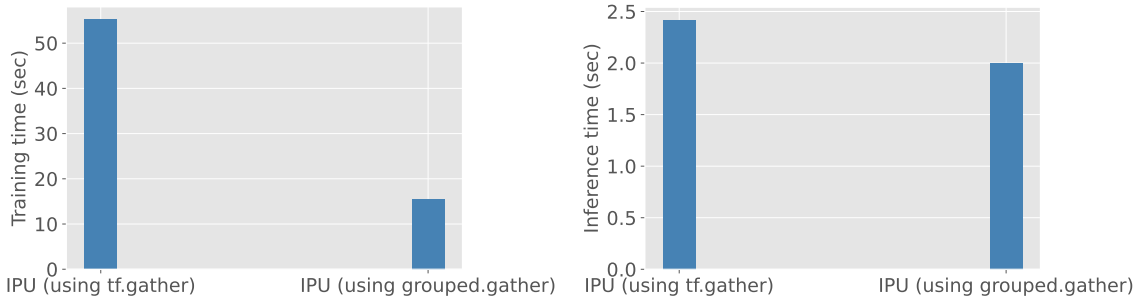
We also observe that the computational efficacy of the A100 surpasses that of the V100. While the initial scalability performance exhibits similarity between the two architectures, the A100 demonstrates a superior peak performance, leading to a reduction of approximately 30% in epoch training duration upon reaching a plateau in computational time.

Table 5.3: Average train/inference time per epoch comparison of IPU using tf.gather and grouped gather with the effect of enabling PopVision for batch size=4.

Method used	IPU (using tf.gather)		IPU (using grouped_gather)	
	Train Time	Inference Time	Train Time	Inference Time
Without PopVision	55.26	2.42	15.5	2
With PopVision Enabled	115.5	5.34	18.44	2.44

The table 5.3 provides a quantitative analysis of the performance impact when using Graphcore's IPU with TensorFlow's tf.gather operation versus the optimized grouped_gather. Moreover, it illustrates the additional overhead introduced when utilizing the *PopVision* profiling tool during both training and inference.

When comparing the two methods, the use of tf.gather demonstrates a notable



(a) (b)

Figure 5.4: **(a)** Average training time comparison of IPU using tf.gather and grouped gather for batch size=4. **(b)** Average inference time comparison of IPU using tf.gather and grouped gather for batch size=4.

increase in training and inference time upon enabling PopVision. Specifically, training time nearly doubles shifting from 55.26 to 115.5 units, while the inference time more than doubles, escalating from 2.42 to 5.34 units. This implies that there is a significant performance penalty because of the extra computations and data logging that PopVision entails. When PopVision analyzes this operation, it comes across a larger number of lower-level operations and interactions that need to be documented and examined, resulting in a greater overhead.

On the other hand, when using the optimized grouped_gather, the extra cost brought on by PopVision is less noticeable, with training time increasing from 15.5 to 18.44 units and inference time from 2 to 2.44 units. This indicates that the grouped_gather operation, since it is an enhancement over the regular tf.gather, lessens the effect of profiling to a certain degree.

The data demonstrates that although profiling with PopVision offers valuable insights for performance tuning, it also imposes a noticeable computational cost for complex computations.

The graphs in Figure 5.4 contrasts the average training and inference times on an IPU when utilizing the standard TensorFlow tf.gather against the optimized *grouped_gather*, which was explained in the Section 4.1.8.1. This optimized gather function is critical, as the standard tf.gather was found to be inefficient due to a lack of specialized kernels for the IPU, leading to excessive communication between

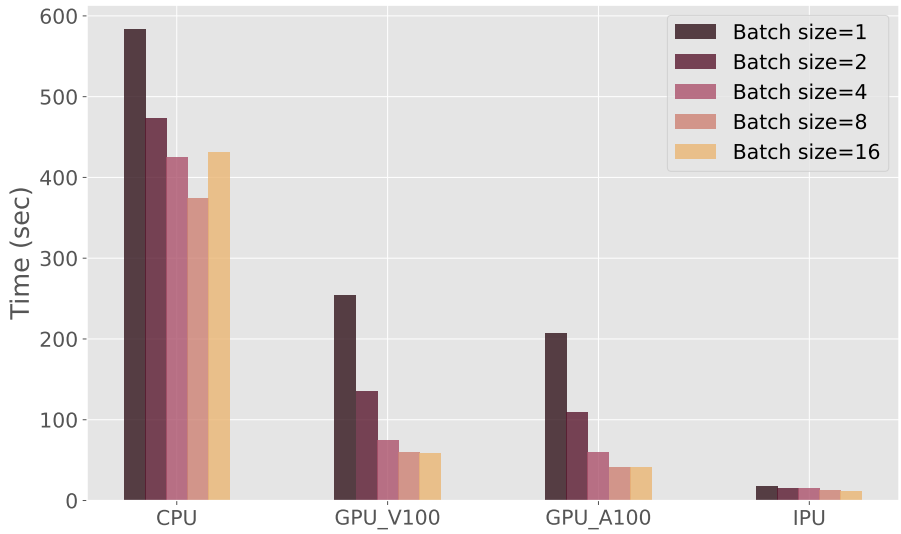


Figure 5.5: Comparison of average training time across CPU, GPU, and IPU for varying batch sizes (1, 2, 4, 8, 16).

tiles and longer run times.

The left side in the figure, which shows the performance using tf.gather, has longer durations for both training and inference. This highlights the need for operations optimized for particular hardware architectures, since standard functions may not make full use of the hardware's capabilities. In contrast, the right side of the figure illustrates the impact of employing the *grouped_gather* operation provided by Graphcore. This implementation leverages the popops library, a part of Graphcore's custom operations, which are detailed in the aforementioned section. The *grouped_gather* operation reduces the execution time for both training and inference. Training time is particularly improved, indicating a substantial performance boost, whereas the speedup in inference is limited as it does not perform a backward pass and is affected less by this improvement.

The results presented in this figure validate the discussions from Section 4.1.8.1 on the benefits of using specialized operations over standard ones. By integrating the *grouped_gather* into the model's training and inference pipeline has led to a significant enhancement in efficiency, underscoring the significance of hardware-aware optimizations in machine learning workflows.

The graph in Figure 5.5 provides a visual representation of the average training and inference times across three types of computing devices: CPU, GPUs, and IPU.

Starting with the CPU, it's clear that it takes significantly longer to perform both training and inference tasks compared to GPUs and IPUs. This discrepancy is likely due to the CPU's general-purpose design, which is tailored for a diverse array of computing tasks but lacks optimization for the parallel processing requirements inherent in machine learning algorithms.

Moving to the GPU, there is a clear decrease in the time it takes to train the model. GPUs are designed for parallel processing, which works well with the needs of machine learning calculations that involve processing large sets of data at the same time. This design facilitates a significant acceleration in machine learning tasks. It is important to highlight that, beyond a batch size of 8, increasing batch size does not yield substantial additional speed gains. The A100 and V100 architectures exhibit comparable scaling characteristics; however, across all batch sizes, the training durations on the A100 are shorter in comparison to those on the V100.

The IPU presents the most efficient training times among the three hardware types. This efficiency can be attributed to the IPU's capability to operate optimally with smaller batch sizes, as emphasized in this study [Mohan et al., 2020]. The remarkable performance of the IPU in training, evident in the graph, can be attributed to its specialized architecture tailored for high efficiency at smaller batch sizes. This is in contrast to GPUs, which necessitate larger batch sizes to fully leverage their parallel processing capabilities.

The graph titled Figure 5.6 displays the comparative speed increases achieved when training the model with different batch sizes (1, 2, 4, 8, 16) using an IPU compared to using a CPU or GPU.

The "IPU/CPU" bar shows the relative performance improvement of the IPU compared to the CPU, which is about 30 times faster. This considerable boost in speed shows that the IPU is much more efficient at handling the training process than the traditional CPU, which is not built for the parallel processing that machine learning algorithms take advantage of. On the other hand, the bar that represents the results for the "IPU/GPU" comparison shows a smaller, but still significant,

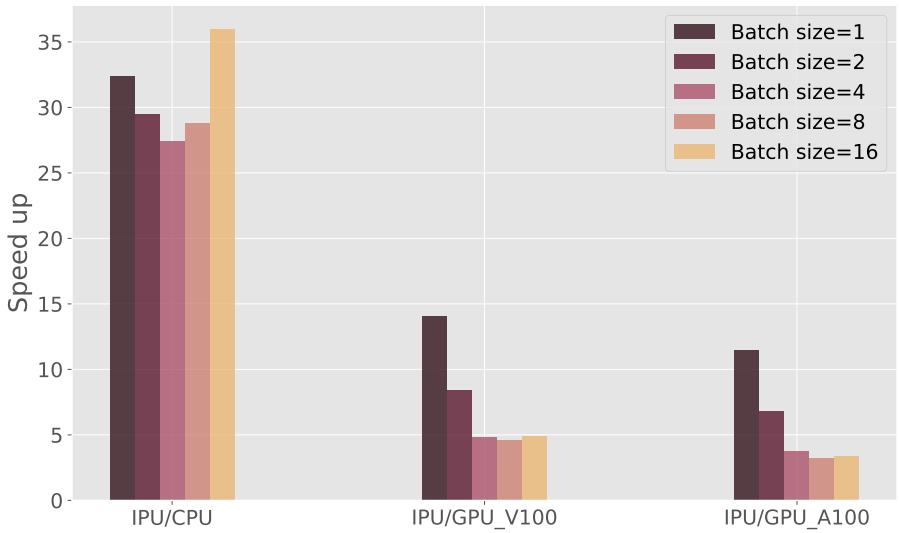


Figure 5.6: Speed up of IPU relative to the CPU and GPU for varying batch sizes (1, 2, 4, 8, 16).

speedup. This indicates that the IPU is faster than the GPU, but the margin of difference is not as great as the margin between the IPU and CPU. Additionally, it's noteworthy that for the smallest batch size of 1, we observe the most significant speedup, approximately a 15-fold increase, which diminishes and plateaus after batch size 4 to around a 5-fold speedup. This suggests that the IPU's efficiency compared to the CPU diminishes with larger batch sizes, while still maintaining a substantial advantage over both CPU and GPU across all batch sizes.

These results imply that for machine learning tasks where the batch size is limited to smaller numbers, IPUs give a considerable performance boost compared to both CPUs and GPUs. The specific architecture of the IPU, which is designed to efficiently process machine learning algorithms, allows it to excel in training speeds at smaller batch sizes. This observation is corroborated by the results presented in [Mohan et al., 2020], where they also demonstrated that IPUs outperform GPUs in scenarios involving smaller batch sizes. However, it is also implied that if the batch size were increased, the GPU might see a relative increase in efficiency due to its parallel processing capabilities, potentially closing the gap in speedup relative to the IPU.

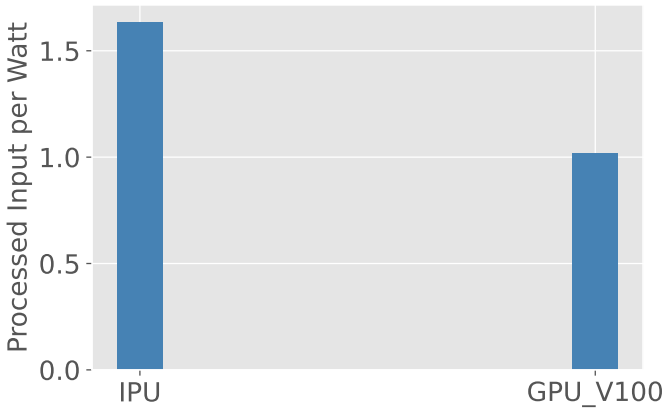


Figure 5.7: Number of processed inputs per Watt for IPU and GPU_V100 for batch size=4.

Figure 5.7 presents a comparative analysis of the throughput-to-power ratio between the IPU and the V100 GPU. The results demonstrate a noteworthy disparity in efficiency, with the IPU exhibiting over a 60% improvement in throughput efficiency compared to the V100 GPU. This indicates that, per unit of energy consumed, the IPU is capable of processing a significantly greater volume of input data. This finding underscores the superior energy utilization and processing efficacy of the IPU architecture.

Chapter 6

CONCLUSION AND FUTURE WORK

This thesis examined the performance assessment of a multiple object tracking model, with a specific focus on comparing the efficiency of Graphcore's IPUs versus traditional GPUs for training a GNN for this application. Through the process of migrating an existing PyTorch implementation to TensorFlow and optimizing it for IPU execution, this study conducted a comprehensive analysis, offering detailed insights into the training and inference performance disparities between these two computing architectures.

The key performance evaluation metrics encompassed the average training time per epoch and average inference time per epoch, utilizing a consistent dataset across all experiments. The findings align with existing literature [Nasari et al., 2022, Mohan et al., 2020, Arcelin,], highlighting IPUs' superior performance over GPUs in scenarios with smaller batch sizes. This advantage is mainly because IPUs can use computing resources efficiently in such situations, while GPUs face limitations in fully harnessing their parallel processing capabilities with smaller batch sizes.

A crucial insight from this research is the impact of device-optimized functions on performance. Specifically, Graphcore's optimized scatter and gather functions showcased a significant acceleration, achieving approximately a fourfold increase in speed when juxtaposed with the default implementations within TensorFlow. Furthermore, IPU-specific settings like I/O tiles and prefetch depth were found to considerably impact performance.

A major challenge was the limited memory capacity of IPUs compared to GPUs. While Graphcore's IPU is designed with the intent for users to deploy multiple units in tandem for large-scale applications, my research focused specifically on investigating the capabilities of a single IPU. This architectural choice constrained my experiments to smaller input graph sizes, since my research was specifically

aimed at exploring the capabilities of a single IPU. Although attempts were made to implement data and model parallelization, the anticipated performance gains were not fully realized within the allocated time frame and project scope. This leaves room for exploration in multi-IPU training in future research.

Moving forward, the research plan involves increasing the scale of the model to a multi-IPU framework to accommodate larger datasets and further improve accuracy. This growth will also provide a chance to evaluate the IPU's capability in parallelization across multiple devices. Furthermore, to enhance performance, the impacts of mixed-precision training should be explored.

This research has significantly deepened my knowledge in several key domains: GNNs, MPNs, Graphcore IPUs, and the nuanced process of model development and optimization. Through this experience, my ability to carry out performance comparisons in high performance computing, as well as my general research skills in machine learning, have been greatly improved.

Moreover, the project highlighted the crucial function of performance profiling. By utilizing profiling techniques, I could pinpoint and correct inefficiencies in the model early on, resulting in enhancements through the integration of IPU-optimized operations.

In summary, this thesis represents a step forward in understanding the dynamics of GNN training on an IPU. Through a comparison between Graphcore IPUs and traditional GPUs, it has highlighted the subtle advantages of IPUs in scenarios demanding efficiency with smaller batch sizes. The challenges encountered and navigated underscore the pivotal significance of hardware optimization in advancing machine learning applications.

BIBLIOGRAPHY

[ex3, 2024] (2024). ex3. `https://www.ex3.simula.no/`. Accessed: January 16, 2024.

[Arcelin,] Arcelin, B. Comparison of graphcore ipus and nvidia gpus for cosmology applications (2021). *DOI: https://doi. org/10.48550/ARXIV*, 2106.

[Battaglia et al., 2018] Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

[Bewley et al., 2016] Bewley, A., Ge, Z., Ott, L., Ramos, F., and Upcroft, B. (2016). Simple online and realtime tracking. In *2016 IEEE international conference on image processing (ICIP)*, pages 3464–3468. IEEE.

[Bilbrey et al., 2022] Bilbrey, J. A., Herman, K. M., Sprueill, H., Xantheas, S. S., Das, P., Roldan, M. L., Kraus, M., Helal, H., and Choudhury, S. (2022). Reducing down (stream) time: Pretraining molecular gnns using heterogeneous ai accelerators. *arXiv preprint arXiv:2211.04598*.

[Brasó et al., 2022] Brasó, G., Cetintas, O., and Leal-Taixé, L. (2022). Multi-object tracking and segmentation via neural message passing. *International Journal of Computer Vision*, 130(12):3035–3053.

[Brasó and Leal-Taixé, 2020] Brasó, G. and Leal-Taixé, L. (2020). Learning a neural solver for multiple object tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6247–6257.

[Brasó and Leal-Taixé, 2020] Brasó, G. and Leal-Taixé, L. (2020). Learning a neural solver for multiple object tracking github. `https://github.com/dvl-tum/mot_neural_solver`.

[Chandrajit et al., 2016] Chandrajit, M., Girisha, R., and Vasudev, T. (2016). Multiple objects tracking in surveillance video using color and hu moments. *Signal & Image Processing: An International Journal*, 7(3):15–27.

[Chen et al., 2022] Chen, C., Wu, Y., Dai, Q., Zhou, H.-Y., Xu, M., Yang, S., Han, X., and Yu, Y. (2022). A survey on graph neural networks and graph transformers in computer vision: a task-oriented perspective. *arXiv preprint arXiv:2209.13232*.

[Cobos et al., 2019] Cobos, R., Hernandez, J., and Abad, A. G. (2019). A fast multi-object tracking system using an object detector ensemble. In *2019 IEEE Colombian Conference on Applications in Computational Intelligence (ColCACI)*, pages 1–5. IEEE.

[Cui et al., 2023] Cui, Y., Zeng, C., Zhao, X., Yang, Y., Wu, G., and Wang, L. (2023). Sportsmot: A large multi-object tracking dataset in multiple sports scenes. *arXiv preprint arXiv:2304.05170*.

[Dwivedi et al., 2023] Dwivedi, V. P., Joshi, C. K., Luu, A. T., Laurent, T., Bengio, Y., and Bresson, X. (2023). Benchmarking graph neural networks. *Journal of Machine Learning Research*, 24(43):1–48.

[Elhoseny, 2020] Elhoseny, M. (2020). Multi-object detection and tracking (modt) machine learning model for real-time video surveillance systems. *Circuits, Systems, and Signal Processing*, 39:611–630.

[Fan et al., 2019] Fan, W., Ma, Y., Li, Q., He, Y., Zhao, E., Tang, J., and Yin, D. (2019). Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426.

[Feng and Meunier, 2022] Feng, M. and Meunier, J. (2022). Skeleton graph-neural-network-based human action recognition: A survey. *Sensors*, 22(6):2091.

[Freund and Moorhead, 2020] Freund, K. and Moorhead, P. (2020). The graphcore second-generation ipu.

[Gao and Hao, 2021] Gao, J. and Hao, L. (2021). Graph neural network and its applications. In *Journal of Physics: Conference Series*, volume 1994, page 012004. IOP Publishing.

[Gao et al., 2019] Gao, J., Zhang, T., and Xu, C. (2019). Graph convolutional tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4649–4659.

[Gilmer et al., 2017] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR.

[Graphcore, 2022] Graphcore (2022). Custom grouped gather scatter. `https://github.com/graphcore/ogb-lsc-pcqm4mv2/tree/main/static_ops`.

[Graphcore, 2020] Graphcore, I. (2020). Introducing the colossus™ mk2 gc200 ipu.

[Guo et al., 2022] Guo, S., Wang, S., Yang, Z., Wang, L., Zhang, H., Guo, P., Gao, Y., and Guo, J. (2022). A review of deep learning-based visual multi-object tracking algorithms for autonomous driving. *Applied Sciences*, 12(21):10741.

[Helal et al., 2022] Helal, H., Firoz, J., Bilbrey, J., Krell, M. M., Murray, T., Li, A., Xantheas, S., and Choudhury, S. (2022). Extreme acceleration of graph neural network-based prediction models for quantum chemistry. *arXiv preprint arXiv:2211.13853*.

[Hosseini et al., 2022] Hosseini, R., Simini, F., and Vishwanath, V. (2022). Operation-level performance benchmarking of graph neural networks for scientific applications. *arXiv preprint arXiv:2207.09955*.

[Islam et al., 2020] Islam, M. M., Islam, M. R., and Islam, M. S. (2020). An efficient human computer interaction through hand gesture using deep convolutional neural network. *SN Computer Science*, 1:1–9.

[Jia et al., 2019] Jia, Z., Tillman, B., Maggioni, M., and Scarpazza, D. P. (2019). Dissecting the graphcore ipu architecture via microbenchmarking. *arXiv preprint arXiv:1912.03413*.

[Kamkar et al., 2020] Kamkar, S., Ghezloo, F., Moghaddam, H. A., Borji, A., and Lashgari, R. (2020). Multiple-target tracking in human and machine vision. *PLoS computational biology*, 16(4):e1007698.

[Kieritz et al., 2018] Kieritz, H., Hubner, W., and Arens, M. (2018). Joint detection and online multi-object tracking. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 1459–1467.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

[Kumaran and Reddy, 2017] Kumaran, N. and Reddy, U. S. (2017). Object detection and tracking in crowd environment—a review. In *2017 International Conference on Inventive Computing and Informatics (ICICI)*, pages 777–782. IEEE.

[Li et al., 2020] Li, J., Gao, X., and Jiang, T. (2020). Graph networks for multiple object tracking. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, pages 719–728.

[Li et al., 2021] Li, M., Chen, S., Chen, X., Zhang, Y., Wang, Y., and Tian, Q. (2021). Symbiotic graph neural networks for 3d skeleton-based human action recognition and motion prediction. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(6):3316–3333.

[Li et al., 2018] Li, P., Wang, D., Wang, L., and Lu, H. (2018). Deep visual tracking: Review and experimental comparison. *Pattern Recognition*, 76:323–338.

[Liu et al., 2019] Liu, P., Li, X., Liu, H., and Fu, Z. (2019). Online learned siamese network with auto-encoding constraints for robust multi-object tracking. *Electronics*, 8(6):595.

[Louw and McIntosh-Smith, 2021] Louw, T. and McIntosh-Smith, S. (2021). Using the graphcore ipu for traditional hpc applications. In *3rd Workshop on Accelerated Machine Learning (AccML)*.

[Luo et al., 2021] Luo, W., Xing, J., Milan, A., Zhang, X., Liu, W., and Kim, T.-K. (2021). Multiple object tracking: A literature review. *Artificial intelligence*, 293:103448.

[Ma et al., 2019] Ma, C., Li, Y., Yang, F., Zhang, Z., Zhuang, Y., Jia, H., and Xie, X. (2019). Deep association: End-to-end graph-based learning for multiple object tracking with conv-graph neural network. In *Proceedings of the 2019 on International Conference on Multimedia Retrieval*, pages 253–261.

[Ma et al., 2021] Ma, C., Yang, F., Li, Y., Jia, H., Xie, X., and Gao, W. (2021). Deep human-interaction and association by graph-based learning for multiple object tracking in the wild. *International Journal of Computer Vision*, 129:1993–2010.

[Meinhardt et al., 2022] Meinhardt, T., Kirillov, A., Leal-Taixe, L., and Feichtenhofer, C. (2022). Trackformer: Multi-object tracking with transformers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 8844–8854.

[Moe et al., 2022] Moe, J., Pogorelov, K., Schroeder, D. T., and Langguth, J. (2022). Implementating spatio-temporal graph convolutional networks on graphcore ipus. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 45–54. IEEE.

[Mohan et al., 2020] Mohan, L. R. M., Marshall, A., Maddrell-Mander, S., O'Hanlon, D., Petridis, K., Rademacker, J., Rege, V., and Titterton, A. (2020).

Studying the potential of graphcore ipus for applications in particle physics. *arXiv preprint arXiv:2008.09210*.

[Nasari et al., 2022] Nasari, A., Le, H., Lawrence, R., He, Z., Yang, X., Krell, M., Tsyplikhin, A., Tatineni, M., Cockerill, T., Perez, L., et al. (2022). Benchmarking the performance of accelerators on national cyberinfrastructure resources for artificial intelligence/machine learning workloads. In *Practice and Experience in Advanced Research Computing*, pages 1–9.

[Papakis et al., 2020] Papakis, I., Sarkar, A., and Karpatne, A. (2020). Gcnnmatch: Graph convolutional neural networks for multi-object tracking via sinkhorn normalization. *arXiv preprint arXiv:2010.00067*.

[Park et al., 2021] Park, Y., Dang, L. M., Lee, S., Han, D., and Moon, H. (2021). Multiple object tracking in deep learning approaches: A survey. *Electronics*, 10(19):2406.

[Poiesi et al., 2013] Poiesi, F., Mazzon, R., and Cavallaro, A. (2013). Multi-target tracking on confidence maps: An application to people tracking. *Computer Vision and Image Understanding*, 117(10):1257–1272.

[Pradhyumna et al., 2021] Pradhyumna, P., Shreya, G., et al. (2021). Graph neural network (gnn) in image and video understanding using deep learning for computer vision applications. In *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*, pages 1183–1189. IEEE.

[Rangesh et al., 2021] Rangesh, A., Maheshwari, P., Gebre, M., Mhatre, S., Ramezani, V., and Trivedi, M. M. (2021). Trackmpnn: A message passing graph neural architecture for multi-object tracking. *arXiv preprint arXiv:2101.04206*.

[Rangesh and Trivedi, 2019] Rangesh, A. and Trivedi, M. M. (2019). No blind spots: Full-surround multi-object tracking for autonomous vehicles using cameras and lidars. *IEEE Transactions on Intelligent Vehicles*, 4(4):588–599.

[Rusch et al., 2023] Rusch, T. K., Bronstein, M. M., and Mishra, S. (2023). A survey on oversmoothing in graph neural networks. *arXiv preprint arXiv:2303.10993*.

[Scarselli et al., 2008] Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2008). The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80.

[Sharma and Jalal, 2021] Sharma, H. and Jalal, A. S. (2021). Visual question answering model based on graph neural network and contextual attention. *Image and Vision Computing*, 110:104165.

[Shekofteh et al., 2023] Shekofteh, S.-K., Alles, C., and Fröning, H. (2023). Reducing memory requirements for the ipu using butterfly factorizations. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, pages 1255–1263.

[Shen et al., 2018] Shen, Y., Li, H., Yi, S., Chen, D., and Wang, X. (2018). Person re-identification with deep similarity-guided graph neural network. In *Proceedings of the European conference on computer vision (ECCV)*, pages 486–504.

[Shi and Rajkumar, 2020] Shi, W. and Rajkumar, R. (2020). Point-gnn: Graph neural network for 3d object detection in a point cloud. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1711–1719.

[Shuai et al., 2021] Shuai, B., Berneshawi, A., Li, X., Modolo, D., and Tighe, J. (2021). Siammot: Siamese multi-object tracking. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12372–12382.

[Shuai et al., 2020] Shuai, B., Berneshawi, A. G., Modolo, D., and Tighe, J. (2020). Multi-object tracking with siamese track-rcnn. *arXiv preprint arXiv:2004.07786*.

[Smeulders et al., 2013] Smeulders, A. W., Chu, D. M., Cucchiara, R., Calderara,

S., Dehghan, A., and Shah, M. (2013). Visual tracking: An experimental survey. *IEEE transactions on pattern analysis and machine intelligence*, 36(7):1442–1468.

[Sumeet et al., 2022] Sumeet, N., Rawat, K., and Nambiar, M. (2022). Performance evaluation of graphcore ipu-m2000 accelerator for text detection application. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, pages 145–152.

[Tang et al., 2022] Tang, S., Zhang, W., Mu, Z., Shen, K., Li, J., Li, J., and Wu, L. (2022). Graph neural networks in computer vision. *Graph Neural Networks: Foundations, Frontiers, and Applications*, pages 447–462.

[Topping et al., 2021] Topping, J., Di Giovanni, F., Chamberlain, B. P., Dong, X., and Bronstein, M. M. (2021). Understanding over-squashing and bottlenecks on graphs via curvature. *arXiv preprint arXiv:2111.14522*.

[Vaswani et al., 2022] Vaswani, K., Volos, S., Fournet, C., Diaz, A. N., Gordon, K., Vembu, B., Webster, S., Chisnall, D., Kulkarni, S., Cunningham, G., et al. (2022). Confidential machine learning within graphcore ipus. *arXiv preprint arXiv:2205.09005*.

[Wang et al., 2021] Wang, Y., Kitani, K., and Weng, X. (2021). Joint object detection and multi-object tracking with graph neural networks. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*, pages 13708–13715. IEEE.

[Wang et al., 2022] Wang, Y., Murnane, D., Choma, N., Farrell, S., Calafiura, P., et al. (2022). Benchmarking gpu and tpu performance with graph neural networks. *arXiv preprint arXiv:2210.12247*.

[Wang et al., 2020a] Wang, Y., Weng, X., and Kitani, K. (2020a). Joint detection and multi-object tracking with graph neural networks. *arXiv preprint arXiv:2006.13164*, 1(2).

[Wang et al., 2020b] Wang, Z., Zheng, L., Liu, Y., Li, Y., and Wang, S. (2020b). Towards real-time multi-object tracking. In *European Conference on Computer Vision*, pages 107–122. Springer.

[Weng et al., 2020] Weng, X., Wang, Y., Man, Y., and Kitani, K. M. (2020). Gnn3dmot: Graph neural network for 3d multi-object tracking with 2d-3d multi-feature learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6499–6508.

[Wilson and Martinez, 2003] Wilson, D. R. and Martinez, T. R. (2003). The general inefficiency of batch training for gradient descent learning. *Neural networks*, 16(10):1429–1451.

[Wojke et al., 2017] Wojke, N., Bewley, A., and Paulus, D. (2017). Simple online and realtime tracking with a deep association metric. In *2017 IEEE international conference on image processing (ICIP)*, pages 3645–3649. IEEE.

[Wu et al., 2023] Wu, W., Shi, X., He, L., and Jin, H. (2023). Turbomgnn: Improving concurrent gnn training tasks on gpu with fine-grained kernel fusion. *IEEE Transactions on Parallel and Distributed Systems*.

[Wu et al., 2013] Wu, Y., Lim, J., and Yang, M.-H. (2013). Online object tracking: A benchmark. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2411–2418.

[Wu et al., 2020] Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24.

[Xiang et al., 2015] Xiang, Y., Alahi, A., and Savarese, S. (2015). Learning to track: Online multi-object tracking by decision making. In *Proceedings of the IEEE international conference on computer vision*, pages 4705–4713.

[Xu et al., 2018] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2018). How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*.

[Xu et al., 2020] Xu, Y., Osep, A., Ban, Y., Horaud, R., Leal-Taixé, L., and Alameda-Pineda, X. (2020). How to train your deep multi-object tracker. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 6787–6796.

[Xu et al., 2019] Xu, Y., Zhou, X., Chen, S., and Li, F. (2019). Deep learning for multiple object tracking: a survey. *IET Computer Vision*, 13(4):355–368.

[Ying et al., 2018] Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. (2018). Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 974–983.

[Yu et al., 2022] Yu, E., Li, Z., and Han, S. (2022). Towards discriminative representation: Multi-view trajectory contrastive learning for online multi-object tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8834–8843.

[Zhang et al., 2022] Zhang, Y., Sun, P., Jiang, Y., Yu, D., Weng, F., Yuan, Z., Luo, P., Liu, W., and Wang, X. (2022). Bytetrack: Multi-object tracking by associating every detection box. In *European Conference on Computer Vision*, pages 1–21. Springer.

[Zhang et al., 2021] Zhang, Y., Wang, C., Wang, X., Zeng, W., and Liu, W. (2021). Fairmot: On the fairness of detection and re-identification in multiple object tracking. *International Journal of Computer Vision*, 129:3069–3087.

[Zhang et al., 2020] Zhang, Z., Leng, J., Ma, L., Miao, Y., Li, C., and Guo, M. (2020). Architectural implication of graph neural networks. *IEEE Computer Architecture Letters*, page 1–1.

[Zheng et al., 2021] Zheng, L., Tang, M., Chen, Y., Zhu, G., Wang, J., and Lu, H. (2021). Improving multiple object tracking with single object tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2453–2462.

[Zhou et al., 2022] Zhou, H., Zheng, D., Nisa, I., Ioannidis, V., Song, X., and Karypis, G. (2022). Tgl: A general framework for temporal gnn training on billion-scale graphs. *arXiv preprint arXiv:2203.14883*.

[Zhou et al., 2020a] Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. (2020a). Graph neural networks: A review of methods and applications. *AI open*, 1:57–81.

[Zhou et al., 2020b] Zhou, X., Koltun, V., and Krähenbühl, P. (2020b). Tracking objects as points. In *European conference on computer vision*, pages 474–490. Springer.

[Zhou et al., 2019] Zhou, X., Wang, D., and Krähenbühl, P. (2019). Objects as points. *arXiv preprint arXiv:1904.07850*.