Parallel Processing Systems for Data and Computation Efficiency with Applications to Graph Computing and Machine Learning

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Li Zhou

Graduate Program in Computer Science and Engineering

The Ohio State University

2019

Dissertation Committee:

Radu Teodorescu, Advisor

Gagan Agrawal

Feng Qin

P. (Saday) Sadayappan

Abstract

Graph computing and machine learning play crucial roles in the Big Data era. Unfortunately, modern computer systems are facing increasingly performance challenges of running different applications that are undermining the benefits of available computational resources, emphasizing the need to explore efficient software/hardware co-design. Furthermore, new applications such as smart homes, smart cities, and autonomous vehicles are becoming increasingly important, driving the increased interest in deploying graph and machine learning applications on different computer systems. This dissertation proposes software/hardware co-design solutions that improve the data and computation efficiency and concurrency with applications to big data and machine learning. Depending on the characteristics of applications and targeting computer systems, dedicated designs are required to fully utilize the computational potentials. This dissertation concentrates on efficiently executing graph and machine learning applications on different computer systems, from a single powerful computer, a distributed high-end servers cluster, to the resource-constrained edge devices. To demonstrate the effectiveness of these techniques, this dissertation presents two graph processing systems with efficient data access and concurrency support, and then proposes adaptive parallel execution of neural networks on heterogeneous edge devices.

To improve the data and computation efficiency for graph applications, this dissertation proposes to organize a graph as a set of edge-sets, and achieve better data locality by consolidating sparse edge-sets with multi-modal graph organization. The framework explores

out-of-core execution by streaming the edge-sets from the disk on several cases of large scale linked data, in which it exhibits up to 10× performance improvement for a single machine as a result of several innovations. Another important contribution of this study relates to the comparison of vertex-centric and edge-centric engines on modern architectures. This work conducts throughout experiments and improves our understanding on the differences among these graph computational models.

To improve the concurrency, this dissertation proposes C-Graph (i.e. Concurrent Graph), an edge-set based graph traversal framework with improved scalability for large scale graphs in a highly concurrent distributed environment. In contrast to most prior works, which focus on accelerating a single graph processing task, in industrial practice we consider multiple graph processing tasks running concurrently, such as a group of queries issued simultaneously to the same graph. The framework achieves both high concurrency and efficiency for k-hop reachability concurrent queries, by maintaining global vertex states and exploring shared access between edge-sets and among queries to facilitate graph traversals. We extend the framework for different types of applications by using a simple message passing mechanism with synchronous and asynchronous communication. We experimentally show that our framework obtains 20× $\sim$ 70× speedup over baselines with up to 300+ concurrent queries.

To utilize the available computational resources of edge devices in Internet of Things (IoT) environments, this study proposes a runtime adaptive convolutional neural network (CNN) acceleration framework that is optimized for heterogeneous IoT environments. The framework leverages spatial partitioning techniques through fusion of the convolution layers and dynamically selects the optimal degree of parallelism according to the availability of computational resources, as well as network conditions. Our evaluation shows that our

framework outperforms state-of-art approaches by improving the inference speed and reducing communication costs while running on wirelessly-connected Raspberry-Pi3 devices. Experimental evaluation shows up to $1.9\times \sim 3.7\times$ speedup using 8 devices for three popular CNN models.

To my family.

Vita

August 2013 - present	PhD Student, Computer Science and Engineering, The Ohio State University, USA.
Dec 2018	MS, Computer Science and Engineering, The Ohio State University, USA.
June 2008	BEng, Electronic Science and Technology, Huazhong University of Science and Technology, China.

Publications

Research Publications

Li Zhou, Hao Wen, Radu Teodorescu, David Hung-Chang Du. Distributing Deep Neural Networks with Containerized Partitions at the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge'19)*, July 2019.

Kristin Barber, Li Zhou, Anys Bacha, Yinqian Zhang, Radu Teodorescu. Isolating Speculative Data to Prevent Transient Execution Attacks. In *IEEE Computer Architecture Letters (CAL)*, May 2019.

Xiang Pan, Anys Bacha, Spencer Rudolph, Li Zhou, Yinqian Zhang, Radu Teodorescu. NVCool: When Non-Volatile Caches Meet Cold Boot Attacks. In *IEEE International Conference on Computer Design (ICCD'18)*, October 2018.

Li Zhou, Ren Chen, Yinglong Xia, Radu Teodorescu. C-Graph: A Highly Efficient Concurrent Graph Reachability Query Framework. In *International Conference on Parallel Processing (ICPP'18)*, August 2018.

Qingsong Wen, Ren Chen, Yinglong Xia, Li Zhou, Juan Deng, Jian Xu, Mingzhen Xia. Big-Data Helps SDN to Verify Integrity of Control/Data Planes. In *Big-Data and Software Defined Networking, IET Book Series on Big Data*, March 2018.

Qingsong Wen, Ren Chen, Lifeng Nai, Li Zhou, Yinglong Xia. Finding Top K Shortest Simple Paths with Improved Space Efficiency. In *ACM SIGMOD/PODS International Workshop on Graph Data-management Experiences and Systems (GRADES'17)*, May 2017.

Li Zhou, Yinglong Xia, Hui Zang, Jian Xu, Mingzhen Xia. An Edge-Set Based Large Scale Graph Processing System. In *IEEE International Conference on Big Data (BigData'16)*, December 2016.

Renji Thomas, Kristin Barber, Naser Sedaghati, Li Zhou, Radu Teodorescu. Core Tunneling: Variation-Aware Voltage Noise Mitigation in GPUs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*, March 2016.

Li Zhou, Avinash Karanth Kodi. PROBE: Predication-based Optical Bandwidth Scaling for Energy-efficient NoCs. In *IEEE/ACM International Symposium on Networks-on-Chip (NOCS'13)*, March 2013.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Dr. Radu Teodorescu
High Performance Computing	Dr. P. (Saday) Sadayappan
Data Management & Mining	Dr. Srinivasan Parthasarathy

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

1.1 Challenges for Graph Computing and Machine Learning on Modern Computer Systems

The ability to run large scale graph analytics on a single machine. Compared to other big data applications, graph analytics typically impose significant performance challenges, which in turn limit the adoption of the technology in some big data scenarios and underutilize the modern computer systems. One of these challenges is driven by the poor data locality due to irregular data access. As a result, graph processing is typically bounded by the I/O latency of a platform [31, 96], not its computational capability. This motivates the deployment of graph processing on single machines (as opposed to distributed solutions), where the high, in-system disk and memory bandwidth can be leveraged to process large scale graphs beyond the memory limit. Notable examples include GraphChi [55] and X-Stream [77]. Our first work explores several optimization opportunities for a single machine out-of-core graph processing system.

The capability of executing concurrent queries for large-scale graphs. Another challenge for graph computing is to efficiently handle concurrent queries. Unfortunately, most existing graph processing frameworks are optimized for a single application, and not capable of responding to concurrent queries. In enterprise applications, a system must gracefully

handle multiple queries at the same time, as well as simultaneous query requests from multiple users. In contrast, graph databases are designed with concurrency in mind, but with poor performance in general, especially in terms of handling large scale graphs or high volumes of concurrent queries [96]. This motives our second work, which extends our single machine graph processing system to distributed environments with concurrency support.

The efficient deployment of machine learning on the resources-constrained devices. Meanwhile, new applications such as smart homes, smart cities, and autonomous vehicles are driving an increased interest in deploying graph computing and machine learning on different platforms from the servers to the edge devices. Unfortunately, deploying deep neural networks (DNNs) on resource-constrained devices presents significant challenges. Prior solutions attempted to address these challenges by either sacrificing accuracy or by relying on cloud resources for assistance. This motives our third work, which explores the parallel execution of Convolutional Neural Networks on a small cluster of edge devices.

1.2 Thesis Statement

Software/hardware co-design solutions that understand the behaviors of applications, can enable effective and efficient graph computing and machine learning by leveraging different computer systems to achieve better performances and scale to large scale datasets.

1.3 Our Contributions

In this thesis, we make multiple efforts to run graph and machine learning applications efficiently on different computer systems. From a single machine, a high-end servers cluster, to a small cluster of edge devices, we propose application-specific optimizations to improve

the performance by utilizing the avaiable computational potentials. We summarize our contributions as follows.

Introducing Edge-set into Large-scale Graph Processing [111]:

In Chapter 3, we present a single machine out-of-core graph processing engine that explores edge-set organization in large-scale linked data, in which it exhibits superior performance as a result of several innovations. In this work, we introduce a novel edge-set based representation for large scale graph processing that is naturally related to the parallel sliding window (PSW) and is straightforward to handle when exchanging/prefetching data between memory and disk. We design a method for storing multiple sparse edge-sets with affinity into the same physical block, so that the data locality can be improved. We use an approach to incrementally update the graph structure while keeping data consistent across edge-sets. A multi-modal representation of an edge-set is explored to integrate with various graph processing algorithms. A comprehensive architectural study of different graph computing models within modern graph frameworks is another contribution of this work.

Scaling Concurrent Graph Reachability Queries to 100 Billion Edges [109]:

In Chapter 4, we present an edge-set based graph traversal framework called C-Graph (i.e. Concurrent Graph). A simple range-based partition is adopted to reduce the overhead of complex partitioning scheme for large-scale graphs. By using multi-mode, edge-set based graph data structures, sparsity and cache locality are optimized in each partition to achieve the best performance for different access patterns. The framework explores data locality between overlapped subgraphs inside/between queries, and utilize bitwise operations and shared global states for efficient graph traversals. In order to solve the memory limitation of concurrent graph queries in a single instance, we utilize dynamic resource allocation during graph traversals. Instead of saving a value per each vertex, we only store values for

the previous and current levels. Synchronous/asynchronous update models are supported for different types of graph applications, such as graph traversals (e.g BFS) and iterative computation (e.g. PageRank). Our system targets the reduction of the average response times for concurrent graph queries on large-scale graphs with up to **100 billion** edges in distributed environments.

These innovations [109, 111] have been incorporated in part into a real industry graph engine called Huawei Eywa [5].

Distributing Deep Neural Networks at the Edge [110]:

In Chapter 5, we propose a collaborative CNN acceleration framework that adapts to the computing resources and network condition in the presence of heterogeneity. We discuss the trade-offs in partitioning DNN inference among multiple lightweight edge devices, and propose an Adaptive Optimal Fused-layer (AOFL) parallelization to reduce the communication cost in an IoT network. We design a dynamic programming-based search algorithm to decide the optimal partition and parallelization for a DNN model. We present a collaborative CNN acceleration framework that adapts to the computing resources and network condition in the presence of heterogeneity. We apply our technique on a distributed IoT cluster consisting of Raspberry-Pi3-based hardware and evaluate image recognition and object detection DNN models.

Chapter 2: Background and Related Work

2.1 Graph Computing and Processing Systems

Graph. A graph is denoted by $G = (V, E)$, where V is a set of vertices and E is a set of edges connecting the vertices; an edge $e = \{s, t, w\} \in E$ is a directed link from vertex s to t, with weight w for a weighted graph. Note that in graph database terminology the weight w can also be referred to as the property of edge e.

Graph Representation. A graph can be represented in various ways in terms of in-memory data organization. Those representations lead to highly variable architectural behaviors, overall performance, and, especially, memory sub-system efficiency. For example, the Coordinate List (COO) format is an intuitive representation of a graph that is consistent with the edge list inputs. Each edge is represented as a three element pair of source, destination and weight on the edge. They are often organized as arrays to improve memory efficiency and performance. One of the most popular data representation structures is Compressed Sparse Row (CSR). CSR organizes vertices, edges, and properties of a graph in separate compact arrays. Comparing to other dynamic data structures with indices, the compact format of CSR reduces memory footprint and simplifies the build/copy/transfer complexity of a graph. It is widely used in multiple systems [55, 63, 65]. However, the compact data structure of CSR brings significant overhead in data movement when structural update occurs

to the graph. In reality, it is not uncommon to dynamically update the topology or properties of a graph. Thus, a flexible data representation is desirable in many cases. For example, GraphChi, as well as several other graph computing frameworks, utilize a vertex-centric structure for graph representation, in which a vertex is the basic unit of a graph, and all vertices form up an adjacency list with indices. However, such an inconsistency between the graph data storage and representation in GraphChi causes high random memory access when processing the in-edges of a vertex. Multi-mode representation has been proposed recently to achieve better performance by dynamically choosing the graph data representation best suited for the behaviors of individual applications [112].

Graph Computing and Frameworks. Graph computing has become increasingly popular in big data applications due to its wide range of use cases including graph databases and data explorations. The large scale problem sets introduce diverse features of graph computing, including frameworks, data representations, and computation types.

Modern graph computing systems usually support generic graph computing frameworks with standard graph programming interfaces. By hiding the details of graph data management, users can easily write applications with primitives provided by the graph computing frameworks. The abstraction of programming interfaces in graph computing frameworks not only simplifies algorithm implementation, but also ensures portability of applications across different graph computing frameworks. Examples of graph computing frameworks include GraphLab [63], Pregel [65], Apache Giraph [3]. They have many similarities in computation models and user primitives.

There are other single machine graph engines using alternative approaches other than GraphChi and X-stream. GraphMat [85] is a single-node multicore in-memory graph framework. It maps vertex programs to high performance sparse matrix operations. Ligra [82]

is a lightweight graph processing framework for shared-memory, which provides two very simple routines for mapping over vertices and edges. TurboGraph [41] saves the outgoing edges of several vertices in pages within a buffer pool in the memory. Vertices are divided into several chunks to ensure data access locality. VENUS [22] enables Vertex-centric Streamlined Processing (VSP) and proposes v-shards to store the source vertices of edges in each shard. GridGraph [113] uses 2D partitioning and fine tunes the partition size to achieve better cache locality. GraphQ [92] aims to answer queries by analyzing partial graphs in stead of computing the whole graph. NXgraph [23] proposed Destination-Sorted Sub-Shard (DSSS) structure to store a graph to ensure graph data access locality and enable fine-grained scheduling. It utilized three update strategies based on the graph size and memory budget.

Computational Models. The vertex-centric computing model performs operations on each vertex of a graph in parallel. It relies on user defined functions and has been widely accepted by most graph systems to implement various graph algorithms. However, the high variability in vertex connectivity generally leads to unbalanced workloads with poor performance in parallel execution. Some graph computing systems, like X-Stream [77] and GridGraph [113], utilize the edge-centric model. Computation is performed on each edge of a graph instead of on each vertex. The edge-centric model is often efficient and concise, but requires more effort from programmers. To achieve both the programmer friendliness of the vertex-centric model and efficiency of edge-centric model, our proposed system implements edge-centric computing, while providing vertex-centric interfaces for convenience.

Hardware Innovations. Graph computing workloads usually show extremely poor performance on modern processors due to the irregular data access. The memory sub-system not only suffers from low data locality, but can also be very small for real-world graphs in the big data era. Researchers start leveraging novel architectures for graph computing. In addition

to GPUs and FPGAs that gain increasing interests for graph applications, recent literature [38] introduces specially designed graph computing accelerators. Emerging high capacity memory techniques inspire some near-data computing efforts, looking into instruction offload for memory sub-systems [7, 8, 69].

2.2 Graph Traversal and Concurrent Queries

Graph Travesal. Graph traversal is the process of visiting a graph by starting from a given source vertex (a.k.a. the root) and then following the adjacency edges in certain patterns to visit the reachable neighborhood iteratively. Examples of basic graph traversal methods include visiting a graph in breadth-first-search (BFS) and/or depth-first-search (DFS) manners. Most graph applications or queries are essentially performing computations on the vertex values and/or edge weights while traversing the graph structure. For example, the single-source-shortest-path (SSSP) algorithm finds the shortest paths from a given source vertex to other vertices in the graph by accumulating the shortest path weights on each vertex with respect to the root.

The *k-hop* reachability query [21] is essentially a local traversal in a graph, which starts from a given source vertex and visits vertices within k hops. It is a widely used building block in graph applications. In practice, the influence of a vertex usually decreases as the number of hops increases. Therefore, for most applications, potential candidates will be found within a small number of hops. In addition, real-world networks are often tightly connected. For example, Figure 2.1 shows the cumulative distribution of path lengths over all vertex pairs in the Slashdot Zoo network. In this network, the diameter (δ) equals 12. The 50-percentile effective diameter ($\delta_{0.5}$) equals 3.51, and 90-percentile effective diameter

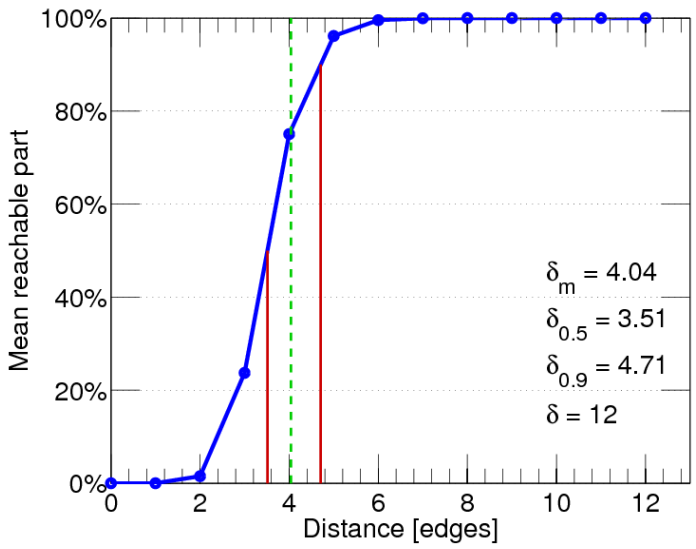


Figure 2.1: The hop plot for Slashdot Zoo graphs [54].

($\delta_{0.9}$) equals 4.71. So most of the network will be visited with less than 5 hops, which is consistent with the six-degrees-of-separation theory in social networks.

The *k-hop* query is frequently employed as an intermediate "operator" between low-level databases and high-level algorithms [64]. Many higher-level functions such as triangle counting, which is equivalent to finding vertices that are within 1 and 2-hop neighbors of the same vertex, can be described and implemented in terms of *k-hop* traversal. Breadth-first search (BFS) is a special case of *k-hop*, where $k \to \infty$. As a result, a graph database's ability to handle *k-hop* access patterns is a good predictor of its performance.

Concurrent Queries in Large-scale Graphs. The ability to handle concurrent queries is very important for industrial big data products. However, adding concurrency in graph databases or graph processing systems is challenging. Neo4j [76] and HyperGraphDB [47] focus on supporting online transaction processing (OLTP) on graph data. However, they are not distributed and cannot support web-scale graphs partitioned over multiple machines.

Titan [4] supports distributed graph traversals over multiple machines, but its performance is a concern due to the complexity of its software stack. For example, graph databases like Titan [4], JanusGraph (based on Titan) [6] and Neo4j [76] are designed with multi-query/users in mind. However, their performance when executing concurrent graph queries is generally poor. In our experiments Titan took 10 seconds on average to complete 100 concurrent *3-hop* queries for a graph of 100 million edges. For some of the queries, the response time was as high as 100 seconds. Other graph databases like Neo4j are not distributed and cannot, as a result, support many real world graphs such as web-scale graphs partitioned over multiple machines.

High memory footprint is another challenge for large-scale graph processing. Concurrent graph queries, generally have high memory usage, which can significantly degrade the response times for all queries. As a result, most of the graph processing systems can't be easily changed to run concurrent queries. These systems are usually highly optimized for certain applications with high resources utilization, but system failures may be triggered when running concurrent queries due to memory exhaustion.

Graph Processing Systems with Query Support. State-of-art graph processing systems often have better performance, however, they generally lack native support for concurrent queries. For example, Gemini [112] is an efficient distributed graph computing system, which outperforms C-Graph in single application performance. However, it cannot handle concurrent queries. Executing the queries serially increases the average response time.

There is an increasing interest in concurrent graph processing including queries for graph processing systems [27, 43, 62, 72, 80, 89, 99, 100]. However, we found that prior works on concurrent queries usually evaluate only small graphs, and don't support distributed environments. MS-BFS [89] introduced level sharing and bitwise operation for efficient

multi-source BFS. It works for small-world graphs, and only supports querying in batches and may therefore not be suitable in an interactive multi-user environment. iBFS [62] supports concurrent queries on multi-GPUs. However, iBFS does not partition the graph, it simply copies it on multiple GPUs, and distributes the queries across all of them. The system is limited to graphs that can fit in their entirety in the GPU memory and therefore it only works for small graphs. Wukong [80] is a distributed graph-based RDF store that leverages RDMA-based graph exploration to provide highly concurrent and low-latency queries over large data sets. Congra [72] extended an existing shared-memory graph processing framework and provided a novel scheme for scheduling concurrent graph processing queries on shared memory based systems. It supports more complex graph algorithms involving graph computation so it was only evaluated with small graphs of at most one hundred millions edges.

2.3 Deep Neural Networks at the Edge

CNN layers. A convolutional neural network consists of an *input*, an *output* layer, and multiple hidden layers. The hidden layers typically consist of a stack of convolution layers, normalization layers, ReLU layers (i.e., activation functions), pooling layers, and fully-connected layers. The **Convolution (*conv*) layer** is the core building block of a CNN. It has of a set of learnable filters (kernels), which convolve across the spatial dimension (width and height) of the input volume and generate a 2-dimensional feature map for each filter by computing the dot product between their weights and a small sub-region of the input. As a result, each layer generates a successively higher level abstraction of the input data. The **Batch normalization layer** normalizes features across spatially grouped feature maps to reduce internal covariate shift. To increase non-linearity, an **activation layer** applies an

element-wise activation function such as rectified-linear unit (*relu*) to the input data, which removes negative values from a feature map by setting them to zero. Both layers are less compute intensive and are often optimized to compute with a previous operator. As such, we group them with the corresponding previous layer that produces their input. The **Pooling layer** (e.g., max pooling (*max*)), performs a down-sampling operation along the spatial dimensions. It is used to progressively reduce the spatial size of the representation, number of parameters, memory footprint and amount of computation in the network. Hence, it also controls overfitting. Finally, after several convolution and max pooling layers, the high-level reasoning in the neural network is done via a dense or **fully-connected (fc) layer**. Neurons in a fully-connected layer are connected to all activations in the previous layer. The value of each output is calculated from the weighted sum of all inputs. The **Softmax** and **argmax** layers produce a probability distribution over the classes for classification and select the one with highest probability as the prediction. fc and *conv* layers are among the most compute- and data-intensive layers of a CNN model.

Models overview. VGG-16 was the runner-up in the ImageNet ILSVRC challenge [78] in 2014. Most importantly, it showed that the depth of the network is a critical component for good performance. The network contains 16 $conv/fc$ layers and features a homogeneous architecture that only performs 3x3 convolutions and 2x2 pooling from beginning to end. A downside of VGG-16 is that it uses a lot more memory and parameters in the first fully-connected layer. **Residual neural network (ResNet)** [42] features special skip connections and makes heavy use of batch normalization. It introduces a so-called "identity shortcut connection" that skips one or more layers to overcome the vanishing gradient issue, enabling much deeper networks with good performance. **You only look once (YOLO)** [74] is a real-time object detection system that is tasked with determining the location of certain

objects on a given image, as well as classifying those objects. Similar to ResNet, YOLO is also missing heavy fully-connected layers at the end of the network.

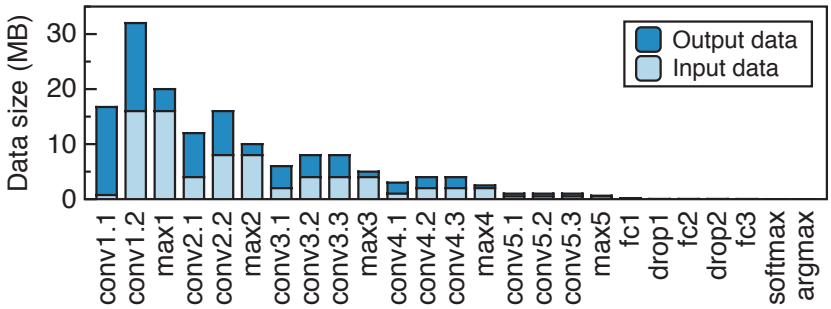


Figure 2.2: The per-layer data size of input and output in VGG-16.

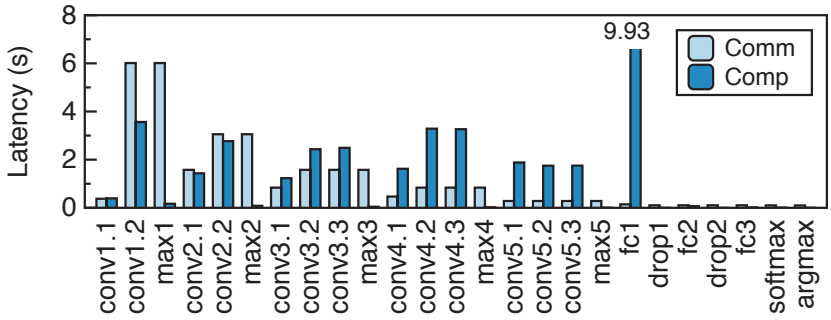


Figure 2.3: The per-layer latency of input communication and layer computation in VGG-16 with a single-core of Raspberry-Pi3 running at 1 GHz in a 25 Mbps local WiFi network.

Key observations. We use VGG-16 to highlight some of the computation and communication characteristics that are typical in CNN models. Figures 2.2 and 2.3 show the per-layer input and output data sizes, as well as the input communication and layer computation latency. This data was collected by executing each VGG-16 layer across two Raspberry-Pi3

devices that had single active core running at 1 GHz and communicated over a 25 Mbps local WiFi network. Overall, we make the following observations from this experiment:

1. Convolution layers dominate the computation time. In VGG-16, 73.8% of the total computation time is spent on *conv* layers. For other models without heavy fc layers at the end of the network like YOLOv2, *conv* layers consume up to 99.93% of the computation time.

2. IoT environments generally rely on wireless communication, which can be slow due to network delays or slow on-device networking hardware. As a result data transfers can be more costly than computation for certain layers as shown in Figure 2.3.

3. The first few layers in the network generate the most output data and are therefore the most communication-intensive. In VGG-16, the cumulative input data size of the first 6 layers represents 66.7% of the total input data size. Figure 2.2 shows the data consumed and produced by each layer in VGG-16, ordered by depth from left to right. We can see that the relative amount of input/output data transferred between layers decreases substantially for deeper layers.

Based on the aforementioned observations, we focus on improving the performance of the convolution layers through paralellization. In addition, device-to-device communication should be avoided for the first few layers of the CNN model where the input and output data sizes are large. Finally, when parallelizing *conv* layers across multiple IoT devices, we need to carefully consider the trade-off between the reduced computation time and the increased communication cost.

DNNs at the Cloud and the Edge. Current techniques enabling DNN-based intelligent applications fall into two categories: *cloud-based* and *edge-only* approaches. Cloud-based

approaches [11, 34, 48, 51, 67, 87, 105] fully (i.e., cloud-only) or partially (i.e., edge-cloud collaboration) offload the computation to the cloud. Neurosurgeon [51] proposes to distribute a DNN model between edge devices and the cloud by deciding a single partition point. In case the model is not pre-installed when offloading remotely, IONN [48] designs incremental model uploading with multiple partition points to overlap the local client and server executions.

Edge-only approaches [30, 37, 40, 46, 66, 71, 98, 108, 110] execute the DNNs on a single edge device with specialized hardware or a small IoT cluster. Many customized accelerators for CNNs have been proposed [9, 18–20, 26, 59, 61, 79]. They focus on improving the performance of CNNs by exploring the potential for resources sharing, and optimizing basic operations. Emerging memory technologies have been explored due to the high memory requirements of machine learning applications.

Our work falls into the category of distributing DNN inference in a small IoT cluster, which have gained increasing research interests recently. NestDNN [30] proposes that multiple compressed models can be dynamically selected based on available runtime resources. Collaborative computing enables a small IoT cluster to run larger models or speedup inference by employing the available idle devices. MoDNN [66] uses layer-wise parallelization, however the MapReduce-like execution results in a large amount of intermediate data to be transferred across devices. Collaborative perception [37] pipelines the computation by partitioning a DNN model and distributing the partitioned blocks to multiple edge devices. Follow-up work in [36] introduces model parallelism methods for both dense and convolution layers in order to address the memory overhead due to model replication. DeepThings [108] reduces the communication cost by fusing the early convolution layers and parallelizing these layers in multiple devices. Our work generalizes the fused-layer approach, proposes a

mechanism for finding optimal layer fusion and device partitioning configurations, and is optimized for heterogeneous IoT environments, which were not addressed in prior work.

Chapter 3: Edge-Set: An Efficient Single Machine Out-of-Core Graph Processing System

3.1 Introduction

In the big data era, the ability to efficiently process large scale graphs is becoming increasingly important, as a great variety of real-world big data scenarios such as social networks, web graphs and etc. are naturally represented as large scale graphs [29,57]. Graph frameworks are now critical components in many big data computing applications, such as Giraph in Hadoop, GraphX in Spark, Gelly in Flink, etc [3,33,94,103].

Compared to other big data systems, graph processing typically imposes significant performance challenges, which in turn limit the adoption of the technology in some big data scenarios. One of these challenges is driven by poor data locality due to irregular data access. As a result, graph processing is typically bounded by the I/O latency [31,96] of a platform, not its computational capabilities. This motivates the deployment of graph processing on single machines (as opposed to distributed solutions) where the high, in-system disk and memory bandwidth can be leveraged.

Single machine graph processing systems are motivated by the high bandwidth of local storage improvement and high communication overhead in distributed environment. Notable examples include GraphChi [55] and X-Stream [77]. GraphChi [55] proposed a

parallel sliding window (PSW) to reduce random access overhead by performing sequential disk I/O operations. However, its performance is poor because of the frequent in-memory graph construction, and the random memory access caused by the vertex-centric graph representation. In contrast, X-Stream [77] proposed an edge-centric graph representation model and optimized the memory behavior through streaming partitions and a novel graph data organization. But its lacks of generic programming interfaces imposes additional burdens when implementing new graph applications.

Prior work paved the way for large scale graph computing. However, those solutions still face efficiency challenges for traversing along the graph structure, such as performing breadth-first search (BFS). In this work, we propose an efficient single machine out-of-core processing system, called Edge-Set. We propose a novel edge-set based graph representation for large scale graph processing that is naturally related to the parallel sliding window (PSW), and is straightforward to handle when exchanging/prefetching data between memory and disk. We store multiple sparse edge-sets with affinity into the same physical block, and utilize multi-modal representation for each edge-set, so that the data locality can be improved. The framework incrementally updates the graph structure while keeping data consistent across edge-sets by streaming in the edge-sets.

Another important contribution of this work relates to the comparison of vertex-centric and edge-centric engines on modern architectures. Modern processors exhibit quite poor performance when running graph applications, mostly due to their random memory access leading to low instruction-per-cycle (IPC) and underutilized memory bandwidth. High variability in architectural behavior was observed for different workloads [14, 16, 49, 70, 95, 97]. To understand the impacts of various processing models on modern architectures, this work conducts a behavioral analysis of CPU and memory sub-systems for graph computing

towards both vertex-centric and edge-centric processing frameworks. In our characterization, we observed the following behaviors: (1) Modern processors show poor instruction-per-cycle (IPC) for graph computing due to significant inefficiency in the memory sub-system. However, the edge-centric model outperforms the vertex-centric model on IPC because of its consistency between graph data organization and processing model; (2) Diverse behaviors were observed among different workloads with respect to the two graph computing models. High variability was observed in the cache hit rate, data translation-lookaside-buffer (DTLB) miss rate, branch miss rate, and the overall performance; (3) Although low hit rates are observed in L2 and L3 caches, the L1 data (L1D) cache can still achieve a high hit rate for small-size meta data; (4) Despite the performance variation of various computation models, they all show inefficient utilization of the cache hierarchy; (5) Big graphs usually result in more page faults compared to medium-size graphs; (6) Graph data volume, density, and connectivity all have significant impacts on multiple architecture parameters. This impact is a function of multiple workload characteristics, As a result, a multifaceted approach is required to greatly improve the performance of graph computing.

The rest of this chapter is organized as follows. In Section 3.2, we introduce our system design and programming interface with implemented graph algorithms. In Section 3.3, we present our methodology, algorithm implementation and analyze experimental results. Finally, in Section 3.5, we conclude this work.

3.2 Edge-set based Graph Processing System

In this section, the main features of our system are covered. To achieve better performance, multiple optimizations have been employed. First, we explore the parallel sliding

window (PSW) based iterative graph computing with our edge-set based graph representation. Second, we discuss the impacts of edge-set consolidation on memory sub-systems and I/O behaviors. Next, we introduce multi-mode graph data organization to provide more flexibility for different types of graphs and applications. Finally, we discuss edge-set based scheduling and prefetching.

3.2.1 Architecture Overview

The architecture of the proposed graph processing system is shown in Figure 3.1. The edge-set generator converts graph data into a set of edge-sets, each consisting of a group of edges for vertices within a certain range. The scheduler loads/preloads corresponding edge-sets from disk to memory for processing. If an edge-set is modified in the edge-set modifier, the resulting edge-set will be persisted into the storage by the evictor. The in-memory edge-set manager maintains the edge-sets that are currently cached in the edge-set buffer and decides which one to evict according to an LRU policy. The edge-set buffer hosts the edge-sets under processing and those prefetched. Graph analysis algorithms are implemented using the gather-apply-scatter (GAS) model, so most current graph algorithms can be easily modified to run on our framework. The compute engine performs the user-defined functions on the edges within each edge-set in parallel.

3.2.2 Edge-set Representation

The edge-set generator in Figure 3.1 produces the edge-sets for an input graph. The input graph is a list of edges consisting of a source vertex, a target vertex, and attributes on the edge. For the sake of simplicity, we assume the vertices are positive integers. This is also the convention of many graph computing frameworks [55, 63]. If viewed as an adjacency matrix, the edge-set representation of a graph is not different from a blocked sparse matrix,

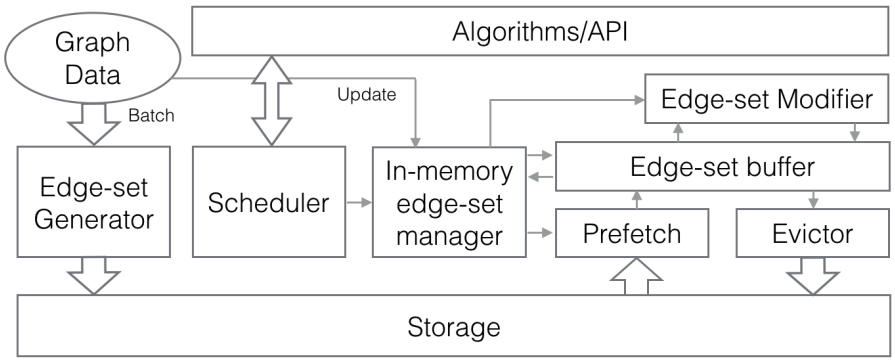


Figure 3.1: Overview of edge-set based processing system architecture.

where the partitioning is determined by evenly distribting the vertex degrees, the same as what used in [55] to determine the vertex ranges for graph sharding.

The edge-sets are naturally related to the parallel sliding window (PSW) in [55], but more flexible. For example, in Figure 3.2, an input graph is represented as three edge lists known as *shards*, each consisting of all the edges to the destination vertices in a certain range. We show the ranges above the shards. To traverse a graph, the PSW works in an iterative manner. For this sample graph, it takes 3 steps to complete the traversal. At the i-th step the edges in the yellow zone are those being processed, which consist of all the edges in the i-th shard and the i-th segments in other shards. These segments are known as the sliding windows, since we always process the edges within a window and the windows simply slide from top to bottom. The yellow zone covers the data to be processed in the current iteration. It is worth noting that, at the i-th step the yellow zone exactly corresponds to the i-th row plus the i-th column of the blocked adjacency matrix. Therefore, traversing a graph is equivalent to scanning the blocked adjacency matrix top-down and left-right simultaneously. Such regularity implies an approach to efficiently prefetch data. Note that the graph sharding

in [55] corresponds to the vertical-only partitioning of the adjacent matrix, which cannot address celebrity vertices with extreme dense incoming edges, but such an issue does not exist in our edge-set based approach where the matrix is partitioned both vertically and horizontally.

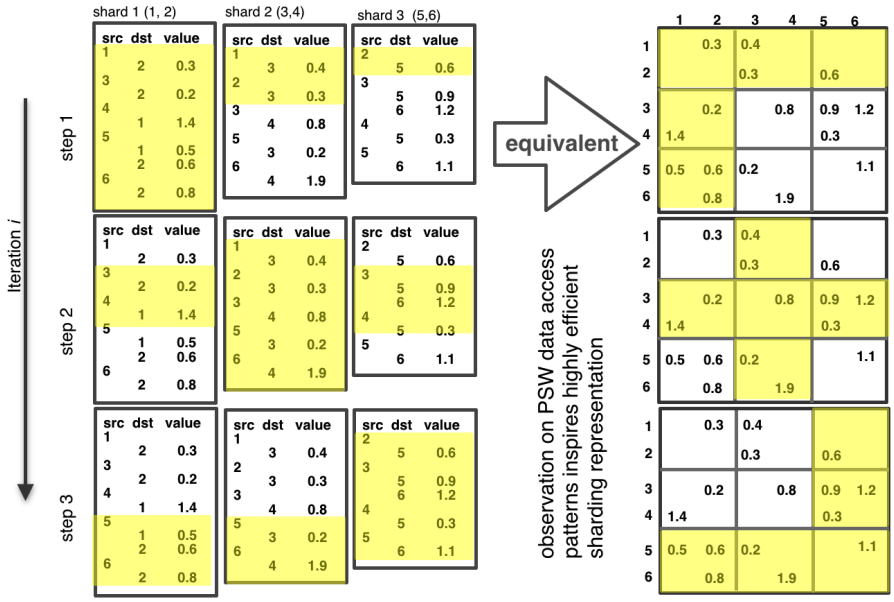


Figure 3.2: Parallel Sliding Window (PSW) in terms of edge-sets. The vertices ranges or matrix blocks in yellow are accessed during each step.

Since all graph algorithms can be implemented using PSW under the Gather-Apply-Scatter (GAS) model (or its variants) [55] and PSW is nothing but all the edge-sets on the same column in the blocked adjacency matrix, we conclude that the edge-set representation of a graph is generic. Generating edge-sets is more straightforward than that in GraphChi where a global sorting is required. In our case, we scan the edge list once to determine the vertex degrees and then we divide the vertices into a set of ranges by evenly distributing the degrees. Then, we scan the edge list again and allocate each edge to an edge-set according

to the ranges where source and destination vertices fall into. Note that both scans can be conducted in a divide-and-conquer manner. Thus, given p parallel threads, the complexity under PRAM is given by $O(m/p)$, where m is the number of input edges. In contrast, GraphChi sorts all edges and then generates the shards. Given sufficient memory (i.e. a single shard for GraphChi) the complexity is $O(m \log m) > O(m/p)$. Note that GraphChi actually utilizes radix sort with complexity $O(km)$, but theoretically $k \leq \log m$. In practice, we also observed improved parallelism and performance using our proposed approach.

Similar with GraphChi, where each shard can fit into memory, the edge-sets containing the same range of vertices (the i-th row or column of the blocked adjacent matrix) can fit into the edge-set buffer at the same time. The system guarantees the first edge-set is evicted only after the compute engine finishes processing the current range of vertices. In addition, the system leaves enough memory to map the vertex properties for those vertices. To further improve the data locality, each edge-set is organized as a set of edge blocks using the same method used for generating an edge-set. The range of vertices of an edge block is decided by the last level cache (LLC) size. The edge block is the smallest unit for processing.

3.2.3 Edge-set Consolidation

The edge-set generator shown in Figure 3.1 can merge small edge-sets. The sparse nature of real large scale graphs can result in some tiny edge-sets that consist of only a few edges each, if not empty. Loading or persisting many such small edge-sets is inefficient due to the IO latency. Therefore, it makes sense to consolidate small edge-sets that are likely to be processed together, so that data locality is potentially increased. Consolidation can occur between adjacent edge-sets both horizontally and vertically, using the following heuristic method. For the sake of simplicity, we look at the horizontal consolidation only. First, we

determine a bound B for the merged set as follows: let k denote the page size of the platform, in term of the number of bytes, and s the size of an edge, then the bound is given by $\lceil \frac{k}{s} \rceil$, which ensures that the resulting set is aligned with the system page, leading to improved IO efficiency. Second, for each edge-set $s_{i,j}$ smaller than the bound, i.e. $|s_{i,j}| < B$, where i, j are the indices of the edge-set in the corresponding adjacency matrix with $N * N$ blocks, it identifies its horizontal neighbor that can minimize the size of the resulting set if merged:

$$\tilde{s} = \min_{j' \in \{j-1, j+1\}, 0<j<N} (|s_{i,j}| + |s_{i,j'}|)$$

If $\tilde{s} < B$, it proposes to merge with the selected neighbor. If two edge-sets select each other, then they are merged. The neighbors of the merged set are the union of the neighbors of the two. We continue the consolidation process repeatedly until no merge occurs anymore. In Figure 3.3, we merge the neighbor sets as long as the size of the merged set is no larger than the given bound, say 4 edges. As a result, edge-sets 1, 2, and 3 are consolidated. Similarly, edge-sets 4 and 5 are merged, and so are 8 and 9.

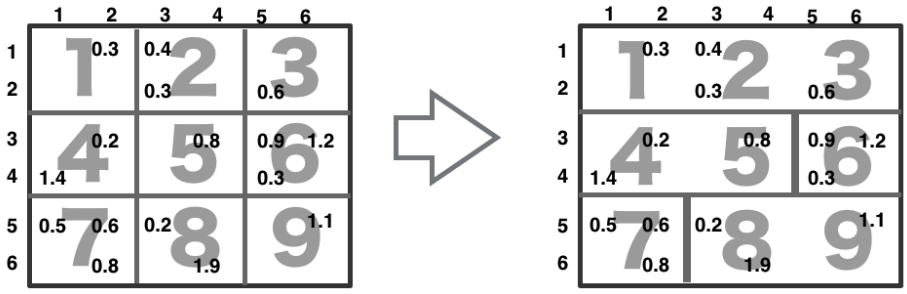


Figure 3.3: An example of horizontal consolidation of logical edge-set to improve data locality.

The horizontal consolidation improves data locality especially when we visit the out-going vertex edges. We can also merge the edge-sets vertically, which benefits the information gathering from the vertex's parents. Note that merging all the edge-sets in a row (column) is equivalent to having the out-going (in-coming) edge list of the vertex. Note that the edge-set consolidation is transparent to users, that is, the users will still see 9 edge-sets when implementing graph algorithms; but physically there are only 5 edge-sets physically stored on disk. The proposed system maintains the mapping between the logical edge-sets and physical edge-sets. Once a logical edge-set is prefetched, the system is aware that all logical edge-sets co-existing in the same physical edge-set become available in memory, which are likely to be processed immediately. Thus, the temporal data locality is improved.

3.2.4 Multi-modal Organization

We allow multi-modal data organization for edge-sets, because of the significant impact the organization format has on individual graph computing algorithms. We take two formats as an example: The coordinate format (a.k.a. COO) in our context is simply a list of edges, each having a source vertex ID, a destination vertex ID, and some attributes on the edge; the compressed sparse row (CSR) in our context sorts COO according to the source vertices and then compresses the list by eliminating the repeated source vertices. Both can be found in the literature for sparse matrices and graphs. The impact of COO and CSR on performance varies according to the graph processing algorithms. Specifically, for the same input graph, we observed better performance for performing PageRank using COO than CSR, although CSR helps the IO a little bit due to compression. However, for breadth-first search (BFS), CSR shows higher noticeable advantage. The reason is that the CSR allows locating a vertex quickly as it is sorted; while COO requires filtering the edge-set when seeking a particular

vertex. However, due to high sparsity, COO may help save the memory required to present a graph than CSR where each vertex has a pointer. Note that in PageRank all edges are visited in each iteration of the algorithm, regardless of the order of the edges; while in BFS, the graph topology must be followed to visit the neighbors of the vertices encountered in the previous iteration.

3.2.5 Scheduling and Prefetching

The scheduler shown in Figure 3.1 applies the user-defined vertex program to the graph and coordinates with the *in-memory edge-set manager*. The manager maintains the buffer of edge-sets. The scheduler notifies the manager which edge-sets to be processed, according to the data access pattern discussed in Section 3.2.2, and the edge-set manager informs the prefetch component to load those edge-sets, as long as the buffer is not full. In the meanwhile, the *evictor* dumps the edge-set that are least recently used. The *edge-set modifier* updates edges and/or its property. Note that the scheduler is aware of the spatial/temporal data locality. If an edge-set is already loaded, it won't be loaded again.

3.2.6 Graph Algorithm and Interface

By hiding the low level graph data organization and providing a generic vertex-centric programming interface, our system easily adopts the gather-apply(-scatter) model from other popular graph frameworks but achieves better performance. The scatter is not necessary since our system targets out-of-core graph computing in shared memory, in which it is automatically handled by cache coherence. The computational model with the interface are described in Algorithm 1. After the graph data is loaded, the compute engine starts to stream the edges into the edge-set buffer and performs the *gather* function on each edge if the source/destination vertex is scheduled. For algorithms like PageRank, all vertices are active

for each iteration. A frontier is maintained for algorithms like breadth-first-search (BFS). In either case, multiple threads may process the same vertex to improve data parallelism. Each thread keeps a local sum copy of the destination vertices within an edge block, and updates vertex value to a shared sum after all the edge blocks within that vertex range are processed. In our experiments, we implemented four common graph algorithms.

Algorithm 1 Edge-Set Programming Interface

```
1: procedure RUN(edge-set buffer)
2:     for edge-set in buffer do
3:         for edge e in edge-set do
4:             if e.dst is scheduled then
5:                 Gather(e)
6:     for vertex in edge-set and is scheduled do
7:         Apply()
```

PageRank (PR): is a well-known algorithm that calculates the importance of websites in a website graph. The pagerank value of each vertex is updated every iteration after gathering all the neighbors' pagerank values. The PSW approach is very efficient for PageRank, since all vertices are active during the computation. In our experiments, we run 10 iterations for performance comparison. An illustration of our implementation is shown in Algorithm 2, where the *sum* value for each vertex initialized to zero.

Algorithm 2 PageRank

```
   procedure GATHER(e)
2:     sum_{e.dst} += e.src/e.src.outdegree
   procedure APPLY( )
4:     e.dst = 0.15 + 0.85 * sum_{e.dst}
```

Breadth First Search (BFS): is a fundamental graph traversal algorithm. It traverses all the vertices in a graph that are reachable from the source vertex. Initially all vertices are set as not visited and the traversal starts at the source. The level of a vertex is stored as its vertex value. We implement BFS in a PSW fashion using pull model, where the level of each vertex is updated with the smallest neighbor level value until convergence. If scheduler is enabled, the neighbors of the source will be put into a frontier and only the unvisited neighbors of the vertices in the frontier will be updated for the next iteration.

Single Source Shortest Path (SSSP): is a graph analytic application that computes the shortest path of each vertex from a source vertex in a graph. In our experiments, we implement Bellman-Ford algorithm. Initially, each vertex value is infinity for all nodes except for the source. The algorithm proceeds by iteratively updating distance estimation starting from the source and maintaining a worklist of nodes whose distances have been changed and thus may cause other distances to be updated. The gather and apply function are provided in Algorithm 3.

Algorithm 3 SSSP

procedure GATHER(e)
    $sum_{e.dst} = min(e.dst, e.src + e.weight)$
procedure APPLY( )
    $e.dst = sum_{e.dst}$

Weakly Connected Components (WCC): identifies the weakly connected components in a graph, where a set of vertices that are reachable from each other. The algorithm assigns a unique label for each vertex and collapses the labels of connected vertices into the smallest one by propagating labels through edges. We also follow the PSW fashion, and each vertex value is updated with the smallest neighboring vertex value until convergence.

3.3 Evaluation

In this section, we conduct experimental evaluations to validate the performance of our single machine out-of-core graph processing system using four graph algorithms. We present a thorough analysis of different optimization stacks by discussing the runtime breakdown, the impact of edge-set prefetch and consolidation, and the architectural characteristics of two computational models. Overall, our system shows improved execution time, better scalability and I/O performance relative to the state-of-the-art. We use GraphChi and X-Stream as our baselines.

3.3.1 Experimental Setup

We evaluate our system on two hardware platforms: a desktop with a quad-core 3.4 GHz Intel i5 processor, 16 GB memory and 256 GB SSD, and a 12-core server with 2.4 GHz Intel Xeon processor, 32 GB memory and 18 TB HDD network storage. The details of machine configuration are shown in Table 3.1. In our experiments, we set an 8 GB memory budget for SDD and 16 GB memory budget for HDD when running the graph applications, leaving sufficient space for OS and other running applications to avoid any system memory pressures.

Both synthetic and real-world datasets are tested in the edge list format (see Table 3.2). We use code from SNAP [57] to generate Kronecker graphs. Three Kronecker graphs [56] with increasing graph sizes are generated with the initiator matrix $[0.9, 0.6; 0.6, 0.1]$ (matrix sum is 2.2) and iteration numbers of 24, 25 and 26 respectively. For example, with iteration number of 24, it generates a Stochastic Kronecker graph on 16,777,216 (2^{24}) nodes with (2.2^{24}) edges. SNAP uses cumulative probability converted from the input matrix. The probability of an edge falling into the cumulative partitions is the cumulative probability

Table 3.1: Hardware configuration

Machine 1		
Processor	Type	Intel(R) Core(TM) i5-4670K
	Frequency	3.4 GHz
	Core#	1 socket x 4 cores x 1 thread
	Cache	32 KB L1, 256 KB L2, 6 MB L3
Host System	Memory	16 GB
	Disk	256 SSD
	OS	Debian GNU/Linux 8.7
Machine 2		
Processor	Type	Intel(R) Xeon(R) CPU E5-2440
	Frequency	2.4 GHz
	Core#	1 socket x 6 cores x 2 thread
	Cache	32 KB L1, 256 KB L2, 15 MB L3
Host System	Memory	32 GB
	Disk	18T HDD Network Storage
	OS	Red Hat 7.4 (Maipo)

over the matrix sum. We use Live-Journal [57], Twitter [17] and Yahoo [2] graphs to evaluate our system for real-world datasets. Live-Journal and Twitter are social graphs, showing the following relationship between users. Yahoo is a web graph that consist of hyperlink relationships between web pages, with larger diameters than social graphs. For small graphs that can fit into memory budget, we still shard and load the graph in batches from disk. We implement four typical graph algorithms covering graph traversals and graph analytics. One of the workloads is to run the PSW-based graph traversal 10 times, which is equivalent to performing PageRank on the graph for 10 iterations [55]. We implement the above workloads and compare our system with GraphChi and X-Stream. Note that the results for X-Stream

on HDD are for 8 cores, since it requires the number of processors to be powers of 2. The results for Yahoo are only available for HDD because the SDD can't hold the entire graph.

Table 3.2: Datasets description

Experimental Datasets	Vertices	Edges
Kronecker (24)	16.7M	165.2M
Kronecker (25)	33.5M	363.5M
Kronecker (26)	67.1M	799.8M
Live-Journal	4.8M	68.9M
Twitter	52.6M	1.4B
Yahoo	1.4B	6.6B

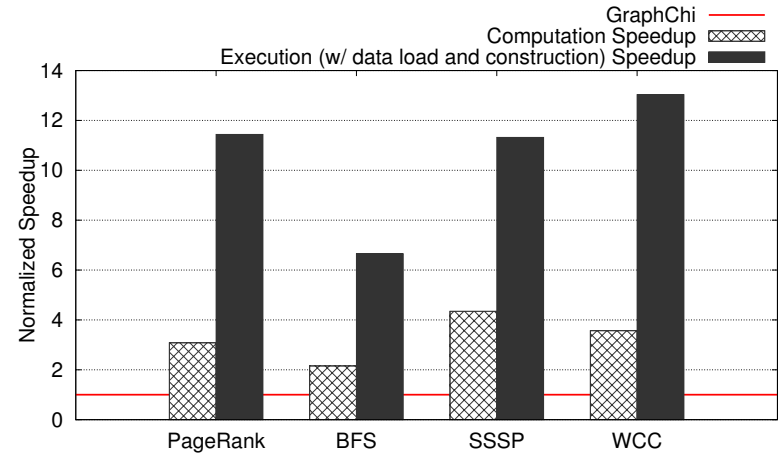
3.3.2 Performance Analysis

Highly promising performance improvements against the baselines are observed in our experiments. Figure 3.4 shows the execution time of our system and GraphChi on both synthetic and real-world datasets. There is approximately $2.2\times \sim 13.0\times$ speedup on Kronecker (25) and Twitter graphs across all four workloads. Such speedups are achieved due to 1) our system eliminates the costly vertex-centric graph construction in GraphChi which requires repeated memory allocation and release; 2) the vertex-centric model causes random access to the memory sub-system 3) our system streams the edge-sets which leads to much better memory behaviors; 4) our system exploits data parallelism in edge-sets by processing multiple vertices simultaneously. Compared to synthetic datasets, real-world datasets have more diversities, in terms of graph sparsity, connectivity and topology, and result in performance gains related to specific graphs. For Twitter graph in particular, the

speedup against GraphChi is slightly lower because the vertex-centric model works well for high-degree vertices after the graph construction, while the edge-centric model generally does not utilize such graph information. The drop of execution speedup against GraphChi for larger datasets is caused by the increased I/O overhead on uncompressed data structures in our system. As a result, for largest dataset Yahoo, our system runs 4.6× faster than GraphChi. This drawback can be mitigated using compression techniques to reduce the graph data load time in our system. In general, our system achieves 10% $\sim$ 30% performance improvement over GraphChi in the preprocessing phase, by eliminating global vertex sorting. To combine the preprocessing and the workload of 10 PSW-based traversals, we achieve 2.1× $\sim$ 3.6× overall speedup.

We also implement PSW-based BFS, SSSP and WCC in our system, and compare with GraphChi. For BFS and SSSP, the root vertices are randomly chosen. We run the experiments multiple times and use average time to mitigate the noise of testing environment. Although it is natural to traverse the graph under vertex-centric model, but it introduces high overhead on graph construction. As a result, our system achieves 2.2× $\sim$ 8.7× speedup. For the PSW-based BFS (including prepocessing), we achieve 1.8× overall speedup on average.

We implement Bellman-ford algorithm for SSSP problem, which computes the shortest paths from a single source vertex to all the other vertices in a weighted graph. The weight on each edge is randomly generated in input graphs. Starting from the source vertex, it traverses all the edges at most |V|-1 times, where |V| is the number of vertices in the graph. The traversal can also end if no distance values are updated since last iteration. The source vertices are randomly chosen, and the average execution time is measured and compared against the baselines. The improvement in execution time is approximately 3.0× $\sim$ 12.4× faster than GraphChi. The overall speedups can vary since the number of iterations



(a) Kronecker (25)

(b) Twitter

Figure 3.4: Performance improvement on execution time against GraphChi on synthetic (a) Kronecker (25) and real-world (b) Twitter datasets.

required to end may vary with different source vertices. When including preprocessing to the PSW-based SSSP, we achieved $1.7\times \sim 2.6\times$ overall speedup.

The data access pattern of WCC is similar to SSSP, where each vertex is assigned a label and updated with the minimum label value of its neighbors. Compared with SSSP, it doesn't require weight value on the edges, so the amount of graph data to be loaded is reduced. The

selective scheduling in GraphChi can reduce the required time for graph construction, but our system still gains $1.4\times \sim 4.1\times$ speedup on computation, and $3.1\times \sim 14.9\times$ speedup on execution.

Our system outperforms GraphChi in both the computation and execution (with graph data load and construction) phases. We illustrate the execution time breakdown for PageRank running different datasets in Figure 3.5. For applications like PageRank where all vertices are active, we observe that about 60% execution time in GraphChi is spent on graph construction for Kronecker graphs. The graph construction accounts for about 40% of the execution time for Twitter since the time spent on disk I/O increases with the input dataset size. Only about 20% time is on computation in GraphChi. Our system spends most of the time towards computation, by reducing the graph load and construction time dramatically. This is because we keep the consistent graph data organization for both graph storage and process. As a result, the system can start computation immediately after the graph data is loaded.

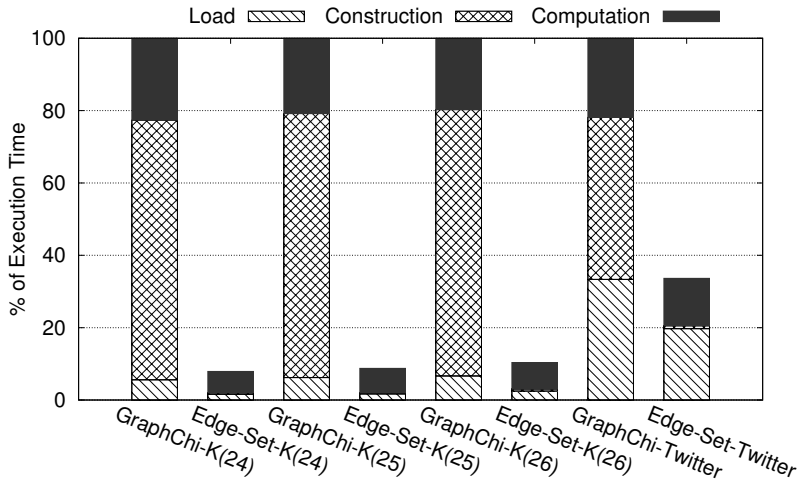


Figure 3.5: Execution time breakdown running PageRank normalized to the results of GraphChi on each dataset. K: Kronecker graph.

We also compare our system with X-Stream, which is an edge-centric graph processing system. X-Stream can stream completely unordered edge lists, and therefore has very little preprocessing overhead. However, it becomes inefficient for running graph analytics repetitively, or concurrently on the same input graph. It also exhibits degraded performance for large graphs. Both GraphChi and our system require preprocessing to generate shards or edge-sets, which is a one-time task given a static graph, but it turns out to be more efficient for above-mentioned situations. To run X-Stream, it requires the exact number of vertices and edges, and privilege to allocate memory. For more details, we present all the execution times for different storage types in Table 3.3 and Table 3.4. For the largest dataset Yahoo, we only test it on HDD since our SDD doesn't have enough space to hold the entire input graph file. Our system achieves approximately $2\times \sim 7\times$ speedup over GraphChi. X-Stream runs much slower, and for some applications it fails to finish the execution within 48 hours.

3.3.3 Impact of Prefetch and Consolidation

We conduct preliminary experiments to evaluate the impact of edge-set prefetch (Section 3.2.5), consolidation (Section 3.2.3), and the edge-set organization (Section 3.2.4). We notice that a larger buffer size usually results in higher performance improvement. The prefetch contributes up to 6% of the overall execution time improvement in our observation. We evaluate the edge-set consolidation with various upper bounds on the physical edge-set size using the three Kronecker graphs. The results are shown in Table 3.5. Figure 3.6 presents the comparison of the execution time and the preprocessing time on Kronecker (26) between two representative graph formats: COO and CSR. COO shows improved performance for the PSW-based workloads since PSW visits all (active) edges (see Figure 3.6). Note that CSR shows advantages for BFS-like traversals since it is essentially an indexed COO (with

Table 3.3: Execution time (in second) with 4-core w/ SDD (8 GB memory budget) for PageRank (10 iterations), BFS, SSSP and WCC. 'n/a' indicates the result is not available for corresponding system, or it failed to finish execution in 48 hours.

	Execution Time (SSD)			
	PageRank	**BFS**	**SSSP**	**WCC**
Kronecker (24)				
GraphChi	149.15	54.86	82.52	91.00
X-Stream	11.92	4.55	6.98	9.04
Edge-Set	11.77	6.29	6.61	6.08
Kronecker (25)				
GraphChi	335.95	123.39	185.57	208.40
X-Stream	257.08	76.73	84.16	117.29
Edge-Set	29.37	18.54	16.40	15.99
Kronecker (26)				
GraphChi	738.47	287.84	327.93	414.29
X-Stream	543.60	160.22	178.95	255.87
Edge-Set	76.58	50.51	78.34	43.47
Live-Journal				
GraphChi	36.31	14.52	29.11	29.94
X-Stream	4.66	2.01	3.51	3.36
Edge-Set	3.77	3.45	3.36	3.09
Twitter				
GraphChi	2553.09	1588.94	2880.97	2297.04
X-Stream	2645.60	1359.01	2066.85	2261.39
Edge-Set	858.81	705.10	959.26	739.14
Yahoo				
GraphChi	n/a	n/a	n/a	n/a
X-Stream	n/a	n/a	n/a	n/a
Edge-Set	n/a	n/a	n/a	n/a

compression) that helps in locating a specific vertex, but it is very inefficient for accessing in-edges of a vertex. It may also require more memory than COO in a highly sparse graph, and degrade the I/O efficiency if no any other preprocessing or optimization is applied.

Table 3.4: Execution time (in second) with 12-core w/ HDD (16 GB memory budget) for PageRank (10 iterations), BFS, SSSP and WCC. 'n/a' indicates the result is not available for corresponding system, or it failed to finish execution in 48 hours.

	Execution Time (HDD)			
	PageRank	**BFS**	**SSSP**	**WCC**
Kronecker (24)				
GraphChi	117.98	41.63	57.21	59.85
X-Stream	32.50	19.72	26.54	25.72
Edge-Se	16.53	9.37	9.51	9.29
Kronecker (25)				
GraphChi	290.83	94.73	134.33	169.20
X-Stream	102.73	128.88	145.82	621.41
Edge-Set	38.99	21.41	21.95	21.34
Kronecker (26)				
GraphChi	689.67	224.91	279.20	355.15
X-Stream	5281.28	649.35	820.98	1349.66
Edge-Set	94.82	51.43	72.07	50.98
Live-Journal				
GraphChi	33.94	13.58	22.69	26.67
X-Stream	31.55	21.71	23.10	25.69
Edge-Set	5.32	4.91	4.64	4.43
Twitter				
GraphChi	6105.19	2181.58	3797.10	2673.63
X-Stream	9240.33	2425.74	9223.33	10262.40
Edge-Set	1561.15	1178.82	2290.37	1148.17
Yahoo				
GraphChi	23177.13	1832.82	6547.83	26501.58
X-Stream	n/a	n/a	123731.25	169788.54
Edge-Set	3827.76	991.17	1557.08	3793.74

Load balancing issues need to be addressed as well when using CSR with vertex-centric parallelism. In this particular case, for PSW-based workloads, the COO format obtains 8.4× speedup over CSR, and 2.9× if preprocessing is considered.

Table 3.5: Impact of edge-set consolidation

Datasets	Performance Improvement
Kronecker (24)	3.2% $\sim$ 9.2 %
Kronecker (25)	3.8% $\sim$ 9.3 %
Kronecker (26)	2.8% $\sim$ 8.6 %

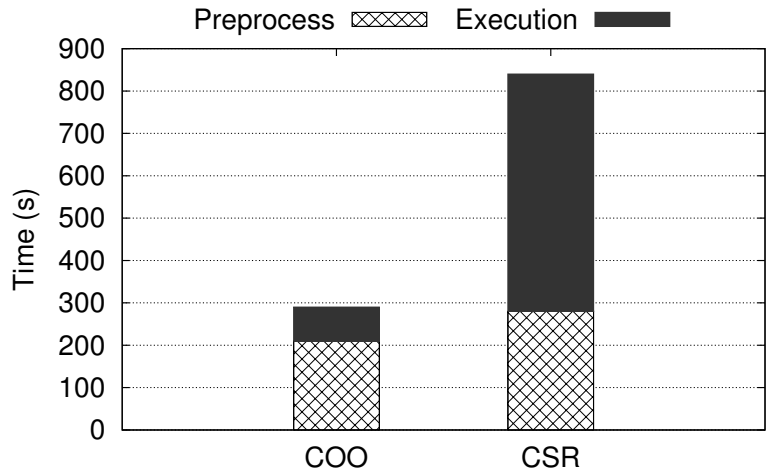


Figure 3.6: Impact on edge-set multi-modal organization on performance running PSW-based workloads (COO vs. CSR).

3.3.4 Scalability and I/O Performance

In Figure 3.7, we illustrate the scalability of our system. The execution time is normalized to 2-CPU GraphChi for each algorithm, and includes both load/construction and computation time. GraphChi's performance is limited by the I/O bandwidth and graph construction. Increasing the number of cores can improve the graph load and construction, but the gains from parallelism are small. Our system utilizes data parallelism by processing multiple vertices

simultaneously in edge-sets. As a result, it shows better scalability on the computation time with increasing number of cores. As our system is optimized for multicores, we start the experiment with two cores, and ramp up to four cores for fair comparison with GraphChi which targets a four-core Mac PC. We notice that for application like BFS, GraphChi shows similar or better computation time with small size graphs since it is more straightforward for such an application after constructing the graph into vertex-centric computational model. However our system still presents better overall performance by eliminating the graph construction and utilizing data parallelism in edge-sets for larger graphs. Moreover, our 2-CPU system achieved 2.0× $\sim$ 3.3× execution speedup over the 4-CPU GraphChi.

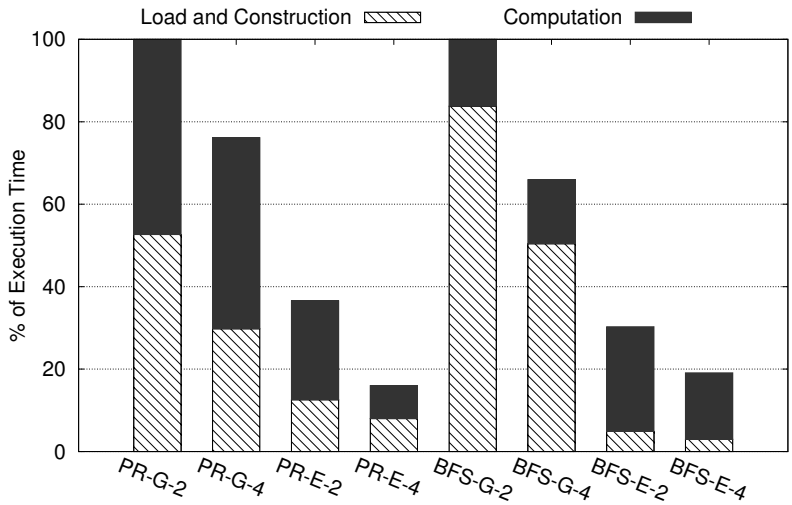


Figure 3.7: Relative runtime when varying the number of threads used by GraphChi and Edge-Set. Experiments were done on a Linux machine with four cores. The results were normalized to each algorithm result of GraphChi 2 cpus. PR: PageRank, G: GraphChi, E: Edge-Set.

Our system exhibits higher I/O efficiency than the baseline. Figure 3.8 shows the disk read bandwidth for Twitter graph over the time of Edge-Set and GraphChi. Edge-Set shows

up to 2× aggregate improvements and more stability in bandwidth utilization. GraphChi has poor I/O efficiency due to the long latency of graph construction.

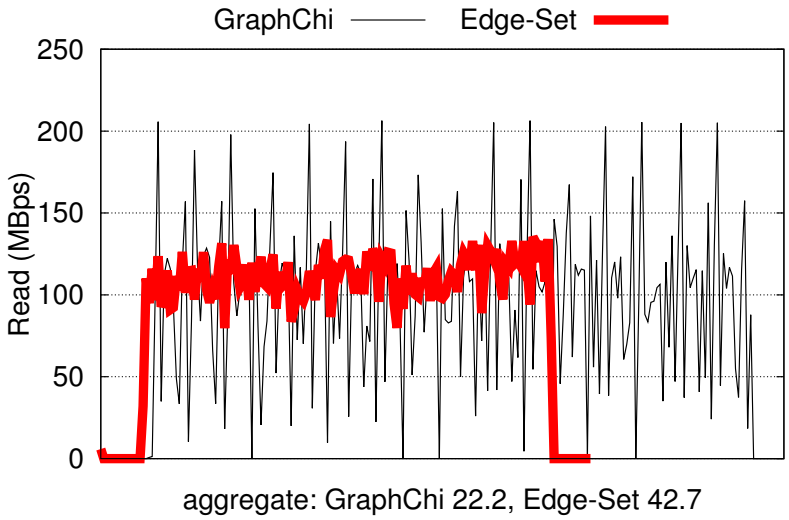


Figure 3.8: Disk read bandwidth over total run time by GraphChi and Edge-Set running Twitter. Edge-Set showed up to 2x aggregate bandwidth and more constant I/O usage.

3.4 CPU Characterization on Vertex-centric vs Edge-centric

In this section, we examine the architectural characteristics of GraphChi and our system. The impacts of different computational models (vertex-centric v.s. edge-centric) on architectural events like cache and TLB misses are discussed. In our experiments, we schedule and pin threads to specific hardware cores to avoid any OS uncertainty.

3.4.1 Workload Characterization

First, we introduce the profiling method we use for architectural events, and the architectural metrics we examine. Then we give a thorough analysis of how the two computational models impact these metrics.

Profiling method: we designe our own profiling tool to read the hardware performance counters and collect detailed hardware statistics. We use the perf_event interface of the Linux kernel to access hardware performance counters. The tool is embedded into the graph engines to capture code sections of interest, specifically the parallel execution regions. To avoid any inaccuracies caused by sampling error or multiplexing of performance counters, we run the experiments multiple times and for each run we only read one counter. In total, around 30 hardware performance counter statistics are collected.

Metrics for CPUs: graph applications are well known for random memory access, but the differences of architectural behaviors between computational models (vertex-centric v.s. edge-centric) have not been fully explored. This work is the first work that provides a comprehensive comparison between the two main graph computational models. In our experiments, we estimate and analyze the L1D, L2 and LLC cache misses-per-kilo-instructions (MPKI) to understand the memory sub-system behaviors. We also measured instruction-per-cycle (IPC), branch missprediction rates, hit rates at all cache levels, and DTLB miss rates.

Core performance: IPC is a key aspect of a processor's performances: the average number of instructions executed for each clock cycle. The per-core IPC when running different graph applications on GraphChi and Edge-Set are shown in Figure 3.9. Processors show lower IPC and higher variations across all the graph applications running on GraphChi. For example, per-core IPC when running SSSP on GraphChi is lower than 0.2, while the number could be around 0.5 when running SSSP. The per-core IPC when running graph

applications on Edge-Set almost doubles, and can reach as high as 0.9 for some datasets. The average per-core IPC for Edge-Set is around 0.7. Running Twitter graph has a lower IPC because of its large memory footprint, which causes a higher DTLB miss rate and more high-latency page walks.

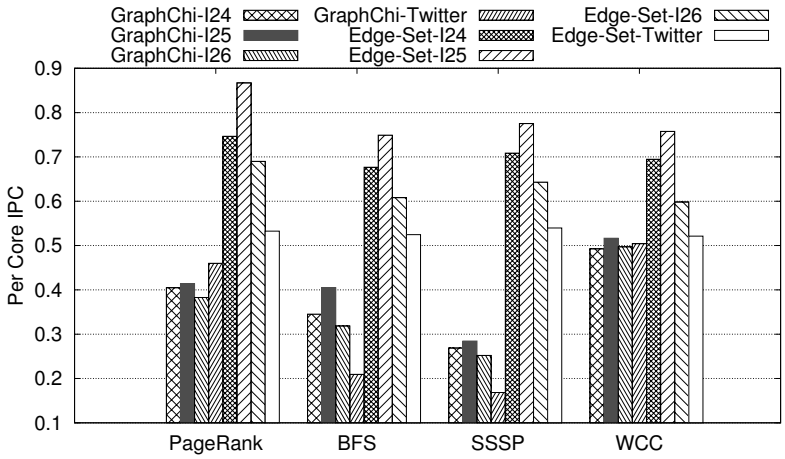


Figure 3.9: Per core IPC of GraphChi and Edge-Set.

Instruction fetch stalls and branch miss predictions are usually the main sources of front-end execution stalls. Generally, a large number of ICache misses or branch miss predictions can significantly affect performance, because modern processors are not optimized for hiding ICache misses, and have large branch miss prediction penalties due to deep pipelines. Other "big data" workloads are known to suffer from high ICache miss rates [49]. Graph applications, on the other hand, have lower ICache miss rates. Our experiments confirm that both computational models show very few ICache misses. As shown in Figure 3.10, the ICache MPKI of each workload is below 0.001, though small variances still exist. The different ICache performance values are caused by design differences of the underlying

frameworks. High ICache MPKI is reported from the incorporation of many other existing libraries and tools in open-source big data frameworks, which results in deep software stacks leading to complex code structures. However, in both GraphChi and Edge-Set, very few external libraries are included and a flat software hierarchy is incorporated. Because of its low code structure complexity, both frameworks show a much lower ICache MPKI.

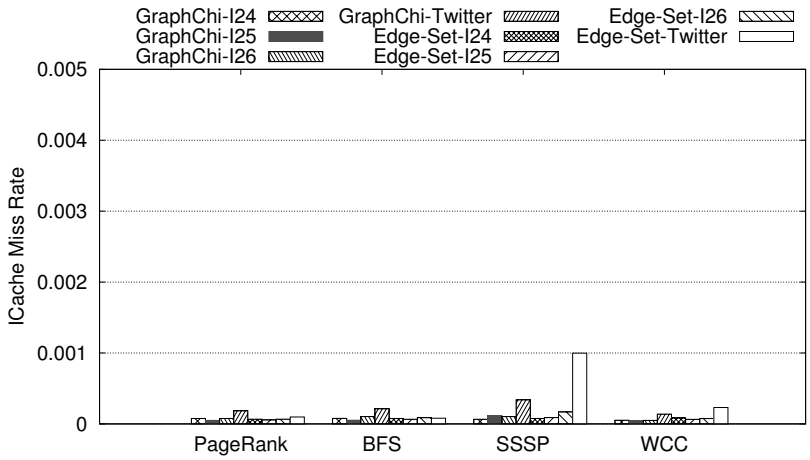


Figure 3.10: Per core ICache miss rate of GraphChi and Edge-Set.

The branch miss prediction rates for GraphChi is relatively low at 2.5% (Figure 3.11) in most workloads. This is mostly due to sequential access patterns in GraphChi. However, Edge-Set has even lower branch miss prediction rate, going as low as 0.1%. This is because the Edge-Set implementation tends to have fewer conditional expressions which leads to fewer hard-to-predict (non-loop) branches.

The DTLB miss rates are shown in Figure 3.12. GraphChi has higher DTLB miss rates ranging from 1% to 12% due to it vertex-centric model, which has lots of irregular accesses to the edge data. Considering the edge-set data organization and sequential access pattern,

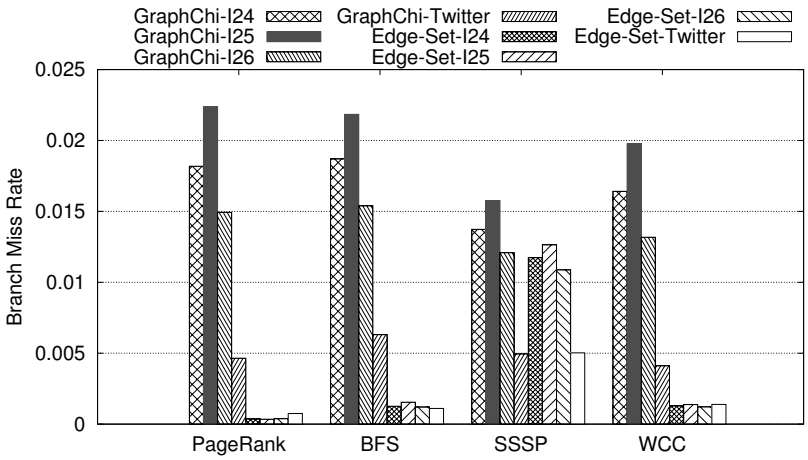


Figure 3.11: Per core branch miss rate of GraphChi and Edge-Set.

Edge-Set has much lower DTLB miss rates for all the datasets. The DTLB misses are mainly resulting from random accesses to the vertex properties. With increasing size of input datasets, the large memory footprint of graph applications causes increased number of memory page walks, which worsens the DTLB miss rate.

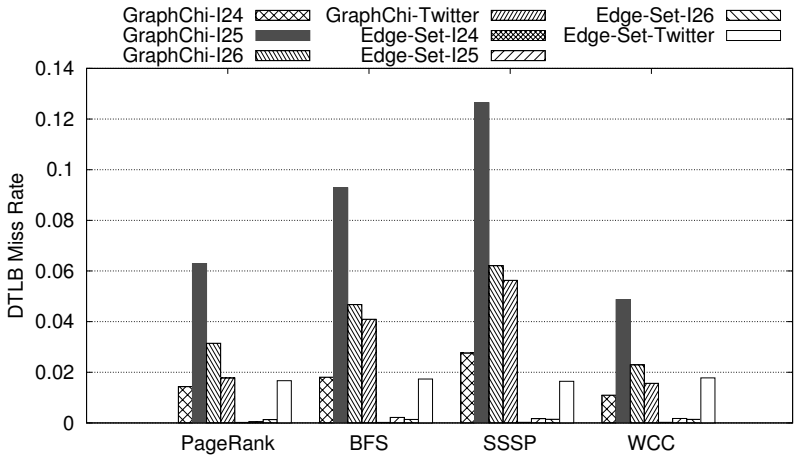


Figure 3.12: Per core DTLB miss rate.

Cache performance: As shown in previous sections, cache plays a crucial role in graph computing performance. In Figure 3.13 $\sim$ 3.18, the MPKI of different levels of caches are shown. GraphChi has higher cache MPKI for all three levels, which ranges from 30 $\sim$ 65 MPKI. Edge-Set shows lower cache MPKI. Both systems have reasonably high L1D hit rates. For specific applications like SSSP, Edge-Set has 16% higher L1D hit rate, while the differences among other applications are small. GraphChi suffers from high L2/LLC miss rates. It has less than 20% hit rate for L2 cache, and less than 6% LLC cache hit rate. Edge-Set has high L2 hit rate but low LLC hit rate. The L2 cache hit rates are above 50% except for Twitter graph. This is due to the limit of L2 cache size, and the high data parallelism in Edge-Set which causes high irregular accesses to the vertex properties. With larger cache size, LLC hit rates in Edge-Set is higher than GraphChi though.

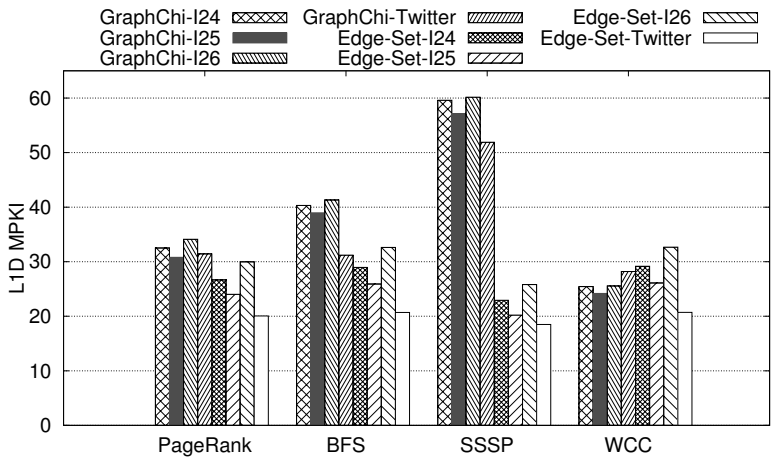


Figure 3.13: L1D MPKI of GraphChi and Edge-Set.

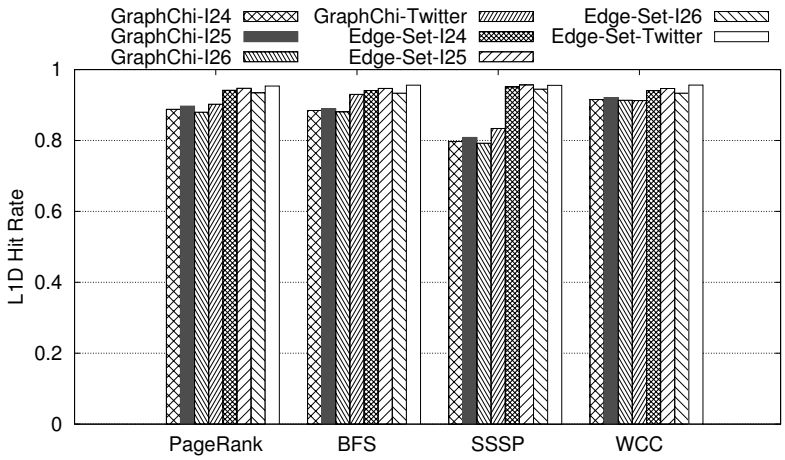


Figure 3.14: L1D hit rates of GraphChi and Edge-Set.

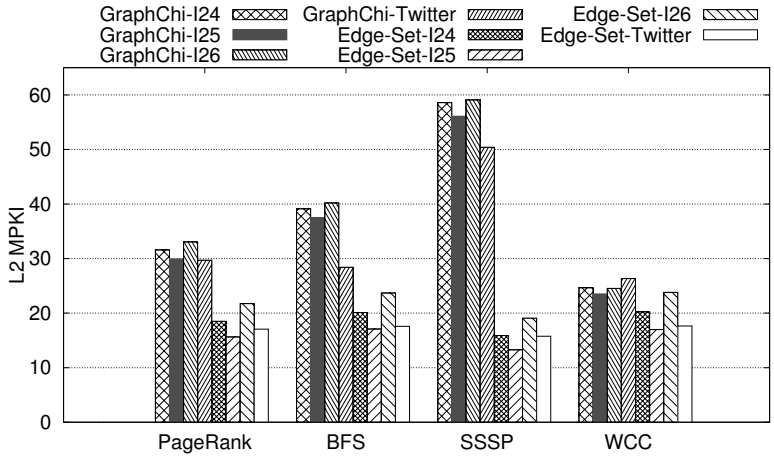


Figure 3.15: L2 MPKI of GraphChi and Edge-Set.

3.4.2 Observations

By measuring a list of architectural factors, we observe multiple different graph computing features on vertex-centric and edge-centric models. The key observations are summarized as follow:

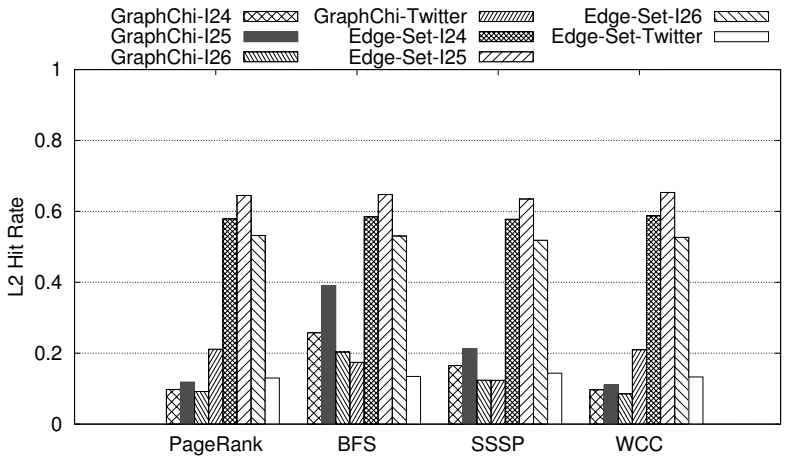


Figure 3.16: L2 hit rates of GraphChi and Edge-Set.

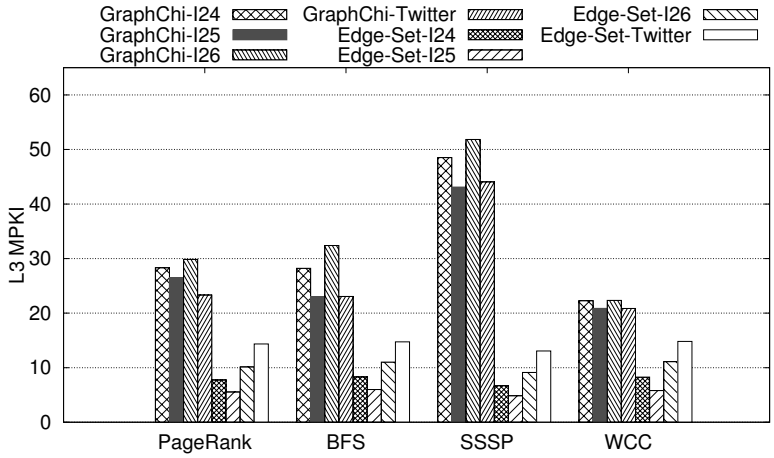


Figure 3.17: LLC MPKI of GraphChi and Edge-Set.

Backend stalls, including memory, are the major bottleneck for both computational models. The ICache miss rate is as low as conventional applications due to the flat code hierarchy of the underlying frameworks. Both graph engines are not cache-friendly as expected for graph computing applications. However Edge-Set outperforms GraphChi by 2x

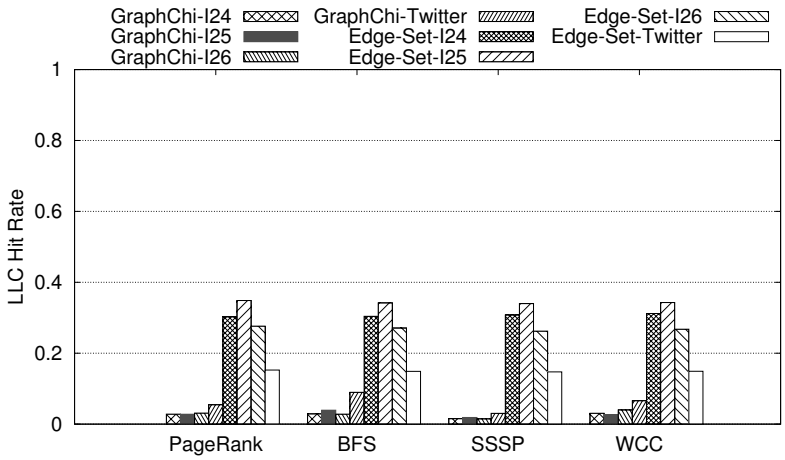


Figure 3.18: LLC hit rates of GraphChi and Edge-Set.

due to the spatial locality of edge-sets in L2/LLC caches. Both computational models have high L1D cache hit rates. DTLB misses are an issue for the vertex-centric model, and can be as high as 12% causing high performance penalty. The edge-centric model shows very low DTLB misses due to its the sequential access model. Input graph datasets have a significant impact on the memory sub-system and overall performance.

The major inefficiency of graph computing is from the memory sub-system. The low cache hit rates introduce challenges for architecture/system design. The differences between the two main computational models provide us a roadmap for framework implementation. Depending on the input graph properties and the applications, choosing the right graph representation and computational model can lead to better performance and higher utilization of architectural resources.

3.5 Summary

We present our efforts on developing an efficient large scale graph processing engine. Our experimental evaluations show very promising results, such as over 10x speedup on traversing a graph using parallel sliding window (PSW). As an industrial system, it provides compatible interfaces to utilize existing vertex programs. The performance improvement is due to several factors, including the edge-set graph representation with multi-modality, data prefetch, and consolidation. We show that our edge-centric computation model dramatically improves the performance over the vertex-centric model by removing graph construction and optimizing the data parallelism.

We also analyze the CPU and memory sub-systems' behaviors of vertex-centric and edge-centric models to show the inefficiency of modern processors for graph computing. Both models show poor L2 and LLC hit rate, while the vertex-centric model shows extremely low IPC and LLC hit rate due to its graph data organization and random memory accesses. The edge-centric model mitigates memory miss delays by generating sequential accesses, but the cache performance is still bad with large memory footprint due to the increased input datasets size.

Chapter 4: C-Graph: A Highly Efficient Concurrent Graph Reachability Query Framework

4.1 Introduction

Graph processing has been widely adopted in big data analytics and plays an increasingly important role in knowledge graph and machine learning applications [35,39,106]. Many real-world scenarios such as social networks, web graphs, wireless network and etc., are naturally represented as large scale graphs [29, 57]. Modeling applications as graphs provides an intuitive representation that allows exploration and extraction of valuable information from data. For example, in recommendation systems, information about neighbors is analyzed in order to predict the user's interests and improve click-through rate (CTR). High performance graph processing also benefits a wealth of important algorithms. For instance, mapping applications make extensive use of shortest path graph traversal algorithms for navigation. In order to effectively manage and process graphs, graph databases such as JanusGraph [6], Neo4j [93] and others have been developed. Graph processing frameworks are also commonly found as critical components in many big data computing platforms, such as Giraph in Hadoop, GraphX in Spark, Gelly in Flink, etc [3, 33, 94, 103].

One of the fundamental operations that a graph processing system must handle efficiently is the graph traversal. For example, the "reachability query" is essentially a graph traversal

to search for a possible path between two given vertices in a graph. Graph queries are often associated with constrains such as a mandatory set of vertices and/or edges to visit, or a maximum number of hops to reach a destination. In weighted graphs, such as those used in modeling software-defined-networks (SDNs), a path query must be subject to some distance constraint in order to meet quality-of-service latency requirements [104].

Many real-world applications rely on *k-hop* [21], a variant of the classic reachability query problem. In *k-hop* the distance from a given node often indicates the level of influence. For example, in wireless, sensor or social networks the signal/influence of a node degrades with distance. The potential candidate of interest is often found within a small number of hops. Real-world networks are generally tightly connected, making *k-hop* query very relevant. According to the "six degrees of separation" principle, which claims that a maximum of six steps are needed to connect any two people, most of the network will be visited within a small number of hops. As a results, *k-hop* reachability often exists as an intermediate "operator" between low-level database and high-level algorithms [64]. Many higher-level analyses can be described and implemented in terms of *k-hop* queries, such as triangle counting which is equivalent to finding vertices that are within 1 and 2-hop neighbors of the same vertex. Therefore, a graph processing system's ability to handle *k-hop* access patterns predicts its performance on higher-level analyses.

Compared to many big data systems, graph processing generally faces significant performance challenges. One such challenge for graph traversals is poor data locality due to irregular data access patterns in many graph problems. As a result, graph processing is typically bound by a platform's I/O latency, rather than its compute throughput [31, 96]. In distributed systems the overheads of communication beyond machine boundaries, such as network latency, exacerbate I/O bottlenecks faced by graph processing systems.

Another challenge for most existing graph processing frameworks is to efficiently handle concurrent queries. These systems are often optimized to either improve performance or reduce I/O overhead, but are not capable of responding to concurrent queries. In enterprise applications, a system usually has to gracefully handle multiple queries at the same time. Also, since multi-user setups are common, several users can send out query requests simultaneously. Graph databases are often designed with concurrency in mind, but they generally have poor performance in graph analysis, especially in terms of handling large scale graphs or high volumes of concurrent queries [96].

In this chapter, we propose an efficient distributed concurrent query framework called *C-Graph*, short for Concurrent Graph processing system, to process concurrent local graph traversal tasks such as those in the *k-hop* reachability query. We take a high level view of the graph processing system design. While improving the efficiency of each processing unit, we consider both disk I/O and network I/O as elements of the storage bandwidth. *C-Graph* is designed as a traditional edge-centric sharding-based graph processing system. The main contributions of our framework can be summarized as follows: (1) A simple range-based partition is adopted to reduce the overhead of complex partitioning scheme for large scale of graphs. (2) Multi-mode, edge-set-based graph data structures optimized for sparsity and cache locality are used in each partition to achieve the best performance for different access patterns. (3) The framework explores data locality between overlapped subgraphs, and utilize bitwise operations and shared global states for efficient graph traversals. (4) In order to solve the memory limitation of concurrent graph queries in a single instance, we utilize dynamic resource allocation during graph traversals. Instead of saving a value per each vertex, we only store values for the previous and current levels. (5) Synchronous/asynchronous update models are supported for different types of graph applications, such as graph traversals and

iterative computation (e.g. PageRank). (6) Our system targets the reduction of the average response times for concurrent graph queries on large-scale graphs with up to 100 billion edges in distributed environments.

The rest of this paper is organized as follows: we discuss the main features and programming interface of our distributed query system in section 4.2. In section 4.3 we evaluate our system performance and scalability. Lastly, we conclude in 4.4.

4.2 C-Graph: Concurrent Graph Query System

In this section, we introduce the main features of our graph processing framework.

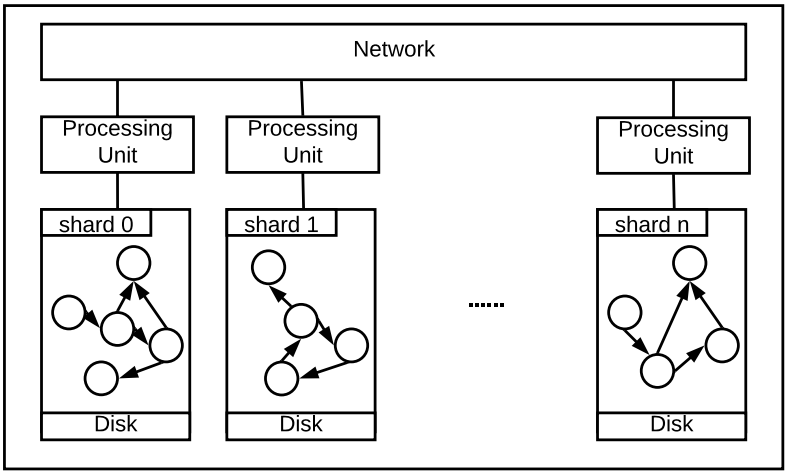


Figure 4.1: Overview of edge-centric sharding-based graph processing system design.

Overview. Figure 4.1 shows an overview of our framework running on a cluster of computing nodes connected by a high-speed network. Each node consists of a processing unit with a cached subgraph shard. The processing units are CPUs in our current framework, and can be extended to GPUs or any other graph processing accelerators. Each subgraph

shard contains a range of vertices called local vertices, which are a subset of all graph vertices. Boundary vertices with respect to a subgraph shard are vertices from other shards that have edges connecting to the local vertices of the subgraph. Each subgraph shard stores all the associated in/out edges as well as the property of the subgraph. The graph property includes vertex values, and edge weights (if the graph is weighted). Each processing unit computes on its own subgraph shard and updates the graph property iteratively. It is also responsible for sending the values of boundary vertices to other processing units. Structuring the graph processing system this way allows us to decouple computation from communication. We focus on improving the computing efficiency of each processing unit based on its available architecture and resources. Then, we treat all communications as an abstraction of the I/O hierarchy (i.e. memory, disk, and network latency). Note that a subgraph shard does not necessarily need to fit in memory; as a result, the I/O cost may also involve local disk I/O.

4.2.1 Range-based Graph Partitioning

Graph partitioning is an important step in optimizing the performance of a graph processing system where the input graphs cannot fit in a node's memory. Many system variables such as workload balance, I/O cost etc. are often considered when designing a graph partitioning strategy. There can be different optimal partition strategies depending on the graph structure and application behavior. Moreover, re-partitioning is often required when graphs change, which is costly for large-scale graphs. Our solution is to adopt a lightweight low-overhead partitioning strategy. Our framework deploys a simple range-based partition similar to GraphChi [55], GridGraph [113], Gemini [112] and etc. Vertices are assigned to different partitions based on vertex ID, which is re-indexed during graph ingestion. Each partition

contains a continuous range of vertices with all associated in/out edges and subgraph properties. To balance the workload, we optimize each partition to contain a similar number of edges. In a p-node system, a given graph $G = (V, E)$ will be partitioned into p continuous subgraphs $G_i = (V_i, E_i)$, where $i = 0, 1, ..., p-1$. In each G_i, V_i are local vertices and E_i is a set of edges $\{s, t, w\}$, where either source s or destination t belongs to V_i. The rest of the vertices in other partitions are boundary vertices. Assigning all out-going edges of a vertex to the same partition is a way of improving the efficiency of local graph traversals. We also store incoming edges when running graph algorithms such as PageRank.

4.2.2 Multi-modal Edge-set based Graph Representations

We adopt multi-modal graph representations in our framework to accommodate different access patterns and achieve best data locality for different graph applications. Compressed sparse row (CSR) is a common storage format to store the graph. It provides an efficient way to access the out-going edges of a vertex, but it is inefficient when accessing the incoming edges of a vertex. To address this inefficiency, we choose to store the incoming edges in compressed sparse column (CSC) format, and out-going edges in compressed sparse row (CSR) format.

To improve cache locality, we explore iterative graph computing with an edge-set based graph representation [23, 55, 113]. Similar to the range-based partitioning, each subgraph partition is further converted into a set of edge-sets. Each edge-set contains vertices within a certain range by vertex ID. As shown in Figure 4.2a, an input graph represented in adjacency matrix format is divided into two partitions, with each partition converted into eight edge-sets. To traverse a graph through out-edges, we scan the blocked adjacency matrix left to right.

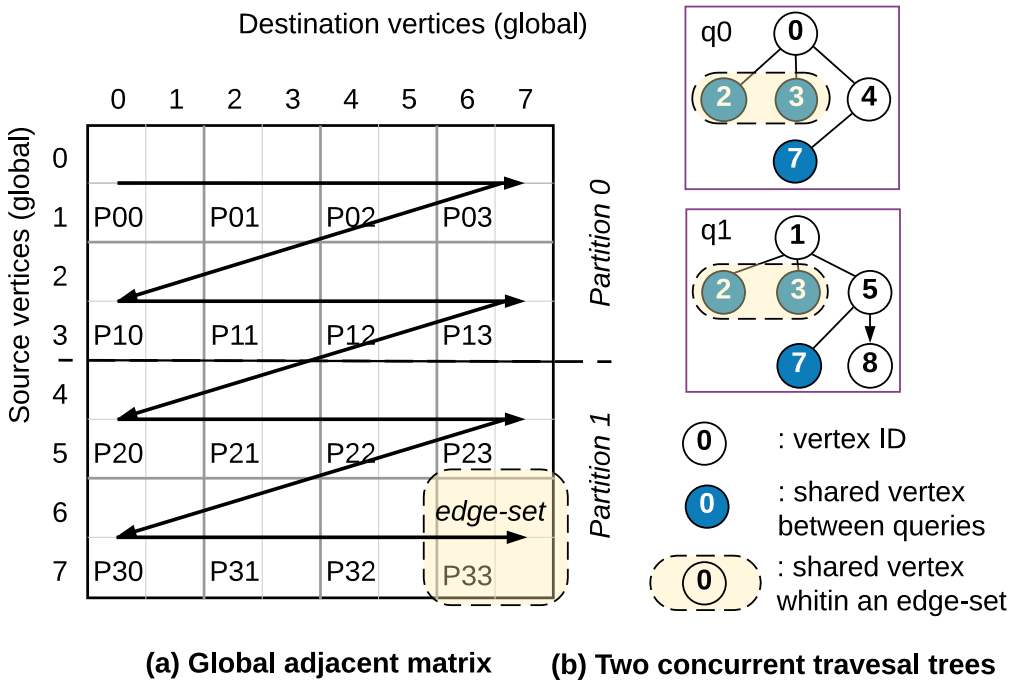


Figure 4.2: An example of edge-set based graph representation. (a) An input graph is divide into two subgraph partitions, and each partition is converted into 8 edge-sets. To traverse a graph through out-going edges is equivalent to scan the edge-sets in left-right pattern. (b) Graph traversal trees for two concurrent queries. First three levels are presented in the example.

Generating edge-sets is straightforward. We first obtain vertex degrees after partitioning the input graph across machines, and then we divide the vertices of each subgraph into a set of ranges by evenly distributing the degrees. Next, we scan the edge list again and allocate each edge to an edge-set according to the ranges into which source and destination vertices fall. Finally, within each edge-set, we generate the CSR/CSC format using local vertex IDs calculated from global vertex ID and partition offset. The preprocessing reduces the complexity of global sorting, and is conducted in a divide-and-conquer manner.

The granularity of an edge-set is chosen such that the vertex values and associated edges fit into the last level cache (LLC). However, the sparse nature of real large-scale graphs can result in some tiny edge-sets that consist of only a few edges each, if not empty. Loading or persisting many such small edge-sets is inefficient due to the I/O latency. Therefore, it makes sense to consolidate small edge-sets that are likely to be processed together, so that data locality is potentially increased. Consolidation can occur between adjacent edge-sets both horizontally and vertically. The horizontal consolidation improves data locality especially when we visit the out-going vertex edges. Vertical consolidation benefits the information gathering from the vertex's parents.

Concurrent graph traversals can benefit from edge-set representation from two dimensions of locality maintained inside an edge-set: 1) shared neighbor vertices of *frontiers* within an edge-set and 2) shared vertices among queries. A simple example is shown in Figure 4.2b, where two concurrent queries are presented, each by a graph traversal tree of three levels. Visiting neighbors of vertex 2 and 3 takes just one pass on edge-sets $P_{1i}, i = 0, 1, 2, 3$, and since these two vertices are shared among both queries, query performance can be improved by making only one traversal on these two vertices. The compute engine performs user-defined functions on edges within each edge-set in parallel. Edge-set graph representation also improves cache locality for iterative graph computations like PageRank from two aspects: 1) sequential accesses to edges within a local graph and 2) write locality preserved by storing the edges in CSC format. Updating the vertex value array in ascending order also leads to better cache locality while enumerating the edges in a edge-set.

4.2.3 Query Processing

Efficient implementation of a distributed graph engine requires balancing computation, communication and storage. Our framework supports both the vertex-centric and partition-centric models. We specifically optimized the partition-centric model to handle graph traversal-based algorithms such as *k-hop*, BFS. The performance of such models depends strongly on the quality of the graph partitions. Figure 4.3 shows the graph traversal iterations in the partition-centric model (which generally requires fewer supersteps to converge compared to the vertex-centric model). In the partition-based model, vertices can be classified into local vertices and boundary vertices. The values for local vertices are stored in the local partition, while boundary vertex values are stored in the remote partitions. Local vertices communicate with boundary vertices through messages. A vertex can send a message to any other vertices in the graph using the destination vertex's unique ID.

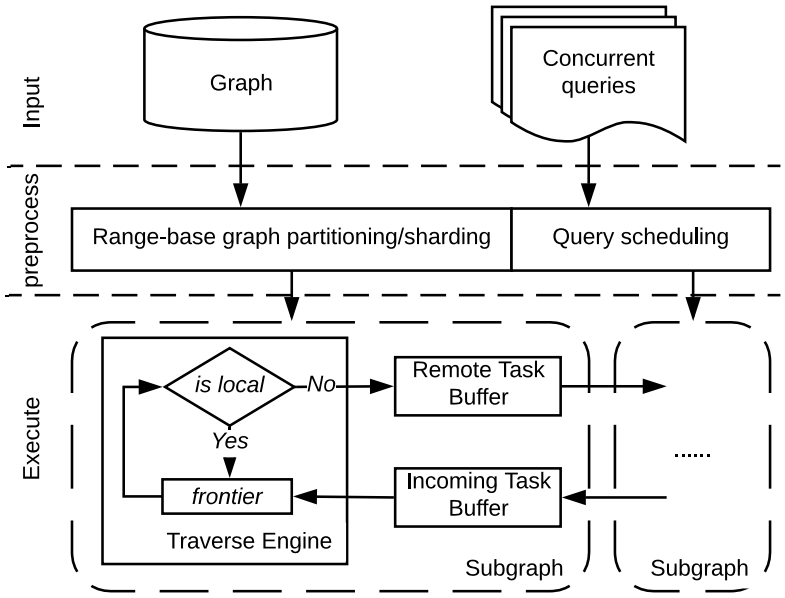


Figure 4.3: Graph query workflow.

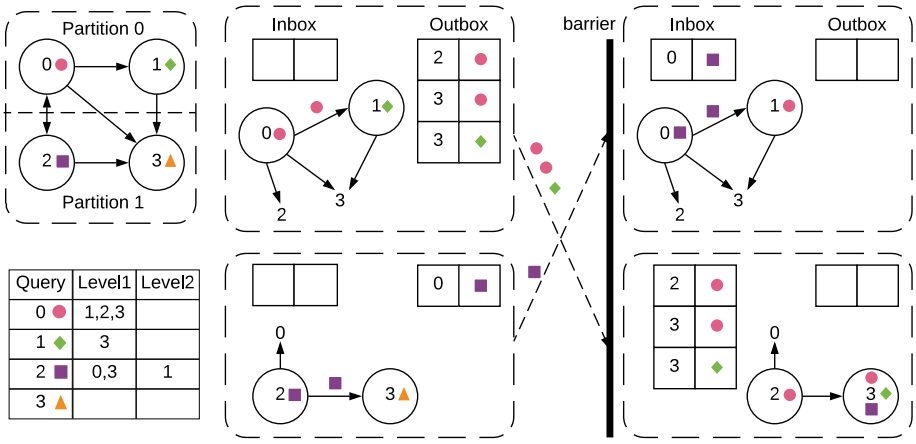


Figure 4.4: A simple two-partition graph example with four concurrent graph traversals starting from all four vertices. Different queries are distinguished using different symbols. Each partition has an inbox buffer for incoming tasks and an outbox buffer for outgoing tasks. Each task is associated with the destination vertex's unique ID. The visited vertices are synchronized after each iteration and won't be visited.

To illustrate the partition-centric model, we consider two operations: local read and remote write, both of which incur cross-partition communications. Local read is performed when reading the value of a boundary vertex. For example, the PageRank value of a local vertex is calculated from all the neighboring vertices, some of which are boundary vertices. In this case, a locally updated vertex value has to be synchronized across all partitions after each iteration. In other cases, a partition may need to update the value of a boundary vertex of the partition. For example, in subgraph traversals involving traversing depth, when a boundary vertex is visited, its depth needs to be updated remotely. The boundary vertex ID with its value along a traverse operator will be sent to the partition to which it belongs. In that partition, the vertex value will be asynchronously updated and the traversal on that vertex will be performed based on the new depth. In a sense, all vertices are updated locally to achieve the maximum performance through efficient local computation, and all changes

```
void abstract compute();
void sendTo(V destination, M msg);
void voteTohalt();
bool ifHasVertex(V vid);
bool isLocalVertex(V vid);
bool isBoundaryVertex(V vid);
Collection getLocalVertices();
Collection getBoundaryVertices();
Collection getAllVertices();
void barrier();
```

Listing 4.1: Partition-centric Model [90]

of the graph property are exchanged proactively across partitions using high speed network connections. A simple example of subgraph traversal is shown in Figure 4.4.

Concurrent queries can be executed individually in request order, or processed in batches to enable subgraph sharing among queries. To mitigate the memory pressure in concurrent graph queries, we utilize dynamic resource allocation during graph traversals. We only need to keep values of vertices in previous and current levels, instead of saving value per vertex during the entire query.

4.2.4 Programming Abstraction

We next introduce the programming API deployed in our framework. We provide an interface for the partition-centric model, which was first introduced by Giraph++ [90] and has been quickly adopted and further optimized in recent works [83, 101]. Listing 4.1 shows the interface of the basic methods call in the partition-centric model.

We provide two functions to accommodate different categories of graph applications: a) graph traversal on graph structure and b) iterative computation on graph property. Graph traversal involves data-intensive accesses and limited numeric operations. The irregular data access pattern leads to poor spatial locality and introduces significant pressure on the memory

```
def Traverse(task queue: Q, hops: k) {
  while any s in Q {
    if (s.hops < k) {
      if (isLocalVertex(s)) {
        for (t in s.neighbors and !visited(t)) {
          t.hops = s.hops + 1
          if (isLocalVertex(t)) Q.push(t)
          else sendTo(t, t.hops)
          visited(t) = true
        }
      }
    }
    Q.pop(s)
  }
}
```

Listing 4.2: k-hop Traversal: For each vertex in a local task queue, visits its neighbors and puts them into two queues based on: local vertices will be inserted into the local task queue, while boundary vertices will be sent to a remote task queue. All neighbors will be marked as visited and shared cross all processing units. The maximum depth of traversal is defined by hops k.

subsystem. Computation on graph property often involves more numeric computation which shows hybrid workload behaviors [70]. The graph traversal pattern is defined in the **Traverse** function, and the iterative computation is defined in **Update** function. An example of *k-hop* implementation is shown in Listing 4.2.

The **Update** function is an implementation of the Gather-Apply-Scatter (GAS) model by providing a vertex-programming interface. A PageRank example using the GAS interface is shown in Listing 4.3. The function looks no different than a normal GAS model graph processing framework. However, our implementation does not generate additional traffic in the gather phase since all edges of a vertex are local.

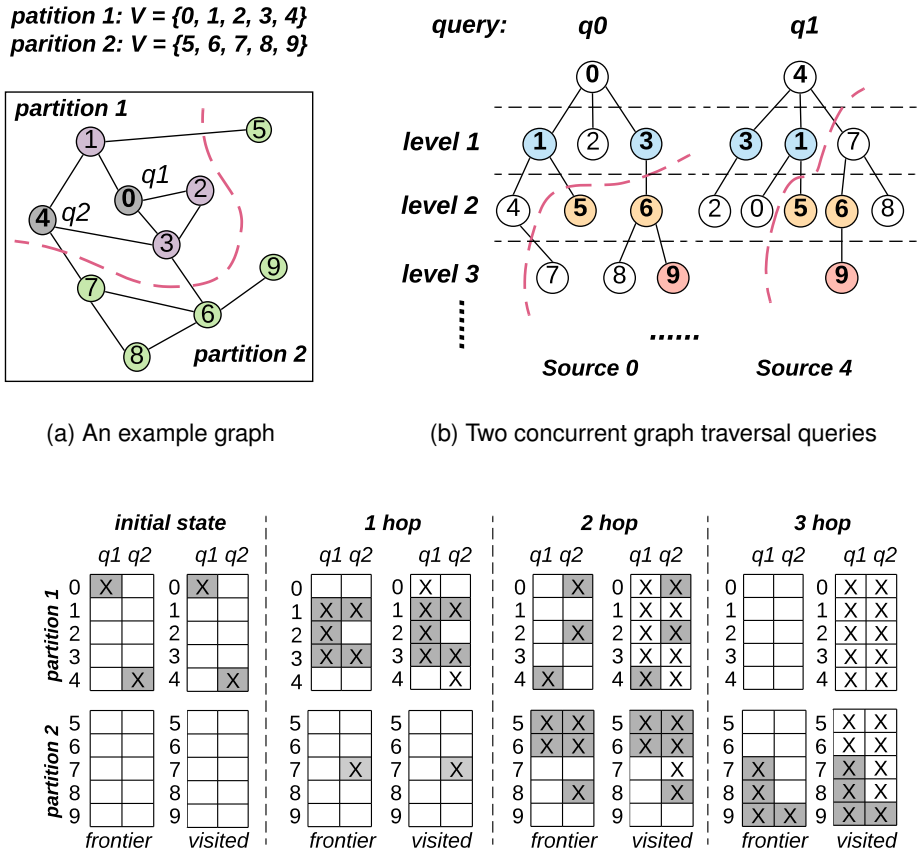
```
def Gather(v, sum) sum += v.val
def Apply(v, sum) v.val = 0.15 + 0.85 * sum
def Scatter(v) v.val / v.outdegree
```

Listing 4.3: PageRank: The gather phase collects inbound messages. The apply phase consumes the final message sum and updates the vertex data. The scatter phase calculates the message computation for each edge.

4.2.5 Concurrent Queries Optimization

We further optimize the concurrent queries by leveraging several state-of-art techniques. In practice, it is inefficient to use a set or queue data structure to store the *frontier* since the union and set operations are expensive with a large number of concurrent graph traversals. In addition, the dramatic difference in *frontier* size at different traversal levels introduces dynamic memory allocation overhead. It also requires a locking mechanism if the *frontier* is processed by multiple threads. Instead of maintaining a task queue or set, we implement the approach introduced in MS-BFS [89] to track concurrent graph traversal *frontier* and *visited* status, and extend it to distributed environments. For each query, we use 2 bits to indicate if a vertex exists in the current or next *frontier*, and 1 bit to track if it has been visited. A fixed number of concurrent queries are decided based on hardware parameters, for example, the length of the cache line. The *frontier*, *frontierNext* and *visited* are stored in arrays for each vertex to provide constant-time access.

An example graph is shown in Figure 4.5. A graph with 10 vertices is divided into two machines using range-based partitioning. Partition 0 contains vertices $V : \{0 \sim 4\}$, and partition 1 contains vertices $V : \{5 \sim 9\}$. Each partition maintains a *frontier* and *visited* bit array for each query. Figure 4.5b shows the traversal tree for each query. In the example, we show two concurrent queries starting from source vertices 0 and 4. Figure 4.5c shows the bit array representations for *frontier* and *visited* nodes. The *frontier* in the current hop is from *frontierNext* in the previous level. Each row represents a vertex, and each column represents a query. The main idea behind this is queries share same vertices in each iteration, and data locality is preserved if updating concurrent queries at same time.



(a) An example graph

(b) Two concurrent graph traversal queries

(c) Frontier and visited bit arrays at each hop

Figure 4.5: An example of bit operations for two concurrent queries.

4.3 Evaluation

To evaluate the efficiency of our system and its optimizations, we measure the system performance using both real-world and semi-synthetic graph datasets. We test our system with various types of graph algorithms, and reported experimental results on scalability with respect to input graph size, number of machines and number of queries. We compare the

performance of our system with open-source graph database Titan [4], and state-of-the-art graph processing engine Gemini [112].

4.3.1 Experimental Setup

Graph Algorithms. In our experimental evaluation, we use two graph algorithms to show the performance of our system running different types of graph applications.

K-Hop Query: is a fundamental algorithm for graph traversals. We use it to evaluate the performance of concurrent queries. Most of our experiments are based on the *3-hop* query, which traverses all vertices in a graph that are reachable within 3 hops from the given source vertex. For each query, we maintain a *frontier* queue and *visited* status for each vertex. Initially all vertices are set as not visited, and *frontier* contains the source vertex. The level of a visited vertex or its parent is recorded as vertex value. The unvisited neighbors of the vertices in the frontier will be added to the *frontier* for the next iteration. The details of the implementation are illustrated in Listing 4.2. The main factor we use to evaluate the performance of the query system is the response time for each query in a concurrent queries environment. We test from 10 to 350 concurrent queries, and report the query time for each query.

PageRank: is a well-known algorithm that calculates the importance of websites in a websites graph. In PageRank all vertices are active during the computation. The vertex page-rank value is updated after gathering all the neighbors' page-rank values. In our experiments, we ran 10 iterations for performance comparison. An illustration of our implementation using the GAS (Gather-Apply-Scatter) API is shown in Listing 4.3, with the sum value for each vertex initialized to zero. Although our system mainly the *k-hop* query, we use

PageRank to evaluate the iterative graph computation applications, which have different access patterns compared to graph traversals.

Software and Hardware Configuration. We conducted most of our experiments on a 9 server machines cluster, each has an Intel(R) Xeon(R) CPU E5-2600 v3, having a total of 44 cores at 2.6 GHz and 125 GB main memory. Our system and all algorithms were implemented in C++11, compiled with GCC 5.4.0, and executed on Ubuntu 16.4. We used Socket and MPI (Message Passing Interface) for network communication.

Datasets. In our evaluation, we experimented with both real-world and semi-synthetic datasets. We used two real world graphs: Orkut and Friendster from SNAP [57], and two semi-synthetic graphs: both are generated from Graph 500 generator with Friendster to test the system's ability to process graphs at different scales. Orkut and Friendster are on-line social networks where users form friendships with each other. Orkut has 3 million vertices and 117 million edges with a diameter of 9, while Friendster has 65.6 million and 1.8 billion edges with a diameter of 32. Both graphs form large connected components with all edges. Two semi-synthetic graphs are generated with Graph 500 generator and Friendster graph. Given a multiplying factor m, the Graph 500 generator produces a graph having m times vertices of Friendster, while keeping the edge/vertex ratio of the Friendster. The smaller semi-synthetic graph has 131.2 million vertices and 72.2 billion edges, and the larger semi-synthetic graph has 985 million vertices and 106.5 billion edges. The details of each graph are shown in Table 4.1.

We used the open-source graph database Titan [4], which supports concurrent graph traversals, as a baseline. Since Titan took hours to load a large graph, we used a small graph Orkut to compare the single machine performance running Orkut on Titan with our system. We used the internal APIs provided by Titan for both graph traversals and PageRank. We

Table 4.1: Datasets description

Experimental Datasets	Vertices	Edges
Orkut (OR-100M)	3,072,441	117,185,083
Friendster (FR-1B)	65,608,366	1,806,067,135
Friendster-Synthetic (FRS-72B)	131,216,732	72,224,268,540
Friendster-Synthetic (FRS-100B)	984,125,490	106,557,960,965

also experimented with the well-known open-source graph database Neo4j [76]. However, this system was even slower to load and traverse a large graph. Therefore, we did not include Neo4j in our comparison.

How does Response Time Impact User Experience? Before we dive into the experimental results, we first discuss an important quality metric of an online business like a website or a database: response time. There is a strong correlation between response time and business metrics since wait time heavily impacts user experience. To quantify the performance impact on a query, the following three thresholds have been defined [1, 81]:

- Users view response time as instantaneous (0.1-0.2 second): Users can get query results right away and feel that they directly manipulate data through the user interface.
- Users feel they are interacting with the information (1-5 seconds): They notice the delay, but feel that the system is working on the query. A good threshold is under 2 seconds.
- Users are still focused on the task (5-10 seconds): They keep their attention on the task. This threshold is around 10 seconds. Productivity suffers after a delay above this threshold.

According to the above thresholds, we would reasonably expect a distributed graph processing system to respond to a set of (say, 100–300) concurrent queries within very a few seconds (say 2 seconds).

4.3.2 System Performance

We compared the concurrent *3-hop* query and PageRank performance with the graph database Titan [4] on a single machine. We run 100 concurrent queries for both systems, with each query containing 10 source vertices. The source vertices are randomly chosen, with each system performing 1000 random subgraph traversals to avoid both graph structure and system biases. The average response time for a query is calculated from the 10 subgraph traversals of each query, and average response times for 100 queries are shown in Figure 4.6, sorted in ascending order.

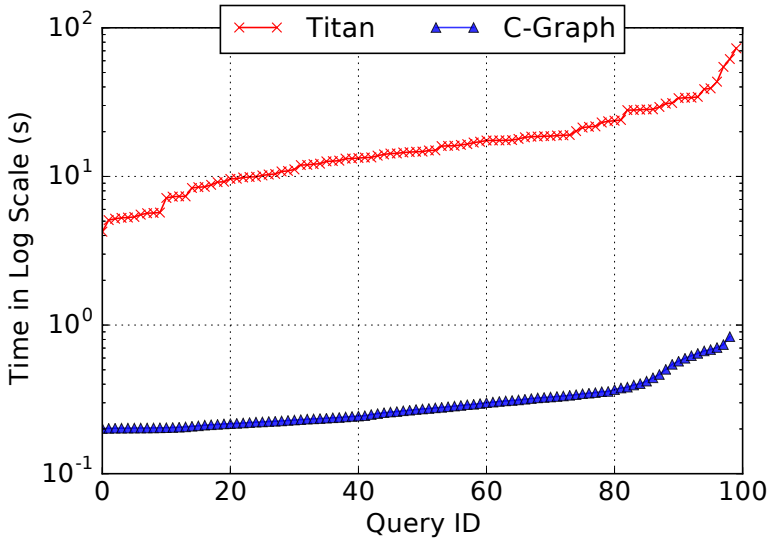
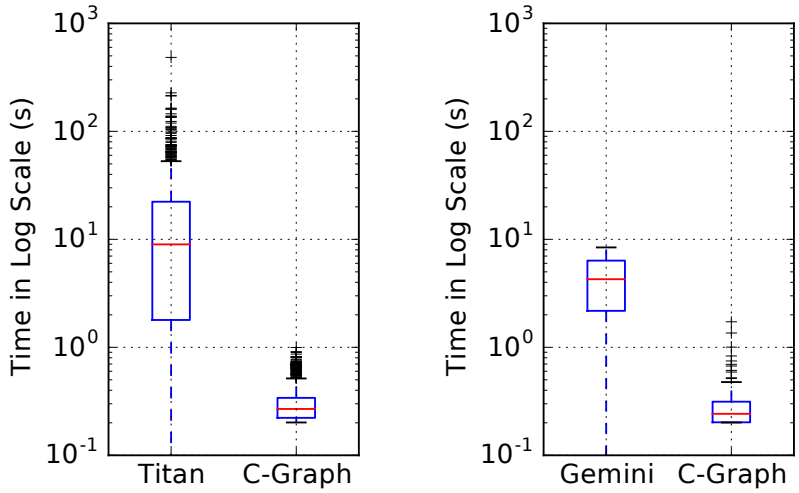


Figure 4.6: Single machine performance comparison of 100 concurrent *3-hop* queries with Titan running OR-100M graph.

The results were encouraging, with C-Graph achieving a 21×–74× speedup over Titan. Moreover, our system exhibited a much lower upper bound on query time, with all 100 *3-hop* queries returning within 1 second, while Titan took up to 70 seconds for some queries. In addition, our system showed much lower variation in response time.

We also compared the distribution of all 1000 subgraph traversal times, with the results shown in Figure 4.7a. The average query response time is 8.6 seconds for Titan, and only 0.25 second for C-Graph. About 10% of the queries in Titan took more than 50 seconds and up to hundreds of seconds. This is likely due to the complexity of the software stack used in Titan, such as the the data storage layers and Java virtual machine. These inefficiencies make the results for PageRank running on Titan even worse. For the Orkut (OR-100M) graph, Titan execution time was hours for a single iteration while C-Graph only took seconds. Overall, our system showed both better and more consistent performance gains compared to Titan.

Most existing graph processing systems lack the ability to handle concurrent queries in large-scale graphs. We use Gemini as an example of how inefficient a design that lacks concurrency can be. Simply using the alternative way in stead of re-design the concurrent support, for example making Gemini start with multiple source vertices, will fail. In these systems, concurrently-issued queries are serialized and a query's response time will be determined by any backlogged queries in addition to the execution time for current query. We used three machines and repeated the 100 queries with the Friendster (FR-1B) graph on both systems. The response time distribution is shown in Figure 4.7b. Even though Gemini is very efficient and only takes tens milliseconds for a single *3-hop* query, the average query response time is around 4.25 seconds due to the stacked up wait time. The average response time for C-Graph is only about 0.3 seconds.



(a) OR-100M (b) FR-1B

Figure 4.7: Response time distribution comparison of 100 concurrent *3-hop* queries. (a) Compared with Titan running Orkut (OR-100M) graph on single machine. (b) Compared with Gemini running Friendster (FR-1B) graph on three machines.

Next we focus on the scalability of our system. We ran experiments with different input graph datasets, increasing number of machines and query counts.

4.3.3 Data Size Scalability

For concurrent queries, an important performance indicator is how the upper bound of the response time scales as the input graph size increases. A good query system should guarantee that all queries return within latencies that are acceptable to the users. To understand how our system scales with increased input graph size, we measured its response times for different datasets: Orkut (OR-100M) with 100 million edges, Friendster (FR-1B) with 1 billion edges, and Friendster-Synthetic (FRS-100B) with 100 billion edges.

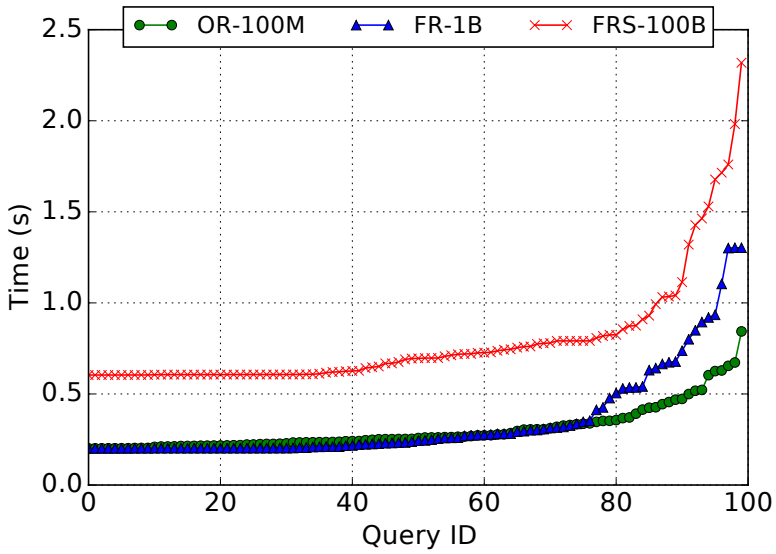


Figure 4.8: Data size scalability results of response times for 100 concurrent *3-hop* queries.

Figure 4.8 shows the histogram of response time for 100 concurrent *3-hop* queries running different graphs with 9 machines. We observed that for both graphs, about 85% queries return within 0.4 second for FR-1B, and for FRS-100B the response time slight increases to 0.6 second for the same percentage of queries. The upper bound of query response time is 1.2 seconds for FR-1B, and for FRS-100B it increases slightly to 1.6 seconds. The upper bound of response time for both graphs is within the 2.0 seconds threshold. Note that the response time highly depends on the average degree of root vertices, which is 38, 27, 108 for OR-100M, FR-1B and FRS-100B, respectively.

Multi-machine Scalability. We studied the scalability of our system with an increasing number of machines. We experimented both types of applications: PageRank and concurrent *3-hop* queries.

We examined the inter-machine scalability using 1 to 9 machines to run PageRank on graph datasets OR-100M, FR-1B and FRS-72B. The results are shown in Figure 4.9. All

results are normalized to single machine execution time of corresponding graph. Overall the scalability is very good. For FR-1B graph, it achieves speedup of 1.8x, 2.4x, and 2.9x using 3, 6 and 9 machines, respectively. With more machines the inter-machine synchronization becomes more challenging. In the smallest graph OR-100M, as expected, the scalability becomes poor beyond 6 machines as communication time dominates the execution. We observed better scalability with the largest graph FRS-72B, achieving up to 4.5x speedup with 9 machines.

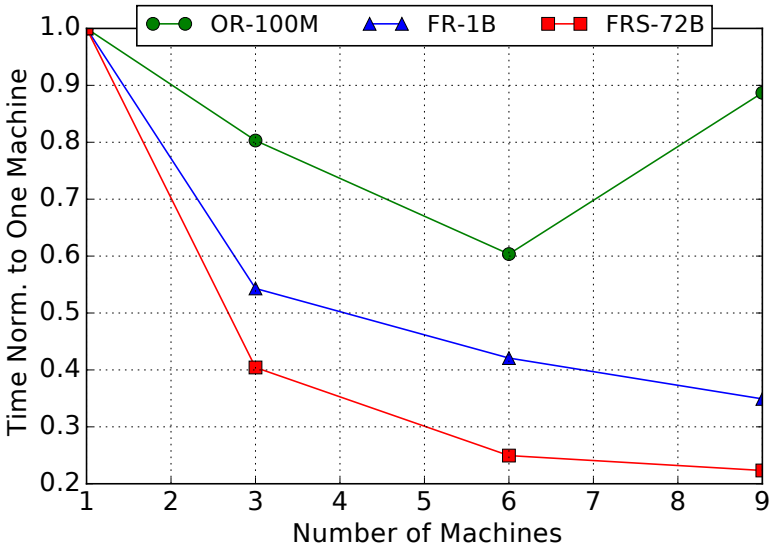


Figure 4.9: Multi-machine scalability results for PageRank.

Figure 4.10 depicts the response time distribution for 100 concurrent k-hop queries on a single graph using different number of machines. While the machine number increases, most of the queries are able to be completed in a short time, i.e., 80% queries receive a response within 0.2 seconds, and 90% queries finish within one second. For a fixed amount of concurrent traversal queries, as the number of machines used grows up, the number of visited

distinct vertices does not vary, while the number of boundary vertices increases significantly. More boundary vertices lead to increased communication overhead for synchronization. In our framework, we employ the partition-centric model combined with the edge-set technique to solve this problem.

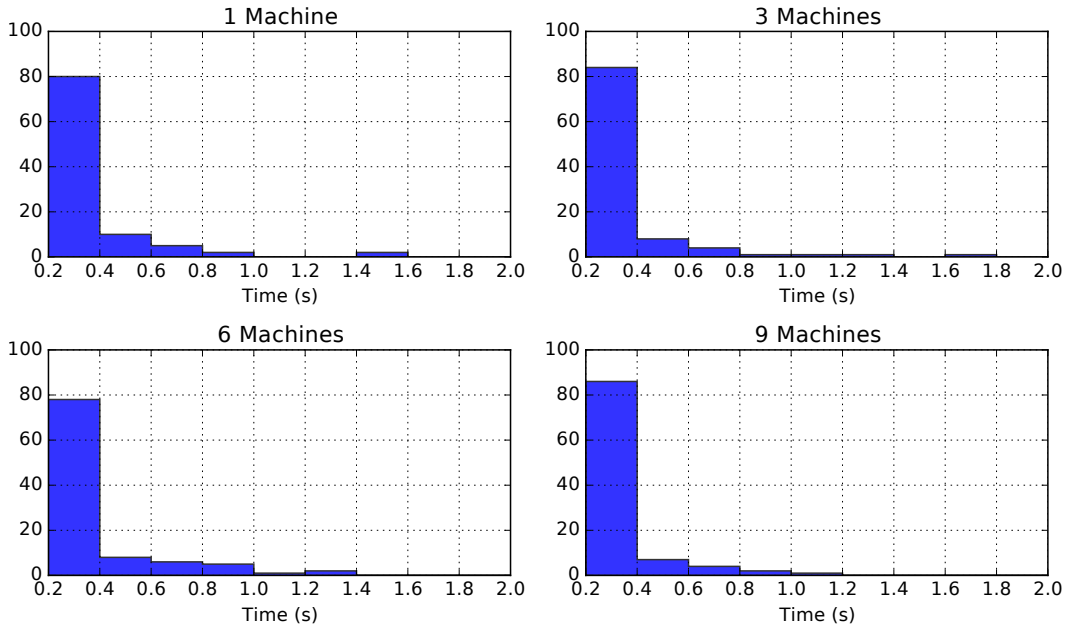


Figure 4.10: Multi-machine scalability results for 100 queries with FR-1B graph.

4.3.4 Query Count Scalability

The main goal of our framework is to execute concurrent graph queries efficiently. To evaluate this property, we study the scalability of our framework as the query count increases. Figure 4.11 shows the response time distribution for increasing number of concurrent *3-hop* queries running the FRS-100B graph on 9 machines. Up to 100 concurrent *3-hop* queries, most of the queries can finish in a short time. 80% of the queries are completed within 0.6

seconds, and 90% queries finish within one one second. When the concurrent query count reaches 350, the performance of C-Graph begins to degrade. About 40% queries are able to respond within one second, 60% queries can finish within the 2 seconds threshold. We have to wait 4 to 7 seconds for the remaining queries. The slowdown of the framework is mainly caused by resource limits, especially due to the large memory footprint required for concurrent queries. Since every query returns with found paths, the memory usage increases linearly with the query count.

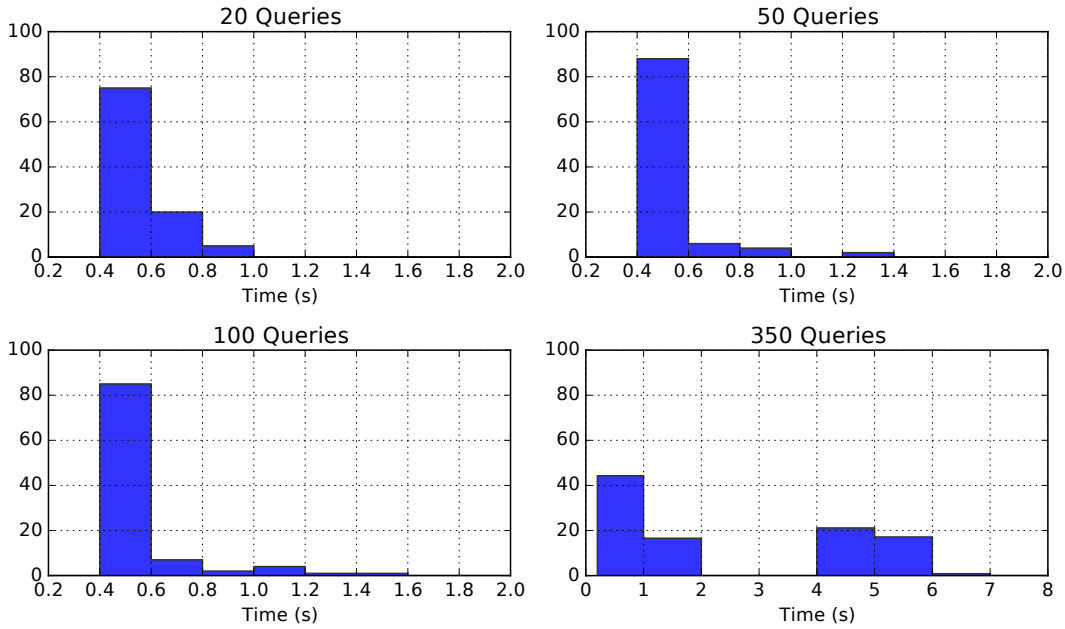


Figure 4.11: 3-hop query count scalability results for FRS-100B graph.

We further compared the performance and scalability of C-Graph to Gemini in order to maximize the query hops. We experimented with 1, 64, 128 and 256 concurrent BFS queries using the Friendster (FR-1B) graph on 3 machines. Since Gemini doesn't support concurrent queries, we reported total execution time for serialized queries running on Gemini. Also,

because our framework reaches the system's memory limit when running higher number of hops with more than 25 concurrent BFS queries, we enabled bit operations in this experiment. We also did not record the query paths. As Figure 4.12 shows, the execution time for Gemini is linear with the number of concurrent BFS queries. C-Graph starts with the same performance for a single BFS which is about 0.5 seconds. However C-Graph execution time increases sublinearly with the number of concurrent BFS queries. As a result C-Graph outperforms Gemini by about 1.7× at 64 and 128 concurrent BFSs, and 2.4× at 256 concurrent BFSs.

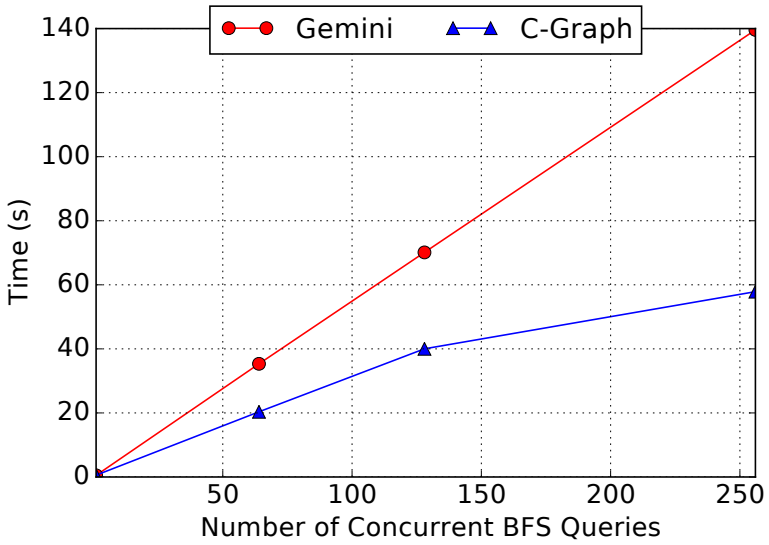


Figure 4.12: Performance comparison of concurrent BFS queries with Gemini running FR-1B graph on three machines.

4.4 Summary

In this chapter, we present our work on a concurrent graph processing framework called C-Graph. This system is designed to meet the industrial requirements of efficiently handling a group of simultaneous graph queries on large graphs, rather than accelerating a single graph processing task exclusively on a server/cluster as in prior work. To achieve this goal, the proposed framework maintains global vertex states to facilitate graph traversals, and supports both synchronous and asynchronous communication interfaces. For any graph processing tasks that can be decomposed into a set of local traversals, such as the graph *k-hop* reachability query, our proposed system exhibits excellent performance.

Chapter 5: Adaptive Parallel Execution of Deep Neural Networks on Heterogeneous Edge Devices

5.1 Introduction

Deep neural networks (DNNs) are rapidly becoming indispensable tools for solving complex problems that include computer vision [53], natural language processing [24], machine translation [13], and many others. Advancements in hardware [32, 46, 71] and lightweight frameworks [12, 88] are making the deployment of machine learning algorithms to new environments possible. In addition, various emerging applications are driving the need for deploying machine learning algorithms to edge devices in smart homes, smart cities, autonomous vehicles, and healthcare [15, 60, 68]. However, in most cases, the bulk of the computation, even for inference problems, is performed entirely within the cloud or through a hybrid combination of edge and cloud computing. In other words, inputs are collected on edge devices, but the processing is mostly offloaded to powerful servers in the cloud.

Cloud-based processing of user-generated data faces several challenges and limitations. For instance, uploading user data to the cloud raises privacy concerns. Consumers are becoming increasingly aware of the privacy implications associated with online services and are likely to be more concerned about devices uploading audio and video data to the internet

for further processing. Furthermore, many IoT applications require frequent decision making that render cloud-based computing impractical due to the communication latency it brings.

In response to the aforementioned concerns, efforts have been made to push machine learning inference from the cloud to the edge. Edge processing has the benefit of keeping data closer to its source to provide real-time responses while protecting the privacy of the end-user [36, 58]. Unfortunately, edge computing for machine learning workloads faces challenges of its own. Most machine learning algorithms are computationally demanding making edge devices inadequate for handling such workloads due to their constrained performance, energy, and memory capacities. To this end, a significant amount of research has investigated efficient approaches for deploying DNNs to the edge. This includes collaborative computation between edge devices and the cloud [37, 51, 87], model compression and parameter pruning [25, 40, 91, 102], or customized mobile implementations [44, 46, 71, 107]. Despite all these efforts, having the ability to scale existing DNNs without sacrificing the model accuracy and processing the collected data streams in real time present ongoing challenges to the deployment of machine learning across edge devices.

In this work, we explore parallel execution of DNN inference across multiple heterogeneous devices that are energy-constrained. A possible application that we envision for our work is local smart home processing. Instead of relaying collected data that includes voice commands, sensor readings, and video streams from a camera to the cloud, our solution leverages other smart home devices, such as speakers, light switches, and hubs to resolve the request. This approach creates a number of research challenges that need to be addressed: (1) how to partition the workload efficiently across devices; (2) how to optimize execution across heterogeneous devices possessing different compute capabilities; (3) how to account for the higher communication latency in wirelessly connected devices.

To answer these questions, we develop a framework for partitioning DNN inference in an optimal way across multiple heterogeneous devices. A simple model of the available devices and their compute capabilities is generated first. Next, our framework uses parameterized performance prediction models for multiple DNN layer types. The framework uses these models to predict the execution time for each layer based on available devices and communication latency, and selects the best partition points and parallelization strategies for each partition. To accommodate the heterogeneity of the compute environment, partition sizes are chosen to match device capabilities. Then at runtime, the partitions are distributed and executed across multiple devices. The framework is deployed and evaluated on the VGG-16 and ResNet-50 image recognition models [42, 84], and the YOLOv2 object detection model [75], achieving speedups of $1.9\times \sim 3.7\times$, relative to the models running on a single device.

Some prior work [36, 66] explored parallelizing DNN inference on edge devices. Their approach, however, relies on layer-granularity parallelism that result in large amounts of intermediate feature maps that are transferred across devices. When communication bandwidth is low, this significantly impacts performance. Other work [108] proposed fusing some of the DNN layers to reduce the communication overhead. Their approach is, however, limited to fusing the first several layers in the DNN. Our work generalizes the problem of layer fusion and proposes a mechanism for choosing the groups of DNN layers to be fused in an optimal way. This is also the first work we are aware of that considers device heterogeneity in this context.

Overall, this work makes the following contributions:

- We discuss the trade-offs in partitioning DNN inference among multiple lightweight edge devices, and propose an Adaptive Optimal Fused-layer (AOFL) parallelization to reduce the communication cost in an IoT network.

- We design a dynamic programming-based search algorithm to decide the optimal partition and parallelization for a DNN model.
- We present a collaborative CNN acceleration framework that adapts to the computing resources and network condition in the presence of heterogeneity.
- We apply our technique on a distributed IoT cluster consisting of Raspberry-Pi3-based hardware and evaluate image recognition and object detection DNN models.

The rest of this chapter is organized as follows: Section 5.2 explores different parallelization strategies in IoT devices and their trade-offs. Section 5.3 describes our approach for finding near-optimal parallelization. Section 5.4 presents the results of our evaluation. Section 5.5 concludes this work.

5.2 Distributing and Parallelizing DNNs Inference at the Edge

We envision the deployment of our system in an environment such as a smart home, in which devices of different types and compute capabilities collaborate to solve a joint task. This creates a number of research challenges that need to be addressed: (1) how to partition the workload efficiently between devices; (2) how to factor in the heterogeneity in compute capabilities of devices; (3) how to account for the higher communication latency in devices that connect wirelessly.

Figure 5.1 provides an overview of the approach we take to solve these challenges. First, a simple model of the available devices and their compute capabilities is generated. Next, our framework uses parameterized performance prediction models for multiple DNN layer types. At runtime, the framework predicts the execution time for each layer based on available devices and communication latency, and selects the best partition points and parallelization

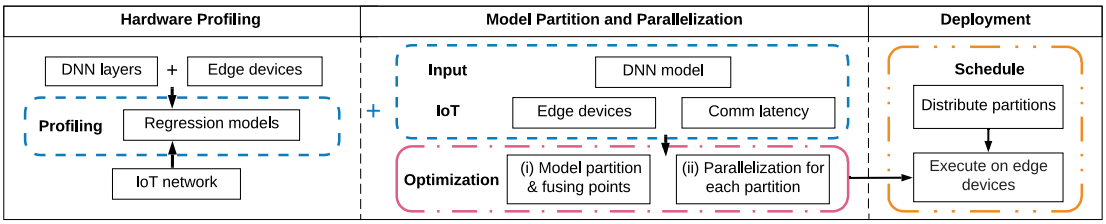


Figure 5.1: Design overview.

strategies for each partition. To accommodate the heterogeneity of the compute environment, partition sizes are chosen to match device capabilities. Then, the partitions are distributed and executed across multiple devices.

5.2.1 Model Parallelism and Partitioning

In this work we employ model parallelism to partition DNN inference among multiple lightweight edge nodes. Model parallelism subdivides the DNN parameters into partitions that can be assigned to multiple devices. Each partition generates a subset of the output feature maps. The partial outputs are then aggregated to form the final output for each layer. This approach can reduce computation latency for a single input and is therefore a good fit for parallelizing machine learning inference.

Partitioning methods. Figure 5.2 shows two partitioning methods that can be used to parallelize a 2-dimensional convolution layer: (1) *channel partitioning* and (2) *spatial partitioning*. As shown in Figure 5.2a, in a *conv* layer each filter generates a feature map (i.e, a channel of the output feature maps). The output feature maps can be partitioned along the *channel* dimension such that each device computes a subset of the output feature maps. This requires mapping the corresponding set of filters to each device. The input maps have to be replicated across all the devices. Channel partitioning is generally more beneficial for

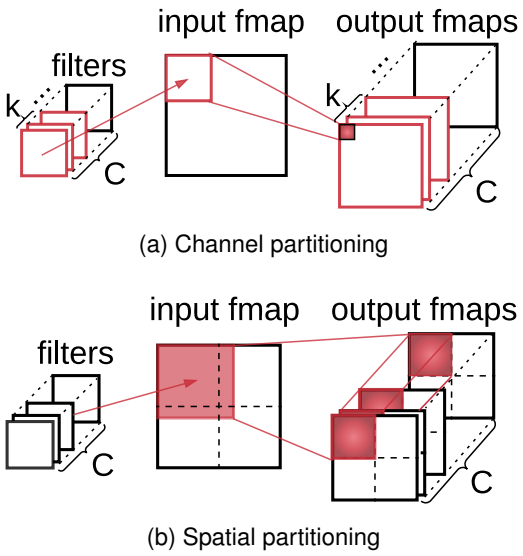


Figure 5.2: Examples of parallelizing a 2D convolution layer (note that, each filter and its corresponding input/output feature map have the same channel size and are not shown in the plots).

training because it reduces communication costs associated with synchronizing network parameters. The need to fully replicate inputs can, however, add substantial communication overhead making channel partitioning less practical for inference.

An alternative approach is to partition the output feature maps spatially (i.e., by height and width) and assign them to multiple devices. Each device keeps a copy of the network and computes a subset of the output feature maps. As Figure 5.2b shows, compared with channel partitioning where the entire input has to be transferred to all devices, in spatial partitioning each device only requires a subset of the input. This greatly reduces communication costs in edge networks that rely on wireless communication.

Note that due to the nature of the convolution operation, for filter sizes that are greater than 1, input partitions are not completely disjoint. Each partition has to be extended by

$\lfloor f_i/2 \rfloor$ (f_i is the size of filters of layer i) along both dimensions to include inputs from neighboring partitions that are required in the computation of the output partition.

5.2.2 Tradeoffs in Parallelizing CNNs in IoT Devices

We now introduce two parallelization strategies used in our framework to parallelize a DNN model.

Layer-wise parallelization. Prior work [52] has shown that the best parallelization strategy depends on the characteristics of the DNN (i.e., layer type, shape of feature maps and size of filters). As a result, layer-wise parallelization [50] has been proposed to allow each layer to be parallelized independently using the appropriate technique for each layer to obtain the best performance. However, in layer-wise parallelization, each device computes part of the output of the current layer and all the subsets of the output need to be merged and re-partitioned before the execution of next layer. This requires output to be gathered by a host node, partitioned and re-sent to all client devices. In a wireless network this results in substantial communication overhead, as we have shown in Figure 2.3. For our system, the benefits of layer-wise parallelization are generally defeated by the communication costs. We use layer-wise parallelization as a baseline for comparison.

Fused-layer parallelization. To reduce the data movement between layers, we propose using fused-layer parallelization. The concept of layer fusion was first proposed in [10] as a method to reduce off-chip data movement in a CNN accelerator. The idea is to send the output of one layer directly to the input of the next layer without going through memory. We propose extending this concept by parallelizing multiple fused layers instead of single layers individually.

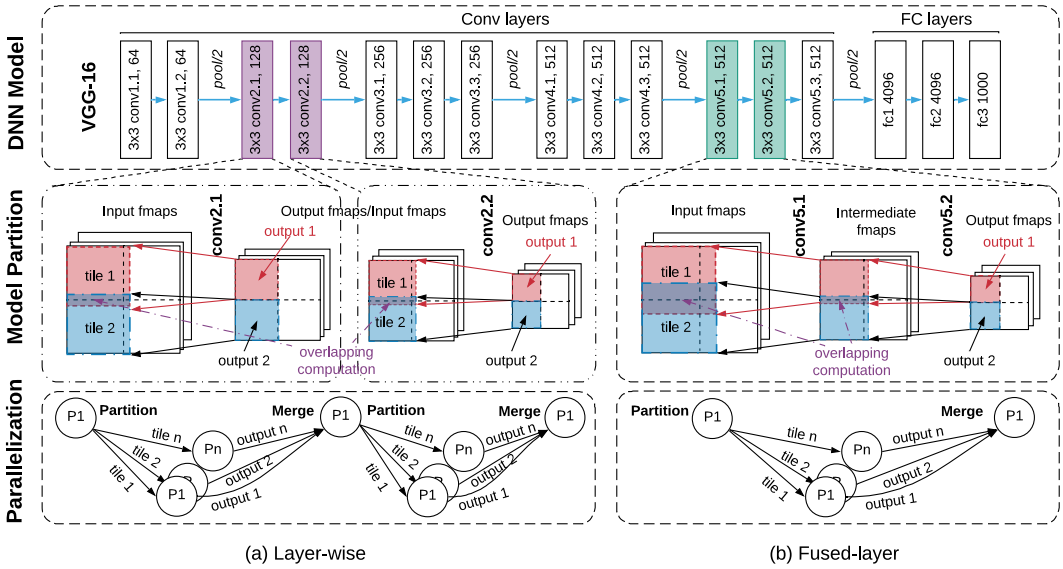


Figure 5.3: Illustration of layer-wise parallelization (a) vs layer fusion (b) for VGG-16.

Figure 5.3 illustrates this concept on the VGG-16 network. A layer-wise parallelization example is shown in Figure 5.3a for layers *conv*2.1 and *conv*2.2. The input maps for *conv*2.1 are partitioned by processor P_1 and assigned to processors $P_1 - P_n$ for computation. The outputs are then gathered at P_1, merged, partitioned and assigned as inputs to $P_1 - P_n$ in order to compute *conv*2.2.

Figure 5.3b shows an example of fused-layer parallelization. Two two convolution layers (*conv*5.1 and *conv*5.2) are parallelized as a single fused-layer block. Partitioning is performed layer-by-layer starting from the last layer in the fused block. Each layer's input is the output of the previous layer. In this example, the last layer (*conv*5.2) is divided into 2x2 disjoint subsets. The required input elements for each partition are calculated based on its output elements. For *conv* layer, we also need to extend each partition's input by $\lfloor f_i/2 \rfloor$ on height and width for overlapping elements. The process is applied recursively

up to the first layer in a fused block (*conv*5.1 in our example). Note that the overlap of the input partitions (highlighted in purple in Figure 5.3) leads to some redundant computation as well as additional communication overhead. As the number of fused layers increases so does the overlap in the input feature map partitions. This is because each input partition has to include all the input elements required to compute the last output partition of the fused block.

The partitions are next distributed to processors $P_1 - P_n$ for computation. All convolution layers in the fused block will now be computed locally and only the output of the last layer will be merged at P_1. This reduces communication costs because only the input of the first layer and the output of the last layer need to be communicated between devices. The more layers are fused, the more communication costs are reduced. However, layer fusion introduces additional costs compared to layer-wise partitioning. The overlap in input partitions adds to the communication cost of distributing those partitions, and adds redundant computation to each node. This cost increases with the number of fused layers. As a result, finding the optimal number of layers to be included in a fused block requires carefully balancing the costs and benefits of layer fusion.

Computation vs. communication trade-off. Figure 5.4 and 5.5 compare computation and communication costs for layer-wise (LW) and fused-layer (FL) parallelization in VGG-16. Beginning at *conv*2.1, fused-layer parallelization fuses between 2 and 12 layers at different partitioning granularities (i.e., 4 (2x2), 9 (3x3), and 16 (4x4)). Computation load is represented in billions floating point operations per second (BFLOPs). We can see that computation increases as expected with the number of layers. When the number of fused layers is small (2-4) the amount of computation performed by FL and LW is very close. This is because the overhead of redundant computation is small as the first couple of layers have

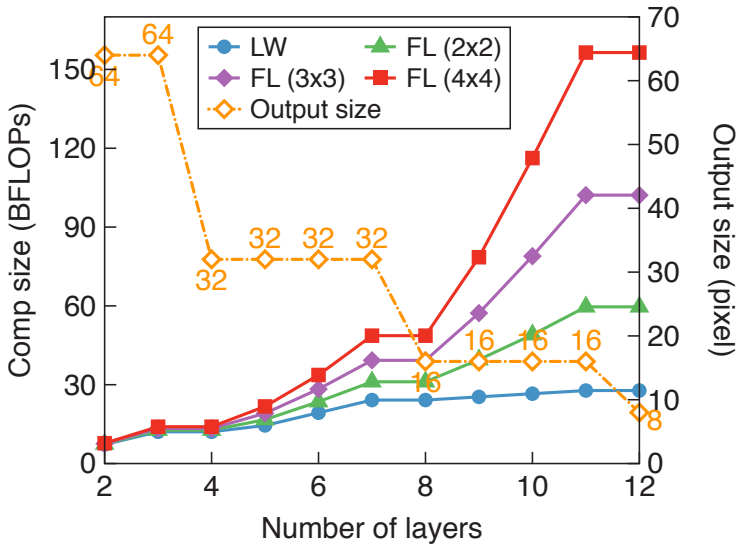


Figure 5.4: Comparison of computation size (BFLOPs) for Layer-wise (LW) and Fused-layer (FL) parallelization over 12 layers starting from *conv*2.1 of VGG-16.

a relatively large spatial output dimension (64x64). The output dimension is shown by the yellow line in Figure 5.4. As the number of fused layers increases, and the output spatial dimension decreases ($\leq$ 16x16) the cumulative computation overhead increases dramatically, up to $3\times \sim 5\times$ when fusing 12 layers. It also leads to more overlapping elements in the first input layer which increases communication costs.

The communication costs of FL are approximated using a 4-device setup by measuring input and output data sizes of a fused block. In LW, the communication cost is calculated as the cumulative input data size of each layer. Figure 5.5 highlights communication costs for the two techniques measured as the amount of data transferred between devices, as a function of the number of layers. We can see that in all cases, FL results in substantially lower communication overhead compared to LW. For instance, when fusing 8 layers, the communication cost of FL is $3\times$ higher than that of FL. This is because FL data is transferred

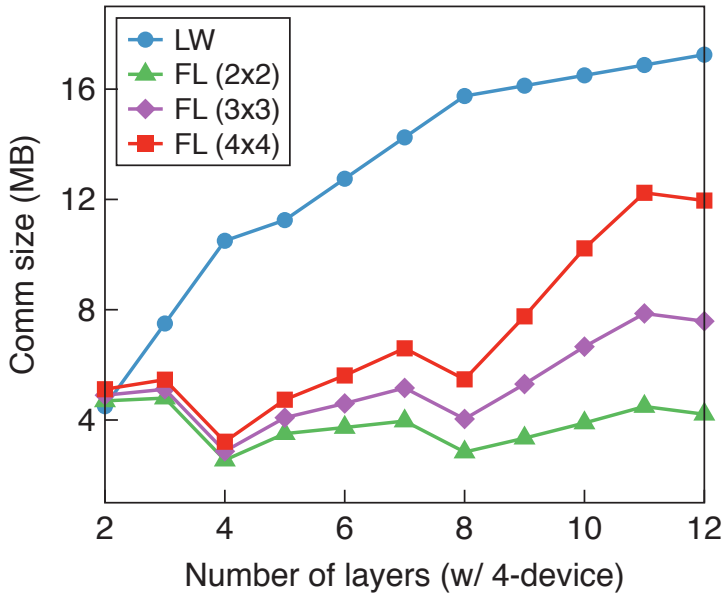


Figure 5.5: Comparison of communication size (MB) for Layer-wise (LW) and Fused-layer (FL) parallelization over 12 layers starting from *conv*2.1 of VGG-16.

only for the first and last layer in the fused block, while LW requires device-to-device data transfers between all layers.

The degree of parallelism (i.e., number of parallel devices) is another factor that affects performance. Different layers or fused blocks, have different computation needs and communication costs, which affect the optimal degree of parallelism. Our target environment is especially sensitive to communication cost which often limits the optimal number of devices. Figure 5.6 shows the estimated per-device computation and total communication costs for a 4-fused-layer block with different number of devices. As the number of devices increases, the per-device computation decreases while the total communication overhead increases. The communication overhead for FL increases more slowly with the number of devices

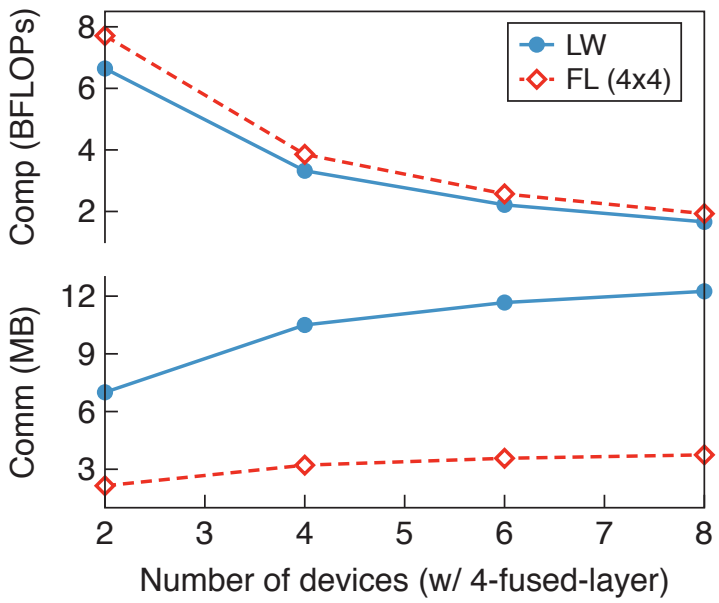


Figure 5.6: Comparison of reduced per-device computation size and communication size for 4 layers on 4 devices.

compared to LW, and it is also lower overall. This suggests that FL is likely exhibit better scalability with the number of devices.

Deploying fused-layer parallelism in our environment in an optimal way requires answering the following questions: (1) which layers should be fused, (2) how many layers to include in each fused block, (3) for each fused and unfused block, how many partitions/devices offer the optimal performance and (4) how to match the partition size to the capability of the device in a heterogeneous environment?

5.3 Adaptive Fused-layer Partition and Parallelization

Answering the aforementioned questions requires solving a multivariate optimization problem. We present a dynamic programming based search algorithm to find the parameter values that are projected to achieve the lowest execution time under our cost model.

Table 5.1: Symbol definitions.

Symbol	Description
$\mathbb{G}$	A CNN model with n layers
$\mathbb{D}$	A list of devices, with known computation abilities and communication latency
$\mathbb{S}$	Partition and parallelization configurations for all layers in model $\mathbb{G}$
$\mathscr{G}^{lw}$, $\mathscr{G}^{fl}$	Layers in model $\mathbb{G}$ using layer-wise or fused-layer parallelization, $\mathbb{G} = \mathscr{G}^{lw} \bigcup \mathscr{G}^{fl}$
$\mathscr{S}^{lw}$, $\mathscr{S}^{fl}$	Partition and parallelization configurations for layers using layer-wise or fused-layer parallelizations, $\mathbb{S} = \mathscr{S}^{lw} \bigcup \mathscr{S}^{fl}$
$l_i, l_{(i,j)}$	Layer(s) in model $\mathbb{G}$, $l_i, l_{(i,j)} \in \mathbb{G}$
$e = (l_i, l_j)$	A tensor e from layer l_i to layer l_j
d_i	A device in IoT cluster, $d_i \in \mathbb{D}$
c_i	A parallelization configuration for layer i, $c_i \in \mathbb{S}$
$t_l(i)$	The cost function for layer l_i using layer-wise parallelization
$t_f(i, j)$	The cost function for a fused block that fuses j consecutive layers from layer i

5.3.1 Problem Definition

Given a CNN model $\mathbb{G}$ with n layers, where $l_i \in \mathbb{G}$ is a layer in the model and edge (l_i, l_j) is a tensor that is an output of layer l_i and an input of layer l_j. The model runs on a list of devices $\mathbb{D}$, and we assume the communication bandwidth of each connection

between two devices (d_i, d_j) is known. For layer-wise parallelization, a parallelization strategy $\mathscr{S}^{lw}$ includes a configuration c_i for each layer l_i which consists of a combination of parameters from $\{height, width, channel\}$, and a subset of devices $\mathscr{D} \in \mathbb{G}$. For layer fusion we add the grouping of layers as an additional dimension to the optimization. For each fused block, a parallelization strategy $\mathscr{S}^{fl}$ is generated. The optimization objective is to find a parallelization strategy $\mathbb{S} = \mathscr{S}^{lw} \bigcup \mathscr{S}^{fl}$ such that the execution time $\mathscr{T}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ is minimized.

5.3.2 Cost Model

We develop a performance model that will be used to guide the optimization search. To construct the model for each layer type, we vary the configuration parameters of the layer and measure the latency for each configuration. Using the profiles, we build a regression model for each layer type to predict execution latency. To predict the communication cost, we use a similar approach by varying the data transfer size and measuring the latency. We define two basic cost functions:

- For each layer l_i and its parallelization configuration c_i from $\{height, width, channel\}$, $t_c(l_i, c_i, \mathscr{D})$ is the time to process layer l_i under configuration c_i on devices $\mathscr{D} \in \mathbb{D}$. This only includes the inference time which is estimated by processing the layer under the configuration multiple times on the devices and measuring the average execution time.

- For each $e = (l_i, l_j)$, $t_x(e, d, k)$ is time to transfer the tensor e between two devices d and k using the size of the data and the known communication bandwidth. $t_x(e, d, \mathscr{D}) = \sum_{k \in \mathscr{D}} t_x(e_k, d, k)$ is the total time of transferring the tensor e_k from a local device d to remote devices $k \in \mathscr{D}$.

The cost functions for each layer l_i in layer-wise parallelization is then defined as the total time of the input/output tensors transfer and layer computation:

$$t_l(i) = t_x(e_{in}, d, \mathscr{D}) + t_x(e_{out}, d, \mathscr{D}) + t_c(l_i, c_i, \mathscr{D}) \tag{5.1}$$

Using this cost function, we define

$$\mathscr{T}(\mathbb{G}, \mathbb{D}, \mathscr{S}^{lw}) = \sum_{l_i \in \mathbb{G}} t_l(i) \tag{5.2}$$

$\mathscr{T}(\mathbb{G}, \mathbb{D}, \mathscr{S}^{lw})$ estimates the execution time of a single inference under layer-wise parallelization strategy $\mathscr{S}^{lw}$.

Next, assuming several consecutive convolution layers are grouped under fused-layer parallelization strategy $\mathscr{S}^{fl}$, where $\mathscr{S}^{lw}(i)$ of layer l_i in the fused block will be replaced with $\mathscr{S}^{fl}(i, j)$. The cost function for a fused block $l_{(i,j)} \in \mathscr{G}^{fl}$ (i.e., fusing j consecutive layers from layer i) is then defined as the total data transfer time of the first layer's input tensor, the last layer's output tensor, and the sum of the computation time of all grouped layers, running on $\mathscr{D}$ devices:

$$t_f(i, j) = t_x(e_{in}, d, \mathscr{D}) + t_x(e_{out}, d, \mathscr{D}) + \sum_{i \leq k < i+j} t_c(l_k, c_k, \mathscr{D}) \tag{5.3}$$

Next we define

$$\mathscr{T}(\mathbb{G}, \mathbb{D}, \mathbb{S}) = \sum_{l_i \in \mathscr{G}^{lw}} t_l(i) \; + \sum_{l_{(i,j)} \in \mathscr{G}^{fl}} t_f(i, j) \tag{5.4}$$

where $\mathscr{T}(\mathbb{G}, \mathbb{D}, \mathbb{S})$ estimates the total execution time of a single inference for a model $\mathbb{G} = \mathscr{G}^{lw} \bigcup \mathscr{G}^{fl}$ on a list of devices $\mathbb{D}$ under a hybrid layer-wise/fused-layer parallelization strategy $\mathbb{S} = \mathscr{S}^{lw} \bigcup \mathscr{S}^{fl}$.

5.3.3 Dynamic Programming-based Optimization

The optimization goal for our framework is finding parallelization strategy $\mathbb{S}$ that minimizes the runtime $\mathscr{T}(\mathbb{G},\mathbb{D},\mathbb{S})$. Given the relatively small optimization space, we use dynamic programming-based search through the space of possible solutions.

We start by determining the optimal parallelization configuration for each layer, and each fused block. We find the optimal $\mathscr{S}^{lw}$ and the optimal $\mathscr{S}^{fl}$ with two tables of cost $t_l(.)$ and $t_f(.)$ with a list of devices $\mathscr{D}$ by varying the used number of devices from 1 to the maximum. Then we use the following dynamic programming based search algorithm to find the optimal placement of fused blocks in a model.

We define the *fused-layer partitioning problem* as follows. Given a CNN model $\mathbb{G}$ with n layers and tables of cost $t_l(.)$ and $t_f(.)$, determine the minimum cost $t_o(i,j)$ (equivalent to $\mathscr{T}_o(\mathbb{G},\mathbb{D},\mathbb{S})$ when $i=0, j=n$) that can be achieved through layer fusion.

We can partition a n-layer model in 2^{n-1} ways, since we have an independent option of fusing, or not fusing, at distance j from the first layer, for $j = 1,2,...,n$. We denote a decomposition into parts using ordinary additive notation. If an optimal solution partitions the model into k parts, for some $1 \leq k \leq n$, then an optimal decomposition $n = i_1 + i_2 + ... + i_k$ of the model into fused layers $i_1, i_2, ..., i_k$, for a minimum time:

$t_o(0,n) = t_f(0,i_1) + t_f(i_1,i_2) + ... + t_f(\sum_{1\leq j\leq k-1} i_j, i_k)$.

More generally, we can frame the optimal value $t_o(0,n)$ for $n \geq 1$ in terms of optimal cost from fusing layers:

$t_o(0,n) = \min(t_f(0,n),\ t_o(0,1)+t_o(1,n-1),\ t_o(0,2)+t_o(2,n-2),...,\ t_o(0,n-1)+t_o(n-1,1))$.

The first argument, $t_f(0,n)$, corresponds to fusing all layers of the model. The other $n-1$ arguments correspond to the minimum time obtained by making an initial partitioning

Algorithm 4 Adaptive Optimal Fused-layer (AOFL) Parallelization Strategy Search Algorithm

Input $\mathbb{G}$: a CNN model with n layers
$\mathbb{D}$: a list of devices
$\mathscr{S}^{lw}(.)$ and $\mathscr{S}^{fl}(.)$: precomputed parallelizations
$t_l(.)$ and $t_f(.)$: precomputed cost functions
Output $\mathbb{S}$: a parallelization strategy that minimizing $\mathscr{T}(\mathbb{G},\mathbb{D},\mathbb{S})$
$t_o(.) \leftarrow MAX,\ fl_o(.) \leftarrow 1$
function AOFL(i, j)
    if $t_o[i][j] < MAX$ **then**
        return $t_o[i][j]$
    if $j = 0$ **then**
        $t_{min} \leftarrow 0,\ l_{opt} \leftarrow 0$
    else if $j = 1$ **then**
        $t_{min} \leftarrow t_l(i),\ l_{opt} \leftarrow 1$
    else
        $t_{min} \leftarrow MAX$
        for $k \in [1, j]$ **do**
            $t \leftarrow t_f(i,k) + AOFL(i+k, j-k)$
            if $t < t_{min}$ **then**
                $t_{min} \leftarrow t,\ l_{opt} \leftarrow k$
    $t_o[i][j] \leftarrow t_{min},\ fl_o[i] \leftarrow l_{opt}$
    return $t_o[i][j]$
function BUILDSTRATEGY($\mathscr{S}^{lw}(.)$, $\mathscr{S}^{fl}(.)$, $fl_o(.)$)
    $i \leftarrow 0$
    while $i < n$ **do**
        if $fl_o(i) = 1$ **then**
            Add $\mathscr{S}^{lw}(i)$ to $\mathbb{S}$
        else
            Add $\mathscr{S}^{fl}(i, fl_o(i))$ to $\mathbb{S}$
        $i \leftarrow i + fl_o(i)$
    return $\mathbb{S}$

of the model into two groups of layers i and $n-i$, for each $i = 1,2,...,n-1$, and then recursively searching for the optimal subpartions of those layer groups. We may view every decomposition of a j-layer part starting from layer i as fusing first part followed by some

Algorithm 5 Update Cost Functions $t_l(.)$ and $t_f(.)$

Input $\mathbb{G}$ and $\mathbb{D}$
Output $t_l(.)$, $t_f(.)$ and $\mathscr{S}^{lw}(.)$, $\mathscr{S}^{fl}(.)$
function UPDATE($\mathbb{G}, \mathbb{D}$)
    $\mathscr{D} \leftarrow \emptyset$, $t_l(.) \leftarrow MAX$, $t_f(.) \leftarrow MAX$
    Rank $\mathbb{D}$ by $freq$ and then by bw between source dev
    ▷ Iterate $\mathbb{G}$ as follow:
    for $i \in [0, n-1]$ **do**
        for $j \in [1, n-i]$ **do**
            $\mathscr{D}_l(.) \leftarrow \emptyset$, $\mathscr{D}_f(.) \leftarrow \emptyset$
            for $d \in \mathbb{D}$ **do**
                $\mathscr{D}' \leftarrow \mathscr{D} \bigcup d$
                Adjust the size of partitions in c_i, if applicable
                if $j = 1$ **then**
                    Predict $t_l(i)^{\mathscr{D}'}$ with regression model
                    if $t_l(i)^{\mathscr{D}'} < t_l(i)$ **then**
                        $t_l(i) \leftarrow t_l(i)^{\mathscr{D}'}, \mathscr{D}_l(i) \leftarrow \mathscr{D}'$
                        Update $\mathscr{S}^{lw}(i)$ with c_i and $\mathscr{D}_l(i)$
                Predict $t_f(i,j)^{\mathscr{D}'}$ with regression model
                if $t_f(i,j)^{\mathscr{D}'} < t_f(i,j)$ **then**
                    $t_f(i,j) \leftarrow t_f(i,j)^{\mathscr{D}'}, \mathscr{D}_f(i,j) \leftarrow \mathscr{D}'$
                    Update $\mathscr{S}^{fl}(i,j)$ with c_i and $\mathscr{D}_f(i,j)$

decomposition of the reminder, and simplify the general equation for dynamic programming as below:

$$t_o(i,j) = \begin{cases} 0 & j = 0, \\ t_l(i) & j = 1, \\ \min\limits_{1 \le k \le j} \left(t_f(i,k) + t_o(i+k, j-k)\right) & otherwise. \end{cases} \tag{5.5}$$

Algorithm 4 shows the pseudocode of our optimization algorithm which uses dynamic programming with memoization to find out the optimal parallelization strategy. Function *AOFL* computes the minimum time and a list of optimal number of fused layers starting from a given layer. The optimal parallelization strategy is built up through function *BuildStrategy*

by iterating the model from the beginning with steps of a list of optimal number of fused layers, and adding corresponding parallelization configuration to $\mathbb{S}$.

Optimizing for heterogeneity. In order to adapt to the heterogeneity in compute capabilities of the IoT network, we need to balance the workload assigned to each device such that they all finish execution at roughly the same time. To that end, the algorithm ranks participating devices $\mathbb{D}$, first by clock frequency ($freq$) then by network bandwidth (bw) between the source and destination devices. The two cost tables of $t_l(.)$ and $t_f(.)$ are updated using function *Update* described in Algorithm 5. Within a fused block, the computation and communication costs of each partition are functions of its input size $\mathscr{P}$, and the execution time of each partition can be represented as

$$t_f(.)_i = t_{comm}(\mathscr{P}_i) + t_{comp}(\mathscr{P}_i) \tag{5.6}$$

where $i = 1, 2, .., N$. The ratio of input sizes of two partitions in a fused block then can be calculated with the regression models by setting $t_f(.)_i$'s equal. To obtain the best performance, a top device is used when predicting the execution time when adding one more device.

5.3.4 Implementation

We use Darknet [73] with NNPACK [28] as backend inference engine kernel to execute convolution layer, and implement a distributed framework integrated with network communication modules using TCP/IP with socket.

Linear regression performance model. We used a linear regression model to predict the runtime of different convolution layers with different configurations and various input shapes based on the partitioning scheme. The model is trained on samples with different convolution related parameters like inputs size and channel, number of filters, size of the

filters, padding and stride. We randomly selected appropriate values from the range of interest for each parameter. We run each sample and get its runtime to generate the training and test sets for linear regression. We train the linear regression model and achieved $> 98\%$ accuracy.

Deployment and device mapping. Assuming a CNN model is divided into blocks, and each block includes one or more fused layers respectively. The local device that captures the input (e.g., a video camera or smart speaker) is responsible for partitioning the input tensor, and distributing the partitioned input among devices. Note that it is possible that the optimal configuration for some blocks is to run on a single device, in which case they will be deployed on a single node. The remote devices start to execute once a task is received and send the results back once the job is completed. All results will be merged at the local device and re-partitioned for the next block.

Reconfiguration. The performance of edge devices in IoT networks fluctuates. Our system periodically recalculates the optimal partition points to adapt to changes in computational capabilities and network conditions. Once there is a change in the model parallelization, we adjust the configurations of partitions and reschedule them. It takes about several seconds to find the optimal partition points and reconfigure the framework. Note that the interval for recalculating the optimal partitions should be carefully selected to avoid performance degradation due to rescheduling overhead. We leave this exploration to future work.

5.4 Evaluation

In this section, we conduct experimental evaluations to validate the effectiveness and robustness of the proposed Adaptive Optimal Fused-layer (AOFL) parallelization. We first explore the performance trade-off at different partitioning granularities and compare the

runtime of different parallelization strategies. We then demonstrate the robustness of our solution to heterogeneous network and processing conditions.

5.4.1 Experimental Setup

Workloads. We focus on parallelizing a total of 13 convolution layers from VGG-16 and 23 layers from YOLOv2, accounting for 73.8% and 99.3% of their total execution time, respectively.

Testbed. We build two IoT clusters for running our experiments using the Raspberry-Pi3 B+ model. Each Raspberry-Pi3 B+ consists of a 1.4 GHz Quad Core ARM Cortex-A53, 1 GB LPDDR2 SDRAM and dual-band 2.4 GHz/5 GHz wireless. The first cluster (Cluster-1) is dedicated to performance evaluation and consists of 8 single core nodes that are fixed to run at 1 GHz in order to represent a realistic low-end edge device cluster. The second cluster (Cluster-2) is configured to represent a heterogeneous edge environment. We use it to evaluate the framework's ability to deal with heterogeneous devices that have different core frequencies between 400 MHz and 1 GHz. In addition, to evaluate the impact of different network conditions, we test with three network configurations consisting of two routers that support 2.4 GHz/5 GHz WiFi. (1) WiFi-1, a fast network with a measured bandwidth of 93.7 Mbps. (2) WiFi-2, a medium speed network with a measured bandwidth of 62.7 Mbps. (3) WiFi-3, a slow network with the measured bandwidth of 25.1 Mbps. Finally, we assume that each device reserves enough memory before it processes any assigned tasks and that all the trained weights are loaded into each device's storage.

Schemes. We compare four different parallelization strategies in our evaluation: (1) Layer-wise (LW) parallelization, which parallelizes the networks layer by layer; (2) Early Fused-layer (EFL) parallelization, an extension of to the implementation of DeepThings [108],

which fuses and parallelizes the first few convolution layers of a model and executes the remaining layers in a single device; (3) Optimal Fused-layer (OFL) parallelization, which selectively fuses convolution layers at different parts of a model; (4) Adaptive Optimal Fused-layer (AOFL) parallelization, which extends OFL by dynamically adapting to the available computing resources and network condition in an IoT network.

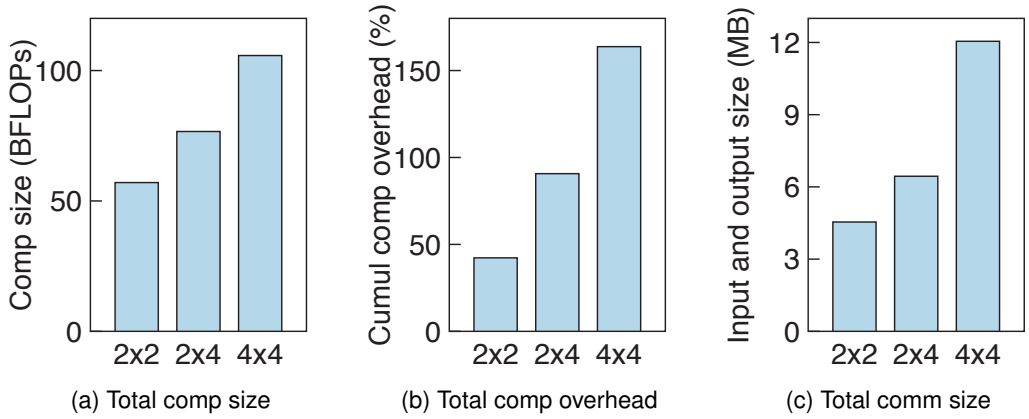


Figure 5.7: An example of total computation size (BFLOPs), computation overhead (%), and communication data size (MB) with same fused blocks in VGG-16 at three different partitioning granularities.

5.4.2 Partitioning Granularity in Fused-layer Parallelization

The performance of fused-layer parallelization depends on partitioning configurations that include the points of interest for layer fusion in a CNN model, the partitioning granularity, and the number of devices. Given the fused blocks in a model, the partitioning granularity of each block affects the computation and communication size. In our analysis, we quantify computation in billion floating-point operations per second (BFLOPs), and communication

in terms of transferred data size in megabytes (MB). For an IoT network where the computational power of devices and the network bandwidth are known, the partitioning granularity represents a qualitative measure of the computation to communication ratio.

Figure 5.7 shows an example of the impact of different partitioning granularities on computation and communication with two fused blocks in VGG-16. The first seven and last six convolution layers in the model are fused. Finer granularity results in lower average computation, however its total computation is increased, as well as the computation overhead. Increasing the granularity from 2x2 to 2x4, the average computation size is reduced by 35.4%, but the total computation size is increased by 34.3% as shown in Figure 5.7a. Because finer granularity creates more redundant computation from overlaps among partitions. Figure 5.7b shows increasing the granularity from 2x4 to 4x4 increases total overhead from 90.7% to 163.7%.

Figure 5.7c shows similar trend for communication size. We observe that with the same fused blocks, the difference of total transferred data size between two partitioning granularities is from the cumulative overlaps among partitions in the input layer of fused blocks. Switching granularity from 2x2 to 2x4, the communication size is decreased by 29.1% on average but increased by 41.8% in total.

We test the three partitioning granularities with Cluster-1 and WiFi-1, and observe that granularity of 2x2 achieves the best performance on 4 devices since it has the least computation and communication cost. Overall, enabling more devices requires increasing the number of partitions. As such, our design adapts to different partitioning granularities by adjusting the fused blocks to reduce the computation cost. For instance, at granularity of 2x4 on 8 devices, breaking the second fused block into two smaller blocks with 3 convolution layers in each can effectively reduce the computation and communication cost by 32.3%

and 4.4% respectively. In general, the choice of the partitioning granularity is a trade-off between computation and communication which varies based on the available computational resources and the condition of the network. Our design uses Algorithm 4 to determine the optimal parallelization configurations.

5.4.3 Runtime Performance

We compare the execution time of our Optimal Fused-layer (OFL) parallelization with two baselines: Layer-wise (LW) and Early Fused-layer (EFL) parallelizations on 1 to 8 devices for VGG-16 and YOLOv2. We set the partitioning granularity to be the same number of used devices.

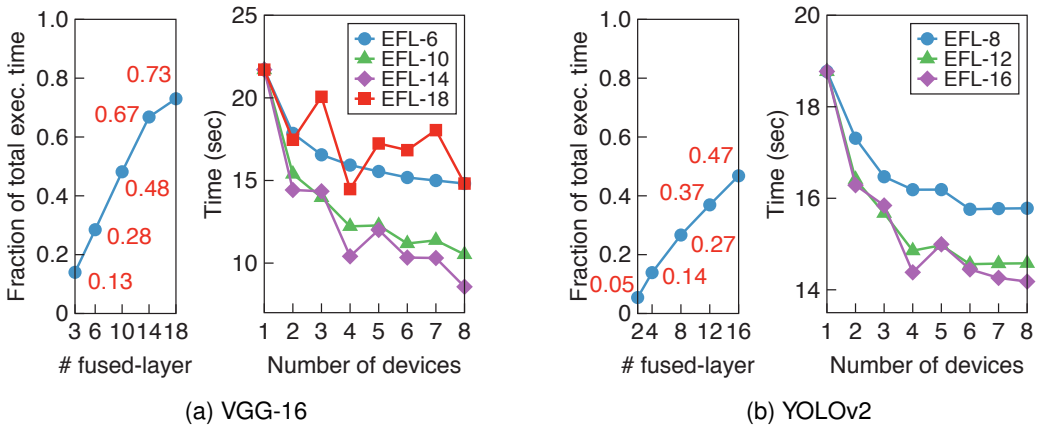
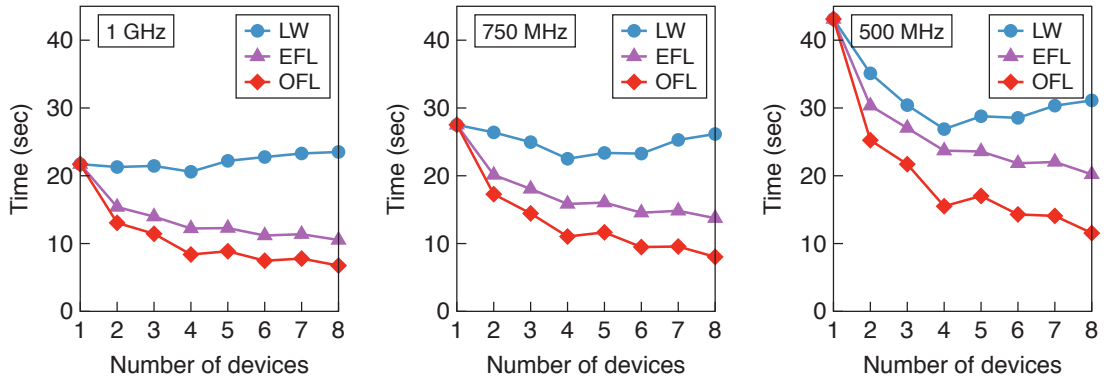


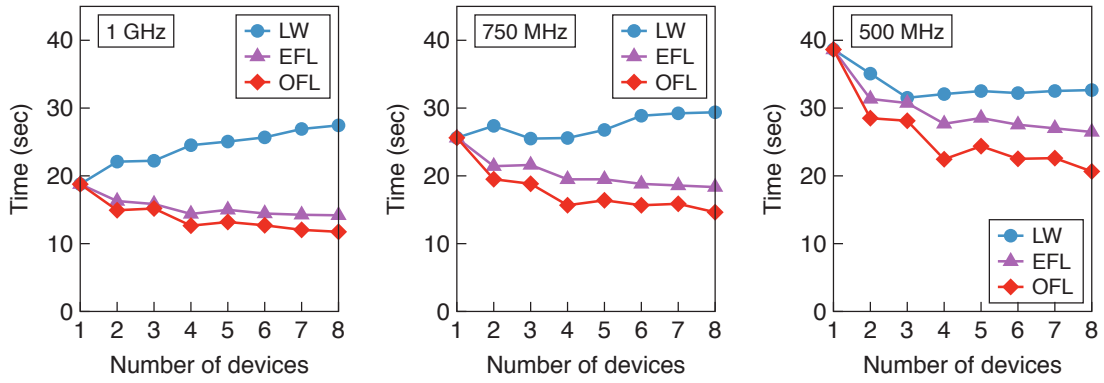
Figure 5.8: Performance of Early Fused-layer (EFL) parallelization with different number of fused layers.

To find the optimal fusing point for EFL, we vary the number of fused layers from the beginning of each model. As shown in Figure 5.8, the performance of EFL initially improves when the number of fused layers and devices is increased. However, EFL doesn't scale well

with deeper CNNs. We observe that fusing 14 and 18 layers in VGG-16 results in similar execution times. The same is observed with fusing 12 and 16 layers in YOLOv2. In some cases, the performance may even degrade if too many layers are fused. For instance, fusing 18 layers in VGG-16 runs slower than fusing 6 layers. Another drawback of EFL relates to the fraction of layers that can be fused within a given model. EFL only works on convolution layers, leaving 27% $\sim$ 33% of VGG-16 and 54% $\sim$ 63% of YOLOv2 execution serialized.



(a) VGG-16: performance



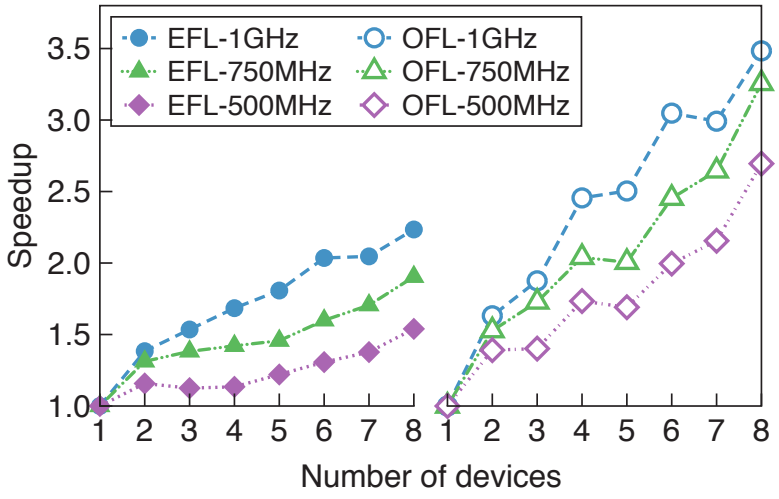
(b) YOLOv2: performance

Figure 5.9: Performance of different parallelizations running VGG-16 and YOLOv2 at different frequencies. The speedup is compared with Layer-wise (LW) parallelizaiton at the corresponding frequency.

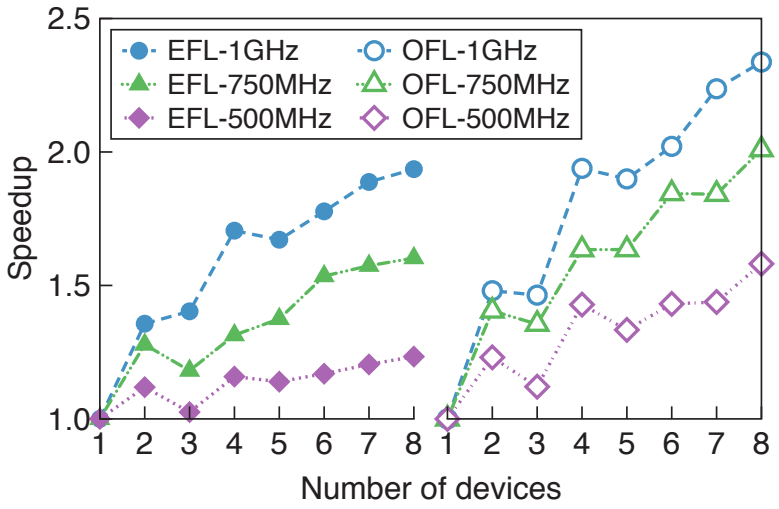
Figure 5.9 shows the execution time of LW, EFL and OFL at different frequencies. In LW, the communication cost increases linearly with the number of devices due to the per layer data distribution and processing synchronization. The latency reaches a minimum on 4 devices across different frequencies in VGG-16. However, the latency increases in YOLOv2 at higher frequencies when using more devices since the model has lightweight convolution layers. Overall, the performance gain from parallelizing computation is easily canceled out because of the the high communication cost. As such, we conclude that LW is not suitable for networks that have a low computation to communication ratio.

By contrast, EFL and OFL reduce the communication cost through layer fusion. Despite the computation overhead, both of them scale their performance with more devices. For example, at 750 MHz EFL reduces the latency by 27.7% $\sim$ 47.5% on 2 to 8 devices relative to LW. OFL further reduces the latency by 14.1% $\sim$ 41.5% on 2 to 8 devices relative to EFL. We observe that OFL always performs better than EFL for the same partitioning granularity due to its flexibility in fusing layers. Unlike EFL, OFL is able to harness parallelization for deeper layers in a model and adapt the fused blocks as the number of devices changes. As a result, OFL is 1.2$\times$ $\sim$ 1.7$\times$ faster than EFL. We also observe that OFL shows better improvement over EFL with slower frequencies. This makes it more suitable for low-end edge devices.

Increasing the number of available devices to boost performance requires the consideration of two factors. First, the execution time is reduced as more devices contribute to the computation. However, this increases the communication time since more data is sent over the network. Second, the overhead introduced by overlapping partitions also increases since we need to use finer partitioning granularity. Overall, as Figures 5.10 shows, OFL



(a) VGG-16: speedup over LW



(b) YOLOv2: speedup over LW

Figure 5.10: Performance speedup of different parallelizations running VGG-16 and YOLOv2 at different frequencies. The speedup is compared with Layer-wise (LW) parallelizaiton at the corresponding frequency.

exhibits much better scalability with the number of devices compared to EFL, especially for VGG-16.

5.4.4 Analysis of Optimal Partitioning Strategies

We analyze the optimal layer fusion configurations under our cost model for VGG-16 and YOLOv2 and observe that OFL fuses more layers from the beginning CNN layers that have large input and output sizes as the number of devices increases. It uses the added compute power to offset the increased computation overhead and achieve large reduction in communication. For instance, we observe that the first 10 and 16 layers in both models are fused respectively. Second, deeper layers in a CNN model have smaller height/width dimensions but a larger channel dimension. This introduces higher computation overhead due to the overlapping of partitions with finer granularity. As a result, OFL fuses a smaller number of layers to balance the computation overhead and communication cost in deeper layers. In addition, large fused blocks are broken into smaller blocks to reduce the computation overhead when the frequency of the devices is decreased.

5.4.5 Robustness

We examine the robustness of our proposed Adaptive Optimal Fused-layer (AOFL) parallelization in terms of scalability, network conditions and heterogeneity.

Scalability. Figure 5.11 shows the scalability of Optimal Fused-layer (OFL) parallelization as a function of devices with different frequencies in WiFi-1. The speedup relative to a single device increases as more devices are added, but is limited by the high communication cost. For VGG-16, OFL runs $1.6\times \sim 3.7\times$ faster on 2 to 8 devices than a single device. The speedup with YOLOv2 is less and ranges from $1.3\times \sim 1.9\times$. This is because the computation of convolution layers in YOLOv2 are lightweight and the communication represents a larger fraction of the runtime. The speedup is also improved when the frequency drops and the ratio of computation increases. For instance, with 8 devices, the speedup increases from

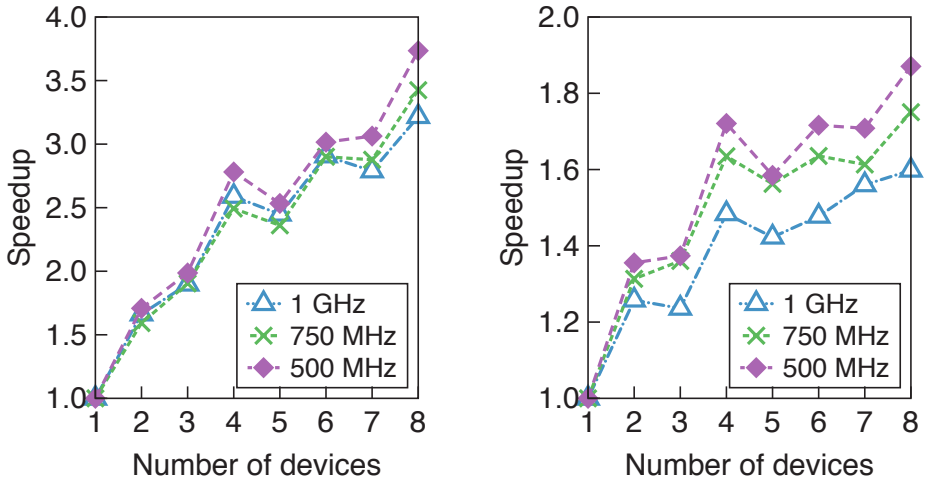


Figure 5.11: Speedup of Optimal Fused-layer (OFL) over a single edge device in WiFi-1 (93 Mbps).

3.2× to 3.7× and 1.6× to 1.9× in each model when doubling the frequency from 500 MHz to 1 GHz. This shows that OFL scales better with higher computation to communication ratio, and can obtain higher speedup with less powerful devices. It is especially suitable for computation-intensive models that operate in a slow network, as well as low-end edge devices.

Impact of the network. Figure 5.12 shows how our framework adapts to changes in the network conditions. LW is heavily impacted by changes in the bandwidth. The execution time increases by 59.8% and 86.4% on average when the network slows down from 93 Mbps to 25 Mbps (WiFi-1 to WiFi-3). On the other hand, EFL is not sensitive to the network conditions. An average slow down of 4.8% is measured when switching from the faster network to the slow one (WiFi-1 to WiFi-3). Since EFL only requires distributing and collecting inputs and outputs once, the change of the communication cost due to different

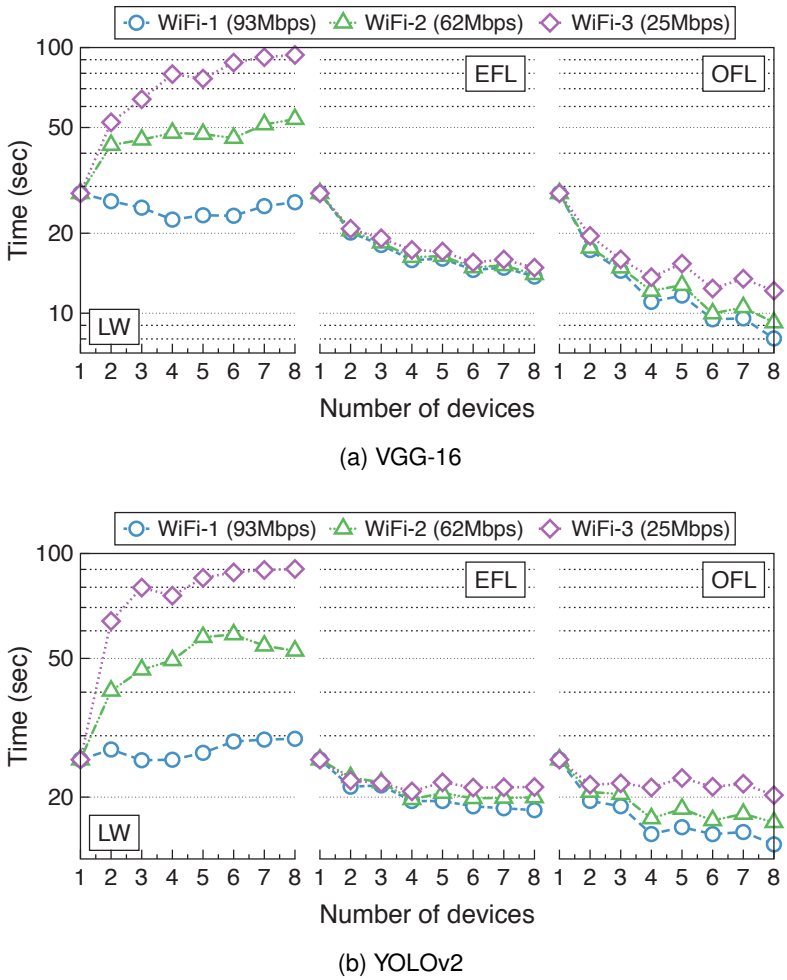


Figure 5.12: Execution time under different WiFi settings.

network conditions has a lesser effect on the total execution time. Finally, OFL experiences latency increases in 14.3% increments as the network is slowed down. However, it still performs better than EFL despite the higher communication cost. This also indicates that OFL takes better advantage of faster networks by distributing the computation of more layers in a model in order to improve performance.

Impact of heterogeneity. In these experiments, we focus on the heterogeneity in computational ability that can lead to inconsistent progress among the workers and slow down the speed of the inference. We use different device speeds available in the heterogeneous Cluster-2 in order to evaluate the robustness of our method in the presence of heterogeneity.

Table 5.2: Case study for heterogeneous IoT.

Cases	Devices in Custer 2: *#Devices* x *Frequency* x *#Cores (= 1)*
1-1	2 x 1 GHz, 6 x 400 MHz
1-2	1 x (1 GHz x 4), ...
2-1	2 x 1 GHz, 2 x 800 MHz, 2 x 600 MHz, 2 x 400 MHz
2-2	2 x 1 GHz, 4 x 800 MHz, 2 x 600 MHz

Table 5.2 lists the different heterogeneous configurations we consider in our study. The results of this study are summarized in Figure 5.13 and can be divided into two main categories.

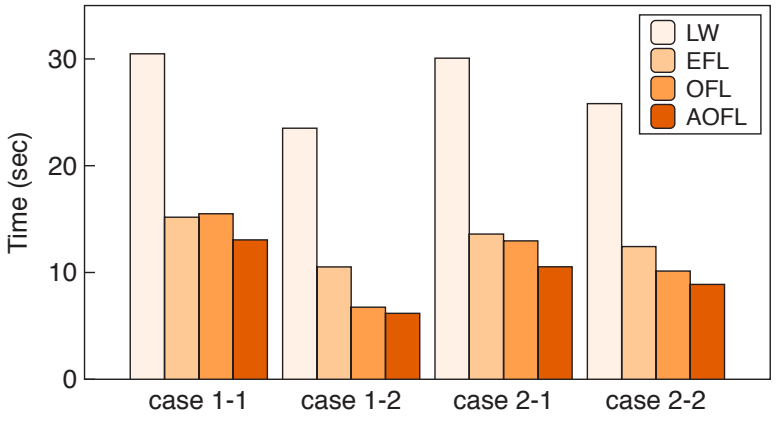


Figure 5.13: Execution time on different configurations of the heterogeneous Cluster-2.

Case 1 represents scenarios that prompt our solution to use a small number of fast devices to achieve better performance. This can be further divided into the sub-cases: 1-1) where the optimal performance of the model is parallelized with the two fastest devices while the rest of the devices are unused. For instance, with two 1 GHz devices, AOFL reduces the runtime by 15.8% compared to a configuration that uses all 8 devices. 1-2) represents a configuration that mimics local server approach where a much faster device is available. In this sub-case, we use a four-core device running at 1 GHz. We observe that the framework recognizes this and sends the entire input to such a device for execution.

Case 2 represents scenarios that prompt our solution to adapt to the availability of devices with different frequency distributions. This can be further divided into the sub-cases: 2-1) when the frequencies of devices span over a wide range, AOFL optimally assigns tasks across all the devices with the exception of the slowest two devices running at 400 MHz. This configuration results in an 18.7% reduction in runtime. 2-2) when the frequencies of devices are within a narrow range, the input partition sizes are adjusted to accommodate the frequency, so each device can spend a similar time in computation. This configuration results in all 8 devices being used leading to a 12.3% reduction in runtime.

5.4.6 Generalizing to More DNNs

Our framework is able to accommodate various CNNs with complex structures, including bypass connections and parallel layers.

Figure 5.14a shows two types of neural network blocks. Plain block stacks convolution layers and goes deeper to get better performance. Both VGG-16 and YOLOv2 have consecutive convolution layers, and achieve significant performance improvement using fused-layer parallelization. However, deep neural networks have the problem of vanishing gradients,

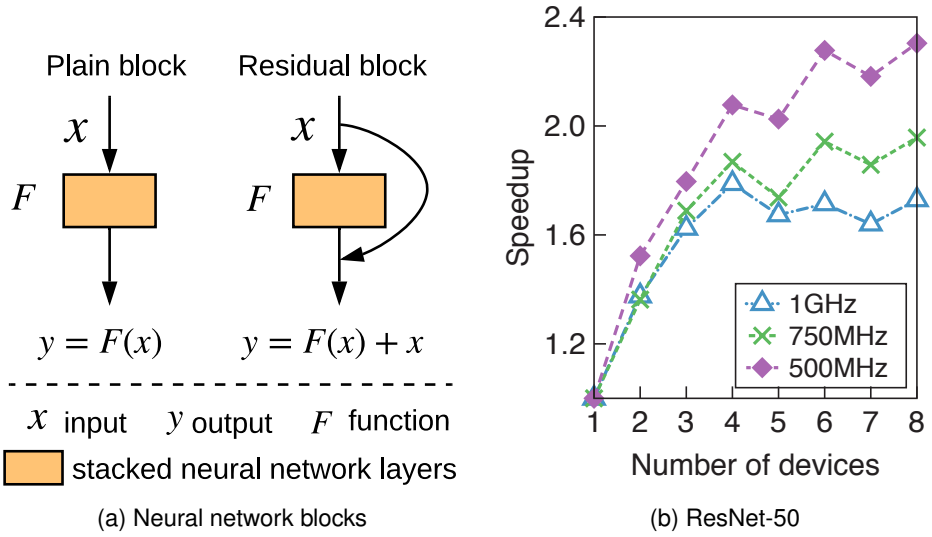


Figure 5.14: (a) Illustration of two different neural network blocks. (b) Speedup of Optimal Fused-layer (OFL) for ResNet-50 over a single edge device in WiFi-1 (93 Mbps).

which may stop the neural network from further training. To overcome the vanishing gradient issue, ResNet features special skip connections to reuse activations from previous layers until the adjacent layer learns its weights, enabling much deeper networks with good performance. The output tensor in the residual block is calculated with element-wise operations from the outputs of previous layers. In fused-layer parallelization, fusing layers in different residual blocks are not allowed unless the input of the current residual block and the output of the next residual block are within the same fused-layer block. This means fusing all the layers in two consecutive residual blocks. Then the bypass connections are duplicated and run on each partition, since there are no dependencies across different partitions. Figure 5.14b shows the performance of Optimal Fused-layer (OFL) parallelization for ResNet-50 as a function of devices with different frequencies in WiFi-1. In our experiments, one or more residual blocks are fused based on the search algorithm, and similar performance gain and scalability

are observed. OFL runs $1.7\times \sim 2.3\times$ faster on 2 to 8 devices than a single device. The results also indicate better speedup when the frequency drops and the ratio of computation increases.

Fused-layer parallelization may be limited by the available fusable convolution layers and model structures. For example, GoogLeNet [86] features inception module where filters with multiple sizes are used to operate on the same level. The outputs of all filter branches are concatenated depth-wise and sent to the next inception module. The network essentially would get a bit wider rather than deeper. A single filter branch may not benefit from fused-layer parallelization due to its number of fusable convolution layers. However, the structure of inception module itself implies a possible parallelism among the filter branches. DenseNet [45] features dense blocks where each layer is connected to every other layer in feedforward fashion. It alleviates vanishing gradients, strengthens features propagation, and encourage features reuse. In order to use fused-layer parallelization, it requires to fuse the entire dense block, and the performance depends on the depth of a dense block. We will explore the applicability to more DNNs in future work.

Model compression, including parameters pruning or quantization, may also benefit from the fused-layer parallelization when the system works under high computation to communication ratio. In the 5G era, much faster communication speed will be available, which can potentially change today's computation and communication ratio at the edge. The benefits of fused-layer parallelization would be further increased by the faster 5G network. However, when network speed is slow, it may be more suitable to run DNNs on a single device. Our algorithm can predict the best parallelization options based on the model characteristics, devices capacity and network condition.

5.5 Summary

In this work, we explore the trade-offs of distributing machine learning applications in IoT, and propose an Adaptive Optimal Fused-layer (AOFL) parallelization for improving the DNN inference at the edge. We design a dynamic programming based search algorithm to find the optimal partition and parallelization for a CNN model on a list of edge devices. We implement a CNN acceleration framework that adapts to the changes of computational resources and network conditions. Experimental evaluations show up to $1.9\times \sim 3.7\times$ speedup are achieved on 8 devices for three popular CNN models.

Chapter 6: Conclusion

Graph computing and machine learning have been widely used in a variety of domains. Unfortunately, these applications often show different behaviors, which place increasingly performance challenges for a range of modern computer systems especially when scaling to large scale datasets. This could lead to degraded system performance due to applications characteristics like the inefficient data access to the memory sub-systems, or hardware-specific restrictions like slow network in Internet-of-things (IoT) environments. This dissertation proposes two efficient graph processing systems respectively to explore the efficient execution of graph applications from a single machine to a distributed cluster. This is accomplished by using application-specific optimization techniques to improve the resources utilization and mitigate the performance bottlenecks. We also analyze the CPU and memory sub-systems' behaviors running graph applications to show the inefficiency of modern processors for graph computing, and provide insightful understanding of the impact of different computational models. Motivated by the emerging interests to deploy machine learning on IoT, this dissertation also presents an adaptive parallel execution framework leveraging a heterogeneous edge devices cluster. The proposed techniques in this dissertation are further summarized as follows.

- Chapter 3 presents an efficient large scale graph processing engine with several performance optimizations, including the edge-set graph representation with multi-modality,

data prefetch, and consolidation. We show that our edge-centric computation model dramatically improves the performance over the vertex-centric model by removing graph construction and optimizing the data parallelism. It provides compatible interfaces to utilize existing vertex programs, and shows up to 10× speedup on traversing a graph as an industrial system.

- Chapter 4 presents a concurrent graph processing framework called C-Graph. This system is designed to meet the industrial requirements of efficiently handling a group of simultaneous graph queries on large graphs. To achieve this goal, the proposed framework maintains global vertex states to facilitate graph traversals, and supports both synchronous and asynchronous communication interfaces. For any graph processing tasks that can be decomposed into a set of local traversals, such as the graph *k-hop* reachability query, our proposed system exhibited excellent performance up to 300+ concurrent queries.

- Chapter 5 explores the trade-offs of distributing machine learning applications in IoT, and proposes an Adaptive Optimal Fused-layer (AOFL) parallelization for improving the DNN inference at the edge. We design a dynamic programming based search algorithm to find the optimal partition and parallelization for a CNN model on a list of edge devices. We implement a CNN acceleration framework that adapts to the changes of computational resources and network conditions. Experimental evaluations show up to 1.9× $\sim$ 3.7× speedup are achieved on 8 devices for three popular CNN models.

The emergence of data science with increment of datasets scale and increasingly interests of Internet-of-things (IoT) require more efficient processing of large scale graph and machine learning. This dissertation provides applications-specific co-design solutions for improving

graph computing and machine learning efficiency. The proposed solutions are based on the applications behaviors and careful examination of computation and communication patterns. The solutions cover a range of computer systems from a single machine, a high-end servers cluster, and the resource-constrained edge devices.

Bibliography

[1] How response times impact business? https://calendar.perfplanet.com/2011/how-response-times-impact-business/, 2011.

[2] Yahoo. yahoo! altavista web page hyperlink connectivity graph, circa 2002. http://webscope.sandbox.yahoo.com/, 2012.

[3] Apache giraph. https://giraph.apache.org/, 2014.

[4] Titan distributed graph database. http://thinkaurelius.github.io/titan/, 2014.

[5] 8th tuc meeting – yinglong xia (huawei), big graph analytics engine. http://ldbcouncil.org/blog/8th-tuc-meeting-%E2%80%93-yinglong-xia-huawei-big-graph-analytics-engine, 2017.

[6] Janusgraph distributed graph database. https://github.com/JanusGraph/janusgraph, 2017.

[7] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 105–117. IEEE, 2015.

[8] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on*, pages 336–348. IEEE, 2015.

[9] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 1–13, 2016.

[10] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer cnn accelerators. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 22. IEEE Press, 2016.

[11] Amazon. Machine learning on AWS. https://aws.amazon.com/machine-learning/.

[12] Apple. Core ML. https://developer.apple.com/documentation/coreml.

[13] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations (ICLR)*, 2015.

[14] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 56–65. IEEE, 2015.

[15] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[16] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A quantitative study of irregular programs on gpus. In *Workload Characterization (IISWC), 2012 IEEE International Symposium on*, pages 141–151. IEEE, 2012.

[17] Meeyoung Cha, Hamed Haddadi, Fabricio Benevenuto, and Krishna P. Gummadi. Measuring User Influence in Twitter: The Million Follower Fallacy. In *Proceedings of the 4th International AAAI Conference on Weblogs and Social Media (ICWSM)*, Washington DC, USA, May 2010.

[18] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 269–284, 2014.

[19] Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379, 2016.

[20] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 609–622, 2014.

[21] James Cheng, Zechao Shang, Hong Cheng, Haixun Wang, and Jeffrey Xu Yu. K-reach: who is in your small world. *Proceedings of the VLDB Endowment*, 5(11):1292–1303, 2012.

[22] Jiefeng Cheng, Qin Liu, Zhenguo Li, Wei Fan, John CS Lui, and Cheng He. Venus: Vertex-centric streamlined graph computation on a single pc. In *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*, pages 1131–1142. IEEE, 2015.

[23] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. Nxgraph: an efficient graph processing system on a single machine. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 409–420. IEEE, 2016.

[24] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning (ICLR)*, pages 160–167, 2008.

[25] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[26] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 92–104, 2015.

[27] Ayush Dubey, Greg D Hill, Robert Escriva, and Emin Gün Sirer. Weaver: a high-performance, transactional graph database based on refinable timestamps. *Proceedings of the VLDB Endowment*, 9(11):852–863, 2016.

[28] Marat Dukhan. Nnpack. https://github.com/Maratyszcza/NNPACK, 2018.

[29] Karthi Duraisamy, Hao Lu, Partha Pratim Pande, and Ananth Kalyanaraman. High-performance and energy-efficient network-on-chip architectures for graph analytics. *ACM Trans. Embed. Comput. Syst.*, 15(4):66:1–66:26, 2016.

[30] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th ACM International Conference on Mobile Computing and Networking*, pages 115–127, 2018.

[31] Ioanna Filippidou and Yannis Kotidis. Online and on-demand partitioning of streaming graphs. In *Proceedings of the 2015 IEEE International Conference on Big Data (Big Data)*, pages 4–13, 2015.

[32] Raspberry Pi Foundation. Raspberry Pi. https://www.raspberrypi.org/.

[33] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI*, volume 14, pages 599–613, 2014.

[34] Google. Cloud machine learning engine. https://cloud.google.com/ml-engine/.

[35] Aditya Grover and Jure Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 855–864, 2016.

[36] Ramyad Hadidi, Jiashen Cao, Micheal S Ryoo, and Hyesoon Kim. Collaborative execution of deep neural networks on internet of things devices. *arXiv preprint arXiv:1901.02537*, 2019.

[37] Ramyad Hadidi, Jiashen Cao, Matthew Woodward, Michael S Ryoo, and Hyesoon Kim. Distributed perception by collaborative robots. *IEEE Robotics and Automation Letters*, 3(4):3709–3716, 2018.

[38] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[39] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2017.

[40] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[41] Wook-Shin Han, Sangyeon Lee, Kyungyeol Park, Jeong-Hoon Lee, Min-Soo Kim, Jinha Kim, and Hwanjo Yu. Turbograph: a fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 77–85. ACM, 2013.

[42] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[43] Ma hias Hauck, Marcus Paradies, and Holger Fröning. Can modern graph processing engines run concurrent eries e iciently? 2017.

[44] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[45] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[46] Intel. Movidius Neural Compute Stick. https://software.intel.com/en-us/movidius-ncs.

[47] Borislav Iordanov. Hypergraphdb: a generalized graph database. *Web-Age information management*, pages 25–36, 2010.

[48] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. IONN: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 401–411, 2018.

[49] Zhen Jia, Jianfeng Zhan, Lei Wang, Rui Han, Sally A McKee, Qiang Yang, Chunjie Luo, and Jingwei Li. Characterizing and subsetting big data workloads. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 191–201. IEEE, 2014.

[50] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. *arXiv preprint arXiv:1802.04924*, 2018.

[51] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 615–629. ACM, 2017.

[52] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[53] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[54] Jérôme Kunegis. Konect: the koblenz network collection. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1343–1350. ACM, 2013.

[55] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 31–46, 2012.

[56] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb):985–1042, 2010.

[57] Jure Leskovec and Rok Sosič. SNAP: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, 8(1):1:1–1:20, 2016.

[58] He Li, Kaoru Ota, and Mianxiong Dong. Learning IoT in edge: deep learning for the internet of things with edge computing. *IEEE Network*, 32(1):96–101, 2018.

[59] Robert LiKamWa, Yunhui Hou, Yuan Gao, Mia Polansky, and Lin Zhong. Redeye: Analog convnet image sensor architecture for continuous mobile vision. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 255–266, 2016.

[60] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88, 2017.

[61] Dao-Fu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Temam, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–381, 2015.

[62] Hang Liu, H Howie Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416. ACM, 2016.

[63] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. Graphlab: A new framework for parallel machine learning. *arXiv preprint arXiv:1006.4990*, 2010.

[64] Peter Macko, Daniel Margo, and Margo Seltzer. Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference*, page 18. ACM, 2013.

[65] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[66] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1396–1401. IEEE, 2017.

[67] Microsoft. Azure machine learning service. https://azure.microsoft.com/en-us/services/machine-learning-service/.

[68] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. Deep learning for IoT big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018.

[69] Lifeng Nai, Ramyad Hadidi, Jaewoong Sim, Hyojong Kim, Pranith Kumar, and Hyesoon Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *Proceedings of the 23rd International Symposium on High Performance Computer Architecture (HPCA)*, HPCA'17, 2017.

[70] Lifeng Nai, Yinglong Xia, Ilie G. Tanase, Hyesoon Kim, and Ching-Yung Lin. Graphbig: Understanding graph computing in the context of industrial solutions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'15, 2015.

[71] Nvidia. Jetson Nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/.

[72] Peitian Pan and Chao Li. Congra: Towards efficient processing of concurrent graph queries on shared-memory machines. In *2017 IEEE 35th International Conference on Computer Design (ICCD)*, pages 217–224. IEEE, 2017.

[73] Joseph Redmon. Darknet: Open source neural networks in c. http://pjreddie.com/darknet/, 2013–2016.

[74] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[75] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. *arXiv preprint*, 2017.

[76] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly Media, Incorporated, 2013.

[77] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[78] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[79] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R. Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 14–26, 2016.

[80] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *OSDI*, pages 317–332, 2016.

[81] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3):265–285, 1984.

[82] Julian Shun and Guy E Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[83] Yogesh Simmhan, Alok Kumbhare, Charith Wickramaarachchi, Soonil Nagarkar, Santosh Ravi, Cauligi Raghavendra, and Viktor Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[84] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[85] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11):1214–1225, 2015.

[86] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[87] Surat Teerapittayanon, Bradley McDanel, and HT Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 328–339, 2017.

[88] TensorFlow™. Tensorflow for mobile and IoT., 2019. https://www.tensorflow.org/lite.

[89] Manuel Then, Moritz Kaufmann, Fernando Chirigati, Tuan-Anh Hoang-Vu, Kien Pham, Alfons Kemper, Thomas Neumann, and Huy T Vo. The more the merrier: Efficient multi-source graph traversal. *Proceedings of the VLDB Endowment*, 8(4):449–460, 2014.

[90] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3):193–204, 2013.

[91] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *in Deep Learning and Unsupervised Feature Learning Workshop, NIPS*. Citeseer, 2011.

[92] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. Graphq: Graph query processing with abstraction refinement-scalable and programmable analytics over very large graphs on a single pc. In *USENIX Annual Technical Conference*, pages 387–401, 2015.

[93] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218. ACM, 2012.

[94] Huanhuan Wu, James Cheng, Yi Lu, Yiping Ke, Yuzhen Huang, Da Yan, and Hejun Wu. Core decomposition in large temporal graphs. In *IEEE Big Data*, pages 649–658, 2015.

[95] Yuduo Wu, Yangzihao Wang, Yuechao Pan, Carl Yang, and John D Owens. Performance characterization of high-level programming models for gpu graph analytics. In *Workload Characterization (IISWC), 2015 IEEE International Symposium on*, pages 66–75. IEEE, 2015.

[96] Yinglong Xia, Ilie G. Tanase, Lifeng Nai, Wei Tan, Yanbin G. Liu, Jason Crawford, and C-Y. Lin. Explore efficient data organization for large scale graph analytics and storage. In *IEEE Big Data*, pages 942 – 951, 2014.

[97] Qiumin Xu, Hyeran Jeon, and Murali Annavaram. Graph processing on gpus: Where are the bottlenecks? In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 140–149. IEEE, 2014.

[98] Zirui Xu, Zhuwei Qin, Fuxun Yu, Chenchen Liu, and Xiang Chen. Direct: Resource-aware dynamic model reconfiguration for convolutional neural network in mobile systems. In *Proceedings of the ACM International Symposium on Low Power Electronics and Design*, page 37, 2018.

[99] Jilong Xue, Zhi Yang, Shian Hou, and Yafei Dai. Processing concurrent graph analytics with decoupled computation model. *IEEE Transactions on Computers*, 66(5):876–890, 2017.

[100] Jilong Xue, Zhi Yang, Zhi Qu, Shian Hou, and Yafei Dai. Seraph: an efficient, low-cost system for concurrent graph processing. In *Proceedings of the 23rd international*

symposium on High-performance parallel and distributed computing, pages 227–238. ACM, 2014.

[101] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[102] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 548–560, 2017.

[103] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. I/O efficient ECC graph decomposition via graph reduction. In *PVLDB*, pages 516 – 527, 2016.

[104] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 87–99, 2014.

[105] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[106] Wen Zhang. Knowledge graph embedding with diversity of structures. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 747–753, 2017.

[107] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[108] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.

[109] Li Zhou, Ren Chen, Yinglong Xia, and Radu Teodorescu. C-graph: A highly efficient concurrent graph reachability query framework. In *Proceedings of the 47th International Conference on Parallel Processing*, page 79. ACM, 2018.

[110] Li Zhou, Hao Wen, Radu Teodorescu, and David H.C. Du. Distributing deep neural networks with containerized partitions at the edge. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, Renton, WA, 2019. USENIX Association.

[111] Li Zhou, Yinglong Xia, Hui Zang, Jian Xu, and Mingzhen Xia. An edge-set based large scale graph processing system. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 1664–1669. IEEE, 2016.

[112] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)(Savannah, GA*, 2016.

[113] Xiaowei Zhu, Wentao Han, and Wenguang Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *USENIX Annual Technical Conference*, pages 375–386, 2015.