Optimizing AI Workloads for the Edge: Balancing Performance, Energy, and Network Constraints

A Dissertation presented
by

Sri Pramodh Rachuri

to
The Graduate School
in Partial Fulfillment of the
Requirements
for the Degree of
Doctor of Philosophy
in
Computer Science
Stony Brook University

August 2026

Sri Pramodh Rachuri

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation.

Anshul Gandhi– Dissertation Advisor
Associate Professor – Computer Science

Shubham Jain - Chairperson of Defense
Associate Professor – Computer Science

Yifan Sun - Committee Member
Assistant Professor – Computer Science

Francesco Bronzino – External Committee Member
Associate Professor – ENS Lyon

This dissertation is accepted by the Graduate School

Celia Marshik
Dean of the Graduate School

Abstract of the Dissertation

Optimizing AI Workloads for the Edge: Balancing Performance, Energy, and Network Constraints

by

Sri Pramodh Rachuri

Doctor of Philosophy

in

Computer Science

Stony Brook University

2026

Usage of Artificial Intelligence (AI) is becoming ubiquitous in our daily lives, with applications such as image classification, natural language processing, generative AI, and many more. In most of these applications, inference is user-facing and latency-sensitive, making it crucial to deliver results quickly. As such, edge computing is a useful paradigm for inference as it brings computing closer to the source of input data, reducing network latency by reducing hops. Edge computing also allows for selective transmission and preprocessing of data, which can significantly reduce bandwidth utilization and improve privacy. Due to these benefits, AI inference and data preprocessing on edge devices is becoming increasingly popular compared to cloud-only solutions. However, edge devices are often resource-constrained and exhibit significant heterogeneity in compute capabilities, energy availability, and network connectivity. These challenges make it difficult to deploy and run AI inference and preprocessing applications efficiently. This thesis develops systems and algorithms to address the challenges of running AI inference and data preprocessing workloads on the edge, focusing on reducing latency and energy consumption while maintaining accuracy.

The frequent deployment of AI inference leads to significant energy consumption despite of each inference being less resource-intensive than training. This is important especially in remote or battery-powered edge environments, where energy availability is limited. To that end, this thesis begins by analyzing how hardware settings (CPU/GPU frequencies) affect energy use across various DNN inference workloads and edge devices. We find that default configurations are rarely optimal, and the best settings depend on workload and usage demands, making manual tuning impractical. To efficiently find such "best" settings, we design EcoEdgeInfer, a system that dynamically balances performance and energy efficiency for DNN inference on edge devices. At its core is EcoGD, an adaptive optimization algorithm that efficiently explores hardware and software configurations to minimize energy and latency under changing workloads. Evaluations show EcoEdgeInfer reduces energy use by 31% and tail latency by 14% compared to existing baselines.

While EcoEdgeInfer optimizes a single edge device, real-world deployments, such as video analytics, often involve distributed, heterogeneous edge clusters with varying network connectivity and compute capabilities. To address the challenges of variable network connectivity and compute capabilities on the edge, we developed OVIDA, a framework that orchestrates video analytics applications across distributed edge nodes, optimizing module placement and replication. OVIDA achieves 51% higher throughput and 28% higher accuracy compared to existing baselines.

Network connectivity can also become a bottleneck in edge data preprocessing and disaggregated storage infrastructures, where data is stored separately from compute resources to optimize costs and resource utilization. This separation is common due to the scale and geographic distribution of data and intermittent compute needs. Near Data Processing (NDP) addresses this by offloading certain operations to storage devices, reducing network load. However, it is not always clear which operations benefit most from this offloading. We develop an analytical model that selectively pushes down operations to minimize job execution time. Experiments on popular data processing systems, namely Spark and Hadoop, demonstrate that our approach can reduce execution time by up to 42%.

Beyond energy and network constraints, edge devices are highly heterogeneous in hardware capabilities, making it difficult to deploy a single AI model across all devices. Large models may not fit on smaller devices, while small models may underutilize more capable hardware or fail to meet latency requirements. Model developers typically release a few variants (e.g., Yolov5 small, medium, large), but these are not always optimal for every device. While EcoEdgeInfer and EcoGD optimize a given model on a specific device, they do not address selecting the best model variant for diverse hardware. OVIDA considers model switching, but only among the limited variants provided by developers.

Thus, as the last part of this thesis, we systematically investigate how model architecture parameters such as the number of layers, layer width, and the number of attention heads in transformer-based models influence performance, energy usage, and accuracy across various edge devices. We develop a benchmarking tool that evaluates these parameters' effects comprehensively with minimum overhead. Using the insights gained, we built a predictive model capable of estimating the performance, energy usage, and accuracy of model variants derived from a base model. Furthermore, we designed ModelRightSizing, a system that leverages this predictive model to automatically select the optimal model variant for specific edge devices and workloads. Evaluations show that ModelRightSizing can reduce energy consumption by up to 85.4%, improve latency by up to 85.2%, and occupy up to 68.5% less memory, all with only 1.4% accuracy loss compared to using the largest model variant.

It is our thesis that well-designed systems combined with algorithms that optimize resource usage in the presence of varying network and compute capabilities can facilitate the successful deployment of AI inference and AI data preprocessing on the edge, delivering real-time results.

Dedication

It is often said that we are complicated functions of our experiences and environment. Within the dataset of my life, you, Dad, are the most significant influence whose guidance and support shaped my trajectory while you were with us.

Although you are no longer with us now, my principal component will stay aligned with your values and teachings, and I will continue to honor your memory in all that I do.

This thesis is dedicated to you, Dad, and the enduring impact you have had on my life.

Acknowledgements

Firstly, I thank my family for their unconditional love and support. I also thank them for their unsaid sacrifices, which have enabled me to pursue my dreams. My parents have always been my pillars of strength, and I am grateful to them for their constant encouragement and support. I thank my brother for being a source of inspiration and motivation. His decisions in his life have always been a guiding light for me. My family is my forever best friends, and I owe them everything I am today.

I thank my advisor, Prof Anshul Gandhi, for being an ideal advisor, who was kind, patient, and encouraging throughout my PhD journey. He understood my strengths and weaknesses and helped me grow as a researcher. But what makes him an exceptional advisor is how he handled the out-of-textbook life situations that arose during my PhD. Year after year, he has also induced his contagious traits, such as curiosity, critical thinking, punctuality, patience, empathy, and humility, in me. I hope to carry them forward in my career.

I am thankful to my thesis committee members, Prof Shubham Jain, Prof Francesco Bronzino, and Prof Yifan Sun, for their invaluable advice and feedback on my projects.

I also thank my undergraduate advisor, Prof. Arzad Alam Kherani, who convinced me to apply to PhD programs when I asked for an LOR for a Master's program. One 'little chat' with him changed the course of my life, and I am grateful to him for that.

I spent my last six years in pursuit of this PhD degree, and I have been fortunate to have met many other wonderful people who have either directly or indirectly contributed to my journey. I thank my friends, roommates, labmates, and colleagues for their constant support and for lending a hand when I needed help. I also thank them for lending me their ears to my rants and frustrations. Your presence in my life has made this journey more enjoyable and memorable. I hope to continue to nurture these friendships in the years to come.

I am also grateful to all others who have contributed to my growth as a researcher and as a person, including the 20 Master's students I had the privilege to mentor. Though I cannot name everyone here, their collective influence has been invaluable.

Table of contents

1 Introduction **1**

1.1 Challenges of AI Inference on Edge Devices 2

1.2 Our Contributions 3

1.3 Thesis Organization 5

2 Background **6**

2.1 Edge Computing Devices for Inference 6

2.1.1 Hardware and Software Knobs for Energy-Performance Trade-offs . 7

2.1.2 Impact of Request Arrival Patterns on Inference 7

2.2 Distributed Video Analytics on Heterogeneous Edge Nodes 8

2.2.1 Diversity of DNN Models and Online Model Selection 8

2.3 Edge Data Processing and Near-Data Optimization 9

2.4 Model Parameter Selection for Edge Inference 9

3 Energy-Efficient Edge Inference **11**

3.1 Introduction 11

3.2 Background 14

3.2.1 Smart Edge Devices for Deep Learning 14

3.2.2 Hardware parameters on edge devices for managing energy 15

3.2.3 DNN Inference Parameters 15

3.2.4 Request Arrival Pattern 16

3.3 Related Work 17

3.3.1 Energy Efficient DNN Training 17

3.3.2 Energy Efficient DNN Inference on Edge 18

3.4 Preliminary Work 18

3.4.1 Experimental Design 19

3.4.2 Preliminary Results 21

3.4.3 Key Takeaways 28

3.5 System Design . . . 28
3.5.1 Core Inference System . . . 29
3.5.2 Metrics Collection . . . 29
3.5.3 Optimizer and optimization policies . . . 30
3.5.4 Predicting the optimal configuration via EcoGD . . . 30
3.5.5 Applying the optimal configuration . . . 32
3.6 Experimental Setup and Methodology . . . 33
3.6.1 Experimental setup . . . 33
3.6.2 Request arrival patterns and traces . . . 33
3.6.3 Comparison baselines . . . 35
3.6.4 Evaluation methodology . . . 36
3.6.5 Hyperparameters settings of different algorithms . . . 36
3.7 Evaluation Results . . . 37
3.7.1 Comparison of algorithms under fixed load . . . 38
3.7.2 Comparison of algorithms under bursty load . . . 42
3.7.3 Key takeaways from synthetic load patterns . . . 46
3.7.4 Performance evaluation under real-world traces . . . 46
3.8 Conclusion . . . 51

4 Network Optimization in Edge Clusters 52
4.1 Introduction . . . 53
4.2 Background and Motivation . . . 55
4.3 System Design and Implementation . . . 58
4.3.1 Microservice-based distributed pipeline . . . 58
4.3.2 Pseudo-central queue design . . . 60
4.4 Placement and Replication . . . 61
4.4.1 Bottleneck Analysis . . . 63
4.4.2 Iterative Approach . . . 63
4.5 Adaptive Model Selection . . . 66
4.6 Methodology and Experimental Setup . . . 66
4.6.1 Experimental Testbed . . . 66
4.6.2 VA Application Pipelines . . . 67
4.6.3 Baselines . . . 69
4.7 Evaluation . . . 70
4.7.1 Placement Evaluation . . . 70
4.7.2 Adaptive Model Selection . . . 72
4.7.3 End-to-end Evaluation of OVIDA . . . 73

4.8 Related work . . . 74
4.9 Conclusion . . . 76

5 Network Optimization in Disaggregated Edge Data Centers 77
5.1 Introduction . . . 78
5.2 Background . . . 81
5.2.1 Data processing in Spark . . . 81
5.2.2 Hadoop Distributed File System (HDFS) . . . 82
5.2.3 An example of a Spark job involving HDFS and NDP . . . 82
5.3 Related Work . . . 84
5.3.1 NDP using specialized hardware . . . 84
5.3.2 Policies for distributing tasks . . . 84
5.3.3 NDP implementations . . . 85
5.4 System Design . . . 86
5.4.1 Extension for Spark . . . 86
5.4.2 Extension for HDFS . . . 88
5.4.3 Analytical model to decide the best pushdown strategy . . . 89
5.5 Evaluation . . . 90
5.5.1 Simulation Results . . . 91
5.5.2 Empirical Results . . . 96
5.5.3 Experimental results . . . 96
5.6 Conclusion . . . 99

6 Model Right Sizing for Edge Devices 100
6.1 Introduction . . . 100
6.2 Related Work . . . 103
6.3 System Design . . . 104
6.3.1 Accuracy Estimator . . . 104
6.3.2 Hardware Performance Estimator . . . 105
6.3.3 EdgeFit Selector . . . 106
6.4 Experimental Methodology . . . 106
6.5 Evaluation . . . 109
6.5.1 Accuracy estimation . . . 109
6.5.2 Performance estimation . . . 110
6.5.3 End-to-end evaluation of model right-sizing . . . 112
6.6 Conclusion . . . 112

7 Conclusion **113**

Bibliography **115**

Chapter 1

Introduction

In the last few years, the usage of Artificial Intelligence (AI) has skyrocketed, with usecases ranging from image classification and natural language processing to generative AI and beyond [118]. Several Deep Neural Network (DNN) architectures have been developed to solve various AI tasks, such as convolutional neural networks (CNNs) for image classification, recurrent neural networks (RNNs) for sequence modeling, and transformers for natural language processing, etc [118]. While these applications have provided significant benefits to the society, they also come with challenges, especially when it comes to inference [95]. Inference is the process of using a trained model to make predictions on new data, and it is often user-facing and latency-sensitive. This means that delivering results quickly is crucial, as delays can lead to poor user experience and lost opportunities. For example, in real-time video analytics applications, such as surveillance or autonomous driving, delays in processing can lead to missed events or unsafe situations [53, 165]. Traditional cloud-based solutions, where data is sent to a remote server for processing, can introduce significant latency due to network hops and bandwidth limitations [165].

To address these challenges, edge computing has emerged as a promising paradigm in which computing resources are brought closer to the source of input data, such as cameras or sensors [54, 165]. By processing data at the edge, latency can be reduced, and bandwidth utilization can be optimized by selectively transmitting and preprocessing data [165, 68, 56]. This is particularly important for applications that require real-time processing, such as video analytics, augmented reality, and autonomous systems [53, 165]. Edge computing also improves privacy by keeping sensitive data closer to the source and reducing the need to transmit it over the network [68, 56]. As a result, AI inference and data preprocessing on edge devices is becoming increasingly popular compared to cloud-only solutions [54]. The amount of compute resources available on edge devices can vary widely, ranging from on-premise data centers to small, battery-powered devices such as IoT devices and embedded

systems. Sometimes, edge devices are even mobile, such as smartphones and drones [7, 54]. In our work, we focus on the *smart edge devices* category (as per Linux Foundation Edge taxonomy [54]), which are not as powerful as cloud servers but are more capable than traditional IoT devices. Sometimes edge devices are also equipped with specialized hardware such as GPUs, TPUs, or FPGAs to accelerate AI inference [7, 30]. For example, NVIDIA Jetson devices are popular commercially available devices that come with GPUs, while Google Coral devices come with TPUs. In rest of the thesis, we will refer to these devices as *edge devices*.

1.1 Challenges of AI Inference on Edge Devices

Edge devices are often resource-constrained and present several challenges for efficient AI inference and preprocessing [135, 85]. Key challenges include:

- **Limited compute capability:** Edge devices typically have slower processors and fewer hardware accelerators compared to cloud servers, making it difficult to run large or complex AI models or process high-throughput data streams quickly [30, 135]. This can result in missed deadlines, degraded application performance, and poor user experience.

- **Memory constraints:** The limited memory available on edge devices can prevent large models from being loaded or executed efficiently, sometimes causing out-of-memory errors or forcing the use of smaller, less accurate models [30, 135].

- **Heterogeneous network connectivity:** In edge clusters, non-uniform network connectivity can create communication bottlenecks, especially when applications are distributed across multiple nodes or require coordination with the cloud. This can lead to increased latency, or reduced throughput [83, 55, 73].

- **Energy limitations:** Remote or battery-powered edge devices may have restricted energy availability, lacking access to reliable power sources. This further constrains their operation, requiring careful management of energy consumption to prolong device lifetime and ensure continuous service [85, 63].

- **Hardware and software heterogeneity:** Edge environments often consist of a diverse set of devices with varying hardware architectures, operating systems, and software stacks. This heterogeneity complicates the deployment, optimization, and maintenance of AI models and applications [54, 163].

These challenges make it difficult to deploy and run AI inference and preprocessing applications efficiently on edge devices, necessitating new software systems and algorithms that can adapt to resource variability and optimize for performance, energy, and accuracy.

1.2 Our Contributions

This thesis develops systems and algorithms to address the challenges of running AI inference and data preprocessing workloads on the edge. Our contributions are summarized below:

- **Energy optimization for inference on a single edge device:** We consider the problem of optimizing energy consumption for DNN inference on a single edge device. We begin by analyzing how hardware settings (CPU/GPU frequencies) and software configurations (batch size) affect energy use across various DNN inference workloads and edge devices. We find that default configurations are rarely optimal, and the best settings depend on workload and usage demands, making manual tuning impractical. To efficiently find such "best" settings, we design EcoEdgeInfer, a system that dynamically balances performance and energy efficiency for DNN inference on edge devices. At its core is EcoGD, an adaptive optimization algorithm that efficiently explores hardware and software configurations to minimize energy and latency under changing workloads. Evaluations show EcoEdgeInfer reduces energy use by 31% and tail latency by 14% compared to existing baselines.

- **Optimizating Placement to avoid network bottlenecks:** To improve the overall performance and reliability of distributed edge applications, it is important to address potential network bottlenecks that can arise due to variable connectivity and resource constraints across edge nodes. While EcoEdgeInfer optimizes a single edge device, real-world deployments involve distributed, heterogeneous edge clusters with varying network connectivity and compute capabilities. To address these challenges, we develop OVIDA, a framework that divides video analytics applications into stateless functions using containerization and uses Kubernetes to orchestrate their execution across distributed edge nodes. OVIDA places the replicas of these functions on edge nodes such that the end-to-end performance is maximized. OVIDA achieves 51% higher throughput and 28% higher accuracy compared to existing baselines.

  Building on the need to address network bottlenecks highlighted in OVIDA, we further explore optimizations for mitigating network bottlenecks in data-intensive edge workloads through Near Data Processing (NDP). Specifically, we target edge data preprocessing and disaggregated storage infrastructures, where separating compute

and storage can exacerbate network limitations. We develop an analytical model that selectively pushes down operations in data preprocessing tasks to minimize job execution time, taking into account both the network capabilities between compute and storage nodes and the reduced compute power of storage nodes. Experiments on popular data processing systems, namely Spark and Hadoop, demonstrate that our approach can reduce execution time by up to 42%.

- **Model variant selection for heterogeneous edge devices:**

  While the above works focus on minimizing energy consumption, latency, and network bottlenecks, they do not address the challenge of hardware-software heterogeneity in edge devices. In current practice, model developers typically release a few variants of their models (e.g., Yolov5 small, medium, large) and edge administrators are forced to select one of these variants based on the device capabilities [76], usually by choosing the largest model that fits on the device or by trial-and-error methods. However, these variants are often limited and may not provide the best trade-off between accuracy, latency, and resource usage for every device or application scenario. While approaches such as pruning can reduce model size, we find that they often lead to significant accuracy loss, especially for smaller models.

  To address this challenge, we design the EdgeFit framework. It first efficiently explores the size-parameter space of ML models to estimate their accuracy, latency, and energy consumption on edge devices without actually deploying each configuration. Then, it leverages model-agnostic kernel-level profiling and tracks the influence of device-agnostic model size-parameters on its accuracy to quickly discover near-optimal model sizes. Our empirical evaluation on three models across three edge devices shows that EdgeFit reduces the cost of parameter exploration by up to 77% compared to an exhaustive search, successfully finding near-optimal model configurations that meet use-case requirements while achieving 36% higher accuracy compared to existing approaches, such as pruning.

  Using the techniques developed in this work, edge administrators and mobile device manufacturers can create custom-distilled models that are tailored to fit their devices to achieve the required performance, accuracy, and energy usage tradeoffs. These techniques are not limited to edge devices, and can also be applied to other AI inference use cases beyond edge computing.

1.3 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, we present the background on edge AI inference, data preprocessing, and energy optimization on edge devices. In Chapter 3, we describe our work on energy optimization for DNN inference on a single edge device, including the design and implementation of EcoEdgeInfer and EcoGD. This work was published under two papers: one in the 2023 International Green and Sustainable Computing Conference (IGSC) and the other in the 2024 Symposium on Edge Computing (SEC) [50, 123]. In Chapter 4, we present our work on optimizing placement and replication of video analytics applications across distributed edge nodes using OVIDA. This work was also published in the 2024 Symposium on Edge Computing (SEC) [134]. In Chapter 5, we discuss our work on optimizing Near Data Processing for edge data preprocessing and disaggregated storage infrastructures. This work was published in the 2022 International Conference on Distributed Computing Systems (ICDCS) [122]. In Chapter 6, we outline our work on model variant selection for heterogeneous edge devices using EdgeFit. Chapters 3, 4, 5, and 6 consist of their own prior work sections, which discuss the related work specific to each chapter. Finally, in Chapter 7, we conclude the thesis and discuss future work.

Chapter 2

Background

With the proliferation of data-intensive and latency-sensitive applications such as real-time video analytics, deep learning (DL) inference, and large-scale distributed data processing, edge computing has emerged as a critical architectural paradigm. Rather than offloading all computation to cloud data centers, edge computing enables processing to be performed closer to the data source—whether it's a camera, sensor, or user-facing mobile device. This chapter presents a consolidated overview of four major areas of relevance: energy-aware DNN inference on smart edge devices, video analytics pipelines on the edge, edge data pre-processing, and model parameter selection. We further explore the challenges and opportunities these paradigms present, especially under constraints of energy, latency, and heterogeneity.

2.1 Edge Computing Devices for Inference

Edge computing brings computation closer to where data is generated, helping reduce end-to-end latency, save bandwidth, and preserve privacy. A growing number of use cases—ranging from autonomous vehicles to augmented reality and industrial automation—now rely on local inference using Deep Neural Networks (DNNs). While cloud-based training of DNNs remains the norm, inference can often be performed on lightweight "smart edge devices" such as the NVIDIA Jetson series (e.g., Xavier NX, Orin Nano), Raspberry Pi with Edge TPUs, or similar platforms.

These devices offer integrated GPUs or tensor accelerators with support for DNN frameworks like PyTorch and TensorFlow, making them suitable for deploying AI workloads at the edge. However, they are severely constrained in terms of compute throughput, memory capacity, and energy budget. For example, Xavier NX comes with a 6-core ARM CPU, 384 CUDA cores, and only 8 GB LPDDR4x memory, operating within a strict 10-20W power

envelope. While powerful enough for moderate inference tasks, these devices require careful resource management to avoid violating latency Service Level Objectives (SLOs) or draining energy budgets.

2.1.1 Hardware and Software Knobs for Energy-Performance Trade-offs

To manage the trade-off between energy consumption and performance, edge devices expose several control knobs. At the hardware level, users can adjust CPU and GPU frequencies or select among predefined power modes (e.g., 10W, 15W, 20W). At the software level, tuning batch size is one of the few practical ways to increase inference throughput.

Studies have shown that increasing batch size enhances GPU utilization, thus improving energy efficiency. However, the benefits taper off as memory becomes a limiting factor, and batching introduces latency trade-offs: larger batches require more time to accumulate, which can hurt tail latency for user-facing requests. Moreover, DL frameworks like PyTorch support static loading (e.g., using TorchScript JIT tracing) for faster warm-up times, but such loading choices have marginal impact on long-term energy trends and are less important compared to batch size and power modes.

2.1.2 Impact of Request Arrival Patterns on Inference

The pattern of incoming inference requests greatly affects how an edge system should be configured. Beyond mere arrival rate, the temporal distribution—such as burstiness or periodicity—impacts queuing delay, batching effectiveness, and energy management.

A low request rate with uniform arrivals allows for aggressive energy-saving configurations (e.g., low GPU frequency). Conversely, a high or bursty request rate necessitates scaling up resources to maintain latency bounds. Importantly, batching is only effective when arrivals are frequent; otherwise, the system experiences long batch accumulation delays. For example, accumulating a batch of 10 requests under an inter-arrival time (IAT) of 100ms takes 1 second, significantly degrading latency. Adaptive systems must therefore tune resource and batching configurations dynamically, based on observed arrival patterns, to optimize for both energy and latency.

2.2 Distributed Video Analytics on Heterogeneous Edge Nodes

Video analytics (VA) represents a flagship edge application domain. VA tasks such as object detection, license plate recognition, pose estimation, and activity classification involve complex pipelines composed of multiple DNN modules. These modules may differ widely in computational requirements and data volume, leading to substantial challenges in deploying end-to-end VA pipelines across edge networks.

Disaggregating the VA pipeline across multiple edge devices allows for better load distribution and modular scaling. However, naive module placement—e.g., placing only one module per edge node—leads to underutilization of critical resources such as GPU. Disaggregated deployments also suffer from high I/O overhead due to frame transfers between pipeline stages. A key insight is that multiple bottleneck modules can be co-located on a single device and scheduled to interleave their use of the GPU. This improves utilization by overlapping compute and I/O, often yielding 2-3.6 $\times$ gains in throughput.

Another challenge lies in the heterogeneity of edge environments. Devices may differ in GPU strength, memory capacity, network connectivity, and power constraints. Hence, intelligent placement, replication, and load-balancing strategies are essential to ensure end-to-end pipeline performance and maximize inference accuracy.

2.2.1 Diversity of DNN Models and Online Model Selection

For any given VA task (e.g., object detection), multiple model variants exist—such as YOLO nano, small, or medium—each offering different accuracy and compute requirements. A lighter model (nano) may perform faster but with lower accuracy, while a heavier model (medium) may be accurate but slow. The choice of model must therefore adapt to the system load: under low traffic, higher-accuracy models are preferable; under high load, simpler models may yield higher accuracy due to reduced frame drops.

However, most model selection frameworks require all candidate models to be preloaded in memory for runtime evaluation. This is infeasible on resource-limited embedded devices. A memory- and compute-efficient approach to input-aware model switching remains an open problem, particularly under varying workloads and limited runtime observability.

2.3 Edge Data Processing and Near-Data Optimization

Beyond real-time inference, large-scale data analytics workloads are increasingly being pushed to edge and hybrid environments using frameworks like Apache Spark. Spark jobs execute in-memory operations over datasets partitioned into blocks and distributed across worker nodes (executors). These jobs typically read data from storage systems like HDFS, where data is split into chunks and stored across multiple datanodes.

The traditional execution model involves transferring full data blocks from storage to computation nodes, where operations like filtering or aggregation are applied. This incurs significant I/O overhead, particularly when only a subset of the data is relevant to the query. Near-Data Processing (NDP) introduces the concept of pushing down simple operations—e.g., filters—into the storage layer, so that data is reduced at the source before being transferred. This technique offers substantial performance and energy savings. For example, if a filter operation reduces a 100 GB dataset to 10 GB, performing the filter at the datanode cuts transfer cost by 90% and speeds up downstream computation.

HDFS supports interfaces such as WebHDFS for client access using HTTP APIs, allowing systems like Spark to exploit storage-level extensions transparently. Integrating NDP into Spark pipelines offers a promising path forward for improving data locality, minimizing latency, and reducing energy consumption in hybrid edge-cloud systems.

2.4 Model Parameter Selection for Edge Inference

With the increasing diversity of edge devices capable of running DNN inference, ranging from TPU/NPU accelerated laptops to low-power ARM-based boards, selecting the right model parameters for a given device is critical for achieving optimal accuracy while meeting latency and energy constraints. Model parameters include architecture choices such as depth, width, and number of attention heads. Each parameter choice affects the model's inference latency, memory usage, and accuracy.

Model makers often provide pre-trained models with different configurations, but it is not possible to exhaustively evaluate all combinations of parameters on every edge device. Instead, a systematic approach is needed to select the best model configuration based on device capabilities, workload characteristics, and performance objectives. Further, techniques such as knowledge distillation and fine-tuning has democratized model training for specific use cases, already allowing developers to create custom models that are optimized for their specific edge deployment scenarios. However, the challenge of finding the right balance

between model complexity, accuracy, and resource consumption remains an open research problem.

Solving this problem requires understanding the trade-offs between accuracy/performance and resource usage (e.g., latency, energy) for different model configurations. Learning the accuracy as a function of model parameters for a given model family is very expensive if done naively, as it requires repeatedly training and evaluating models with different parameter settings. Hence, there is a need for efficient methods to explore the parameter space without exhaustive search to be able to predict the accuracy, latency, and energy consumption of different model configurations on a given edge device. This can further be used to select the best model configuration for a given edge device and workload under constraints of latency, energy, and accuracy.

Chapter 3

Energy-Efficient Edge Inference

This chapter examines the challenges and opportunities of deploying Deep Neural Network (DNN) inference workloads on edge devices. It presents a systematic study of how workload parameters (such as batch size, layers, and quantization) and hardware knobs (including CPU and GPU frequency) affect energy consumption and accuracy across various DNN models and edge platforms. The findings reveal that default configurations are often suboptimal for energy efficiency, and that targeted tuning can achieve substantial energy savings without compromising accuracy.

Building on these insights, the chapter then introduces EcoEdgeInfer, a system designed to balance performance and sustainability for DNN inference at the edge. We describe the adaptive optimization algorithm, EcoGD, which efficiently explores the configuration space to minimize energy consumption and latency, and demonstrate its effectiveness through evaluation on real-world workloads.

In this chapter, Sections 3.1 to 3.3 provide the relevant motivation, background, and related work for this chapter. Section 3.4 presents the initial study on hardware and workload parameters, while Sections 3.5 to 3.7 detail the design and evaluation of the EcoEdgeInfer system. Finally, Section 3.8 concludes the chapter with a summary of findings and future work directions.

3.1 Introduction

The adoption of Deep Neural Networks (DNNs) has skyrocketed in recent months and years with applications in image recognition, speech recognition, and natural language processing [118]. While these applications have brought many benefits to society in general, the underlying compute requirements of executing the DNNs have had a severe environmental impact, leading to *significantly high energy consumption and carbon emissions* [147]. This

negative impact of training large and expensive DNNs has been highlighted in recent works, and has already resulted in research efforts to regulate the monetary and environmental cost of DNN training, for example, by leveraging hardware techniques [155, 61].

However, the sustainability costs of ***DNN inference*** have not received as much attention, yet. This is likely because DNN inference has much lower computational needs (usually completes in seconds or minutes, compared to hours or days for training). Unlike training, inference is usually a one-shot operation, where the input data is fed to the DNN and the output is generated without any further adjustments to the model or need for complex backpropagation operations; this makes inference a lightweight and short-lived task. Unfortunately, with the increasing popularity of online tools like ChatGPT [110] and GitHub Copilot [57], the usage of inference has significantly grown, and will likely continue to grow. Consequently, even though it has lower computational demand, inference is executed much more frequently than training [95], and can thus have a substantial environmental impact. *Addressing the high energy costs of the quickly scaling inference applications is thus a crucial and timely problem.*

DNN inference, unlike training, is ***user-facing***, and thus requires low (mean and tail) latency. The inference latency experienced by an end-user can be broadly categorized into network latency (to and from the cloud service) and computation latency. By *deploying DNN inference on edge devices*, the network latency can be significantly reduced as the input and output data does not have to be transferred to/from the cloud. This is especially useful for real-time and privacy-conscious applications like autonomous vehicles, augmented reality, and smart cities. Depending on the application, edge devices may be deployed in remote locations with limited access to power sources, and some might even be solar- or battery-powered [63]. As such, minimizing the energy consumption of DNN inference is important in the edge computing setting.

Addressing the ***energy costs of DNN inference on the edge*** is a challenging problem for several reasons:

(1) The energy consumption of inference on edge depends on both ***hardware and software configurations***. While there are tuning knobs that can influence energy consumption, such as GPU operating frequency and batch size, their impact on energy may be interrelated, and thus they cannot be optimized independently.

(2) There is an ***inherent tension between reducing energy consumption and improving inference latency***, resulting in a non-trivial optimization to balance the energy-latency tradeoff. For example, setting the GPU frequency to the maximum allowable value can reduce inference latency but at the expense of high energy.

(3) Although the effect of batch size on energy has been studied, it is ***not always clear how batch size affects inference latency***, especially under dynamic workload.

(4) Inference applications are user-driven, and thus, the ***request load can change dynamically and unpredictably***, requiring an agile solution that can adapt to changing conditions quickly and with low overhead to prevent high inference tail latencies.

There have been several recent works that focused on reducing the energy consumption of DNN training. However, these solutions are not directly applicable to DNN inference as the requirements (e.g., memory, compute capacity, and tail latency) and usage patterns (e.g., request rate and burstiness) for training and inference are different. Moreover, most of the prior works on DNN inference have focused on server and cloud environments [154], and have not considered the unique challenges of running inference on edge devices, such as limited power/energy and low-overhead requirements. Finally, while there have been works that analyze the impact of hardware and software parameters on energy consumption of DNN execution [135, 50], they are offline studies that neither consider dynamic workload conditions nor focus on designing an adaptive solution for practical deployments. As such, there is a *gap in research on adaptive solutions for energy-efficient DNN inference on the edge*. We discuss related works in detail in Section 3.3.

Our contributions: In this work, we first present we first present a preliminary systematic study of workload and hardware parameters that affect the energy consumption of DNN inference on edge devices. Then, we introduce*EcoEdgeInfer*, a *framework for DNN inference serving on edge devices that regulates their energy consumption and latency at runtime*. EcoEdgeInfer works by dynamically and transparently tuning hardware and software parameters based on recommendations from optimization algorithms. We have *implemented EcoEdgeInfer on NVIDIA Jetson devices* using Python, ensuring seamless integration with existing DNN inference libraries like PyTorch. The code for EcoEdgeInfer is publicly available as a GitHub repository to promote further research and facilitate reproducibility.[1]

Our key contribution is an *optimization algorithm, EcoGD, specifically designed for sustainable inference on edge deployments*. EcoGD is inspired by Gradient Descent, and enhances its functionality via real-time adaptation to changing workloads, estimating the cost of unexplored configurations and gradients, and maintaining low overhead. To prevent fluctuations in performance and maintain low tail inference latency, EcoGD limits its exploration space; to react to evolving workload conditions quickly, EcoGD maintains a limited memory of explored configurations.

[1]https://github.com/PACELab/EcoEdgeInfer

We *experimentally evaluate EcoGD against existing solutions with different DNN inference models*. Our results show that EcoGD converges to superior configurations under fixed and bursty synthetic load patterns, reducing energy consumption by 22% and lowering mean inference latency by 13%, on average, compared to existing solutions. Further, EcoGD's achieved energy and latency values are within 2% and 10%, respectively, of those achieved by an (impractical) offline optimal solution. Importantly, EcoGD effectively adapts to changing load conditions, consistently outperforming other solutions by lowering energy consumption by as much as 66% (with 31% average reduction) and reducing tail latency by as much as 91% (with 14% average reduction) under real-world workload traces. To the best of our knowledge, *ours is the first work that jointly optimizes energy consumption and latency of DNN inference on edge devices at runtime while adapting to changing workload conditions*.

3.2 Background

In this section, we provide the necessary background on the problem of energy-efficient DNN inference on edge devices. We first discuss the types of edge devices we consider in this work and the control knobs they offer to regulate energy consumption. We then discuss workload parameters that can also be adjusted to regulate energy consumption. Finally, we discuss the significance of the request arrival pattern on our problem and how it can impact energy consumption.

3.2.1 Smart Edge Devices for Deep Learning

Edge computing is the practice of processing data near the source of data generation rather than relying on centralized data centers. While a range of devices, from IoT sensors to on-premise workstations, can be considered edge devices, we focus on edge devices with limited computational resources, such as single-board computers, called *smart edge devices* [54]. These devices are typically equipped with low-power processors, small amounts of memory, and are energy-limited.

With the launch of NVIDIA's Jetson series [6], running Deep Learning (DL) models on edge devices has become feasible due to the integrated GPUs with CUDA support. For example, NVIDIA Xavier NX has 6 ARM cores, 384 CUDA cores, 48 Tensor cores, and 8GB of LPDDR4x memory; it consumes at most 20W of power. Table 3.1 provides the specifications of two popular smart edge devices, the Jetson Nano and Xavier NX, which we use in our preliminary study and evaluation. However, the limited computational resources on these devices also make it challenging to run large models in an efficient manner.

Table 3.1: Specifications for Jetson Nano and Xavier NX [104, 105]

Specification	Jetson Nano	Jetson Xavier NX
CPU	4-core ARM A57	6-core Nvidia Carmel
CPU Freq. range	102 MHz – 1.48 GHz	115 MHz – 1.9 GHz
CPU Freq. step	100 MHz (15 steps)	77 MHz (25 steps)
GPU	Nvidia Maxwell	NVIDIA Volta
CUDA Cores	128	384
Tensor Cores	-	48
Memory	4 GB LPDDR4	8 GB LPDDR4
GPU Freq. range	76 MHz – 921 MHz	114 MHz – 1.1 GHz
GPU Freq. steps	77 MHz (count 12)	90 MHz (count 15)
Throughput	472 GFLOPs	21 TOPs
Power Modes	5W, 10W	10W, 15W
Jetpack	4.6.3 [L4T 32.7.3]	4.5.1 [L4T 32.5.2]
Framework	PyTorch 1.10.0, OpenCV 4.6.0	PyTorch 1.10.0, OpenCV 4.6.0
Libraries	CUDA 10.2 + cuDNN 8.2.1	CUDA 10.2 + cuDNN 8.0.0
Operating Sys.	Ubuntu 18.04	Ubuntu 18.04

3.2.2 Hardware parameters on edge devices for managing energy

To regulate energy consumption, NVIDIA provides three power modes (10, 15, 20 Watts) as default presets to change hardware parameters of the Xavier NX device. However, we can also manually change hardware parameters for greater and finer control. While running DL inference, we noticed a significant impact on energy consumption by varying the GPU frequency, a moderate impact by varying the CPU frequency, and almost no impact when varying the number of CPU cores. This is because the GPU is the primary resource used by DL inference. Further, Python being single-threaded, the CPU is not used for parallelism; hence, the number of cores does not significantly impact energy or performance. So, we only focus on GPU and CPU frequencies in our experiments.

3.2.3 DNN Inference Parameters

In addition to hardware parameters, there are also software or workload parameters that can be adjusted to regulate energy consumption. When a user submits an inference request, the DNN model and their weights are fixed by the user. However, the inference system or framework can still adjust knobs like the batch size and how the model's task graph is loaded (statically or dynamically). Static loading using Pytorch's JIT Trace does not have

a significant impact on energy consumption in the long run when compared to dynamic loading because it only shifts the model initialization and loading [5]. So, we ignore the model loading mechanism and focus on the batch size as a potential tunable parameter.

Da *et al.* [89] showed that increasing the batch size always leads to increased throughput (not necessarily latency) due to higher parallelism and utilization of the GPU. They also showed that increasing the batch size can lead to energy savings, but these savings taper off after a certain point. The maximum batch size a model can handle is limited by the device's memory. Given the limited memory on edge devices, the batch size is kept low to prevent out-of-memory errors.

3.2.4 Request Arrival Pattern

Another factor that impacts DNN inference performance, especially on the edge, is the arrival pattern of requests. This is also a factor that has ***not been thoroughly investigated by prior works*** that focus on regulating DNN inference energy. Note that the arrival pattern of requests does not just refer to the arrival rate, but can also include the distribution of requests and factors such as burstiness of arrivals.

Consider an inference serving system. When the rate of inference requests coming in is low, the system can adjust its parameters (such as CPU and GPU frequency) to strike a balance between inference latency and energy consumption. However, when the request rate is high, the system may have to tune its parameters to maximize performance to handle the high load. In fact, for resource-limited edge devices, it is possible that a long queue starts building up quickly at high request rate as the serving capability of the device may be limited, thereby constraining the system to exclusively pick the highest performance configurations. As such, the optimal system configuration and the potential for tuning the configuration parameters are affected by the request rate.

The request arrival pattern also indirectly affects the system performance through the batch size. While batch size can help increase the throughput and energy efficiency [89], it also leads to increased tail latency due to the time taken to accumulate the batch before processing [23]. For example, if we have a constant stream of requests with an inter-arrival time (IAT) of 10ms, it will take $\sim$100ms to accumulate a batch size of 10. But, if the IAT is 100ms, it will take about 1s to get to the same batch size. So, a lower batch size may be preferable for a high IAT workload to avoid large accumulation delays and long tail latencies. However, if the arrival pattern is bursty, the above conclusions can change; a burst of 10 (or more) uninterrupted requests can immediately lead to an accumulation of the required batch size, with additional requests now waiting to be served. Worse, the request arrival pattern

can change unpredictably, especially for user-facing workloads [46], necessitating an agile solution that adapts to arrival pattern changes at runtime.

Finally, the content of requests can also impact inference latency and energy. For example, images with different blurring and brightness may have different processing times and energy consumption for image recognition. For this work, we do not consider such content-based workload changes, and limit our focus to changes in arrival rate and request distribution.

3.3 Related Work

Optimizing the energy or carbon usage of Deep Learning workloads has been an important topic of research in the past few years. However, most of the work has focused on model training on server-grade GPUs and machines. There are very few works that focus on inference workloads on edge devices; we discuss these at the end of this section.

3.3.1 Energy Efficient DNN Training

Google's CICS [124] focuses on minimizing the carbon footprint of data centers by temporally delaying flexible workloads to greener times. Zeus [155] optimizes the energy efficiency of DNN training by finding optimal job and GPU-level configurations. PowerFlow [61] is a GPU cluster scheduler that reduces the average job completion time under an energy budget. Cloud providers have also started offering Machine-Learning-as-a-Service (MLaaS) platforms that run on heterogeneous GPU clusters and automatically optimize the hardware, software, cluster configuration, and hyperparameters for training DNNs to reduce the energy consumption and improve the performance of the training jobs [149].

Works like PowerFlow [61] and Zeus [155] change the frequency of server-grade GPUs using power-limit APIs and thereby reduce energy consumption. Since we are focusing on a single device optimization, Zeus is the closest work to ours. Zeus uses Multi-Armed Bandit (MAB) to find the optimal batch size and performs exhaustive search to find the optimal GPU power limit for server-grade GPUs. Zeus can afford to run an exhaustive search on GPU power limit settings since it does not consider a change in workload (i.e., only considers a static training setting). However, in our inference case, due to the varying workload patterns, an exhaustive search is not feasible, as we show in Section 3.7. The MAB approach is feasible, and we compare against MAB (as a baseline) in our evaluation in Section 3.7.

Prashanthi et al. [135] is one of the first works focusing on changing CPU and GPU frequencies for DNN workloads on edge devices. The authors study the impact of power modes (presets of frequency limits by NVIDIA) on CPU and GPU utilization, training time,

and energy usage. However, the work focuses on DNN training. Instead, our work here focuses on DNN inference, which is better suited for resource-constrained edge devices than (computationally heavy) DNN training.

The observations made for training workloads do not readily apply to inference workloads. For example, Zeus [155] does not consider their system to have a queue of jobs that need to be scheduled because training workloads are usually long running and are not sent by users in a queue. Inference workloads, on the other hand, take a lot less time to complete but are more frequent and arrive dynamically. Hence, we must consider the real-time queueing effects on latency when optimizing for user-facing inference workloads.

3.3.2 Energy Efficient DNN Inference on Edge

JEDI [74] focuses on edge devices and considers inference workloads. The authors create a TensorRT-based pipelining framework to increase resource utilization and performance. They consider parameters like multi-threading, multi-device (CPU, GPU, DLA) processing, and buffer assignment. While JEDI results in lower energy consumption, this is not the primary objective and is a by-product of their optimization. As such, they did not attempt to tune energy-saving knobs, like CPU and GPU frequencies, which we believe are powerful and lightweight and lightweight tools to optimize energy consumption.

Dutt *et al.* [50] focus on the inference side of DNN workloads and profile the effects of processor frequency tuning on energy consumption. However, they do not consider workload parameters and do not consider latency or performance as an objective. Further, only a static offline workload is considered for evaluation. By contrast, we specifically focus on the tradeoff between energy consumption and inference latency, and consider dynamic changes in inference workload.

Based on our literature review above, we believe there are very ***few prior works on jointly optimizing the energy consumption and latency of DNN inference workloads on edge devices by tuning both hardware and software parameters***. Further, we are unaware of existing research that considers the ***impact of request arrival pattern on energy consumption and inference latency for edge devices***.

3.4 Preliminary Work

In this preliminary work, we study the impact of workload parameters and hardware knobs on the energy consumption of DNN *inference*, specifically for *edge* devices. We consider two edge devices—an entry-level Jetson Nano and a mid-tier Jetson Xavier NX—and

five different DNN models for various inference tasks ranging from objection detection to language modeling. For workload parameters, we investigate different batch sizes, different number of layers and quantization precision, and other factors such as lazy loading and static graph execution; since some of these parameters often impact inference accuracy as well, we also investigate their energy-accuracy tradeoff. For hardware-related knobs, we explore different CPU and GPU frequency settings.

Our experimental results show that we can reduce inference energy by as much as 19% compared to DVFS by optimally tuning the CPU and GPU frequencies. We also find that batch size can impact energy consumption significantly (to the tune of 40–45%), even within the limited range of batch sizes supported by resource-scarce edge devices. Further, the above factors are independent and can be thus be *combined* for even greater energy savings.

When considering accuracy, we find that a lower quantization can save 26%–65% energy while negligibly impacting accuracy. We also find that by *jointly* tuning multiple workload parameters (*e.g.*, batch size and quantization), we can achieve *both* energy savings (as high as 42%) and higher accuracy.

3.4.1 Experimental Design

Experimental Setup

We used Jetson Nano and Jetson Xavier NX for our experiments; the device specifications are shown in Table 3.1. Both devices provide multiple onboard sensors measuring power for different components. Each device has a separate sensor for full module power, which we used for reporting our results, unless otherwise noted. We manually polled and logged these readings via `sysfs` every 100ms; the energy overhead of 100ms polling was less than 0.5%. We found that more frequent polling (*e.g.*, 10ms or 1ms) resulted in high energy overhead of more than 2%.

DNN Workloads

We chose diverse DNN workloads that have practical inference applications in edge devices [113, 166]. While benchmarking suites like MLPerf [103] do exist, we assembled our own benchmarking framework as we needed finer control over workload parameters and inputs for evaluation.

For image classification, we used AlexNet [84], ResNet-18 [65], and MobileNet-v2; AlexNet and ResNet represent milestones in CNN model development, and Mobilenet-v2 is a prominent DNN model specially targeted for resource-constrained devices [130]. For natural language classification, we used DistilBERT [131] and BERT-Tiny [144]; DistilBERT is

Table 3.2: DNN workload specifications

Model	Layers	Params	Ops (GFLOPs)	Batch Size	Input
AlexNet	8	61M	0.727	4, 8, 16, 32, 64	Tensor (3,224,224)
ResNet-18	18	11M	2	4, 8, 16, 32, 64	Tensor (3,224,224)
MobileNet-V2	53	3.4M	0.57	4, 8, 12	Tensor (3,224,224)
BERT-Tiny	4	4.4M	0.0353	4, 8, 16, 32, 64	String (512 words, 1.1kb)
DistilBERT	6	43.2M	4.3	4, 8, 16	String (512 words, 1.1kb)

a popular light-weight Transformer-based smaller version of BERT base and BERT-Tiny is a knowledge-distilled version of BERT base which is significantly smaller. Table 3.2 lists the specifications for our models; all models were implemented in Python. The inputs were carefully selected so that input size stays about the same across different models for better comparison and parity. In our evaluation, we primarily varied the batch size (shown in Table 3.2); however, for experiments studying the impact on accuracy, we also varied the number of layers and quantization. We select batch size values in increments of powers of 2, till the device runs out of memory.

Hardware Knobs

Both devices have three main hardware knobs that can be changed during runtime—CPU clock frequency, GPU clock frequency, and the number of cores. The number of cores did not impact the power consumption in our experiments as the DNN models are implemented in Python, which is effectively single-threaded due to the Global Interpreter Lock (GIL) mechanism employed. We thus focused on CPU and GPU frequencies as the hardware knobs.

Both devices offer a large range of CPU and GPU frequencies, as noted in Table 3.1. Jetson Nano has 180 possible GPU and CPU frequency combinations (15 CPU freq $\times$ 12 GPU freq) and Xavier NX has 375 possible configurations (25 CPU freq $\times$ 15 GPU freq). We set static CPU and GPU frequencies via the `sysfs config files`. The devices also offer a default dynamic frequency module, DVFS, which we compare against for energy savings. DVFS allows the system to dynamically adjust the voltage and frequency levels of the CPU and GPU, during runtime, based on workload demands. The devices also provide a handful of Power Modes, representing a much smaller subset of the frequency knobs; we chose to instead explore the full frequency range.

Experimental Methodology

We executed the DNN inference workload and logged the execution checkpoints, CUDA events (*e.g.*, model loading and execution), and the requisite timestamps. In a separate thread in the background, we logged power readings using a Python process that reads the sysfs power entries. Merging these logs and correlating with inference events (*e.g.*, model initialization, inference start and end) allowed us to compute the power consumption, execution time, and energy for specific intervals of interest, such as model loading and model inference. During testing, we disconnected all peripherals from the device and killed unnecessary background processes, including the GUI.

Each experiment consisted of running a DNN inference workload with a specified batch size under a specific CPU and GPU frequency setting. The workload in every experiment was kept constant at 3,200 inferences. Each experiment was repeated 10 times and average values were reported; the variation between different runs was low (much less than 5%).

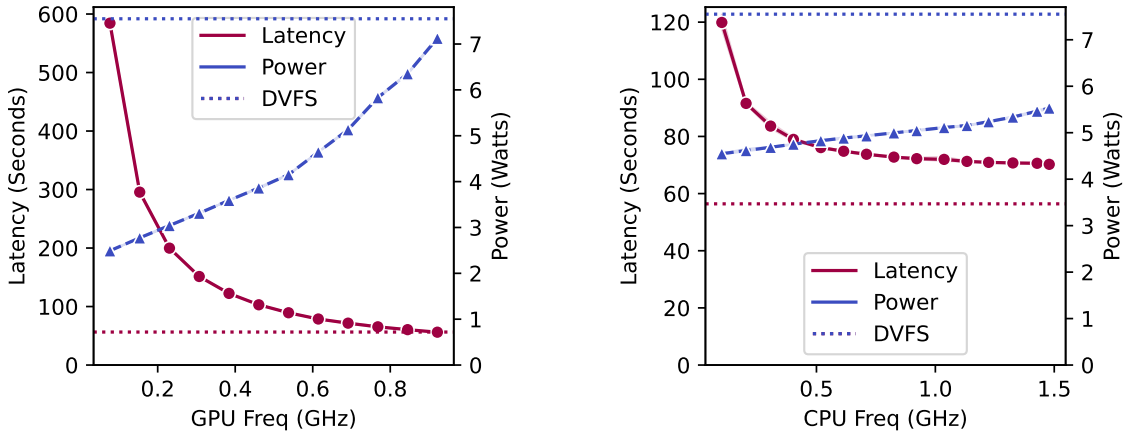
3.4.2 Preliminary Results

This section presents our experimental results on the impact of workload and device parameters/knobs on DNN inference energy consumption. We also analyzed the energy consumption vs. inference accuracy tradeoff for different model parameters. Finally, we present some observations on other non-trivial factors impacting inference energy on edge devices.

Frequency Scaling Sweep and DVFS

We start with a simple experiment to understand the impact of frequency scaling on power and inference time; in the next subsection, we analyze the impact on energy consumption, which is the aggregated power consumption over the workload execution. Figure 3.1(a) shows the time taken for inference (left y-axis) and power consumed (right y-axis) as a function of GPU frequency for Jetson Nano using AlexNet inference with batch size of 8; here, we fix the CPU frequency at 1132.8 MHz. Note that the inference time, or latency, is the time to complete the full workload of 3,200 inferences. We see that power consumption increases almost linearly with GPU frequency. However, the *decrease in inference latency starts to plateau out at higher GPU frequencies* as GPU is no longer the bottleneck and the performance is limited by other components such as memory and I/O bandwidth; similar effects have been noted in prior work on servers [148].

Both power and latency vary greatly as GPU frequency changes from lowest to highest; the span of power consumed is 4.7W, whereas the span of latency is 532s. Figure 3.1(b) shows a similar result, but for changing CPU frequencies; here, we fix the GPU frequency at



(a) Changing GPU frequency

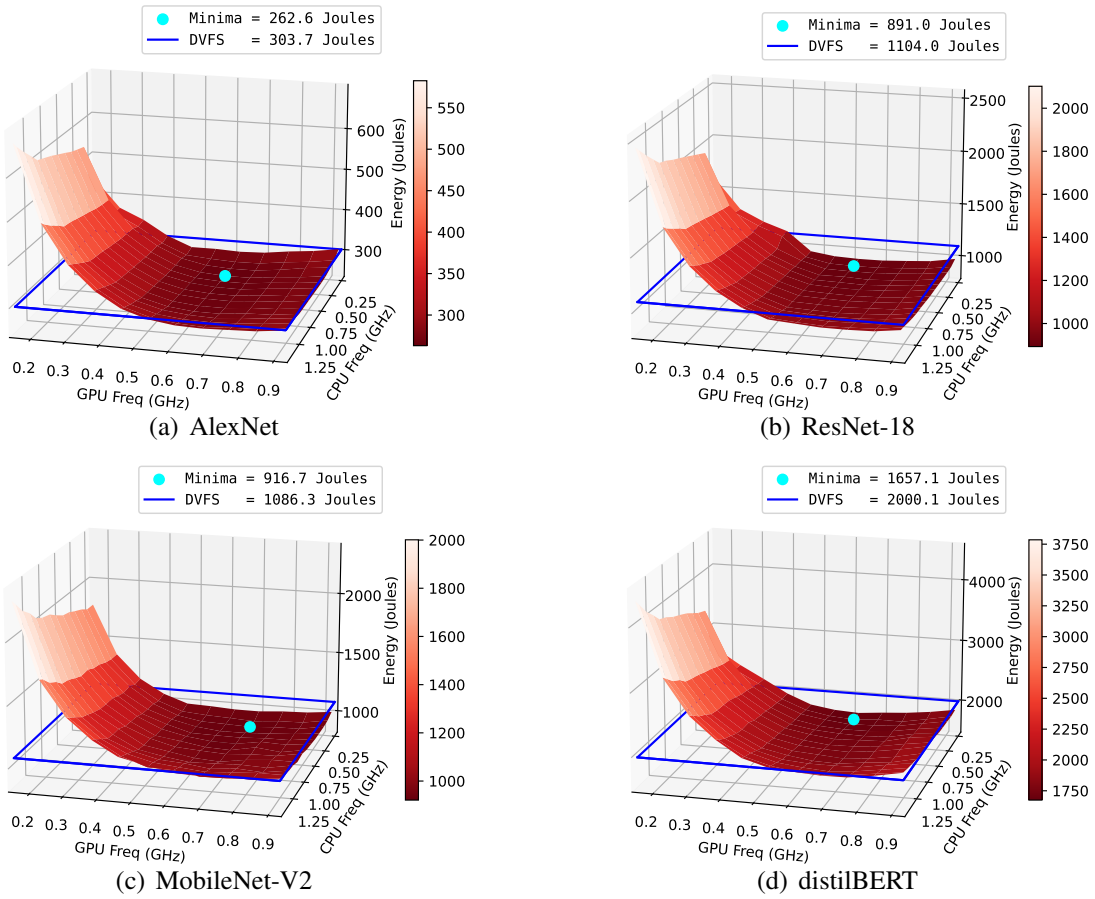
(b) Changing CPU frequency

Figure 3.1: Comparison of inference latency and power consumed under different CPU/GPU frequencies when running AlexNet on Jetson Nano.

0.6912 GHz. While the trend is somewhat similar, we see that the span of power (1W) and span of latency (60s) is much more narrower, indicating that ***CPU frequency has a smaller impact compared to GPU frequency***. This is because computationally intensive operations, such as tensor operations, are done by the GPU, whereas the CPU is responsible for less intensive tasks such as data preprocessing, initialization, and control flow.

Next, we evaluate the impact of DVFS on power and latency. The dashed lines in Figure 3.1 show the behavior when using DVFS instead of manually setting the frequencies; note that the DVFS values are same in both subfigures. We see that DVFS affords low latency but incurs very high power consumption, suggesting that *DVFS opts for higher frequencies*. This was confirmed by profiling, which revealed that DVFS operated at the highest CPU frequency 89% of the time and at the highest GPU frequency 83% of the time. Note that DVFS power usage can be slightly higher than that observed by manually setting the highest frequency because DVFS aggressively scales up the operating voltage when increasing frequency.

The above results are for the default 'schedutil' governor for CPU and and the default 'nvhost_podgov' governor for GPU DVFS. We also experimented with the other DVFS governors available (*e.g.*, 'powersave', 'performance', 'ondemand'). We found that all GPU DVFS governors perform similarly, with the power consumption and latency under other governors being within 1% of those under the default governor. With CPU DVFS, the best among other governors had 2.9% lower power, but with 2.6% higher inference latency. Given the small difference, we use the default DVFS governors going forward.



(a) AlexNet (b) ResNet-18

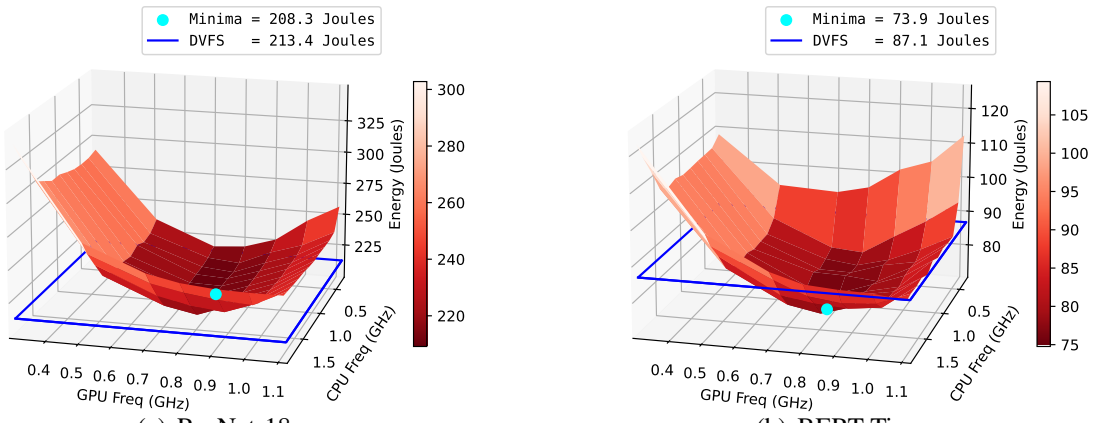
(c) MobileNet-V2 (d) distilBERT

Figure 3.2: Inference energy consumption as a function of CPU and GPU frequency under Jetson Nano; the color shade denotes the energy consumption, also shown on z-axis. Also highlighted is the point at which energy is minimized. The blue line represents the energy consumed under DVFS.

Energy Topology under Jetson Nano

While the power and latency plots illustrate how frequency impacts these metrics, they do not convey the energy impact. Figure 3.2 plots the energy consumed for inference as a function of CPU and GPU frequency for all 6 workloads for Jetson Nano. The batch size is fixed at 16 for all workloads except MobileNet-v2, for which we use batch size of 8. Generally, we see that energy does not change much with CPU frequency, for a given GPU frequency. However, for a fixed CPU frequency, the ***energy consumption does change substantially with GPU frequency***. While not always visible, the *impact of GPU frequency on energy is not monotonic*; there exist some moderately high GPU frequencies at which the energy is minimized, as shown by the cyan dot in the plots.

The minimum energy obtained by sweeping over all the CPU and GPU frequencies does lower energy significantly when compared to DVFS, as noted in the legend for each subfigure. The percentage reductions in energy afforded by the minima over DVFS for



(a) ResNet-18 (b) BERT-Tiny

Figure 3.3: Inference energy consumption as a function of CPU and GPU frequency under Xavier NX; the color shade denotes the energy consumption, also shown on the z-axis. Also highlighted is the point at which energy is minimized. The blue line represents the energy consumed under DVFS.

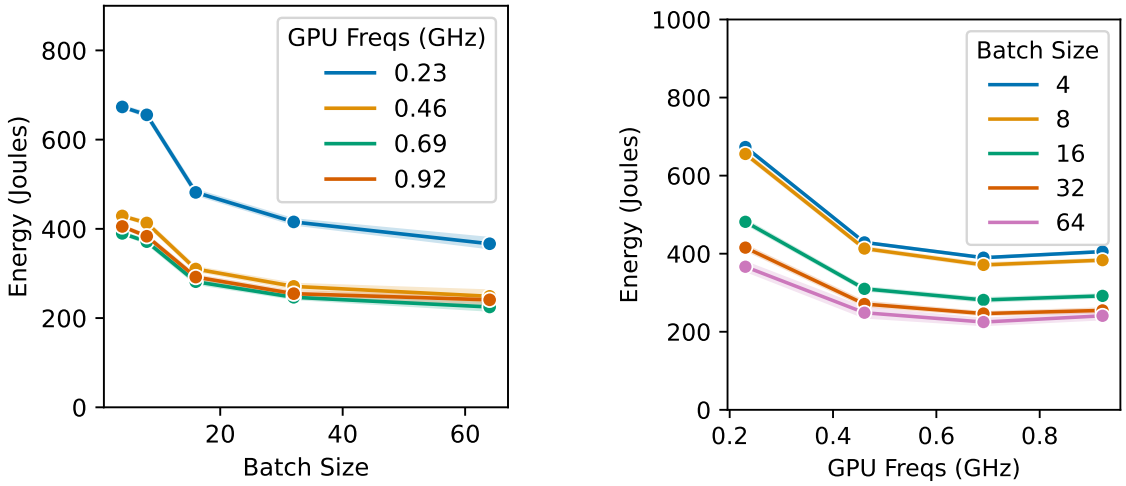
Figures 3.2(a)–3.2(d) are 13.5%, 19.3%, 15.6%, and 17.2%, respectively. However, we found that the latency under the minima configuration is typically 28%–35% higher than that achieved under DVFS. The minima usually occurred at the GPU frequency of 614.4MHz, with the optimal CPU frequency varying across the workloads.

Energy Topology under Jetson Xavier NX

Figure 3.3 shows a similar energy vs. frequencies plot for three workloads with batch size 16, but under Xavier NX. We see similar trends here as with the Nano (thus omitting figures for other workloads), with the minima affording an energy reduction of 2.5%, and 15.1% for ResNet-18, and BERT-Tiny, respectively; the reduction afforded for other models was in the 13–15% range. The energy minima usually occurred at GPU frequency of 803.25MHz.

Comparing the energy topologies of Nano (Figure 3.2) and Xavier NX (Figure 3.3), we clearly see that the ***energy consumption of Xavier NX is significantly lower***, often by at least 2×, and sometimes as much 4×. Given the newer generation of Xavier NX, this is not wholly unexpected. We also observe a greater increase in energy (compared to the minima) at higher frequencies for Xavier NX, resulting in a steeper trough-like surface; by contrast, the energy values appear to plateau and only slightly increase at higher frequencies under Nano.

However, we found that the latency under the minima configuration is typically 17%–23% higher than that achieved under DVFS.



(a) Impact of batch size

(b) Impact of GPU frequency

Figure 3.4: Comparison of energy consumed for different batch sizes and GPU/CPU frequencies when running AlexNet inference on Jetson Nano.

Impact of Batch size

Figure 3.4(a) shows the impact of batch size on energy consumption under Jetson Nano for AlexNet for a fixed CPU frequency (of 1.48GHz) and different GPU frequencies. We observe that energy drops almost linearly with batch size, with the energy consumed under the highest batch size (64) being 40–45% lower than that under the smallest batch size (4). Since the workload is constant (3,200 inferences) across batch sizes, a higher batch size reduces the total number of mini-batches, resulting in the GPU performing less I/O and preprocessing. The drop is less pronounced at higher batch sizes because the GPU computation throughput starts to saturate as the batch size increases. Note the higher energy profile for the lowest GPU frequency; this is consistent with the higher z-axis values for lower GPU frequencies in Figure 3.2(a).

Results for the impact of batch size on inference energy were qualitatively similar when we fixed a GPU frequency and varied the CPU frequencies; however, the energy difference between different CPU frequency curves was much smaller, consistent with the observation from Figure 3.2(a) that CPU frequency has a small impact on inference energy.

Figure 3.4(b) presents the same results but with GPU frequency on the x-axis and distinct curves for different batch sizes. We see that a larger batch size reduces inference energy; visually, the energy curve for a larger batch size is "shifted" lower, without impacting the trend. This suggests that the *energy-saving impact of batch size is independent of the energy-saving impact of GPU frequency*, and can thus be ***combined for greater energy savings***. Results were qualitatively similar for other workloads, and under Xavier NX.

Table 3.3: Top-5 validation accuracy for different ResNet versions

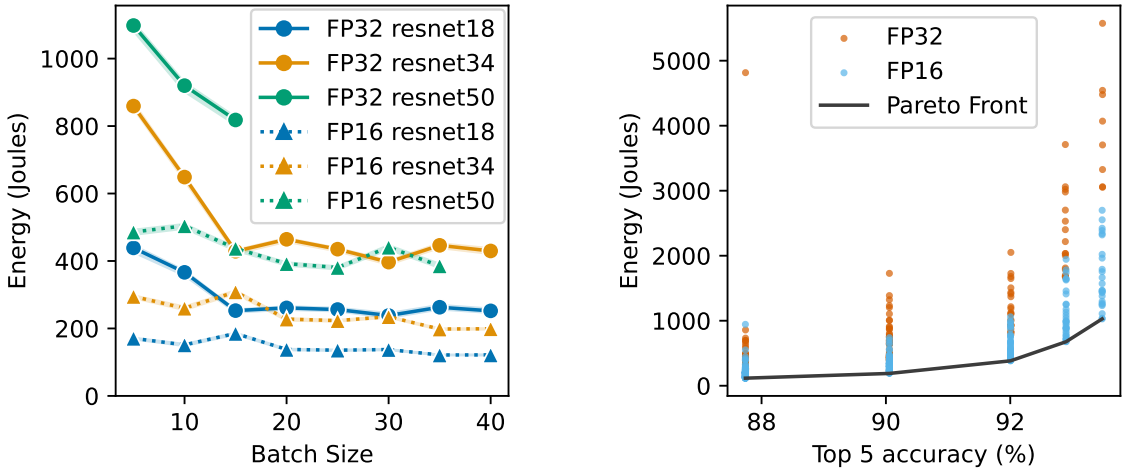
Layers in ResNet	FP32 Accuracy	FP16 Accuracy
18	87.74	87.74
34	90.06	90.06
50	92.01	92.00
101	92.89	92.89
152	93.48	93.47

Impact of Quantization and Layers

We next analyze the impact of two workload parameters—number of layers in the model and quantization. Increasing the number of layers for a DNN can improve accuracy by allowing them to capture more complex patterns, but at the expense of increased energy consumption. Quantization is the process of representing real numbers as the closest possible value allowed by the data structure. Pre-compiled PyTorch for the Jetson series supports only FP32 (float32, single precision) and FP16 (half precision) quantizations. Lower quantizations reduce precision (*e.g.*, 1234.1234 in FP32 becomes 1234.0 in FP16) but save memory, and their execution can be accelerated on supported hardware, leading to energy savings.

Interestingly, both parameters impact inference energy *and* inference accuracy, unlike the other parameters we evaluated thus far. We consider the ResNet model under Xavier NX for this study. We first measured the validation top-5 accuracy of different versions of ResNet (with different number of layers and quantization) pre-trained on Imagenet 1K dataset [119, 47] and report them in Table 3.3. We see a ***negligible difference in top-5 accuracy when we change quantization from FP16 to FP32***; the top-1 accuracy had the same trend as well. This observation is consistent with prior work [157]. However, the number of layers does show an impact on accuracy.

To investigate the impact of these workload parameters on inference energy, we fixed the CPU and GPU frequencies (to 1.9 GHz and 1.1 GHz, respectively) and varied the batch size. Figure 3.5(a) shows the inference energy as a function of batch size for different ResNet versions. We make two interesting observations. First, The *FP16 variant of a version has 26%–65% lower energy consumption than its FP32 counterpart*. This is because, for a given workload, the FP16 inference latency is lower than that of FP32 while their power usage is similar. The lower latency under FP16 is due to lower memory and I/O requirements, hardware support that accelerates computation for lower precision [62], and the fact that due to its lower memory footprint, we were able to use higher batch sizes for FP16 (*e.g.*, FP16 ResNet-50 can go till batch size 35 before encountering OOM, whereas FP32 Resnet-50 is limited to a batch size of 15).



(a) Energy consumed vs. batch size

(b) Energy-accuracy tradeoff

Figure 3.5: Analyzing the effect of the number of layers and quantization for the ResNet model on Xavier NX.

Second, a larger version of ResNet with FP16 can consume *lower* energy than a smaller version with FP32. For example, Resnet-34 with FP16 at batch size 35 consumes 18.7% ***lower energy*** than the most energy-efficient Resnet-18 with FP32 (which occurs at batch size 30); this is significant since Resnet-34 with FP16 provides 2.6% ***more accuracy*** than Resnet-18 with FP32 (see Table 3.3). Similarly, (not shown Figure 3.5(a)), ResNet-152 with FP16 at batch size 15 saves 38% energy and provides 0.6% more accuracy compared to ResNet-101 with FP32. Other frequency combinations reveal such interesting cases as well. The maximum energy savings are obtained for ResNet-152 with FP16 with optimal batch size, which saves 42% energy and provides 0.6% higher accuracy compared to the most energy-efficient batch size configuration of Resnet-101 with FP32 under 1.9GHz CPU frequency and 0.2GHz GPU frequency. Thus, by *jointly* tuning batch size, quantization, and number of layers, we can uncover superior model configurations.

To demonstrate the benefits of jointly tuning workload parameters, we consider the scatter plot of inference energy and accuracy for different Resnet versions in Figure 3.5(b). There is a clear tradeoff between energy and accuracy, with higher accuracy generally requiring more energy. The Pareto front shown in Figure 3.5(b) allows us to pick the most energy-efficient ResNet version for a given accuracy target; as discussed before, the FP16 versions are superior to FP32 ones.

Impact of other factors

The implementation of the DNN model can also impact its inference energy. We perform a comparison of AlexNet implementation in PyTorch with our implementation of AlexNet

using LibTorch in C++. We find that the ***PyTorch implementation consumes 16% more energy compared to LibTorch***, likely due to Python overheads. However, LibTorch is not commonly used during prototyping as it needs to be compiled every time a model parameter changes. LibTorch is more commonly used in high-performance systems [114, 145]. We also investigated other optimizations in our experiments such as turning off lazy-loading and garbage collection, and found a 1.82% and 1.65% reduction in total energy, respectively.

3.4.3 Key Takeaways

This work analyzes the impact of workload parameters and hardware knobs on the energy consumption of DNN inference on smart edge devices. Our experimental results show that significant improvements in energy consumption (as much as 19%) can be achieved over default DVFS settings by choosing the right frequencies. We also find that workload parameters (*e.g.*, the batch size) and model settings (*e.g.*, graph initialization strategies) can impact inference energy. Interestingly, when accuracy is considered, the tuning of workload parameters plays an important role, revealing model configurations that can achieve higher accuracy while consuming lower energy.

Out of the parameters we studied, we decided to focus on CPU frequency, GPU frequency, and batch size in the rest of the thesis because they have the most significant impact on energy consumption, with non-trivial locations of optimal configurations in the search space.

3.5 System Design

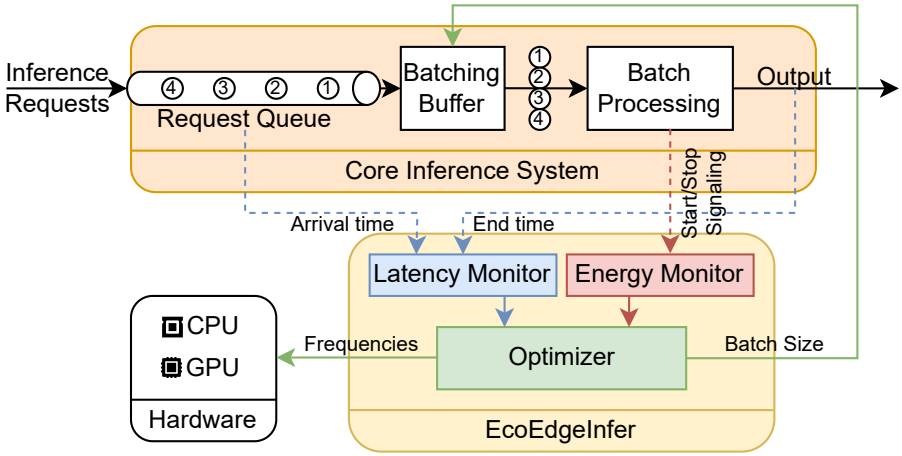


Figure 3.6: Illustration of EcoEdgeInfer's system design.

To address the challenges of finding non-trivial optimal configurations for DNN inference on edge devices, we present EcoEdgeInfer. We primarily break down the features of EcoEdgeInfer into: *(1) Core Inference System*, *(2) Metrics Collection*, *(3) Optimizer*, and *(4) Applying the optimal configuration*. Figure 6.2 illustrates the system design of EcoEdgeInfer and the interactions between these components.

3.5.1 Core Inference System

The core inference system is the component of EcoEdgeInfer that actually performs the DNN inference and batching, and is the only component that interacts with the developer's code. It also allows us to collect the metrics that are used by the optimizer.

Every time the inference method is called, the core inference system receives the input data and adds it to the request queue. On a separate thread, it waits for the request queue to have enough requests to form a batch (in the batching buffer) and then performs the inference on the batch. The batch size is determined by the optimizer and can be changed dynamically. The request queue, the batching buffer, and the tunable batch size together allow the system to process requests with varying arrival rates by staging the incoming requests; see Figure 6.2. The core inference system sends signals to the monitoring system to start collecting metrics while processing each batch.

3.5.2 Metrics Collection

The monitoring system is responsible for collecting (at least) the latency and energy consumption metrics based on the signals received from the core inference system. The latency monitor uses the arrival timestamp of each request in the request queue and the completion timestamp of the request to calculate the request's latency. The energy monitor starts and stops logging the power sensors based on the signals received from the core inference system. We log the power sensors every 10 ms over the i2c interface and the sysfs interface provided by NVIDIA's API [106]. The latency and energy metrics collected for a single batch may be too small and can lead to noisy values. So, the monitoring system stores the metrics in a temporary cache till enough samples are collected. In our experiments, we found that monitoring for $\sim$400 requests is enough to remove the noise and get a good estimate of the metrics. This duration (of 400 requests) is referred to as an *Optimizer Step* in the rest of the work.

3.5.3 Optimizer and optimization policies

The optimizer is the core component of EcoEdgeInfer that determines the optimal parameter values to be used. It uses the metrics collected in the previous optimizer step to calculate a cost function. In our work, we consider the cost function to be the mean of homogenized latency and homogenized energy consumption; we aim to minimize this cost function. In general, any combination of these two metrics could be employed as needed. To homogenize the latency and energy metrics, we divide the observed latency and energy values in each experiment by those obtained when all parameters are set to their maximum value. Similar approaches have been used in prior work to balance multiple metrics in the objective function (latency and energy, in our case) [155, 129].

The calculated cost is then used by the optimizer to predict the optimal CPU Frequency, GPU Frequency, and Batch Size that can be used for the next optimizer step. In addition to our algorithm, EcoGD (described in the next subsection), we have implemented five other optimization policies inspired by prior works as baselines, including Grid Search, Multi-Armed Bandit, and Bayesian Optimization. These baselines are detailed in Section 3.6.3.

3.5.4 Predicting the optimal configuration via EcoGD

We design a specific optimization algorithm, EcoGD, for the problem of optimizing the parameters of inference on edge. Our EcoGD algorithm is inspired by the gradient descent algorithm. However, unlike standard gradient descent problems, in the inference on edge setting, we do not have access to the gradient of the cost function nor do we readily have all the data points needed to calculate it. As such, the traditional gradient descent algorithms cannot be used as-is; similar observations have been made by prior works addressing different problems as well, such as the problem of tuning the parameters of transactional memory[49]. EcoGD can be considered a variant of the broad family of local-search heuristic algorithms, which includes algorithms like hill climbing and simulated annealing, with Tabu search being the closest algorithm to EcoGD [58].

Algorithm 1 shows the pseudo code for EcoGD along with the input, output, and parameters of the optimizer (lines 1–4). At every optimizer step, EcoGD either explores the neighboring configurations (lines 7–9) or, if the neighborhood has been sufficiently explored, jumps to a new configuration (lines 11–20).

Every configuration in the 3-dimensional space of CPU Frequency, GPU Frequency, and Batch Size has 26 neighbors: 6 directly adjacent (i.e., change in only one dimension), 4 diagonally adjacent in the CPU-GPU plane (i.e., change in both frequency dimensions), and 16 other diagonally adjacent configurations (i.e., change in batch size and at least one

Algorithm 1 Pseudo code for EcoGD

1: **Input:** History matrix H containing the cost of all explored configurations.
2: **Output:** Next configuration to run $C = (CPU, GPU, Batchsize)$
3: **Parameters:** Memory size Mem_{size}, Maximum loops Max_Loops, Starting Config C_{start}
4: **Initialize:** $loop_counter \leftarrow 0$, $last_center \leftarrow C_{start}$

Function $EcoGD(H)$:
5: $H = copy(H, Mem_{size})$
6: ▷ Trim history to last Mem_{size} configurations
7: $unexplored_nbhr \leftarrow \{nbhr \in 6\text{-}nbhrhd(last_center)$ s.t. $nbhr \notin H\}$
8: ▷ Finding unexplored neighbors in 6-neighborhood
9: ▷ neighbour, neighborhood abbreviated as nbhr, nbhrhd
10: **if** $Len(unexplored_nbhrs) > 0$ **then**
11: $C \leftarrow unexplored_nbhrs[0]$
12: **return** C ▷ Return the first unexplored nbhr if any
13: $nbhrhd_history \leftarrow explored_nbhrs_history(last_center, H)$
14: $+ unexplored_nbhrs_history(last_center, H)$
15: ▷ History of unexplored neighbors are estimated
16: $best_config \leftarrow \text{argmin}(nbhrhd_history)$
17: **if** $best_config \neq last_center$ **then**
18: $loop_counter \leftarrow 0$
19: **else**
20: $loop_counter \leftarrow loop_counter + 1$
21: **if** $loop_counter > Max_Loops$ **then**
22: $best_config \leftarrow random_nbhr(last_center)$
23: ▷ Jump to a random neighbor if stuck
24: $loop_counter \leftarrow 0$
25: $last_center \leftarrow best_config$
26: **return** $best_config$

frequency dimension). However, EcoGD only evaluates the cost of the 6 directly adjacent neighbors of the current configuration for exploration (line 6). This reduces the number of configuration evaluations from 26 to just 6, thereby significantly reducing the exploration time. The 4 configurations that are diagonally adjacent in the CPU-GPU plane are instead estimated (line 10) using the following equation:

$$Cost(i+x, j+y, k) = \frac{Cost(i, j+y, k) + Cost(i+x, j, k)}{2} \tag{3.1}$$

where $Cost(i, j, k)$ is the cost of the configuration with CPU Frequency i, GPU Frequency j, and Batch Size k. Note that $x, y \in \{-1, 1\}$ indicates the diagonal neighbors of the current configuration in the CPU-GPU plane.

The remaining 16 diagonally adjacent configurations are not considered for estimation because we could not find a reliable and accurate approach to estimate the cost of such configurations. With this approach of selective exploration and estimation, EcoGD is able to converge to the optimal configuration quickly while following a path with reasonable costs. This is a key feature of EcoGD that distinguishes it from Tabu search and other local-search heuristic algorithms.

If the neighborhood is already explored, EcoGD jumps to the neighbor that has the lowest cost and the process is repeated (line 11). We do not let EcoGD jump farther than the neighboring configurations to avoid the optimizer from moving to a vastly different configuration. In practice, we found that jumping farther than the neighboring configurations does not lead to a significant improvement in the cost function and can cause repeated overshooting over the optimal configuration. Note that a learning rate like mechanism cannot be used and tuned because the configuration space is discrete and only the immediate neighbors are possible due to quantization.

To avoid re-exploring the same configurations repeatedly, EcoGD maintains a list of recently explored configurations and the corresponding costs. We limit the number of configurations remembered to a fixed number to allow adaptation to changing workload conditions (lines 12–18). And to avoid getting stuck in a local minima, EcoGD jumps to a random neighbor when the same configuration is predicted more than a fixed number of times and all the neighbors have been explored already (line 17).

EcoGD maintains only a few configurations and costs in its history to minimize physical memory overheads and also to quickly adapt to changing workloads by discarding stale configurations (line 5). The details of the hyperparameters of EcoGD are discussed in Section 3.6.5.

3.5.5 Applying the optimal configuration

After the optimizer predicts the optimal configuration, it sets the optimal CPU frequency using the cpufreq driver and the optimal GPU frequency using NVIDIA's API [106]. In parallel, the optimizer sends the optimal batch size value to the core inference system, to be used for the subsequent inferences.

3.6 Experimental Setup and Methodology

In this section, we describe the experimental setup, workload traces, baseline algorithms, and the methodology we employ for our experimental evaluation (presented in Section 3.7).

3.6.1 Experimental setup

We perform all evaluations on an NVIDIA Jetson Xavier NX device; see section 3.2.1 for the technical specifications of the device. We use PyTorch v2.1.0, CUDA v11.4, and cuDNN v8.6 for our experiments.

We primarily use Resnet50 from the torchvision library [15] to extend our evaluation to a different model; for this model, we use an image of size 224×224 as the input request. We also use a transformer-based model, BERT-Tiny [33, 143], to extend our evaluation; for this model, we use a random text of size ∼1KB as the input request.

3.6.2 Request arrival patterns and traces

A key contribution of our evaluation is considering the impact of the request arrival pattern. As such, we perform our evaluations under three different inference request arrival patterns. First, we consider fixed inter-arrival times (time between successive requests); these conditions represent a controlled workload setting. We choose inter-arrival times of 50ms and 90ms, representing high and low load, respectively; these values were also chosen to ensure a stable system, as the minimum inference service time is about 40ms for Resnet50. Second, we consider bursty arrivals, representing a more extreme workload setting. Specifically, requests arrive in bursts of 10 requests every 500ms or every 900ms, thus still maintaining an average inter-arrival time of 50ms and 90ms, respectively, but with a more bursty arrival pattern.

Finally, we also evaluate using snippets from three real-world time-varying load traces. Publicly accessible inference request traces are not widely available. As such, we consider the following traces in our evaluation.

1. **Bellevue trace:** Number plate reading is a popular inference task at edge devices and smart cameras, and is used in applications such as traffic monitoring and parking management. The number of inference requests arriving at such edge devices is proportional to the number of vehicles passing through the camera at any given time [121]. To generate realistic traffic for this inference task, we use the Bellevue Traffic Video Dataset [3], which contains video data of vehicular traffic in Bellevue, Washington. We extract the number of vehicles per second from the dataset and use this as a proxy for the incoming rate of inference requests.

2. **Twitter trace:** Social media content moderation is another popular inference application to perform large-scale tasks such as sentiment analysis and fake news detection. In this application, each request for inference corresponds to a tweet that needs to be analyzed. To simulate the load for such an application, we use Twitter's sampled stream of tweets [16, 4]. We select a 3-hour period surrounding midnight on December 31, 2020, and January 1, 2021, from the East Coast of the United States to create a localized traffic trace. The number of tweets per second serves as a proxy for the inference request rate.

3. **Azure trace:** Owing to the increasing popularity of private edge data centers, well-known cloud paradigms such as serverless inference [1, 80] are expected to be employed at the edge in the near future [161]. We use the Azure Functions Traces [164] to represent the load on such an edge system. The rate of serverless invocations is used as a proxy for the arriving request rate at the edge device.

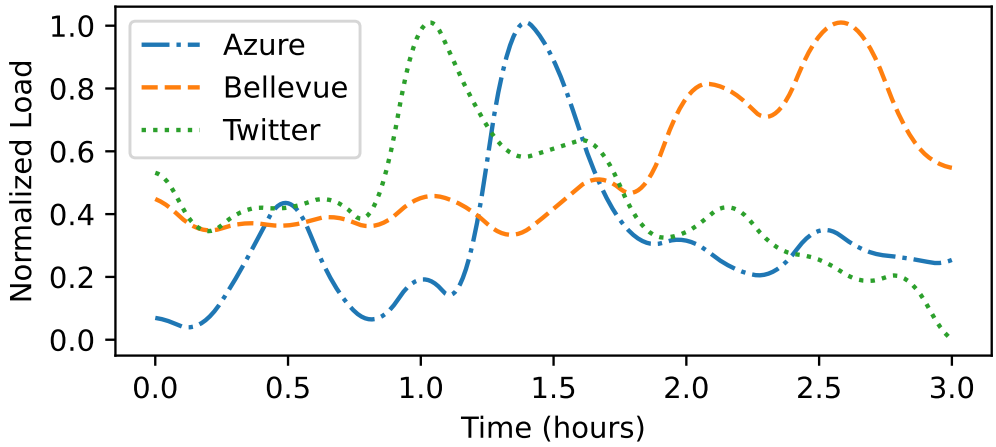


Figure 3.7: Timeseries of normalized load for the traces used in our evaluation.

In all traces, we select 3 hours of data that exhibit diverse traffic patterns. Figure 3.7 shows how the load varies over time in the three traces; the load is normalized to the maximum for each trace for ease of comparison. We see that the Azure trace has a small surge at the start and then a more significant surge to the maximum load, and then a drop back to moderate load. The Bellevue trace has a more stable load with a gradual increase towards the end and then a sudden drop. In the Twitter trace, we see a sharp surge to maximum load and then a gradual drop to a very low load. The trace data is scaled to our device capacity and is used to create a timeseries of request rates arriving at our edge device. In terms of average load (or request rate) post-scaling, the Bellevue trace has the highest load, followed by the Twitter and the Azure traces.

3.6.3 Comparison baselines

In our experimental evaluation, we compare EcoGD with five other baseline algorithms.

1. **Grid Search** is a simple optimization policy that exhaustively tries all possible combinations of CPU Frequency, GPU Frequency, and Batch Size, and then selects the one with the lowest cost. Grid Search is easy to implement and should result in the optimal configuration but it requires significant running time to sweep over all possible combinations of parameters. For example, on the NVIDIA Xavier NX, the search space is $25 \times 15 \times 16 = 6,000$ combinations. While Grid Search may be impractical for real inference systems (requiring a full Grid Search to be re-performed every time the workload conditions change), we use this policy as a baseline in our evaluation.

2. **Linear Search** starts with a default configuration sweep through one dimension at a time (from among batch size, CPU frequency, and GPU frequency) and selects the configuration in that dimension that gives the lowest cost. It then moves to the remaining dimensions iteratively to optimize their configuration, while fixing the optimal configuration for the prior dimensions. Once the search is complete, a configuration which is locally optimal in each dimension is reported. The policy is much faster than Grid Search as it only searches through $25 + 15 + 16 = 56$ configurations. However, like Grid Search, Linear must be rerun every time the workload changes.

3. **DVFS** is the default policy (enabled by default) used in servers and edge devices to dynamically set the CPU and GPU frequency depending on the incoming load. The Xavier NX device ships with many governor policies for automatic DVFS control for CPU and GPU. Among them, schedutil and nvhost_podgov are the default governors for CPU and GPU, respectively. We also experimented with other governors, but there were no remarkable differences in performance. As such, we report DVFS results under their default CPU and GPU governors. DVFS, in theory, automatically adapts to changing workloads and is thus an adaptive policy. For example, schedutil constantly monitors the CPU utilization and adjusts the CPU frequency accordingly [11]. Note that DVFS does not manage the batch size, and so we use a reasonably large (for edge devices) batch size of 16 and a moderate batch size of 8 in our experiments.

4. **Bayesian Optimization** is an optimization technique that has been often used in systems literature [22, 24, 43] to optimize black-box functions. It uses a probabilistic model to predict the cost function and then uses an acquisition function to decide the next configuration to try. We use the GaussianProcessRegressor and RadialBasisFunction kernel from the scikit-learn library to implement Bayesian Optimization [117].

5. **Multi-Armed Bandit (MAB)** is another popular technique used in systems literature to find optimal configurations for systems [155, 136]. We chose the most popular variant of MAB, the epsilon-greedy algorithm, for our evaluation. At every step, the MAB algorithm probabilistically selects either exploration or exploitation. When exploration is selected, the algorithm tries new random configurations, and when exploitation is selected, the algorithm selects the configuration that has given the lowest cost so far. When the same configuration is selected multiple times, we choose to update the cost history using exponential moving average as it is the most common technique used in the literature when conditions are changing [100].

We informally refer to Bayesian Optimization, MAB, and EcoGD as *learning-based algorithms* since they learn from the cost of the configurations they have tried so far and use this information to decide on the next configuration to try.

3.6.4 Evaluation methodology

For our evaluation, we consider the energy consumption, inference latency, and cost as the primary metrics (see Section 3.5.3 for our cost definition); we consider mean and tail values for these metrics in different evaluation settings. When running Grid Search and Linear Search, we run the experiments till the end of the search space. DVFS, Bayesian Optimization, MAB, and EcoGD are run for 150 optimizer steps in the case of fixed and bursty request arrival patterns. For the traces, we run the experiments for 3 hours as that is the length of the traces. We also repeat each experiment 3 times. The error bars in the bar plots of Section 3.7 represent the standard deviation of the 3 runs.

All experiments are given 5 optimizer steps as a warm-up period to get rid of bootstrapping effects such as PyTorch's lazy initialization. All algorithms and result interpretations ignore these 5 steps. We also limit all algorithms' optimizer search space to prevent configurations that are known to be bad and can cause queue build-up (and may crash the experiments); these bad configurations comprised about 1% of the total possible search space.

3.6.5 Hyperparameters settings of different algorithms

Different algorithms have different hyperparameters that need to be tuned and set to achieve their best performance.

- **Grid Search** has no hyperparameters.
- **Linear Sweeps** has hyperparameters to define the order of parameters (batch size, GPU frequency, CPU frequency) to sweep through. We tried all 6 permutations of the order

and found that the differences in the results, especially the cost achieved, were negligible. As such, we present the results for the permutation that performed the best (within the narrow margin of results): batch size sweep, then CPU frequency sweep, and finally GPU frequency sweep.

- **DVFS** only optimizes the CPU and GPU frequencies. So, we manually set the batch size to a fixed value. We initially consider batch sizes of 8 and 16, and show both sets of results in our evaluation. We find that batch size 16 is superior, so we fix the batch size to 16 for DVFS under trace-driven experiments.
- **Multi-Armed Bandit** (MAB) has hyperparameters to define the exploration/exploitation probability. While higher exploitation probability results in lower energy consumption as the algorithm sticks to the best configuration, it may take much longer to find the best configuration. We set the exploitation probability to 0.9, exponential decay to 0.9, and the number of configurations when cold starting to 10, as they gave the best results in our experiments.
- **Bayesian Optimization**, similar to MAB, needs some initial configurations to start the optimization when cold starting. We set this value to 10 to make it comparable with MAB.
- **EcoGD** has hyperparameters to set the memory size of configuration cost history and the number of maximum loopbacks allowed to jump out of local minima. Having a higher memory size results in better performance when running under fixed inter-arrival patterns, but it harms the performance when the arrival pattern is dynamic. Setting a lower value for maximum loopbacks may result in faster convergence but can also lead to random spikes in the cost achieved. We set the memory size to 10 and maximum loopbacks to 10, as they gave the best results in our experiments. Unlike MAB and Bayesian, EcoGD does not need any initial configurations to start the optimization when cold starting.

3.7 Evaluation Results

In this section, we present our evaluation results. We first consider Resnet50 under fixed and bursty load, in Sections 3.7.1 and 3.7.2, respectively, to analyze the performance and characteristics of different algorithms. This performance analysis is summarized in Section 3.7.3 for reference. We then present our main results in Section 3.7.4 using Resnet50 and BERT-Tiny under three time-varying, real-world request traces; we analyze both mean and tail results under these trace-driven experiments.

3.7.1 Comparison of algorithms under fixed load

Best configuration achieved

We start our evaluation by comparing the performance of the *best* configuration achieved by each algorithm. For Grid Search and Linear, the best configuration is the one found after sweeping their respective parameter search spaces. In the case of learning-based algorithms (i.e., MAB, Bayesian, and EcoGD), the best configuration is the one that is selected by the algorithm after it converges; the algorithms are said to have converged when the same low-cost configuration is selected multiple times in a row. For DVFS, the frequency selection is done automatically by the governors based on various external factors that are not in our control, and DVFS does not have a distinct convergence phase. For a fair comparison, we select as 'best' configuration for DVFS the one that DVFS picks after it has taken the same number of steps as that taken by EcoGD to converge.

Figure 3.8 shows the energy consumed, latency achieved, and cost incurred by each algorithm once it converges to the best configuration; we show results separately for a fixed inter-arrival time (IAT) of 50ms and 90ms. All values for all algorithms are normalized by that under Grid Search for ease of comparison. Note that the cost is obtained based on homogenized energy and homogenized latency values (see Section 3.5.3); as such, the cost values are not simply a mean of the energy and latency values shown in the figure.

We see that Grid Search consumes the lowest cost in both the cases (tied with EcoGD for 50ms IAT), as it sweeps through all possible configurations to find the best configuration. This is to be expected as Grid Search is essentially an exhaustive search solution. Note that Grid Search does not necessarily consume the lowest energy in all cases; for example, it consumes more energy than Linear in the case of 50ms IAT. However, it does achieve the lowest *cost*, as that is the objective function being optimized; see Section 3.5.3 for the cost definition. In the case of 50ms, it does so by obtaining a low latency to make up for the slightly higher energy consumption.

While Linear Search achieves roughly the same cost as Grid Search in the case of 50ms, it does so by sweeping through only a fraction of the configurations. But in the case of 90ms, Linear incurs more energy, latency, and cost than Grid Search. This is because Linear *independently* optimizes the three dimensions (batch size, CPU frequency, GPU frequency) and does not consider the interactions between them. As such, ***Linear Search may converge to a sub-optimal configuration***.

DVFS with batch size 16 has higher energy consumption, latency, and cost compared to Grid Search and Linear because it does not have access to workload information (e.g., batch size) and cannot optimize to it. DVFS with batch size 8 incurs the lowest latency among

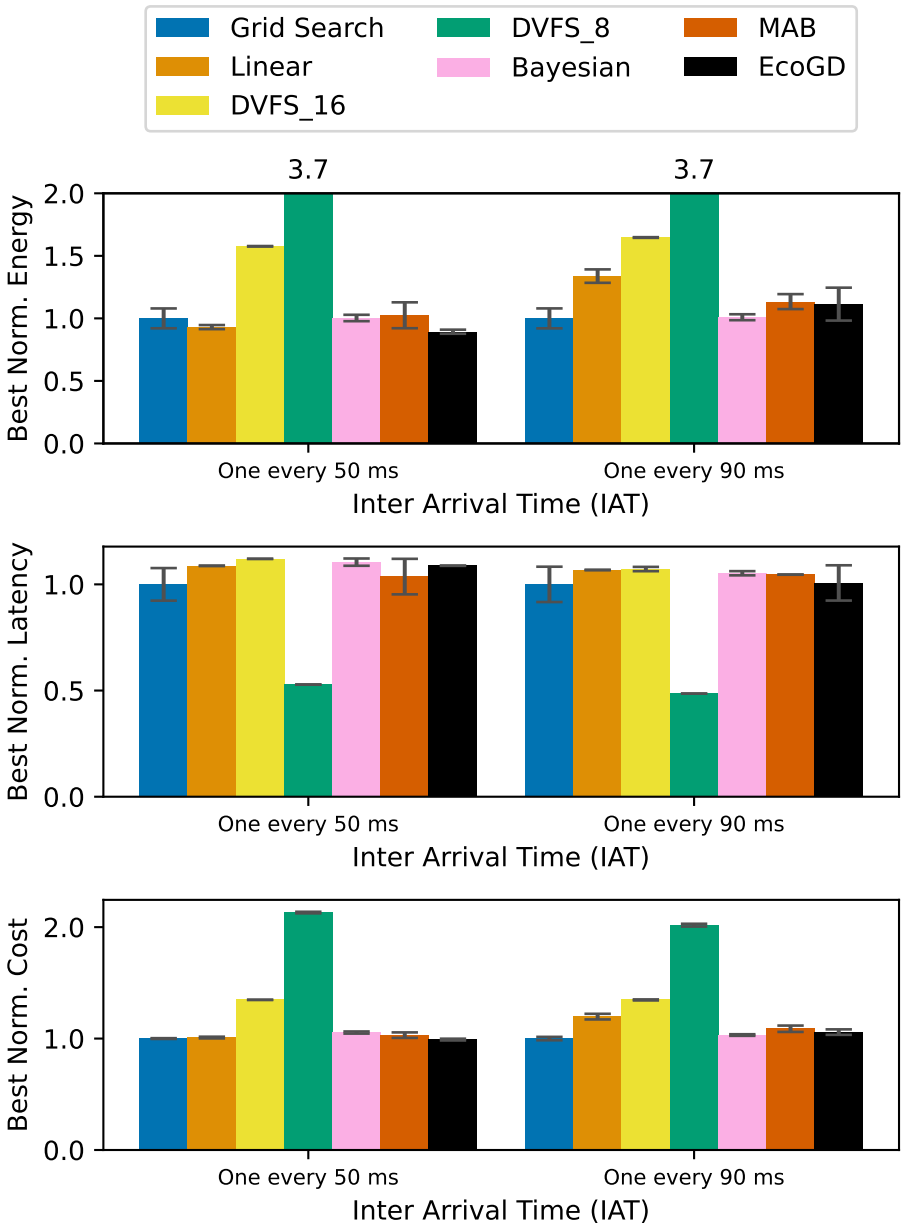


Figure 3.8: Energy, latency, and cost incurred by all algorithms after convergence under fixed request inter-arrival times.

all algorithms and approximately 2× lower when compared with Grid Search because it has a lower batch accumulation delay. However, it incurs as much as 3.7× higher energy consumption. Consequently, it has the highest cost among all algorithms. This result highlights the ***need for joint optimization*** rather than only focusing on a single metric (latency or energy).

On further inspection, we found that this high energy consumption is because of the smaller batch size (8) for this variant of DVFS. All other algorithms converged to a high

batch size (closer to 16) for their optimal configurations. In general, energy efficiency improves with batch size (due to amortization), and so a smaller batch size increases energy consumption. Prior works analyzing the impact of batch size in server-class machines for training and inference have made similar observations [89, 155]. This highlights the ***importance of optimizing workload parameters along with hardware parameters for jointly optimizing energy and latency***.

The learning-based algorithms (Bayesian, MAB, and EcoGD) typically achieve a cost only slightly higher than that achieved by (the exhaustive) Grid Search. They also achieve lower cost than sub-optimal configurations found by Linear in the case of 90ms IAT, and a much lower cost than that achieved by DVFS for both IATs. In general, we can conclude that the ***learning-based algorithms all achieve a fairly low cost under fixed IATs***. In fact, EcoGD achieves the lowest cost (tied with Grid Search) in the case of 50ms IAT.

Total cost till convergence, including overhead

The above best configuration results do not account for the energy, time, and cost incurred to arrive at the best configuration (via exploration, exploitation, and search space sweeps). To get a comprehensive view of the results, we now consider the cumulative energy consumption, inference latency, and cost incurred during the time it took each algorithm to converge to the best cost configuration. Figure 3.9 shows these metrics for each algorithm normalized by that under Linear Search; we chose Linear Search in this case for normalization because the performance of Grid Search is quite poor and would make it difficult to compare algorithms if normalized by Grid Search.

We see that ***Grid Search incurs prohibitively high overhead***, evidenced by the high values for all three metrics; this is to be expected as it needs to sweep through the 6,000 configurations (modulo the ones we discard due to predictably poor performance) to find the best one. Linear Search has much lower consumption when compared to Grid Search, as it only sweeps through 56 configurations, which represents more than a 100$\times$ reduction in search space size. In fact, Linear results in the lowest cost under 50ms IAT (but not under 90ms IAT).

Bayesian has high energy consumption, latency, and cost overhead; this is because Bayesian does not seem to converge well and keeps exploring newer configurations, resulting in frequent jumps in cost. This is because Bayesian Optimization does not have a mechanism to exploit the best configurations it has found so far and instead keeps exploring the search space. While Bayesian Optimization is useful in offline optimization (unlike our focus), it may not be very effective in online scenarios [59, 88]. While MAB has lower energy, latency, and cost compared to Bayesian, it still lags behind EcoGD in cost (and energy).

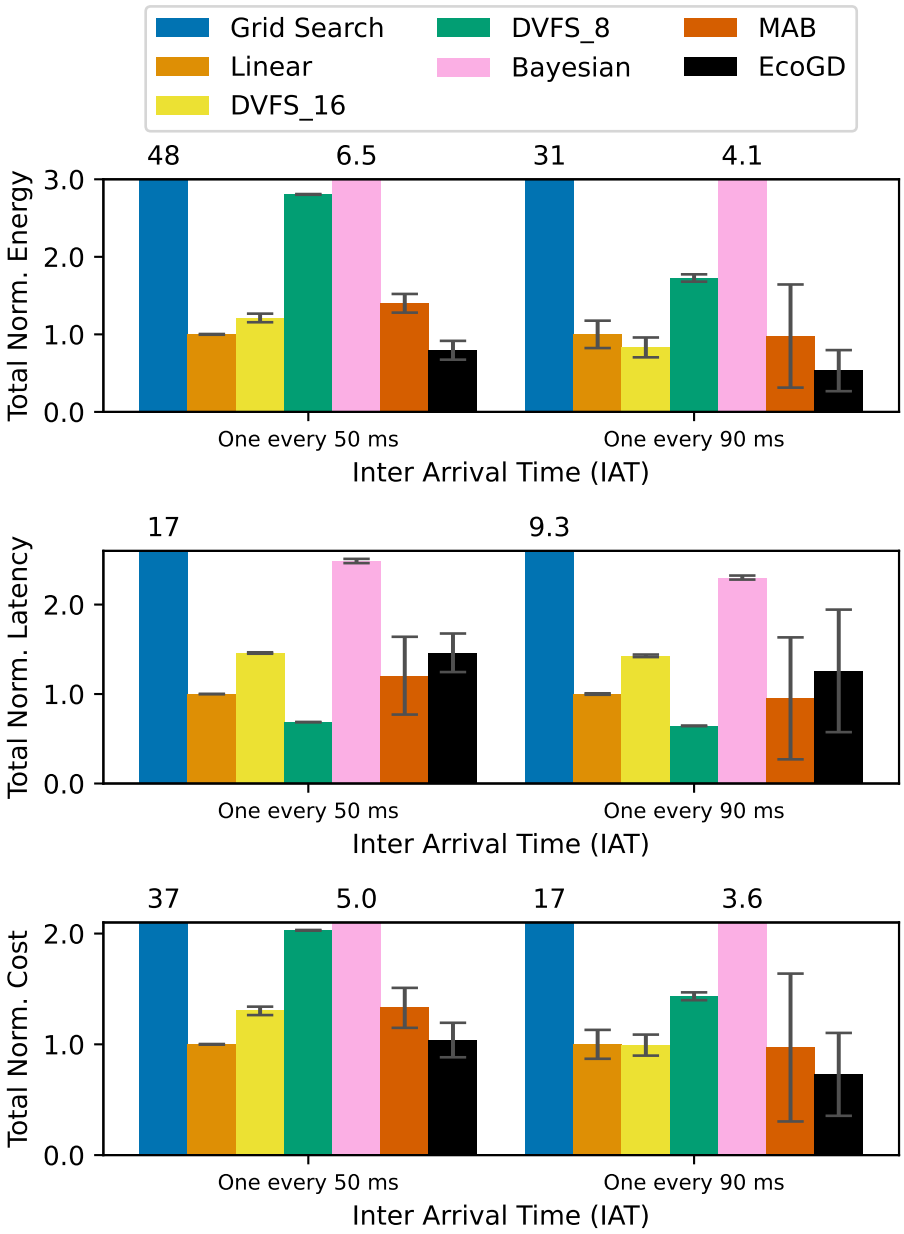


Figure 3.9: Total energy, latency, and cost with overhead incurred by all algorithms till convergence under fixed request inter-arrival times.

DVFS with batch size 16 has higher energy consumption, latency, and cost than EcoGD. This is because DVFS does not have access to workload information and cannot optimize for it. Similar to the observations for best cost, we see that DVFS with batch size 8 has a lower latency because of the lower batch accumulation time but has much higher energy consumption and cost.

EcoGD has the lowest energy consumption among all algorithms. Further, and more importantly, ***EcoGD achieves the lowest cost under 90ms IAT and second-lowest cost (only 4% higher than Linear) under 50ms IAT***. This superior performance of EcoGD is because it is able to explore the search space through an efficient route using (observed and estimated) gradients while converging to the best configuration.

We also notice that the standard deviation is higher for MAB and EcoGD in case of 90ms IAT than other algorithms. This is because they explored the search space through different, diverse routes, although they individually converged to the same neighborhood of configurations eventually. For example, in case of 90ms IAT, EcoGD converged to the neighborhood of configurations [1.8, 0.96, 14], [1.91, 0.6, 15] and [1.19, 0.41, 15] in its three runs. Here, the first, second, and third elements of the configuration triplet refer to the CPU frequency (in GHz), GPU frequency (in GHz), and batch size, respectively. We see that all three runs converged to high batch sizes; two of the runs converged to low GPU frequencies; two other runs converged to high CPU frequency. Despite the differences in the configurations, we find that their converged costs are within 4% of each other. This shows that ***multiple, local optima with similar cost values can exist***; since we optimize for cost, the actual converged configurations may be slightly different as long as they have similar cost values. In the case of MAB, the converged configurations were [1.57, 0.6, 13], [1.88, 0.51, 15], [1.91, 0.8, 15], and all resulted in converged costs that were within 5% of each other.

3.7.2 Comparison of algorithms under bursty load

We now move on to the case where the inter-arrival time of inference requests is bursty.

Best configuration achieved

Figure 3.10 shows the energy consumed, latency achieved, and cost incurred by each algorithm once it converges to the best configuration; we show results under bursty inter-arrival times of 500ms and 900ms between a batch of 10 successive requests. Grid Search incurs almost the lowest cost, energy, and latency in both cases of IATs as it sweeps through all configurations to find the best configuration. Linear Search takes more energy, latency, and cost than Grid Search in both IATs as it settles for sub-optimal configurations.

Both versions of DVFS have higher costs than Grid Search and Linear. We again find that DVFS with batch size 8 has the highest energy and cost among all algorithms (because of its lower batch size, as explained in Section 3.7.1). We also see that DVFS with batch size 16 has significantly higher latency; this is because, with 10 requests arriving every 500ms

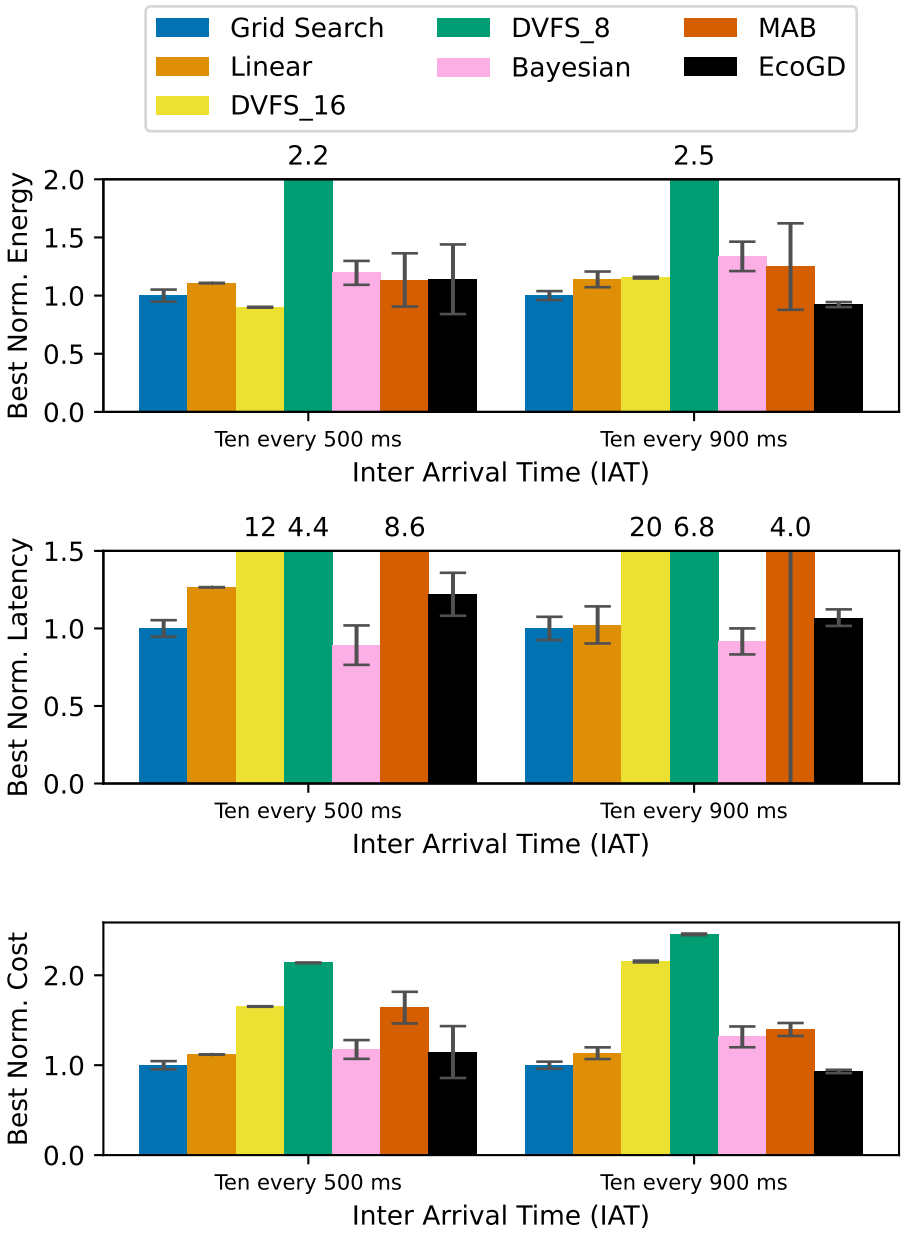


Figure 3.10: Energy, latency, and cost incurred by all algorithms after convergence under bursty request inter-arrival times.

and 900ms, it has to wait for enough batches to accumulate 16 requests before it can begin processing them.

MAB has significantly higher latency among the learning-based algorithms for both IATs. We find that ***MAB results in highly variable behavior under bursty load, often converging to sub-optimal configurations***. It also exhibits high standard deviation in latency under

900ms IAT due to the variable exploration routes it takes. Bayesian avoids the very high latency that MAB incurs, but still has higher cost than EcoGD.

Among the learning-based algorithms, EcoGD has the lowest cost and energy consumption for both bursty IATs. In fact, it has the lowest cost among all algorithms for 900ms IAT. It also has the second-lowest latency for both IATs among learning-based algorithms.

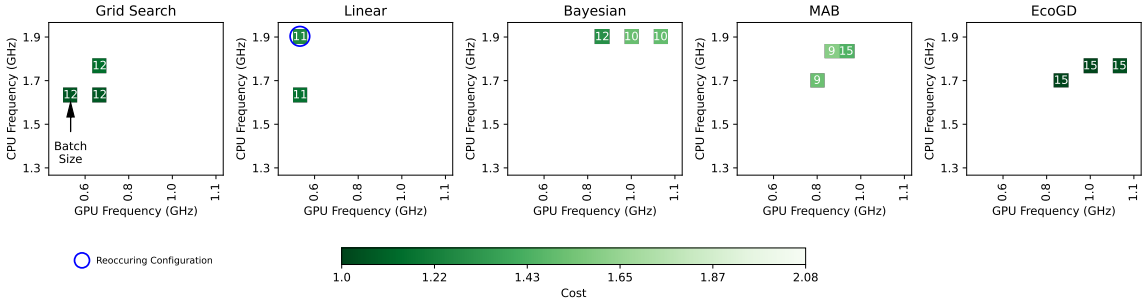


Figure 3.11: Heatmap of converged cost, shown as squares, achieved by all algorithms under bursty request inter-arrival time of 900ms between even batch of 10 requests. The axes indicate the CPU and GPU frequency for the configurations; the batch size is denoted as the numbers inside the squares. The color shading indicates converged cost (see colorscale), with darker squares indicating lower cost. Circles indicate re-occuring converged configurations across runs.

Figure 3.11 shows a heatmap of the converged cost achieved by each algorithm (see Figure 3.10) under the bursty 900ms IAT over the configuration parameters being optimized; we omit DVFS algorithms as their cost is quite high. The figure caption explains the axes and notations used. We see that different algorithms converge to different configuration neighborhoods. For example, Grid Search converges to lower GPU frequencies whereas EcoGD converges to higher GPU frequencies. Nonetheless, they both have low converged costs. This suggests that ***there are multiple local optima that could result in low cost***. Bayesian and MAB converge to smaller batch sizes, compared to EcoGD, which results in higher energy consumption as energy efficiency typically improves with batch size [155, 89].

Total cost till convergence, including overhead

Figure 3.12 shows the total energy consumed, latency achieved, and cost incurred by each algorithm until it converges to the best configuration; we show results under bursty inter-arrival times of 500ms and 900ms between a batch of 10 successive requests. All metrics for each algorithm are normalized by that under Linear Search.

Grid Search again performs quite poorly due to its very large search space. Linear continues to perform better than Grid Search. However, compared to fixed load (Figure 3.9), Linear does not perform as well in terms of total cost; for example, DVFS (with batch size 16) and EcoGD have lower cost than Linear for both IATs. *This suggests that a static algorithm like Linear is not well suited for bursty arrivals.*

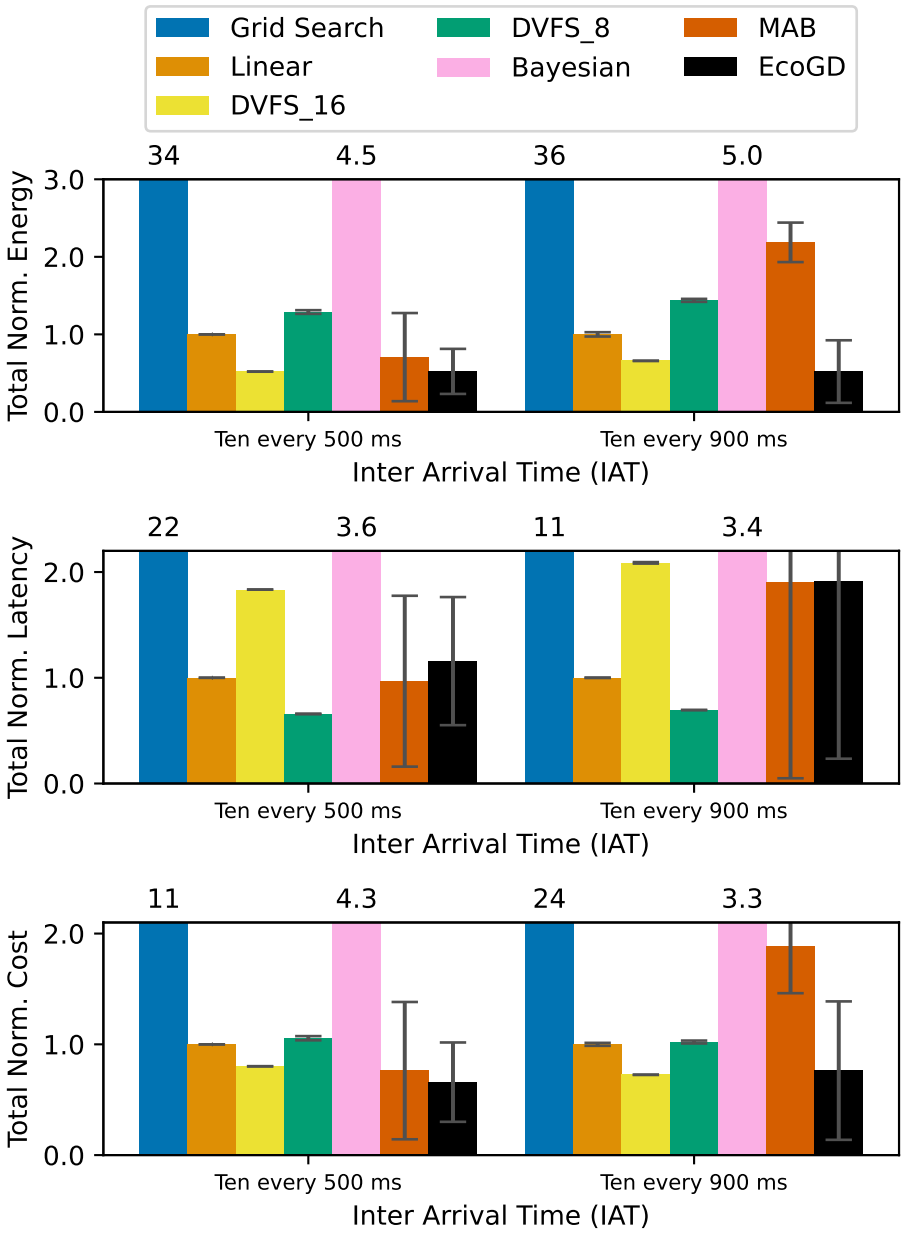


Figure 3.12: Total energy, latency, and cost with overhead incurred by all algorithms till convergence under bursty request inter-arrival times.

Among the DVFS configurations, DVFS with batch size 8 has higher energy than DVFS with batch size 16 but much lower latency than DVFS with batch size 16 and all other algorithms. This is because a single burst of 10 requests can easily satisfy the batch size 8 requirement, resulting in very low batch accumulation time. But, as discussed in Section 3.7.1, this comes at the cost of high energy consumption. Note that the cost values shown are based

on homogenized versions of energy and latency (see Section 3.5.3 for cost definition), and so a very low (pre-homogenized) latency does not immediately translate to low cost.

Among the learning-based algorithms, we again see Bayesian performing poorly, similar to the case of fixed IATs, due to its inability to converge to good configurations. MAB again has a lower cost than Bayesian, but higher than EcoGD.

EcoGD continues to achieve very low cost, even under bursty load. Specifically, EcoGD has the lowest cost under 500ms IAT and the second-lowest cost (only 5% higher than DVFS with batch size 16) under 900ms IAT. This is primarily due to its low energy consumption coupled with its reasonably low latency. In fact, EcoGD achieves the lowest energy consumption for both IATs (tied with DVFS with batch size 16 for 500ms IAT). This shows that EcoGD performs well for static and bursty conditions.

3.7.3 Key takeaways from synthetic load patterns

The performance evaluation of algorithms from the above subsections can be informally summarized as follows. Grid Search, not surprisingly, converges to low cost configurations, but is not a viable option in practice due to its high overhead. Linear can be sub-optimal and is not well-suited for bursty workloads. DVFS with batch size 8 has low latency but high energy consumption and cost, whereas DVFS with batch size 16 has lower energy consumption but higher latency and high cost. Among the learning-based algorithms, Bayesian has low post-convergence cost but high overhead. MAB has higher energy overhead, which impacts its total cost; it also converges to configurations with very high latency in bursty IAT conditions, hurting its post-convergence cost. EcoGD achieves low post-convergence cost with low overhead in both fixed and bursty load conditions, typically achieving the lowest total cost among all algorithms.

3.7.4 Performance evaluation under real-world traces

We now present our key performance evaluation results by comparing the dynamic algorithms on real-world request traces described in Section 3.6.2. One objective here is to evaluate how the algorithms perform under unpredictable and potentially abrupt changes in request arrival rate. Another objective is to study the impact on mean and tail metrics. We do not consider the static Grid Search and Linear Search algorithms in this subsection. Grid Search has very high overhead, as established in Figures 3.9 and 3.12; this overhead will be incurred much more frequently in trace-driven experiments, making Grid Search impractical. While Linear has lower overhead than Grid Search, its overhead and post-convergence performance are still sub-optimal, especially under bursty workload; this will get worse under dynamic traces.

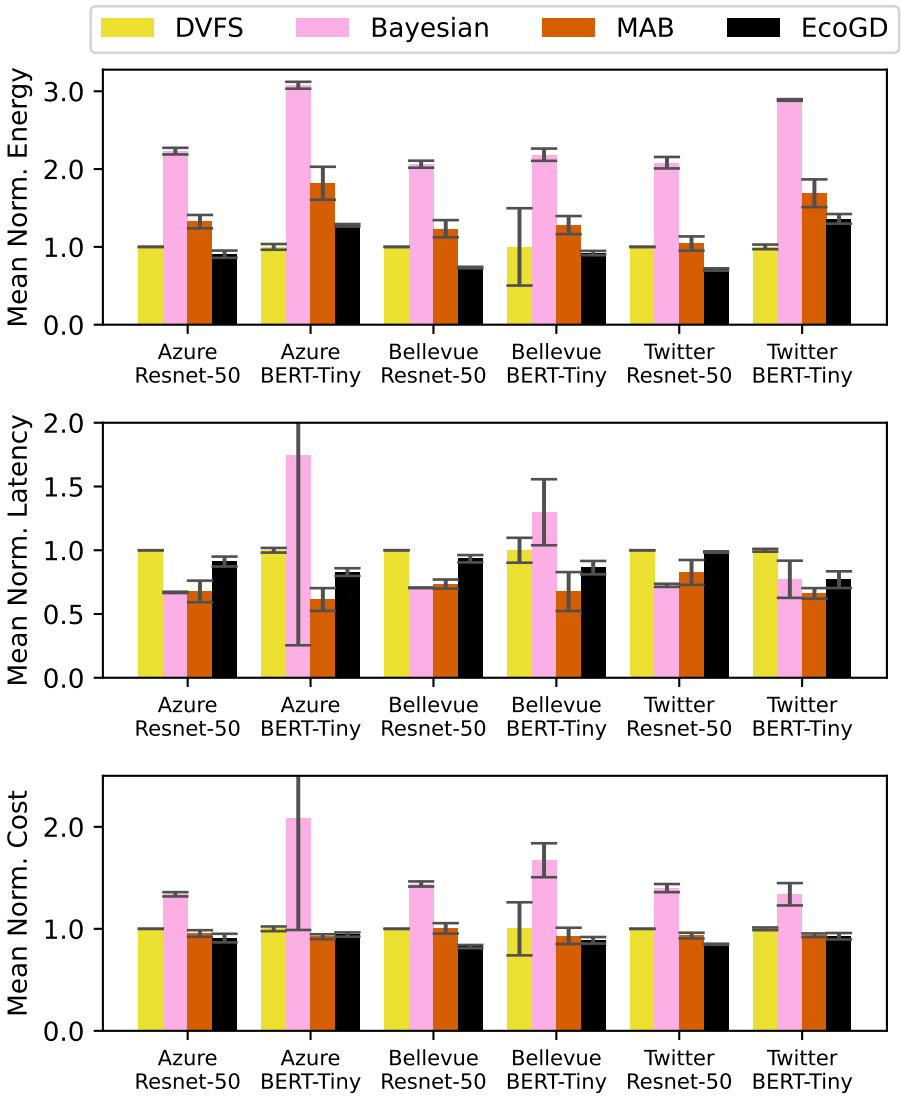


Figure 3.13: Mean energy, latency, and cost incurred by all algorithms under real-world request traces using Resnet50 and BERT-Tiny.

Comparison for mean metrics

Figure 3.13 shows the energy consumption, latency, and cost achieved by different dynamic algorithms, normalized to those achieved under DVFS, for the three different traces we consider using Resnet50 and BERT-Tiny. Since DVFS with batch size 16 outperformed DVFS with batch size 8 for the fixed and bursty load conditions, we only consider DVFS with batch size 16.

We start by again noticing the poor performance of Bayesian, which incurs very high energy consumption, resulting in its high cost. We find that Bayesian does not converge to good configurations, which is in agreement with our findings from prior subsections. Coupled

with the fact that the request rate changes frequently in the traces, ***Bayesian results in the highest cost*** in all six scenarios we consider.

DVFS maintains low energy consumption via its dynamic modulation of CPU and GPU frequencies. However, under the dynamic request traces, its fixed batch size contributes to high accumulation delay during periods of low request rate, resulting in high inference latency and (consequently) cost. MAB, on the other hand, is able to dynamically change the batch size and maintain lower latency compared to DVFS. However, it is unable to achieve the low energy consumption of DVFS, resulting in a somewhat similar (albeit slightly lower) eventual cost as DVFS.

EcoGD consistently achieves low cost across all scenarios by balancing energy consumption and latency. We see that EcoGD achieves the lowest energy among all algorithms for all Resnet50 scenarios. While it does not achieve the lowest energy for BERT-Tiny scenarios, it makes up for it with lower latency. On close inspection, we find that EcoGD has the lowest cost (which is our optimization metric) for all scenarios, except for Azure traffic with BERT-Tiny, where it has the second-lowest cost (2% higher than MAB). Averaged across all six scenarios, EcoGD reduces cost compared to DVFS, Bayesian, and MAB by 11%, 41%, and 6%, respectively. We find that the reduction is slightly higher for Resnet-50 scenarios (14%, 38%, and 10%, respectively).

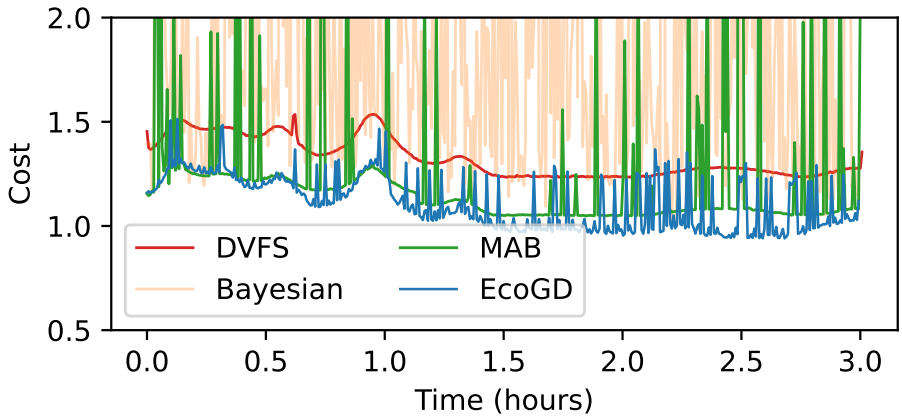


Figure 3.14: Cost timeseries of DVFS, MAB and EcoGD for Bellevue trace with Resnet50 model.

Analyzing the adaptive behavior of algorithms

To better understand the adaptive behavior of the different algorithms, we plot their cost as a function of time for the Bellevue trace with Resnet50 model in Figure 3.14. We see that DVFS exhibits consistently high cost, largely because of its inability to adapt the batch size. EcoGD almost always achieves the lowest cost but does exhibit spikes in cost, occasionally

resulting in momentarily higher costs than DVFS. These spikes are a result of our algorithm's configuration explorations as it attempts to look for better search spaces. MAB also exhibits such spikes for a similar reason, though the cost spikes quite significantly under MAB, often doubling its cost momentarily. This is because MAB has a mandatory exploration step, where it jumps to a random configuration from the entire search space, potentially resulting in a momentarily high cost. Bayesian (lightly colored to maintain visibility of other lines) can be seen to exhibit very variable cost behavior over time, with almost continual spikes. This is because Bayesian Optimization's expected improvement acquisition function is not designed to be adaptable to changing workloads [109]. As the load pattern changes under request traces, previously known configurations may become irrelevant, which Bayesian Optimization does not take into account.

The above observations provide insights into the workings of the algorithms and also highlight the consistently superior performance of EcoGD throughout the trace duration. We observe similar behavior for the other traces as well.

We also evaluated the performance of the algorithms under a synthetic trace where the inter-arrival time changed abruptly and immediately from 50ms to 90ms. We again found similar results with EcoGD performing the best (and obtaining 10% lower cost than the next best algorithm, MAB), suggesting that EcoGD is able to adapt to rapid changes in the workload. However, we note that the real-world traces we considered in Section 3.7.4 did not exhibit such abrupt changes.

Comparison for tail metrics

In real-world scenarios, it is often important to optimize for tail metrics, such as tail latency, in addition to mean metrics [46, 99, 45]. As such, we now analyze, in Figure 3.15, the 99th percentile energy consumption, latency, and cost achieved by different dynamic algorithms under all six trace-driven scenarios.

We see that the tail energy consumption under Bayesian and MAB is very high compared to DVFS and EcoGD; this was not the case for the mean energy consumption in Figure 3.13. We also find that the tail latency is high for Bayesian, especially for the BERT-Tiny model. MAB also exhibits high tail latency in some scenarios.

In terms of cost, Bayesian, as expected, has a high tail cost in all scenarios. MAB also results in a high tail cost, especially under the Bellevue request trace. This is because of MAB's probabilistic random exploration, which can result in high cost configurations being chosen over time. By contrast, ***EcoGD achieves the lowest 99th percentile cost for all scenarios, except for Azure traffic with BERT-Tiny, where it has the second-lowest cost (13% higher than MAB)***. Unlike MAB, EcoGD does not have a mandatory exploration phase

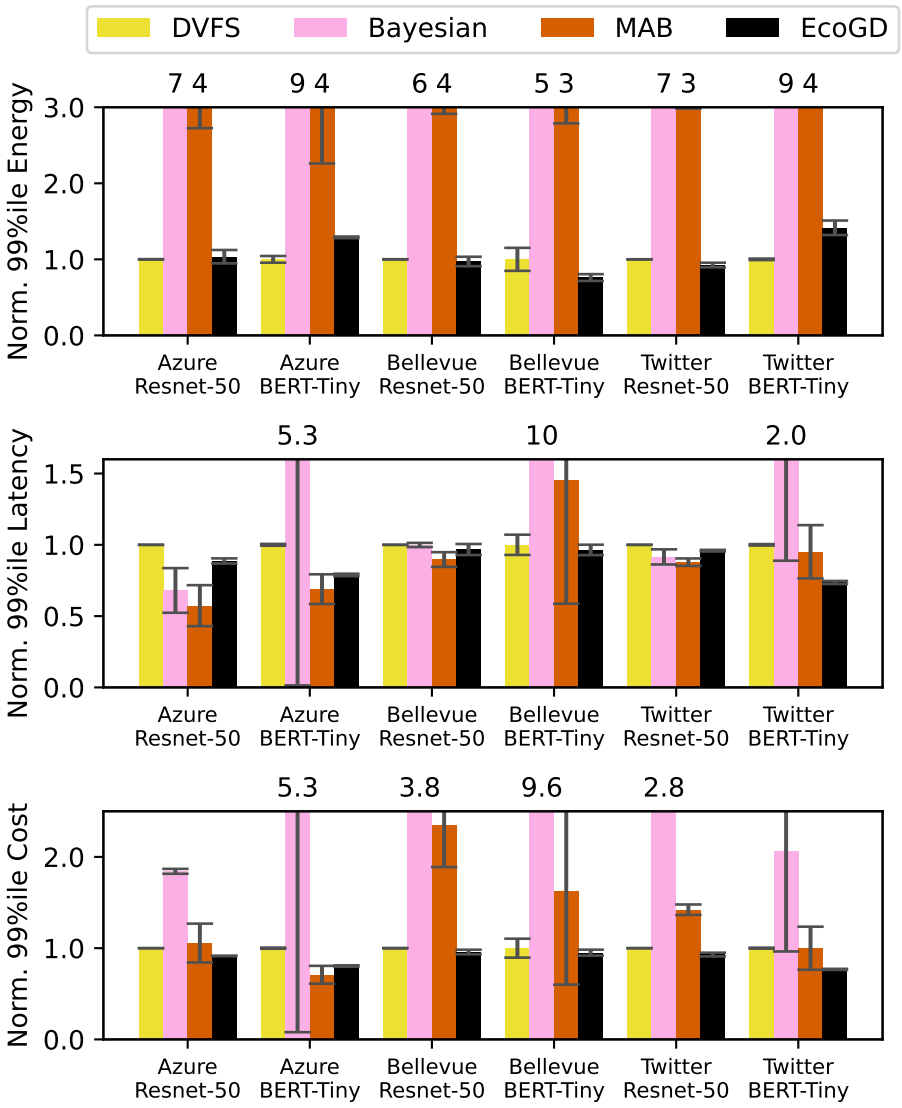


Figure 3.15: 99th percentile of energy, latency, and cost incurred by all algorithms under real-world request traces using Resnet50 and BERT-Tiny.

and instead can decide to either continue exploiting the best known configuration or explore a new one based on its knowledge of the neighborhood. This allows EcoGD to avoid high cost configurations. Averaged across all six scenarios, EcoGD reduces tail cost compared to DVFS, Bayesian, and MAB by 11%, 72%, and 27%, respectively. We thus conclude that EcoGD achieves low mean and tail cost, making it readily applicable in real-world scenarios.

We also find that different algorithms perform slightly differently depending on the DNN model being employed. For example, MAB has lower latency and higher energy consumption under BERT-Tiny compared to Resnet50 in Figure 3.13, whereas Bayesian experiences high tail latency under the BERT-Tiny model in Figure 3.15. This is because BERT-Tiny has

higher computational and processing requirements than Resnet50, resulting in different opportunities for optimization. However, the difference in performance for a given algorithm across traces is not that remarkable. This is likely because all real-world traces do exhibit similarly significant levels of load variations (see Figure 3.7). We do see slightly high tail latencies and costs under the Bellevue trace and slightly low tail latencies and costs under the Azure trace for MAB in Figure 3.15. This is likely because, among the traces we employed, Bellevue had the highest average load and Azure had the lowest average load.

3.8 Conclusion

This work focuses on dynamic and energy-efficient DNN inference on edge. We present EcoEdgeInfer, a framework that optimizes the tradeoff between inference latency and energy consumption by tuning hardware and software parameters dynamically and in response to changes in workload. Our core contribution is EcoGD, our adaptive parameter tuning optimization algorithm, inspired by Gradient Descent, that quickly converges to near-optimal configurations with low overhead while avoiding fluctuations in achieved cost. Evaluation results under three different request arrival traces and two different DNN workloads show that EcoGD consistently outperforms existing baselines, lowering the mean cost objective by as much as 55% (with 19% average reduction) and the tail cost objective by as much as 90% (with 36% average reduction).

Chapter 4

Network Optimization in Edge Clusters

Millions of video cameras are deployed globally across major cities for learning-based video analytic (VA) applications, such as object detection. Video streams from the cameras are either sent over the wide-area network to be processed by the cloud or are (at least partially) processed in a local edge workstation, incurring significant latency and elevated financial costs. In this work, to minimize reliance on the cloud and overcome the unavailability of high-compute workstations on edge, we investigate the use of heterogeneous and distributed embedded devices as edge nodes shared by multiple cameras to fully serve the video processing needs of a VA application (without requiring cloud support).

We present OVIDA, an edge-only orchestrator to deploy VA application(s) on a distributed edge environment to maximize accuracy. Given the resource-constrained nature of edge nodes, OVIDA disaggregates the VA application pipeline into multiple modules. OVIDA's core functionality and contributions are: (i) optimizing the placement and replication of the VA application modules across the edge nodes to maximize the throughput, and in turn, accuracy; and (ii) an adaptive model selection algorithm for VA modules based on accuracy-throughput trade-off to maximize accuracy in response to varying load conditions. To further improve performance, OVIDA employs a central-queue–based design (instead of the usual push-based design), which also obviates the need for complex load balancing algorithms. We implement OVIDA on top of Kubernetes and evaluate its performance for three VA applications, supported over a heterogeneous edge cluster under varying network conditions. When compared against several baselines in our evaluation, we achieve throughput and accuracy gains of at least 51% and 28%.

This chapter contains of work that was done in collaboration. Such works and their outcomes are either clearly marked in text or removed with a note.

4.1 Introduction

A significant rise in the deployment of cameras to support diverse applications in surveillance, health-care, transportation, and safety has led to the wide adoption of video analytics systems [53]. These systems use advances in computer vision and machine learning, fueled by the growth in edge and cloud computing paradigms. The global video analytics market is estimated to grow from $5 billion in 2020 to $21 billion by 2027, at a CAGR of 22.70% [18]. This level of growth necessitates efficient systems that can perform live Video Analytics (VA).

Traditionally, cloud computing has been favored due to the intensive computational requirements posed by VA applications using Deep Neural Networks (DNNs). However, this computatonal capability comes at a cost of significant latency, elevated financial costs, and privacy issues [165, 68, 137]. For example, there are 15,576 CCTV cameras in London Underground train stations; enabling live streams of these to the cloud would require significant capital and operational expenditures [19]. In contrast, the edge computing paradigm offers reduced latency and enhanced privacy by bringing data processing closer to the source [68, 56], albeit with weaker compute nodes. Edge computing is also beneficial for remote cameras that may not have a high bandwidth connection to the cloud and in settings where using the cloud is not economical [165].

Unfortunately, edge devices, such as Jetson Nanos, Jetson Orins [7] and Raspberry Pis [10], are significantly less capable than cloud servers in terms of processing power, memory size, and connectivity. Since DNNs have high memory and GPU requirements, a single edge node may not have sufficient resources to execute an entire VA application with satisfactory performance; in the VA context, performance is typically measured in terms of accuracy or throughput. In response, recent works have suggested breaking (i.e., disaggregating) the VA pipeline into its individual constituent functions (referred to as 'modules' in this work) and deploying them as microservices (or 'services') on multiple edge nodes [121, 163, 159, 162].

However, disaggregating VA pipelines across the edge poses four key challenges. (i) *Reduced data locality*: The input and output data sizes of VA modules in a pipeline can vary significantly, resulting in severe data transfer delays over diverse edge networks if modules are not placed appropriately. (ii) *Heterogeneity*: Modules may have different resource requirements (e.g., memory, GPU), and can have variable performance on heterogeneous edge nodes. (iii) *Underutilization of resources*: Due to disaggregation, each module has to spend a significant amount of processing time in I/O operations, which results in the underutilization of GPU resources. (iv) *Large model space*: Often, there are multiple models that can be employed to perform a single VA task on a specific compute node. These models,

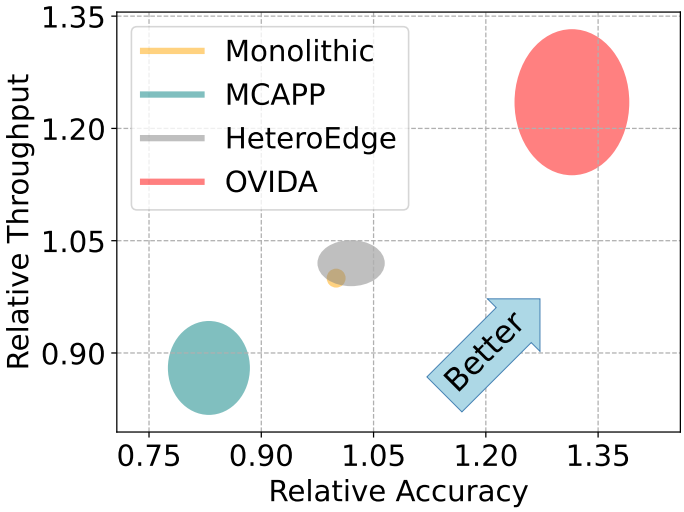


Figure 4.1: A summary of accuracy and throughput performance results, comparing OVIDA with other placement policies, normalized against Monolithic. The center of each ellipse represents the median, while the radius in each direction indicates the standard deviation.

however, can vary in resource consumption, service rates, and content-specific accuracy. Due to their differences, the models might react differently to real-world dynamism in arrival rate and frame content, making selecting the appropriate model an important challenge.

These challenges necessitate intelligent resource allocation, placement policies, and efficient model selection for a distributed edge. Unfortunately, existing solutions are either not optimized for multi-function pipelines[128, 32], assume that there will only be one neural network-based VA-application service [163], or are not able to optimally use the resources on an edge node [27]. Several works offer techniques to select the optimal model based on the given constraints [75, 125, 128, 21, 97]. Most of them depend on the inference of the "golden configuration" – the best possible configuration. Such an approach requires all or most models to be loaded into the memory simultaneously, which is infeasible in embedded devices such as smart cameras due to memory limitations.

This work presents the design and implementation of a distributed framework, OVIDA (**O**rchestrator for **VI**deo analytics on **D**isaggregated **A**rchitecture), for efficiently deploying disaggregated VA pipelines over network-connected heterogeneous edge nodes to maximize the throughput and accuracy of the VA application. OVIDA also features a lightweight pseudo-central queue-based mechanism that minimizes queuing delays, reduces per-frame latency, and obviates the need for complex load balancing in heterogeneous settings. OVIDA is enabled with a lightweight solution to dynamically choose and switch models on edge devices in response to changes in arrival rate and frame content.

OVIDA presents a complete deployment solution for distributed VA on the edge. We disaggregate the VA pipelines into stateless modules and make the following contributions:

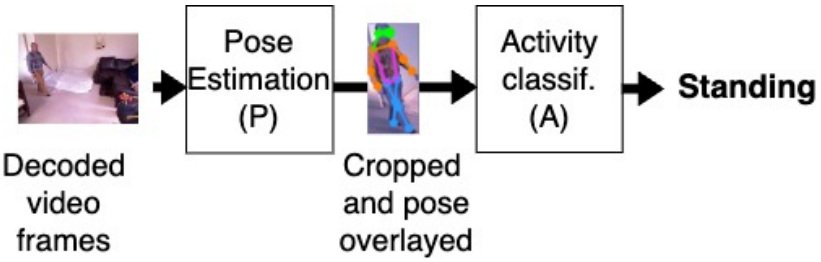
- *Replication and Placement:* OVIDA performs bottleneck function detection and horizontally scales bottleneck modules by deploying multiple instances. These instances are placed across edge nodes to maximize the end-to-end throughput (proportional to accuracy) of the VA pipeline, while optimizing resource utilization at each node.
- *Adaptive model selection:* OVIDA continuously estimates the best DNN model based on load–accuracy trade-off, leading to the best possible accuracy, without requiring to search through the entire space every time.
- *Load balancing:* We use insights from queuing theory to implement a lightweight central queue-based mechanism that minimizes queuing delays (leading to a lower per-frame latency) and obviates the need for complex load balancing.

We implement OVIDA over Kubernetes [9] using Docker [2] to containerize each module instance. We demonstrate the performance of OVIDA for three VA application pipelines, supported over a heterogeneous edge cluster under varying network conditions. We also compare OVIDA's performance against several baselines, including those from recent works [163, 27]. Based on experimental evaluations, we show that OVIDA's placement achieves up to 41% improvement in accuracy compared to the next-best placement policy. Figure 4.1 shows the accuracy and throughput of OVIDA compared to other baseline policies across various deployment scenarios (see Section 4.7.1 for experimental details). Combining OVIDA's placement and dynamic model selection further increases the accuracy gains by 15%.

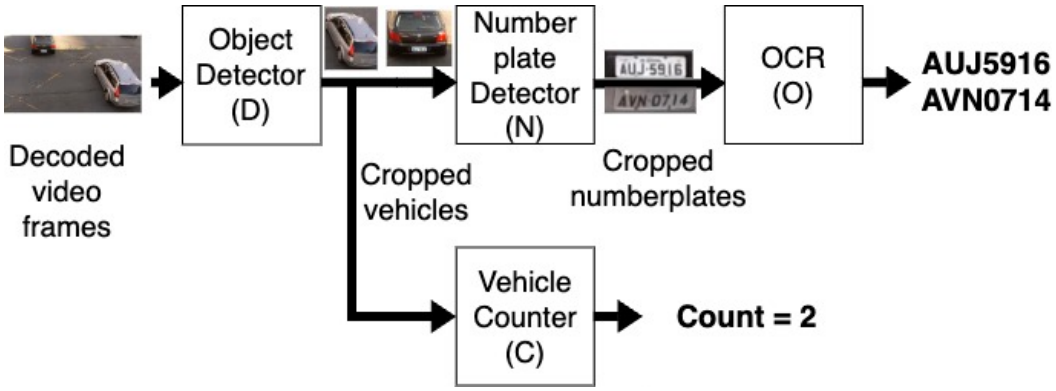
4.2 Background and Motivation

This section provides essential background on video analytics, edge computing, and the availability of diverse models for VA applications. It also motivates the need to maximize resource utilization of edge nodes and implement input-specific dynamic model switching to enhance accuracy.

Video Analytics (VA) refers to analyzing video content using computer vision techniques. A VA application typically consists of several vision functions, chained together to form a pipeline, where the output of one function is the input to the next. The VA pipelines used in this work for the evalution study are Activity Recognition (AcRg), Number Plate Recognition (NPR), and Vehicle Counting (VC), as illustrated in Figure 4.2. Note that NPR and VC have a common function – the object detector. The functions in VA pipelines may differ significantly in resources required (e.g., CPU, GPU, and memory) and their input/output data sizes.



(a) Activity recognition pipeline.



(b) Number plate recognition and vehicle counting pipelines; object detector is a common function for these two pipelines.

Figure 4.2: Illustration of the three VA pipelines implemented and evaluated in this work.

Edge Computing aims to reduce the bandwidth and latency requirements when compared to the cloud by using computing resources near the data source [137]. VA applications are excellent candidates for edge computing, as it eliminates the need to send large video content over the network to remote cloud servers. Moreover, processing data on hardware that is already owned and paid for by stakeholders such as cities, service providers, or the users, gives an economic and privacy incentive to use edge computing. However, edge-only systems do have some drawbacks when compared to a cloud data center; they have limited resource availability (e.g., memory and compute) and are typically heterogeneous in terms of computing and connectivity.

Since edge nodes are underpowered and have limited resources, it is essential to maximize their utilization. We conduct an experiment to investigate the opportunity to host multiple functions on a single edge node. Our key insight is to leverage the time spent on I/O operations by each function. As can be seen in Figure 4.3, the GPU is significantly underutilized when only a single instance is supported. This is due to high I/O overhead in a disaggregated pipeline—frame transfers over the network and unavoidable hardware limitations such as copying an image to GPU memory before it can be processed. However, when two module instances (or replicas) of the bottleneck service are placed together, they work in parallel

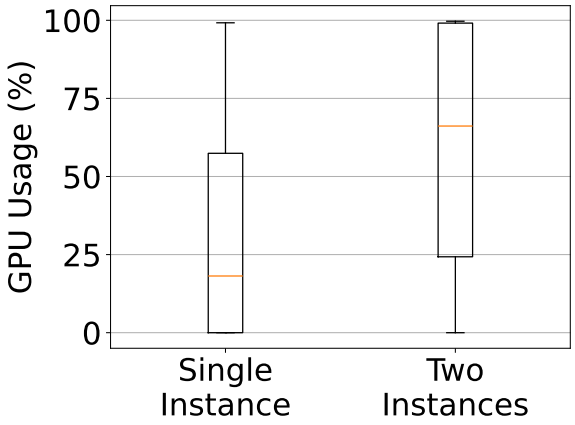


Figure 4.3: Improved GPU utilization obtained by placing multiple bottleneck modules on an edge device.

and share the GPU, resulting in much higher ($\sim 3.6\times$) GPU utilization due to interleaving. Further, deploying two instances of the bottleneck module on the same device under the same load nearly *doubles* the achieved throughput. Thus, multiplexing DNN-based modules over a GPU-enabled embedded edge device can improve resource utilization and overall pipeline throughput and accuracy. Prior works, however, do not exploit this feature and only deploy a single pipeline module per application on an edge device [27].

Diverse model implementations. A given VA task can be serviced at varying levels of performance by different model implementations. In particular, the computer vision community offers a variety of DNN models with different levels of accuracy, inference times, and resource consumption for performing the same tasks. For example, models like YOLO [77], which are used for detection, pose estimation, classification and other applications are available in multiple sizes (e.g., nano, small, and medium) and differ in their accuracy and resource consumption because of differences in their number of model parameters. At low load, a simpler and faster model (such as a nano variant) might perform worse than a more powerful model (such as a medium variant). However, as load increases, for example, due to additional video streams, the medium model variant will drop more frames due to slower inference times. Subsequently, it may end up performing worse than the nano model, which may not have to drop any frames due to its lightweight needs.

Prior works have proposed techniques to select the best possible model to perform a VA task in response to changes in input [125] and required quality of service [128, 75] (see Section 4.8 for a discussion of related works). All of these require all models to be pre-loaded into the memory for a periodic comparison. Unfortunately, this is not possible in memory-constrained embedded devices.

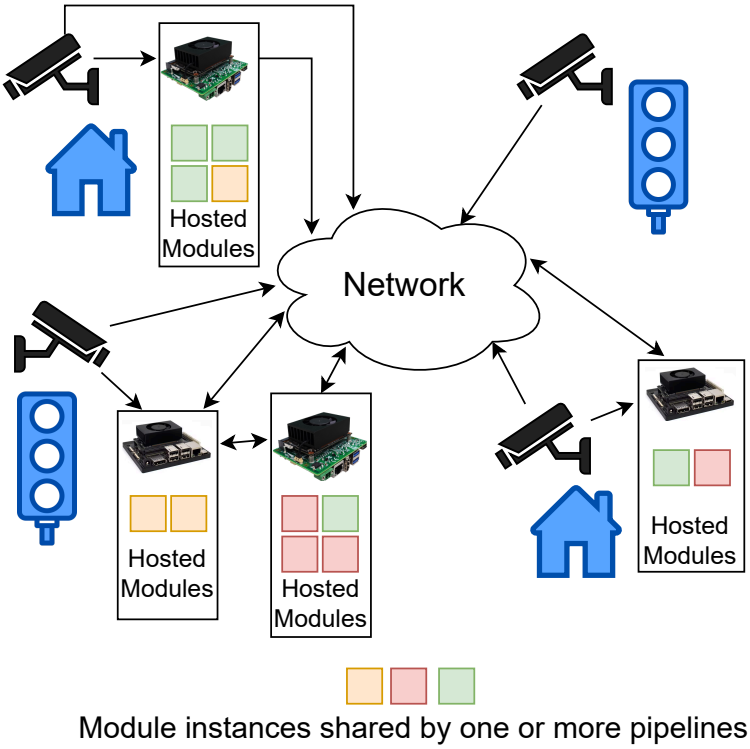


Figure 4.4: Overview of distributed edge devices featuring multiple VA pipelines. Each colored box represents a VA module hosted on the edge device. The modules are replicated as needed.

4.3 System Design and Implementation

This section discusses the fundamental system design decisions behind OVIDA, including the disaggregation of the VA pipeline (Section 4.3.1) and the central queue-like architecture (Section 4.3.2). We discuss the key contributions of OVIDA— module instance placement and model selection—in Sections 4.4 and 4.5, respectively.

4.3.1 Microservice-based distributed pipeline

The functions in a VA pipeline can either all be placed on a single device in a monolithic fashion or across multiple devices in a distributed fashion. A monolithic deployment has the following major drawbacks, especially when deployed on the edge: (i) Each edge node may lack sufficient memory to host all required function models (note that our choice of pipelines and edge nodes still allows us to evaluate monolithic deployment in Section 4.7); (ii) New frames start processing only after the current input has been fully processed through the pipeline; (iii) The architecture cannot scale effectively with the bottleneck function of the pipeline.

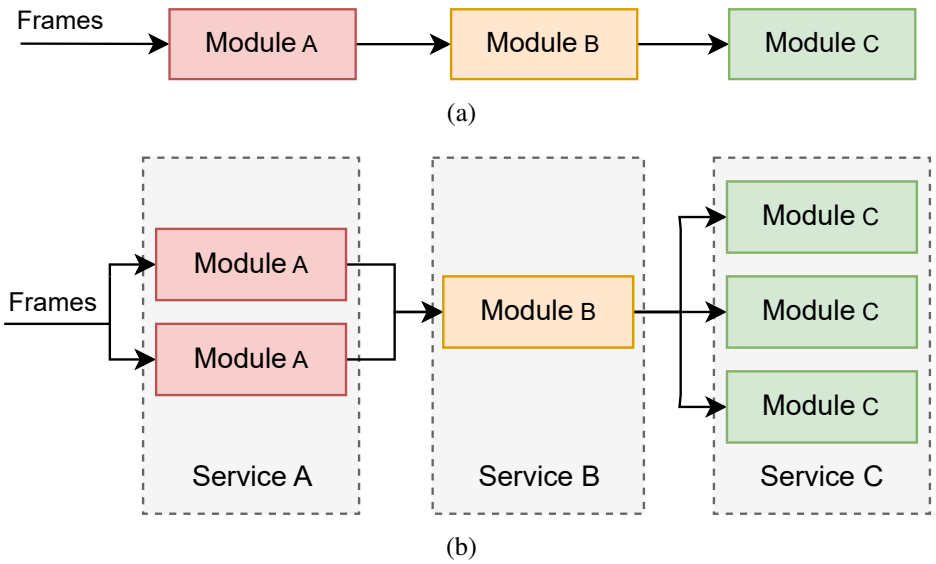


Figure 4.5: Example illustrations of (a) a pipeline with 3 modules and with one instance each, and (b) the same pipeline with 3 modules but with multiple instances for modules A and C.

To avoid the limitations of a monolithic deployment on the edge, we design a *distributed pipeline* that splits a VA application into multiple functions, where each function works independently. OVIDA packages each VA pipeline function as a microservice (or service), and each service may include one or more replicas of containerized functions called 'modules' (see Figure 4.5(b)). Module instances (or replica) can be distributed across network-connected nodes as needed, avoiding resource limitations (e.g., memory availability, illustrated in Figure 4.4). The distributed design also provides another key benefit—*scalability*. Since edge devices are resource-constrained, the throughput of a specific module may be limited; by horizontally scaling a module using multiple instances, we alleviate bottlenecks. In contrast, a monolithic deployment requires replicating the entire pipeline, not just the bottlenecked modules. However, a drawback of disaggregating the pipeline is the *additional I/O costs of transferring frames between the modules*. With careful module placement (Section 4.4), our evaluation in Section 4.7 shows that this cost can be minimized, and is significantly outweighed by its benefits.

Leveraging the horizontally scaled instances of a module requires careful resource allocation among them to avoid processing and queueing delays. Each module instance can have varying throughput based on the compute power of the edge node, the other modules sharing the node, and the current input load. Thus the resource availability at an edge node can change dynamically and abruptly. We next discuss how OVIDA's architecture design is inherently tailored to handle such dynamic scenarios.

4.3.2 Pseudo-central queue design

In general, distributed applications employ a "push" based architecture, where input data, such as frames or objects, are pushed from the source to the servers (or module instances, as in our case); such push-based architectures have been employed in recent VA works, such as Distream [159], VideoEdge [71], and Hetero-Edge [163], among others. This architecture works well if there is only one instance (as shown in Figure 4.5(a)) or if each of the multiple instances of a module has the same processing rate. However, this is not true in heterogeneous edge-based VA systems where different instances may have different resource capacities and availability (due to CPU and GPU sharing), and thus different processing rates. Consequently, an instance with a lower processing rate must either maintain a possibly ever-increasing queue or drop some frames. On the other hand, an instance with a higher processing rate might idle.

To address the issue of heterogeneous processing rates, OVIDA employs a *central queue implementation*, which is pull-based instead of push-based. Under this implementation, an instance pulls frames to process only when it is ready to process them, i.e., once it has completed processing the previous frames. Queueing literature informs us that a central queue-like policy (single queue, multiple servers, such as $M/M/k$) provides significantly lower frame processing time than a round-robin, push-based policy [64, Chapter 14.4]. This is because, with a push-based policy, incoming jobs (or frames) can get stuck behind a slow job at an instance; this is not the case with a pull-based central queue.

An idealized central queue concept assumes that there is no communication delay between the central queue and the module instances. In real edge deployments, however, network link delays are not negligible. To minimize these delays in practice, OVIDA makes use of buffers that we call entry and exit queues, as explained below.

Design details

OVIDA amortizes data transfer delays by pulling multiple frames (equal to the buffer size of the entry queue) from the central queue to an idle instance. We refer to this architecture with the buffers as a *pseudo-central* queue.

Figure 4.6 shows the major components of OVIDA's pseudo-central queue design.

- *Entry and Exit Queue* are the shared data structures that serve as buffers among API Server, Fetcher, and Function.
- *API Server* runs an HTTP server and provides API for receiving frames and handling frame requests. The received frames are stored in the entry queue ③. The requested frames are sent from the exit queue ⑥.

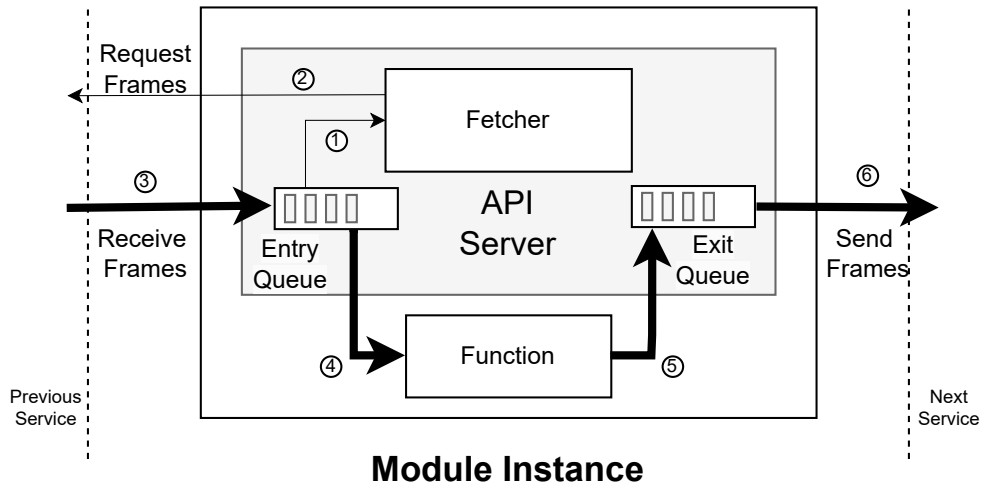


Figure 4.6: Module instance in pseudo-central queue design.

- *Fetcher* runs on a separate thread inside the API Server. If the entry queue is not filled ① and has n empty buffer spaces, it requests n frames to be sent from a module instance of the previous service (using the latter's Server Process API) ②. The Fetcher keeps requesting frames from instances of the previous service (see Figure 4.5(b)) in a round-robin fashion until the entry queue is full.

- *Function* runs the main task of processing a frame. It pops the unprocessed frame from the entry queue ④, processes it, and pushes it into the exit queue ⑤. If the exit queue is filled up to a set limit (generally shallow), the worker process will stop popping from the entry queue and cease the Fetcher. For the first module in the pipleline, if the entry queue is full, incoming frames from the frame decoder are dropped until there is space available in the entry queue.

We evaluate the impact of the pseudo-central queue on a distributed and heterogeneous deployment of a Number Plate Recognition pipeline (more in Section 4.7) and find that the pseudo-central queue design achieves as much as 4× lower latency compared to push-based under high loads. As such, we employ the pseudo-central queue in our implementation of OVIDA (and all baselines, see Section 4.6.3).

4.4 Placement and Replication

The algorithms discussed in this section are not my original work. They were developed by my collaborators. They are left here for completeness.

In general, the throughput of a VA pipeline can be increased by: (1) using faster edge nodes, (2) optimizing the utilization of the available edge nodes, or (3) employing a faster

variant of the DNN used in a module. Unlike the cloud, upgrading to a faster edge node on the fly (option 1 above) is not practical due to the limited availability of edge devices. In this work we explore the other two options. We first discuss option 2 in this section. This is aimed at improving the utilization of available resources via careful model placement and replication. In Section 4.5, we will follow up with option 3—dynamic switching between faster-but-less-accurate and slower-but-more-accurate versions of DNNs.

Recall that our objective in this work is to *maximize the accuracy* of the pipelines deployed on distributed, diverse edge devices. Higher throughput leads to increased accuracy by reducing the number of dropped frames [21, 112]. Frame drops primarily come from an imbalance of processing rates at different stages of the pipeline, giving rise to excessive queuing at one or more stages. This can be alleviated by a 'bottleneck removal' approach. We iteratively replicate and strategically place the *bottleneck modules*. Bottleneck modules are those modules generating the highest delay of all modules in the pipeline, where the delay of a module is the sum of the network transfer delay plus the processing time for that module. While we do not directly optimize latency, it is achieved indirectly by identifying and alleviating the bottleneck. To prevent queue buildup, the queue size is kept very small, thereby capping maximum queuing delays. The proposed strategy must be executed on already resource-constrained edge devices, and therefore, it must have a low overhead. So, we look for low-overhead, fast heuristics that work well in practice.

Identifying the bottleneck module(s) is important before replicating it and determining optimal placement. A key challenge in a dynamic setting is that the *bottleneck may shift* with load or with a change in deployment. For example, consider the pipeline shown in Figure 4.5(a). For simplicity, assume each new instance of a module has equal throughput. If the inference time per request for each module is in the ratio $2 : 1 : 3$ (corresponding to Module A, B, and C), then Module C is the bottleneck. Adding a new instance of Module C and distributing the load equally changes the ratio to $2 : 1 : 1.5$, shifting the bottleneck to Module A. There are also other challenges. The bottleneck depends also on the actual placement of module instances, even if the number of module instances remains unchanged. For example, the inference time ratios in the above example may depend on the actual devices the modules are located on and their resource availability, which in turn depends on hardware configurations and workloads. Overall, instead of trying to achieve a global optimal over all possible replications and placements which is an np-complete problem [41], we use a simple iterative approach based on bottleneck analysis that we will describe in the following subsections.

Table 4.1: Notations

E	The set of all available edge devices
$\tau_{m,e}$	Mean processing (service) time of module m on edge device e
λ_k	Fraction of aggregated (input) arrival rate of all frames over all pipelines that is attributed to pipeline k
$d_{m,e}$	Mean network delay experienced by module instance m on edge device e
$f_{k,m,e}$	Fraction of frames of pipeline k processed by module m on edge device e

4.4.1 Bottleneck Analysis

We define our problem as minimizing the 'cost' of the bottleneck module while keeping GPU and memory usage for all modules within predetermined bounds to avoid resource contention. These bounds are set by available edge resources.

The cost of a module instance is the sum of network delay d for data transfer from the previous module and the inference time τ of the module instance, given the current placement choices being considered. This cost is scaled by the fraction of frames sent to the module instance computed over all frames arriving at the entire system over all active pipelines (similar to the use of 'visit ratios' in classical bottleneck analysis.) This scaling factor depends on respective arrival rates of different VA pipelines, the number of pipelines sharing a specific module and the number of instances of the module used. The notion of cost above thus signifies the 'service demand' on the module.

Similar to other works [128, 21], we perform offline profiling to determine the inference time for each module for each edge device. We similarly use profiling to estimate the amount of data transferred between modules to estimate the network transfer delay. We use a simple estimate of the network bandwidth to compute the delay. For infrequent changes, such as adding a new camera with a different stream resolution, offline profiling and subsequent placement can be redone. For more real-time changes in arrival rate and frame content, we propose dynamic model switching (Section 4.5) that is not captured in the solution approach we discuss here.

4.4.2 Iterative Approach

The necessity to solve replication and placement problems jointly while being mindful of heterogeneity makes this problem challenging. Besides, using compute-intensive optimization methods such as mixed integer linear programming (MILP) is not feasible in resource-constrained edge devices, encouraging the need for lightweight heuristics.

OVIDA uses an iterative approach for bottleneck identification and removal. An initial placement is iteratively improved by first identifying the bottleneck module and then replicating that module and placing the new module instance, assuming the needed edge resources are available to accommodate the new instance. This in turn changes the costs of some or all modules as the service demand changes due to changes in visit ratios, and speficially reduces the bottleneck cost. The iterations continue until a *saturation point* is reached, i.e., no improvement in the bottleneck cost is possible or no further resources are available to replicate and place new instances. Note that the algorithm reduces the bottleneck cost monotonically, thereby guaranteeing convergence.

Initialization

Algorithm 2 Initialization

```
1:  for s ∈ S do
2:      cost ← ∞
3:      node ← None
4:      m ← new module instance of service s.
5:      for e ∈ E do
6:          lowestcost ← d[m][e] + τ[m][e]
7:          if lowestcost < cost then
8:              cost ← lowestcost
9:              node ← e
10:     Place e ← m
```

To initialize the pipeline, OVIDA iterates through all services in the order of their position in the VA pipeline, placing one module instance per service. Iteration in the order of positon in the pipeline is critical because the networking cost calculation for each module depends on the placement of modules in the previous services. Each module is placed on the edge node with the lowest cost for that specific module. This initialization algorithm is illustrated in Algorithm 2.

Bottleneck detection and interative replication and placement

Algorithm 3 describes the iterative approach mentioned before. For notational convenience we assume that the same module is not replicated multiple times on the same edge device.[1] A data structure in the form of a cost matrix is maintained — $cost(m,e)$ that presents the cost when a module instance m for service s is placed on edge device e. The visit ratios are

[1]This is only to simplify presentation, the algorithm implemented and evaluated can handle any possibility.

also maintained in a similar form $visit_ratio(m,e)$, presenting the fraction of the aggregated arrival frame rates seen by this instance. Note that $visit_ratio(m,e) = \sum_k \lambda_k f_{k,m,e}$, where λ_k is the fraction of total arrival frame rates attributed to the VA pipeline k that uses this instance and $f_{k,m,e}$ is the fraction of the frames of the pipeline k processed by this instance. $f_{k,m,e}$ is determined by the algorithm.

$$cost(m,e) = visit_ratio(m,e) \times (d_{m,e} + \tau_{m,e}).$$

Algorithm 3 Placement and replication

```
1:  (m,e) = arg max cost[][] /* Identify the bottleneck */
2:  e_c = φ; lowestcost = ∞
3:  for e' ∈ E do
4:      Copy cost[][] to cost'[][]
5:      Copy visit_ratio[][] to visit_ratio'[][]
6:      if enough remaining capacity on e' to accommodate m then
7:          /* Check whether the bottleneck can be removed*/
8:          Update visit_ratio'[m][e] for all (m,e) for
            which cost'[m][e] is defined, assuming a
            new instance of m is now mapped on e'.
9:          Recalculate cost'[][] based on updated
            visit_ratio'[][]
10:         if cost'[m][e] < cost[m][e] and max cost'[][] < max cost[][] then
11:             /* This choice removes the bottleneck*/
12:             /* Remember the choice */
13:             if max cost'[][] < lowestcost then
14:                 e_c = e'; lowestcost = max cost'[][]
```

Algorithm 3 presents one single iteration. In each iteration, the largest element (m,e) of the cost matrix represents the current bottleneck. An additional instance of the bottleneck module m is created. A check is performed to see whether this new instance, when placed in edge node $e' \in E$, alleviates the bottleneck, assuming that such placement is at all doable (i.e., e has enough remaining capacity). For each check, the visit ratios are (temporarily) updated by recalculating $f_{k,m,e}$, for all e where an instance of the module m is already placed (including the new e'). For simplicity, $f_{k,m,e}$ is kept in inverse proportion to the delay $(d_{m,e} + \tau_{m,e})$. After these checks are performed, the best option is selected (e_c in the Algorithm). The best option is the one that reduces the cost of the bottleneck and any new bottleneck (to be treated again in the next iteration) has a lower cost than before.

Once the best option is determined, the visit ratios and cost matrix are updated (not directly shown in the Algorithm). At this point, the estimated GPU and memory usage of

the chosen edge node e' are adjusted based on the offline profiled GPU and memory usage. Thus, the remaining capacity of e' is available for the next iteration.

The iterations continue until there is no further decrease in the bottleneck cost or a new module instance cannot be created/placed due to resource saturation. This heuristic approach results in higher throughput and accuracy compared to previously known approaches, as demonstrated in Section 4.7. The replication and placement algorithm runs when a VA application or edge device arrives or departs. However, in real scenarios, such events are rare, and the computation overhead is amortized over the system's total runtime.

4.5 Adaptive Model Selection

This section is not my original work and hence, completely removed from the thesis. For the sake of completeness, please refer to the brief description below.

We develop a technique for model selection that dynamically adapts to changes in arrival rate and frame content. Using the profiled inference times and their impact on pipeline accuracy for all models, we estimate the relationship between arrival rate and accuracy. When the pipeline is operational with live traffic, each module periodically assesses its arrival rate and checks for any changes in frame content. Changes in frame content, such as day, night, and camera out-of-focus conditions, can be detected using low-overhead classical computer vision methods [96, 116]. If a model other than the current one is predicted to have better accuracy, it is loaded in the background. OVIDA switches to this new model when the model is loaded and ready, with very minimal disruption to the pipeline processing. Note that memory limitations can be managed during the loading of the new model by employing memory swap, albeit with a temporary drop in performance.

Please refer to the original work for more details: OVIDA [134].

4.6 Methodology and Experimental Setup

4.6.1 Experimental Testbed

Our heterogeneous compute testbed consists of five edge compute devices; two 16 GB NVIDIA Orin NXes (orin nx), two 8 GB NVIDIA Orin Nanos (orin nano), and one 4 GB NVIDIA Jetson Nano (jetson nano) [7]. The orin nanos and orin nxes acted as edge nodes and the jetson nano was used for streaming. To further increase heterogeneity, orin nxes were set to 10-watt and 25-watt power profiles each, while orin nanos were set to a 15-watt power profile. All devices were connected over a switched LAN with a peak throughput of

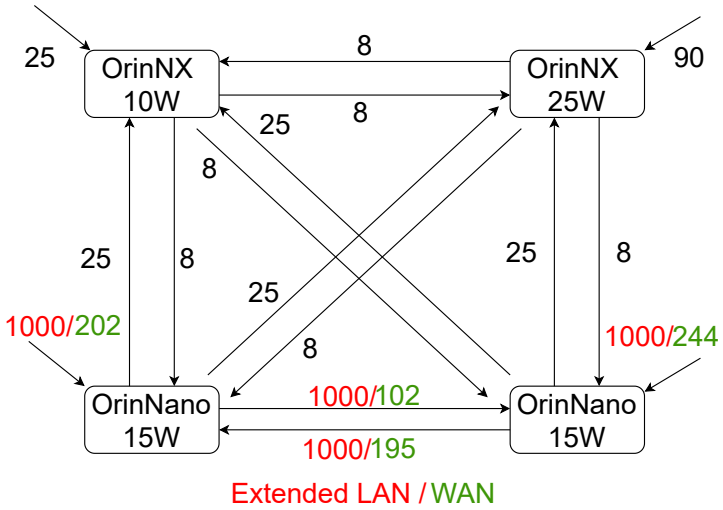


Figure 4.7: Illustration of the WAN and Extended LAN network topologies we experimented with.

1 Gbps and a link delay of less than a millisecond. We emulated various limited bandwidth network conditions using tc packet filtering [14]. We used Kubernetes (K8s) [9] to connect and manage the cluster of edge devices through a virtual network. Each function was containerized using Docker [2]; K8s manages the deployment and networking of these containers. We considered varying network topologies, load conditions, and deployment types for evaluations.

Network Topologies

We used three different network topologies: local area (LAN), wide area (WAN), and a combination of the two (Extended LAN). For LAN, we do not limit the network in any way. For WAN, download and upload speeds range from 25–244 and 8–195 Mbps, respectively. Similarly for Extended LAN, download and upload speeds range from 25–1000 and 8–1000 Mbps, respectively. Upload and download speeds for WAN and Extended LAN are illustrated in figure 4.7. Differences between the two are highlighted in red and green, while identical links are shown in black. The selection of link delays and bandwidth was motivated by the Ookla speed test report for fixed networks [17].

4.6.2 VA Application Pipelines

To evaluate our contributions, we implemented three VA pipelines: (i) Number Plate Recognition (NPR), (ii) Vehicle Counting (VC), and (iii) Activity Recognition, (AcRg).

The NPR pipeline consisting of three unique DNN based services: object detector (D), number plate detector (N), and optical character recognition (O). The object detector detects

vehicles in frames. It forwards the cropped images of the detected vehicles to the number plate detector module(s), which then detects number plates in the cropped images. The detected number plates are further cropped and forwarded to the optical character recognition module(s), which recognizes the characters in the number plate images and saves the results. The DNN models used are YOLOv5-medium [76] for the D modules and YOLOv4-tiny [34] for the N and O modules. We use the unique number plates detected as the accuracy metric for NPR pipeline. Due to the absence of a ground truth in the dataset employed, we consider the set of number plates detected when no frames are dropped ("golden configuration") as the ground truth.

The VC pipeline consists of DNN-based object detector (D) and a counter (C). The object detector service is the same as the one used by the NPR pipeline. The counter counts and reports the number of vehicles detected by the object detector. We use the number of cars detected as the accuracy metric.

The AcRg pipeline, inspired by the work of Huang et al. [70], consists of two DNN-based services: pose-estimator (P) and activity recognition (A). Pose-estimator detects people in the image and generates cropped images with pose overlaid on them. These cropped images are sent to the activity recognition module(s), which classifies the activity in the cropped images. The pose-estimator uses yolov8-pose variants (medium (*m*), nano (*n*), extra nano (*xn*)) [77], trained on the coco dataset [91]. The medium and nano variants are off-the-shelf models whereas the extra nano variant was trained ourselves with a lower depth and width parameters compared to the nano variant. We fine-tuned two more variants of yolov8m-pose (*md* and *mb*) for pose estimation under specific conditions: (i) low light, and (ii) out-of-focus frames. This was achieved by reducing the gamma value and adding Gaussian blur to the training images from the coco dataset for each respective variant. The activity recognition uses a yolov8-cls nano [77] model fine-tuned on the training set of the fall dataset [20]. Since we have access to the ground truth, we use F1-score as the accuracy metric for the AcRg pipeline.

Figure 4.2 illustrates the three pipelines discussed above. All models used were converted to TensorRT [8] format, which improves inference throughput without affecting accuracy.

Datasets

We used the VehicleRear dataset [44] which provides videos captured at 30FPS with 1920x1080 resolution to evaluate the NPR and VC pipelines. To evaluate the AcRg pipeline, we used the fall dataset [20] and compiled a single video from all the images in the dataset.

Load Conditions

The load for the VA pipelines was varied by adjusting the number of simultaneous streams processed by the system, ranging from 1 to 3. Increasing the number of streams beyond 3 resulted in impractically low accuracy due to the limited performance of the edge devices.

4.6.3 Baselines

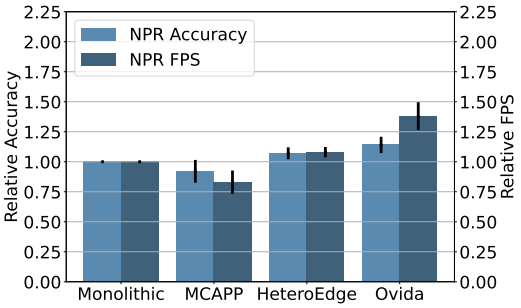
We compare OVIDA against the following baselines:

- *Monolithic* is the baseline deployment, where there is no disaggregation of VA pipeline, wherein the entire pipeline is implemented as a single code block on a single compute device. Unlike a disaggregated deployment, there is no parallelism and an input frame is processed completely through the pipeline before a new frame can start processing. Each stream has its own monolithic module and there is no sharing of frames among them.
- *MCAPP* is a solution framework for the multi-component application placement problem proposed by Bahreini et al. [27]. The placement is done in two steps: (i) Matching the modules and edge nodes using the well known Hungarian matching algorithm [86]. The cost for each combination of edge node and a module is the sum of service cost and the communication cost between the user and the edge node. (ii) Considering inter-component communication costs to make swaps in module-node mappings such that the overall cost of the pipeline is reduced. The number of instances of each module deployed corresponds to the number of streams handled by the deployment. While MCAPP only allows a single module per edge device, we enhance its performance by allowing the sharing of an edge device via modules processing separate video streams.
- *HeteroEdge* is a placement policy based on Hetero-Edge's fast heuristic [163] that places the modules on edge devices with GPU usage less than a predefined threshold. Although the original heuristic does not support multiple DNN-based modules, we made minor tweaks to allow that. The policy traverses over the module instances in pipeline order for placement (for example, D-N-O order in NPR pipeline). Similar to MCAPP, the number of module instances corresponds to the number of streams handled by the deployment.
- *OVIDA* employs the placement policy from Section 4.4.

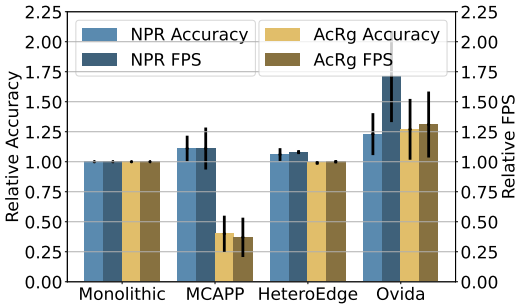
To ensure a fair comparison of placement policies, we use central-queue–based load balancing (see Section 4.3) for all baselines in our evaluation.

4.7 Evaluation

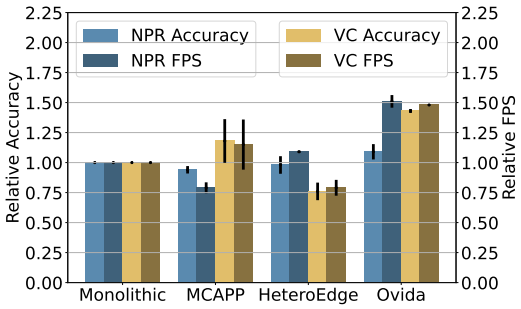
We evaluate the effectiveness of OVIDA by measuring performance and resource consumption via our testbed deployment. We first evaluate the two main components: OVIDA's placement heuristic and adaptive model selection, followed by an end-to-end evaluation that combines the two.



(a) Single pipeline.



(b) Multiple pipelines without common modules



(c) Multiple pipelines with common modules.

Figure 4.8: Performance comparison between OVIDA and other policies under different deployment scenarios. All metrics are normalized with respect to Monolithic.

4.7.1 Placement Evaluation

Figure 4.8 shows the median achieved throughput (in FPS) and VA application accuracy under various conditions (see Section 4.6) for three deployment types: (i) single pipeline (NPR pipeline), (ii) multiple pipelines without a common module (NPR and AcRg pipelines), and (iii) multiple pipelines with common modules (NPR and VC pipelines). The plots are normalized using *Monolithic* as the baseline, with all reported metrics divided by the baseline values. In the case of multiple pipelines with common modules, *OVIDA* has the ability to share common modules, whereas other policies do not.

We see that *MCAPP* performs poorly, achieving almost the lowest FPS and accuracy compared to other policies in all three deployments. Deploying two pipelines without common modules (Figure 4.8(b)) results in a 60% lower accuracy for the AcRg pipeline and a minimal improvement of 10% for the NPR pipeline compared to *Monolithic*. This is to be expected as *MCAPP* only places one module instance per stream on an edge device, ***thereby incurring communication cost every time a frame is transferred between module instances.***

HeteroEdge performs slightly better than *Monolithic* for single pipeline and multiple pipelines without common module deployments. Unlike *MCAPP*, *HeteroEdge* is able to place multiple module instances on a single device (until GPU or memory saturation), resulting in lower inter-module communication delays. ***Due to the parallelism of a disaggregated pipeline, HeteroEdge outperforms Monolithic*** by 7% in terms of accuracy in the best case (Figure 4.8(a)). ***The unbalance caused by saturating a device before moving on to the next one, overshadows the gains of parallalism and results in performance worse than Monolithic in case of pipelines with common modules (Figure 4.8(c)).***

OVIDA outperforms *Monolithic*, *MCAPP*, and *HeteroEdge* in all deployment cases. In the case of single pipeline deployment (Figure 4.8(a)), *OVIDA* achieves 14% higher accuracy and 38% higher throughput compared to *Monolithic*, and 7% higher accuracy and 27% higher throughput than *HeteroEdge*, the next best placement policy. For multiple pipelines without common modules (Figure 4.8(b)), *OVIDA* achieves 14% accuracy and 58% throughput gain for the NPR pipeline, and 27% accuracy and 31% throughput gain for the AcRg pipeline compared to the next best policy, *HeteroEdge*. Finally, for multiple pipelines with common modules (Figure 4.8(c)), *OVIDA* achieves a 13% accuracy and 50% throughput gain for NPR pipeline and 41% accuracy and 48% throughput gain for VC pipeline compared to *Monolithic*. Sharing common modules provides an additional accuracy boost of 4% for the NPR pipeline and 20% for the VC pipeline, compared to *Monolithic*. Here, VC and NPR pipelines' common service, object detector, was the bottleneck. Increasing the number of object detectors significantly boosts VC's performance, as it is the only resource-intensive module in the pipeline.

OVIDA outperforms the baseline approaches because it alleviates the bottleneck by selectively scaling the module instances. Further, it takes into account the heterogeneous communication delays and service times among edge nodes while performing the placement. We also find that *OVIDA* achieves high GPU utilization (but comparable to HeteroEdge), as shown in Figure 4.9, which shows the median GPU utilization for all policies across all deployments. By more efficiently placing modules, *OVIDA* makes better use of GPU resources.

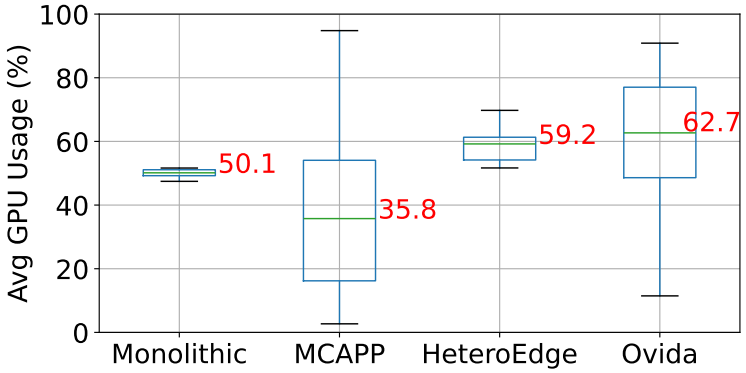


Figure 4.9: GPU usage across all devices for the evaluation shown in Figure 4.8.

4.7.2 Adaptive Model Selection

This section is not my original work and hence, completely removed from the thesis. For the sake of completeness, please refer to the main takeaway of this section below.

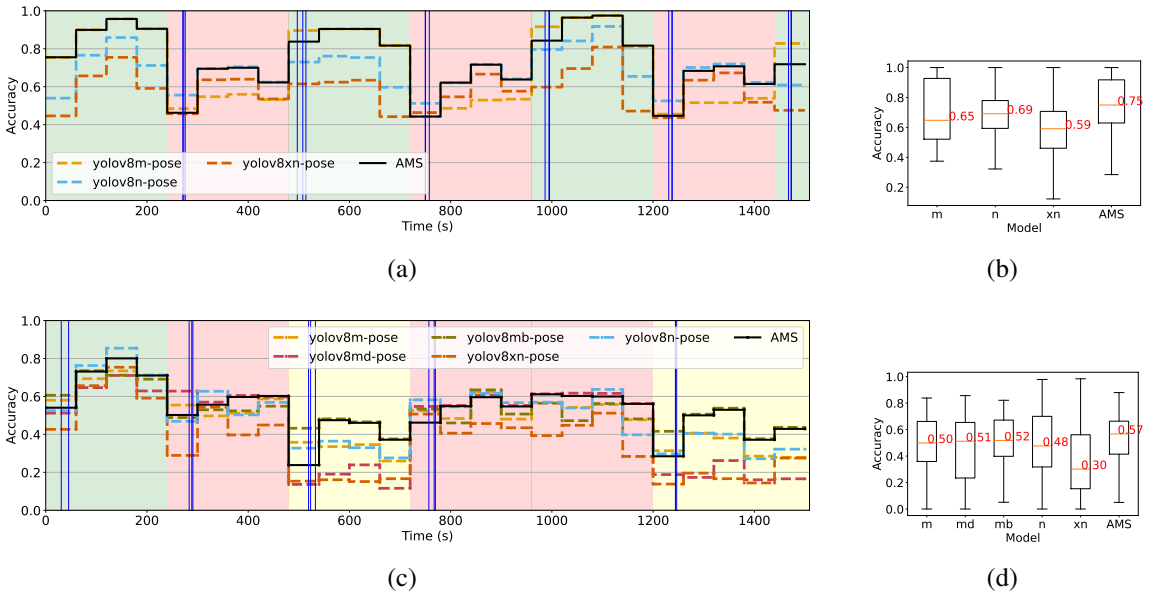


Figure 4.10: (a) Selecting the appropriate models in response to changes in arrival rate. (b) Distribution of accuracy for AcRg pipeline in (a), sampled every 30s. (c) Selecting the appropriate models in response to changes in frame content. (d) Distribution of accuracy for AcRg in (c), sampled every 30s. Note that 'AMS' here refers to OVIDA with Adaptive Model Selection.

Figure 4.10 shows the results of our adaptive model selection evaluation. Dynamic model switching in response to changes in arrival rate achieves a 15% improvement in accuracy over the default placement using only the *m* model and a 10% improvement over the *n* model, the best static choice in this case (see Figure 4.10(b)). And dynamic model switching in response to changes in frame content achieves a 14% improvement in accuracy over the

default placement using the *m* model and a 10% improvement over the *mb* model, the best static choice in this case (see Figure 4.10(d)).

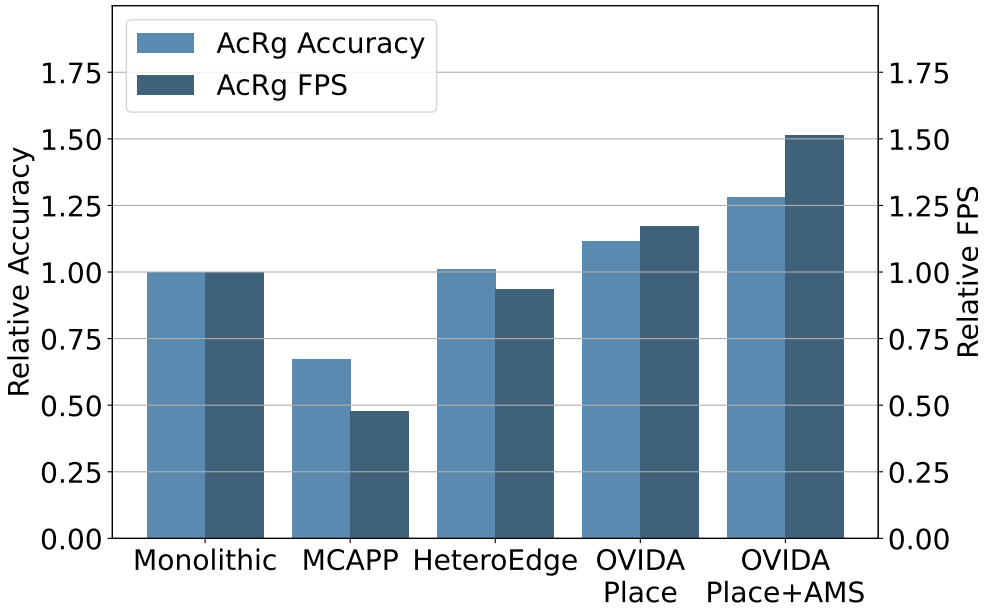


Figure 4.11: End-to-end performance improvement afforded by OVIDA with placement and model selection for the pose estimation pipeline.

4.7.3 End-to-end Evaluation of OVIDA

We now evaluate the benefits of combining OVIDA's placement and adaptive model selection against other placement schemes. Similar to Section 4.7.2, we used the pose-estimation pipeline. The input load traces are the same as those employed in Figures 4.10(a) and 4.10(c).

Figure 4.11 shows the relative FPS and accuracy achieved by different policies for the above-mentioned load traces. The 'Monolithic', 'MCAPP', 'HeteroEdge', and 'OVIDA Place' policies shown only perform the initial placement using the fixed *m* model. OVIDA Place with fixed *m* model achieved throughput and accuracy gain of 17% and 11% over Monolithic. Our end-to-end system, OVIDA with placement and adaptive model selection ('OVIDA Place+AMS') achieves significantly better performance, with throughput and accuracy gains of 51% and 28%, respectively, over Monolithic. ***Compared to OVIDA's placement-only policy, OVIDA's placement+AMS thus obtains additional throughput and accuracy gains of 29% and 15%, respectively, highlighting the benefits of combining model placement and dynamic model selection***.

4.8 Related work

Table 4.2 presents a summary of the important features that set OVIDA apart from existing work. In this section, we first discuss prior works that focus on VA placement and then those that tackle the VA model selection problem.

Module Placement

The placement problem in the VA context refers to scheduling module instances on available edge devices to maximize performance (throughput and accuracy). Earlier works have explored different system designs to address this problem. (i) *Edge-cloud collaboration:* Edge devices are only either used for partial processing or for filtering out frames before sending them to the cloud for serving the remaining VA operations [137, 36, 162]. (ii) *Hierarchical processing:* Video streams go through multiple layers of processing infrastructure such as smart cameras, nearby workstation cluster, and the cloud, each of them hosting some VA function(s) [71]. (iii) *Local edge:* Completely processing video streams at the edge using the devices in the same local area network [159, 163, 93, 102, 37]. While many prior works have explored several distinct combinations of system designs, apart from Bahreini et al. [27] none have considered the possibility of a ***distributed edge node system hosting a multi-component VA pipeline***. Further, prior works often make certain limiting assumptions, such as assuming (i) the availability of powerful workstations or server-class machines with discrete GPUs to execute neural network-based VA functions, and (ii) all edge nodes being present in a local network connected with high-speed links.

Among the above works, we will now discuss those most closely related to ours. Bahreini et al. [27] propose a generic heuristic for placing multi-component pipelines on distributed edge nodes. However, the heuristic is limited to placing *only one* module per pipeline on a node, which can greatly limit performance (as we show in Sections 4.2 and 4.7.1).

Hetero-Edge [163] presents a system that works with heterogeneous edge nodes and places the bottleneck module on a single GPU, if that GPU's usage is less than a threshold, before moving to other alternative devices. If no GPU is available, the other functions are parallelized over the CPU of the same or a different node. This heuristic assumes only one module instance for every service and further assumes that there is only one module in the pipeline that requires GPU access. These assumptions do not hold for many VA tasks (Section 4.7.1).

Distream [159] proposes (i) dynamic load balancing among smart cameras, and (ii) finding an optimal point to slice the pipeline such that the first half is placed on a smart camera and the second on a powerful GPU-enabled workstation that is assumed to be present

Table 4.2: Feature comparison with most relevant works.

Features	**MCAPP [27]**	**Distream [159]**	**HeteroEdge [163]**	**OVIDA (Ours)**
Distributed edge nodes	✓	✗	✗	✓
Hosts pipelines with multiple DNN based modules	✓	✓	✗	✓
Adaptive load balancing	✗	✓	✗	✓
Bottleneck detection and scaling	✗	✗	✗	✓
Does not require a workstation	✓	✗	✓	✓
Adaptive model switching	✗	✗	✗	✓

in the same local area network. However, Distream is quite rigid in its system design in that it requires a powerful workstation and a high-speed connection among all edge devices, which may not always be available in an edge-only scenario.

Oakestra [32] is a general-purpose orchestrator for edge deployments, capable of placing disaggregated applications [31]. Oakestra's scheduler places microservices on suitable edge nodes based on an extensive set of parameters including memory, network, GPU, and CPU requirements. However, as a generalizable solution, it lacks application-specific knowledge and does not support scaling through bottleneck replication if better placement options are unavailable.

Model Selection

Existing works have proposed strategies for dynamic model selection with distinct system designs for embedded devices [153, 97] and workstations on edge [125, 75, 128, 21]. We next discuss the closely related works that focus on model selection on heterogeneous distributed edge computing deployments.

INFaaS [128] allows developers to register multiple models per application; the submitted models are then hosted on available edge devices. A user's query to INFaaS contains the image to process, the VA application to use, the maximum allowed latency, and the minimum required accuracy. INFaas then selects the best possible model-device combination to satisfy

the query. However, INFaaS was designed and evaluated only on high-performing edge devices with enough memory to load multiple models simultaneously.

Proteus [21] uses a similar system design to INFaaS, except that it optimizes model-device selection for maximizing accuracy as opposed to maintaining minimum accuracy and maximum latency bounds. However, it also requires loading multiple model variants on an edge device. Further, Proteus models the optimization as a mixed integer linear programming problem, whose solution is computationally intensive. Proteus' high memory and computing demands make it infeasible for edge deployments lacking powerful workstations.

LiteReconfig [153] proposes an approach to select an inference model based on light features (frame resolution, number of objects detected) and heavy features (histogram of color, histogram of oriented gradients, and feature extraction using a DNN among others) to maximize accuracy for a minimum latency constraint. It utilizes a custom-trained DNN model to estimate the additional latency cost of using the heavy features and determines whether to employ them. Based on this estimated latency cost, it selects the inference model to maintain both accuracy and latency within specified bounds. RAVAS [125] proposes a low-overhead technique to share the GPU among different models processing multiple video streams. RAVAS uses reinforcement learning while receiving periodic accuracy feedback from the "golden configuration"—the best configuration of models. Both LiteReconfig and RAVAS require all the available models to be pre-loaded into memory for quick switching and comparison, which is not feasible in edge deployments due to memory constraints. In contrast, OVIDA only keeps one model in memory and makes a swap when required.

4.9 Conclusion

In this work, we presented the design and evaluation of OVIDA. To our knowledge, OVIDA is the first fully disaggregated video analytics design that has been evaluated on a testbed of multiple embedded devices for three VA applications and three network topologies, under varying load conditions. Our experiments show the OVIDA's placement policy can outperform all baseline techniques in terms of median accuracy. Compared to the next-best policy, OVIDA achieves accuracy gains of up to 7%, 27%, and 41% for single pipeline, and multiple pipelines with and without common modules, respectively, under various network and load conditions. We also present an adaptive model selection technique that selects the most appropriate model for a VA task and seamlessly switches to it as needed, in response to changes in load. OVIDA's adaptive model selection can provide additional accuracy gains of 15%. We believe that OVIDA's design can easily be scaled to efficiently support large VA systems on a network of distributed edge nodes.

Chapter 5

Network Optimization in Disaggregated Edge Data Centers

Resource disaggregation (RD) is an emerging paradigm for data center computing whereby resource-optimized servers are employed to minimize resource fragmentation and improve resource utilization. Apache Spark deployed under the RD paradigm employs a cluster of compute-optimized servers to run executors and a cluster of storage-optimized servers to host the data on HDFS. However, the network transfer from storage to compute cluster becomes a severe bottleneck for big data processing. Near-data processing (NDP) is a concept that aims to alleviate network load in such cases by offloading (or "pushing down") some of the compute tasks to the storage cluster. Employing NDP for Spark under the RD paradigm is challenging because storage-optimized servers have limited computational resources and cannot host the entire Spark processing stack. Further, even if such a lightweight stack could be developed and deployed on the storage cluster, it is not entirely obvious which Spark queries would benefit from pushdown, and which tasks of a given query should be pushed down to storage.

This work presents the design and implementation of a near-data processing system for Spark, SparkNDP, that aims to address the aforementioned challenges. SparkNDP works by implementing novel NDP Spark capabilities on the storage cluster using a lightweight library of SQL operators and then developing an analytical model to help determine which Spark tasks should be pushed down to storage based on the current network and system state. Simulation and prototype implementation results show that SparkNDP can help reduce Spark query execution times when compared to both the default approach of not pushing down any tasks to storage and the outright NDP approach of pushing all tasks to storage.

5.1 Introduction

Data centers are typically composed of racks of general purpose servers, with each server equipped with an adequate amount of compute, memory, and storage resources to execute a variety of applications. However, when a specific application (e.g., a compute-intensive job) is executed on a general purpose server, the non-dominant resources (memory and storage, in this example) remain severely underutilized. To avoid resource fragmentation and improve utilization, data centers are moving towards a *disaggregated infrastructure* (DI) model, also referred to as resource disaggregation, whereby servers are built to optimize for a specific resource, such as compute or storage [132]. DI also helps to make upgrades to technologies independently and reduces time to adoption as the development of new motherboard designs and integration testings are no longer needed.

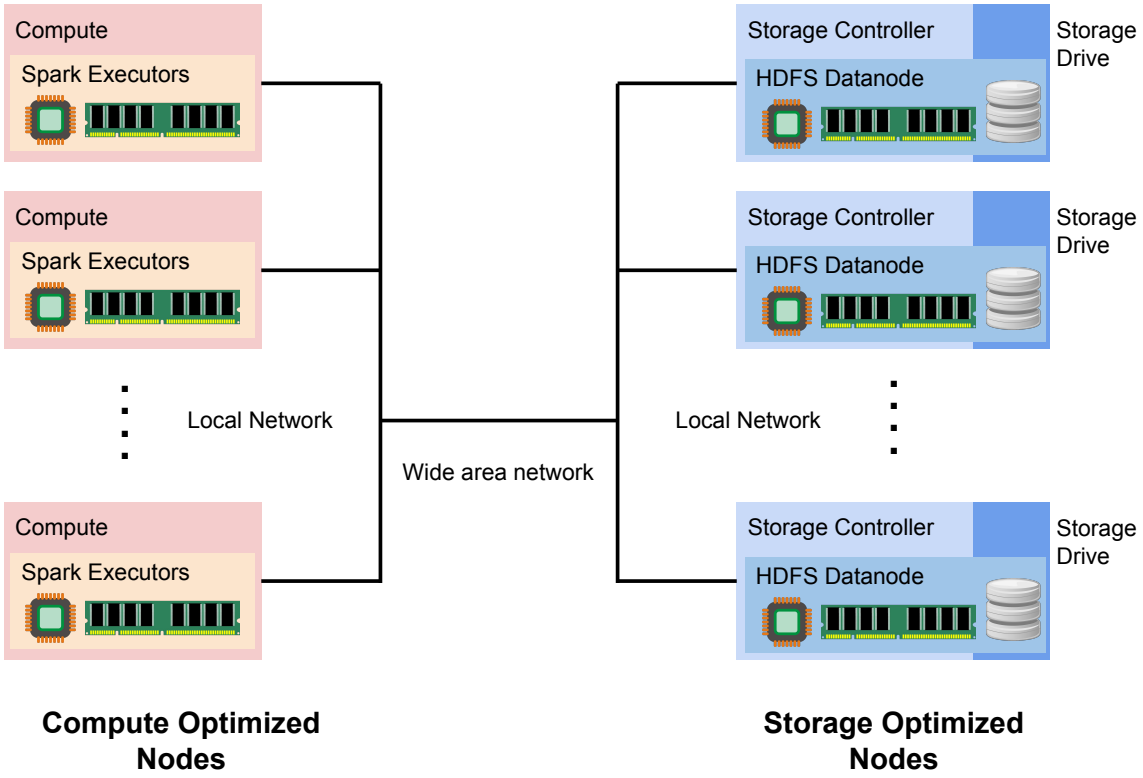


Figure 5.1: Illustration of a disaggregated infrastructure (DI) deployment of Spark over a cluster of compute optimized nodes (running Spark executors) connected over the network to a cluster of storage optimized nodes (hosting HDFS datanodes).

Under the DI model, resource-optimized servers are interconnected over the network. As such, data transfer between groups of optimized servers can quickly overwhelm the network, resulting in a performance bottleneck [55, 73, 83]. A popular application that is often deployed under the DI model in enterprise settings is Spark data processing [83, 40].

Consider the DI model shown in Figure 5.1 which consists of a cluster of compute-optimized nodes (servers or VMs or containers) running Apache Spark [139] connected via the network to a cluster of storage-optimized nodes hosting data on HDFS [133]. Typically, the Spark executors on the compute nodes would pull the entire data, in the form of partitions, from HDFS datanodes over the finite-bandwidth network [158]. Since modern datasets can be several hundred or gigabytes or even petabytes in size [152], the data transfer overhead poses a significant barrier for deploying Spark over DI.

Near-data processing (NDP) attempts to alleviate network latencies for large data transfers by moving computation closer to the source data [146]. In the aforementioned Spark example, consider the dataset to consist of order history for a bookstore. A user interested in analyzing the dataset might first filter the data based on a date range and then apply analysis on the filtered data; the analysis itself might involve some compute-intensive ML modeling tasks. Without NDP, the data partitions on HDFS will have to be transferred over to the Spark executors before the individual filter and analysis operations can be executed on them. However, if we can leverage any available compute power at the storage nodes to execute the filter operation at the storage nodes, then only the filtered data (and not the full dataset) will have to be transferred over the network to the compute nodes.

Prior work has shown that NDP has the potential to reduce query execution time for Spark over DI by "pushing down" the Spark operations to the storage cluster [60]. However, this outright pushdown of all Spark operations to storage may not work well for all queries as the available compute resources at the storage cluster may get quickly saturated, thereby harming the end-to-end performance. This is especially problematic because the overhead of supporting Spark operations on storage nodes already reduces the available compute resources on the storage cluster that can be leveraged to execute the pushed down operations. What is needed is a careful accounting of the end-to-end performance implications of pushdown and insight into which queries will benefit more from pushdown.

As such, to fully leverage NDP for Spark, there are three key questions that must be addressed: (1) *what software support is required to enable NDP at HDFS nodes?* (2) *which queries will benefit from pushdown?* and (3) *which operations of a given Spark query should be pushed down to HDFS to maximize the NDP benefits?* By default, HDFS nodes do not provide any support for executing parts of a Spark job. Further, since storage-optimized nodes may have very limited computing power, Spark executors or other such regular processes cannot be deployed on the HDFS nodes. What is needed is a lightweight software capable of executing parts of a Spark query. Even if such support is available, it is not entirely obvious whether a given Spark query, are parts of the query, should be pushed down for execution

at the HDFS nodes. Different queries will have different compute requirements, and the optimal pushdown strategy will likely be dictated by current network and system conditions.

In this work, we present the *design and implementation of SparkNDP*, a software framework that enables NDP for Spark. SparkNDP works by deploying a lightweight library of SQL operators on storage-optimized HDFS nodes. To provide sufficient query-level information to HDFS for performing NDP, we create a datasource extension for Spark. This extension converts Scala job queries on-the-fly to SQL commands to be executed at HDFS nodes. Our SparkNDP prototype is publicly available [127].

To determine which queries can benefit from pushdown, SparkNDP considers the reduction in data size after each query operator to decide whether the reduction in data transfer size over the network is worth the slow execution time at storage. To figure out which operations should be pushed down to HDFS, SparkNDP relies on an *analytical model* that we develop to estimate the latency benefits of each potential operation pushdown. Our model takes into account the throughput at HDFS and Spark clusters and the network delay in transferring intermediate data between the clusters. We integrate this model with SparkNDP to enable runtime decision making for optimal pushdown for any Spark query. Although we designed SparkNDP on top of HDFS, our system design and model can work with other services such as AWS S3, Azure Data Lake Storage, etc.

To evaluate the pushdown decision making of SparkNDP under different conditions, we develop a practical, data-driven simulator that allows us to consider arbitrary queries and cluster sizes. Our simulation results show that SparkNDP reduces the query execution time by upto 71% compared to the systems without NDP and 6% compared to the already existing NDP solutions. It also reduces the query execution time as much as 38% compared to the already existing NDP solutions when there are multiple jobs running simultaneously.

To validate the real-world potential of SparkNDP, we implement a prototype on top of Spark and HDFS and empirically evaluate its benefits using TPC-H Spark queries. Using our experimental setup, we consider a few different cluster sizes, different network bandwidths, and different operating frequencies for the storage cluster. Our empirical results show that SparkNDP can make the right choice between no pushdown and full pushdown in most cases, and can also sometimes determine the optimal selective pushdown when such an option provides significant benefits. Compared to the default strategy of no pushdown, SparkNDP reduces average query execution time by around 42%. Further, the query execution time under SparkNDP is within 5% of that under the offline Oracle policy, highlighting the near-optimal behavior of SparkNDP.

To summarize, the contributions of this work are:

- We design and implement SparkNDP, a lightweight system built on top of Spark and HDFS that enables selective pushdown (to disaggregated storage) of several Spark query operations.
- We develop an analytical model for SparkNDP to determine which operations of a Spark query should be pushed down as a function of network and system conditions.
- We empirically evaluate SparkNDP on a distributed cluster setup using TPC-H Spark queries and show that SparkNDP can reduce query execution times (by as much as 25%) compared to the best of non-NDP and full-NDP Spark deployments.

The rest of this work is organized as follows. Section 5.2 provides the necessary background on how Spark works in an DI environment. Section 5.3 summarizes relevant prior works in the areas of Near Data Processing and Disaggregated Infrastructure. Section 5.4 discussed our design and implementation of SparkNDP. We evaluate SparkNDP in Section 5.5 using simulation and empirical results under various network and system conditions. We discuss related work in Section 5.3 and conclude the work in Section 5.6.

5.2 Background

5.2.1 Data processing in Spark

Distributed data processing applications exchange and manipulate data by formatting them into objects or data structures they understand. Spark provides multiple such data structures and we use DataFrame (DF) among them. DFs are optimized for storing the required data on memory and allowing Spark to perform operations like filter, project, aggregate-sum, map, sort, etc. As DFs are generally present in memory (unless memory space is limited), the disk I/O delay is not incurred and data transfers are quick. Spark can load data into DFs either from a local file system or from network storage applications like Hadoop Distributed File System (HDFS) [133]. This gives a possibility for users to disaggregate the persistent storage requirements from computational requirements by using systems like HDFS.

The data processing flow of Spark jobs can be visualized as a Directed Acyclic Graph (DAG) in which every vertex is a user-specified operation on an DFs. In order to use all the available cores, Spark increases the parallelism of a given job by breaking down the input DataFrame into multiple partitions based on memory availability on the executors. An executor is a spark daemon running on worker nodes and are the in charge of running the spark tasks. Consider a Spark job that needs to work on a file of size 100 GB and each executor can handle a maximum of 256 MB of data. Spark master divides the 100GB data

into 400 *partitions* and add them into a task queue along with the operations meant to be performed on them. Every time an executor is available and free, it picks up a task from the queue, copies the data from the storage stack, runs the operations on this partition and submits the intermediate data to the next stage. Here, we found an opportunity to pushdown the operations into the storage stack and run them before the copying process.

A spark job can be written in either in Scala or Python. Code written in scala for spark are known to be faster and once compiled, they can be launched on a Spark using spark-submit utility.

5.2.2 Hadoop Distributed File System (HDFS)

HDFS is a popular utility to store persistent data on multiple nodes in a distributed fashion. An HDFS cluster primarily consists of a namenode that handles all connection initializations and maintains information about the file blocks and a datanode that stores the actual file blocks. HDFS can be configured to add redundancy by copying the same file blocks into multiple datanodes for fault tolerance. It also provides an interface called "WebHDFS" that allows clients (like Spark) to access the contents using an HTTP restful API without installing any Hadoop extensions/libraries. In our implementation, we take advantage of the replication factor to increase the number of datanodes that can perform our pushdown operations and intercept the WebHDFS communication between the client and the datanode to run NDP operations.

```
1  sales_record = spark.read.format("csv")
2                      .load("hdfs://namenode/sales_record.csv")
3  sales_record_filtered = sales_record.filter($"date" >= "1994-01-01"
4                                           && $"date" < "1995-01-01")
5  total_sale_1994 = sales_record_filtered.agg(sum($"price"))
```

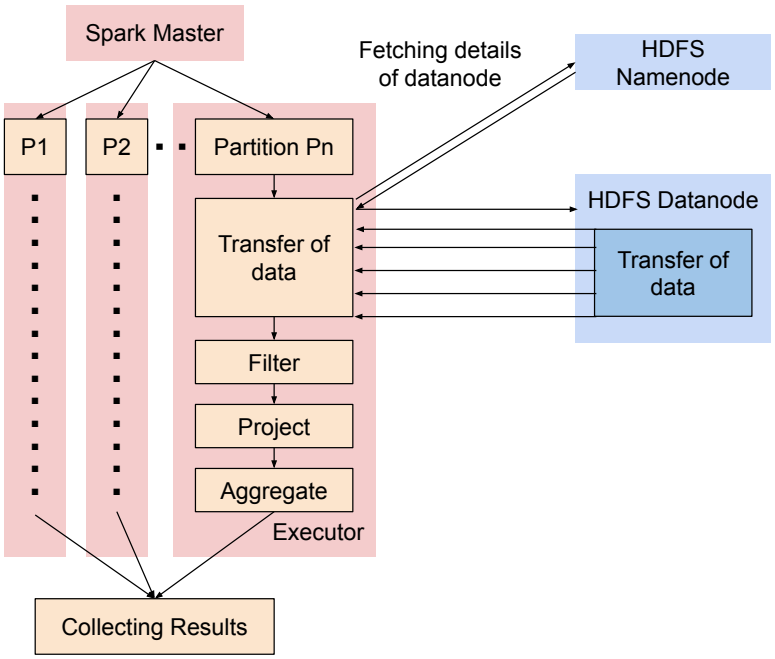
Figure 5.2: An example spark code

5.2.3 An example of a Spark job involving HDFS and NDP

Figure 5.2 shows an example code written in scala to load a CSV file containing sales records of a store and perform "filter" and "aggregate" operations to output the sum of prices of items sold in the year 1994. Spark first divides the dataset into partitions of manageable sizes; then, each executor working on a partition loads the dataset from a datanode, filters the data based on the date, and aggregates the sum of entries in column "price". Figure 5.3 illustrates this flow of data. We note that running the filter operation before transferring the records will reduce the transfer size/time between Spark and HDFS. For instance, if the sales

record dataset contains all records from start of 1990 to end of 1999, then the filter operation may reduce the data size by about 10×; if the filter operation is pushed down to storage, then only the filtered data will have to be sent to the compute cluster for running the aggregate operation.

Spark and HDFS in DI without NDP



Spark and HDFS in DI with NDP

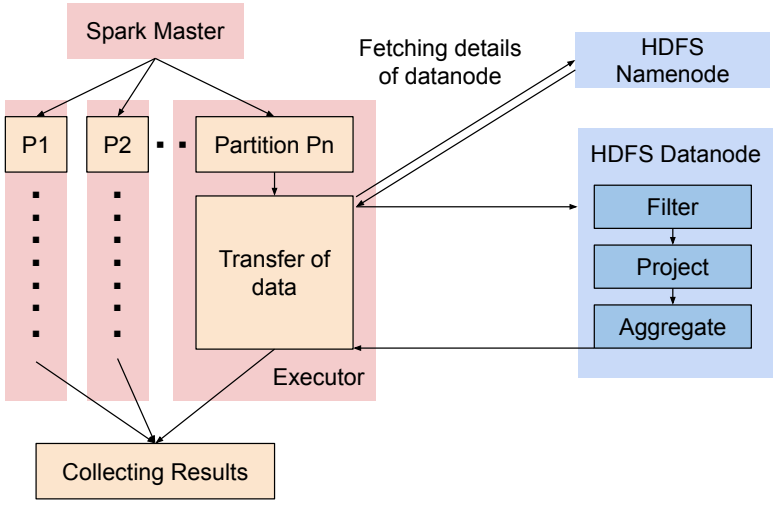


Figure 5.3: Overview of how Apache Spark and HDFS share data.

5.3 Related Work

This section discusses relevant works in the fields of NDP and DI. Subsection 5.3.1 discusses works that perform NDP using specialized hardware. Subsection 5.3.2 discusses works that design strategies for task distribution to improve performance. Subsection 5.3.3 discusses works that have implemented NDP for Spark.

5.3.1 NDP using specialized hardware

The concept of Near-Data Processing has been around recently and is known to have the potential to improve efficiency of computation in data centres [29]. Modern storage devices built on NVMe and Flash technologies offer great bandwidth and random access to data. For example, nKV [146] proposes a design to implement a fast key-value store for smart storage devices by directly controlling the physical data to achieve high I/O parallelism. Their system design also provides necessary data structures and meta data to the compute elements present on these smart storage devices to allow performing NDP. Experiments show 1.4×–2.7× better performance on complex graph-processing algorithms. This work differs from our work by executing operations on different hardware based on their optimality.

Likewise, Huang et al. [69] propose an in-network computing architecture which has a three-phase processing procedure, offloading the computation near data in the memory network for execution and aggregating the results along their routing path. Their in-network NDP architecture, Active-Routing, can achieve up to 7× speedup with a geometric mean of 60% performance improvement, and can reduce energy-delay product by 80% on average across benchmarks.

Similar to nKV [146], JetStream [120] incorporates storage data structure in the form of OLAP data cubes, so data is stored for analysis near where it is generated. Experiments show that the end-to-end latency is within a few seconds, even when bandwidth drops by a factor of two. The work differs from our work by implementing data cubes which capture aggregates of data encapsulated within them to serve queries faster.

While the above works propose useful NDP designs, they require specialized hardware architecture support and are thus expensive to deploy in existing data centers.

5.3.2 Policies for distributing tasks

Gaia [67] implements a geo-distributed ML system that employs an intelligent communication mechanism over WANs to efficiently utilize the scarce WAN bandwidth. Experiments show a 1.8×-–53.5× speedup over two state-of-the-art distributed ML systems. This work

specifically deals with optimizing ML workloads by storing an approximate copy of the saved model locally at each computing node for decreasing the communication required between the nodes.

Given the scarcity of works that develop policies for NDP, we aim to fill this gap by developing an analytical model for NDP that takes decisions on which operations should be placed where (compute versus storage clusters).

5.3.3 NDP implementations

Octopus [39] proposes a technique of query pushdown for performing NDP by allowing Spark to remotely run SQL queries at the MySQL instances located on the storage devices. Experiments show that Octopus outperforms the recent Spark version 1.4.0 by about 5.25× in terms of running time to process an aggregation query. The heuristic solution proposed leverages the data locality principle which minimizes the amount of data movement. While Octopus generates subqueries that seek to reduce the overall query processing time, our work reduces the time by deciding the exact subset of operations to be pushed down.

PushdownDB [156] implements a new DBMS which pushes down parts of analytics queries into the S3 Select engine of AWS. A program written in C++ and python works as a driver to access the results after performing SQL queries on the DB. An extensive experiments have shown that PushdownDB is 30% cheaper and 6.7× faster than a baseline without it. In our work, we use HDFS as the storage service, SparkNDP as the near-data processing system, Spark for running the jobs and develop an analytical model that can decide which operations should be pushed down to the storage cluster.

An important related work to our project is λFlow [60]. The authors present a framework to perform NDP and push operations like map, flat map, and filter to OpenStack Swift on the storage cluster from Spark on the compute cluster. They implement this as an extension to Crystal [42], a software-defined storage framework for OpenStack Swift. As the application of NDP is in software, the deployment can be inexpensive. λFlow can reduce data transfers significantly and achieve 1.47×–3.39× speedup in job completion times. The evaluations report a 90% reduction in network bandwidth and memory requirements.

In our work, we show that always pushing down every operation to the storage stack is not beneficial due to resource constraints. Instead, by selectively pushing down operations depending on the network and system state, SparkNDP can achieve 25% improvement over complete pushdown strategies like λFlow.

5.4 System Design

This section describes the design of our NDP-enabling Spark framework, SparkNDP, and the analytical model we construct to decide on which queries and operations to push down to storage.

In a general use case of Spark, the user launches a Spark driver that processes the incoming queries and distributes them to its worker nodes that may be present on different physical hosts or VMs or containers. When the worker nodes start working on their part of the query, they make individual read requests to the storage system to get the required data and then process it as shown in Figure 5.3. After receiving the requested data, the workers perform operations on the tables.

This approach of transferring the complete dataset is inefficient as it is slowed down by the network capacity. To alleviate the network load, we investigated supporting the execution of specific operations at the HDFS datanodes, such as Filter, Project, and some Aggregate operations. These operations are simple enough and can be ported into the storage cluster, essentially performing Near Data Processing for Spark. Once NDP is implemented, the data flow in Spark and HDFS clusters is as illustrated in Figure 5.3. To perform NDP at HDFS, we create extensions to Spark and HDFS instead of modifying them. The extensions we made are described in the following subsections.

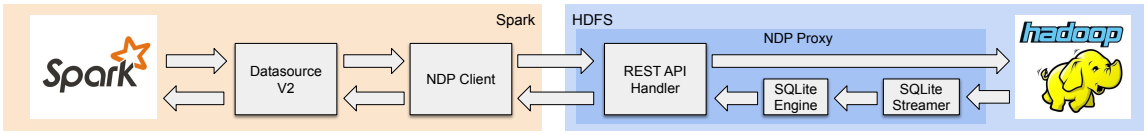


Figure 5.4: Dataflow diagram of our SparkNDP implementation.

5.4.1 Extension for Spark

For pushdown operations to be run at HDFS, we require Spark to provide sufficient information to HDFS while executing the query. Spark provides an API called the Datasource V2 API to allow building a custom handler to any data source (like HDFS). We leverage this API to create an *NDP Datasource* extension for Spark. Our extension not only supports the requirements of the API but also talks to our NDP-enabled HDFS cluster to perform pushdown. Our implementation allows pushdown of Filter, Project, and some limited Aggregate operations like sum, average, max, and min. When an aggregate operation is pushed down to our NDP-enabled HDFS, the datanodes are responsible for performing the operation only for the partition it is currently handling. We handle the overall aggregation of these individual results in our NDP Datasource Extension.

In our implementation, when a query is being processed by Spark, it first interacts with the datasource extension to get metadata about the tables it needs and gets the partition details from the HDFS and our NDP client extension. The Spark master shares these details with the executors so they can independently communicate with the HDFS cluster. When an executor tries to connect to the datanodes, our NDP proxy intercepts the requests and performs NDP. More details about the NDP proxy is given in the next subsection. We also made an "NDP client" extension for Spark to enable communication with our NDP Proxy on HDFS datanodes.

The actual access to HDFS data blocks in executors is achieved through Java's InputStream [111]. We added a new parameter called "readParam" to HDFS's open API (that initializes InputStream) to send XML data required for filesystem operations and NDP pushdown. For performing NDP at the datanodes, we use SQLite. We convert the queries in Scala jobs to equivalent SQL commands on the fly at NDP Datasource to be sent to the NDP Proxy. For example, the following Scala can be converted to an SQL code as shown below.

```
Scala –
employee. filter ($‘‘department’’=‘‘HR’’). select ($‘‘name’’)
SQL –
SELECT name FROM employee WHERE department = ‘‘HR’’
```

In this query, "FROM employee" gives us the table to be read from storage. "WHERE department = "HR"" is the filter operation and "SELECT name" is the project operation. We send this SQL query through the "readParam" parameter in XML data format. Below is an example of how the XML data looks like. We add the column names of the CSV table we wish to process to the "Scheme" element, the SQL query to "Query" element and other miscellaneous parameters like delimiters required to parse the CSV file as a table.

```
<?xml version=’1.0’ encoding=’UTF−8’?>
<Processor>
<Name>dikeSQL</Name>
<Version>0.1 </Version>
<Configuration>
<Schema>l_orderkey LONG, l_partkey LONG, l_suppkey LONG, l_linenumber LONG, l_quantity
    NUMERIC, l_extendedprice NUMERIC, l_discount NUMERIC, l_tax NUMERIC,
    l_returnflag STRING, l_linestatus STRING, l_shipdate STRING, l_commitdate STRING,
    l_receiptdate STRING, l_shipinstruct STRING, l_shipmode STRING, l_comment STRING
</Schema>
<Query><![CDATA[SELECT SUM("l_extendedprice" ∗ "l_discount") FROM CaerusObject s
    WHERE l_shipdate IS NOT NULL AND s."l_shipdate" >= ’1994−01−01’ AND s."l_shipdate"
```

```
    < '1995-01-01' AND s."l_discount" >= 0.05 AND s."l_discount" <= 0.07 AND s."l_quantity
    " < 24.0 ]]>
</Query>
<BlockSize>134217728</BlockSize>
<FieldDelimiter>44</FieldDelimiter>
<RowDelimiter>10</RowDelimiter>
<QuoteDelimiter>34</QuoteDelimiter>
</Configuration>
</Processor>
```

5.4.2 Extension for HDFS

HDFS provides options to connect using the protocols Hadoop RPC (Google proto-buff based) and Hadoop WebHDFS (REST API based). We decided to work with WebHDFS because of its similarity with AWS S3 (for possible future compatibility) and since it allows for a more flexible architecture. For implementing NDP at HDFS datanodes, we chose to implement a reverse proxy (or called Application Proxy in Hadoop's terms) and deploy it on every datanode to intercept the communication between HDFS processes and Spark executors. We implemented this reverse proxy, which we call *NDP Proxy*, using the POCO framework [25].

While stock WebHDFS open/read APIs contain all the parameters required to provide access to file blocks, they have no provision to pass the XML file containing the additional parameters needed to perform NDP. The following is the syntax of the API provided in the official documentation [140].

```
http ://< HOST>:<PORT>/webhdfs/v1/<PATH>?op=OPEN[&offset=<LONG>][&length=<LONG
    >][&buffersize=<INT>]
```

We work around this problem by injecting the XML code mentioned in the previous subsection into the HTTP message header. Our NDP Proxy on datanodes inspects the header of incoming requests to decide if it should perform NDP or simply forward the request (as is currently the case). In case it receives a request with the NDP header, it performs the following actions to perform NDP.

1. Extract and validate XML configuration from HTTP Header.
2. Connect to HDFS Datanode process to get the data.
3. Launch a new SQLite instance with a streaming plugin.

4. Start processing data on the fly while retrieving data using SQLite.

5. Send results back to the Spark executor.

6. Destroy SQLite instance.

Figure 5.4 summarizes the flow of requests through our extensions in Spark and HDFS. We have open sourced our implementation [127].

5.4.3 Analytical model to decide the best pushdown strategy

While our aforementioned SparkNDP design enables selective push down (of a given operator in a query), it is not entirely obvious which operators should be selectively pushed down. Further, the decision to push down an operator is not deterministic and depends on the network and system state; for example, if the network is expected to be congested, then pushdown may be preferred to minimize the subsequent data transfer size from storage to compute cluster.

To help decide on selective pushdown, we construct an analytical model for SparkNDP. Our analytical model, ***Net-Aware***, works by estimating: (i) the execution time of an operation (across all partitions), including any queueing time, if served at the compute cluster; (ii) the execution time of an operation (across all partitions), including any queueing time, if served at the storage cluster; (iii) the time needed to transfer the input data for the operation across the network; and (iv) the time needed to transfer the output data for the operation across the network. Typically, the ***reduction factor***, the ratio of input data size to output data size for an operation, is large enough in practice that item (iv) can be considered negligible. If the sum of items (i) and (iii) is greater than the items (ii) and (iv), then Net-Aware recommends SparkNDP to push down the operator (associated with the operation) to the storage cluster. For a given query, Net-Aware iteratively makes this decision for each operation in the query while taking into account the implementation requirement that if an operator is not pushed down, then all subsequent operations cannot be pushed down either.

To estimate the execution time of an operator at a cluster, Net-Aware considers the outstanding queue size, Q (in terms of number of partitions) at that cluster and the "drain rate", X (in units of partitions/s) at the cluster. If the operator being considered for pushed down is working on P partitions, then the execution time at a cluster, in seconds, is $T_c(Q,X) = \frac{Q+P}{X}$. For simplicity, the queue size and drain rate are considered for the entire cluster (as opposed to per core or per node) and the competing network load is assumed to be constant.

To estimate the transfer time for a given data of size D GB over a network with bandwidth B Gbps, Net-Aware also considers the number of simultaneous data transfers at the network,

S. We assume that the number of queries, S_{max}, that can simultaneously transfer data between HDFS and Spark clusters is limited by the application configuration parameters, e.g., the max-jobs-per-context parameter in Spark. Alternatively, the cluster operator might decide to set the concurrent queries limit to the ratio of number of executors to number of datanodes, or something similar. Regardless, if $S < S_{max}$, then the network transfer time is simply $T_n(D) = \frac{D \times 8}{B/S}$ seconds, assuming that new data transfers cannot overtake existing transfers in the network. Note that some active transfers can complete before our target data transfer but we ignore the subsequent potential increase in network bandwidth for simplicity. However, if $S \geq S_{max}$, then the query in consideration will first have to wait until sufficient outstanding data, D_S GB, can be transferred before it can start sharing the network (i.e., until the number of active transfers falls below S_{max}) to transfer its data. In this case, we have $T_n(D) = \frac{(D_S+D) \times 8}{B/S_{max}}$ seconds.

Based on the above, Net-Aware decides to push down an operator if $T_c(Q_{Spark}, X_{Spark}) + T_n(D_{input}) > T_c(Q_{HDFS}, X_{HDFS}) + T_n(D_{output})$. Here, D_{input} and D_{output} denote the data size, in GB, for the input and output of the operator, respectively; and the subscript of Q and X represent the cluster whose queue size and throughput they denote, respectively. Finally, considering a query is made up of a sequence of n operators, $o_1, o_2, \ldots, o_n$, Net-Aware will push down the first i operators if it determines (using the above decision equation) that operations 1 through i should be pushed down but not operation $(i+1)$. For example, if operations 1 and 2 benefit from pushdown but not operation 3, then Net-Aware will disregard the decision for subsequent operations and only push down operations 1 and 2. This is because, in practice, once the query is sent for execution to the compute cluster (Spark), it cannot subsequently be pushed down again to the HDFS cluster. While other query topologies, like trees, are possible, Net-Aware currently only consider the bottleneck path as the sequence of operations.

5.5 Evaluation

We now evaluate SparkNDP via simulations and prototype experiments. We start by exploring a wide range of scenarios via simulation and then present empirical results based on our prototype implementation (as described in Section 5.4) on our experimental setup. We run our Net-Aware analytical model before launching every query to estimate the latest optimal pushdown strategy for that query. We compare our work against two policies mainly: *No-Pushdown* where all the data is brought to the compute cluster for processing (i.e., no NDP), and λ*Flow* [60] which pushes down all the compatible operations (Filter, Project, and Aggregate) to the storage cluster (full NDP). In some results, we show the *Oracle*

policy, which is the offline optimal pushdown policy, for comparison. In some cases, we also include the results from partial pushdowns. *Filter Pushdown* policy performs only the Filter operation at the storage cluster where as *Filter+Project Pushdown* performs Filter and Project operations at the storage cluster but not Aggregate. We use the terms jobs and queries interchangeably in this section as we run "jobs" on Spark by submitting a TPC-H "query". Note that, for all jobs, the input dataset initially resides at the storage cluster and the final query output should be available at the compute cluster (since the user interfaces with Spark).

5.5.1 Simulation Results

We designed a discrete event simulator to simulate how Spark processes tasks using an HDFS cluster in a DI setting. A discrete event simulator models a system in the order of occurrence of events instead of progression of time. This allows us to simulate clusters of any size and very long events in a very short time, making our simulator highly scalable. We use SimPy [12] to create the simulator.

We simulate resources like CPU cores as simple discrete resources that can be requested by jobs. If a core is taken up by a task, all other task requests to the same core will be added to a queue and they will be served in a FIFO manner. Simulating network congestion is an important requirement since, in real life, concurrent data transfers between Spark executors and datanodes end up sharing the total available bandwidth. We chose to simulate the network by breaking the data transfers into smaller chunks, similar to the Maximum Transmission Unit (MTU) in computer networks, that reserve the network channel for a very short duration. We found that this abstraction handles network congestion quite well and thus the executors of the same job will also compete for network bandwidth similar to real-world networks.

Parameter	Value
Number of machines in Spark Cluster	35
Number of cores per machine in Spark Cluster	2
Clock Frequency of cores in Spark Cluster	2.27 GHz
Number of machines in HDFS Cluster	4
Number of cores per machine in HDFS Cluster	1-4
Input file size	150 GB
Clock Frequency of cores in HDFS Cluster	2.27 GHz
Bandwidth between the clusters	4 Gbps
Input size of a task	256 MB

Table 5.1: Cluster configuration of our experimental setup

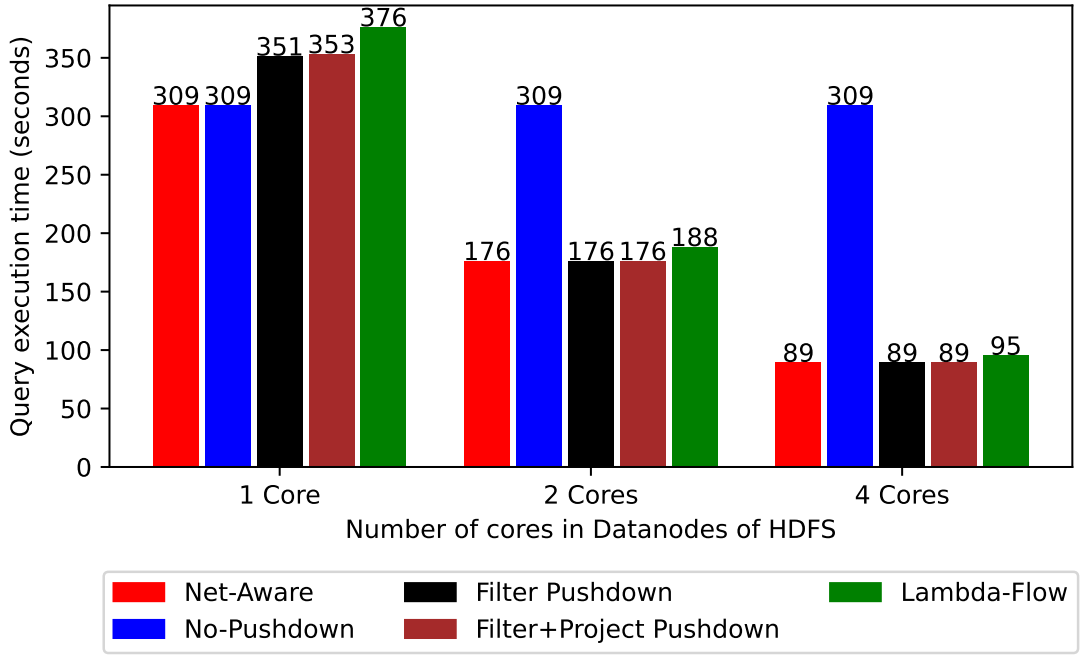


Figure 5.5: Simulated query execution time when a single job is being processed.

Single query results

We first simulate a cluster that has specifications similar to our experimental setup (see Section 5.5.2) and execute a single job. Table 5.1 shows the specifications of our experimental setup, which are also used in our simulator. We report simulation results averaged over 500 runs. Each query consist of three operations: Filter, Project, and Aggregate with respective reduction factors (ratio of input to output data size) of 60, 1, and 10,000 respectively. These numbers are inspired by the reduction factors of real-world Spark jobs that we encountered in production.

Figure 5.5 shows the average query execution time under different pushdown policies as a function of the number of cores for the HDFS cluster nodes. In the case of 1 core HDFS nodes, we see that the default No-Pushdown policy performs the best, and λFlow performs the worst. However, for the 2 core and 4 core HDFS nodes, No-Pushdown is the worst option. While λFlow performs well in these cases, since storage nodes have reasonable computing power, selective pushdown (where we push down a subset of the operators to HDFS cluster) actually performs the best. This is because selective pushdown optimally balances the tradeoff between reducing the data transfer time by pushing down the initial operations to storage and reducing the compute time by leveraging the increased processing capability (70 cores) of the compute cluster. This shows that NDP is beneficial when the

storage cluster has sufficient processing capacity; as the processing capacity at the storage cluster increases, the gains from NDP also increase.

In all cases, Net-Aware rightly picks the optimal policy, despite the optimal pushdown strategy being different. Compared to the No-Pushdown policy, Net-Aware reduces query execution time by about 71% for the case of 4 core HDFS nodes. Similarly, compared to the λFlow policy, Net-Aware reduces query execution time by about 18% for the case of 1 core HDFS nodes. Clearly, Net-Aware is superior to No-Pushdown and λFlow. Even compared to the best of No-Pushdown and λFlow, Net-Aware still provides about 6% improvement for the 2 and 4 core HDFS nodes cases.

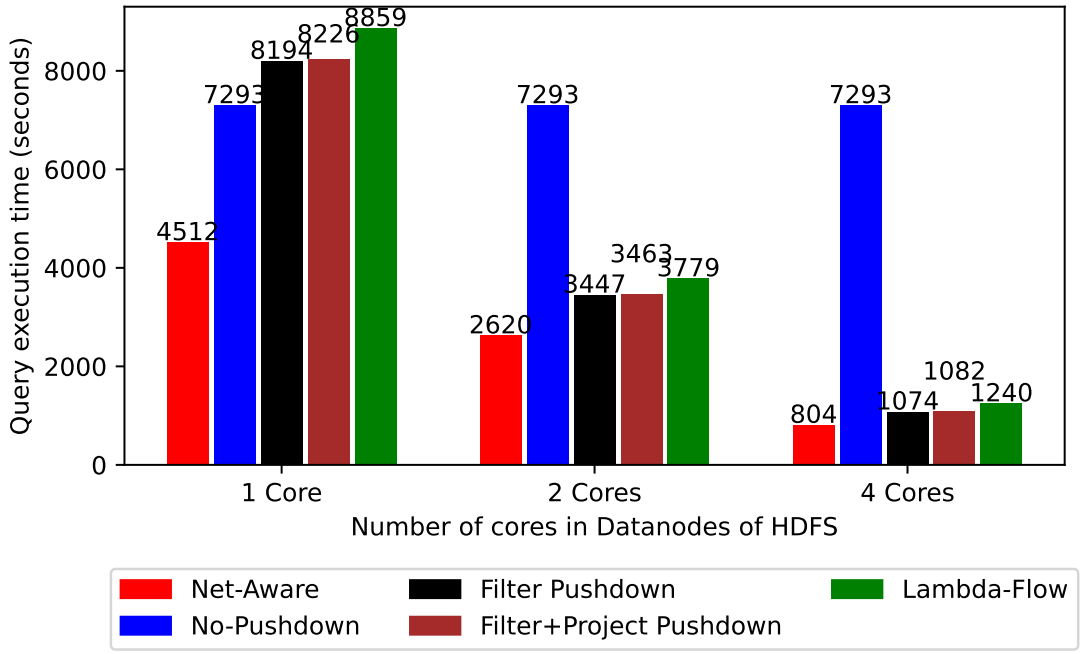


Figure 5.6: Simulated query execution time when multiple jobs are processed.

Multiple query results

We now consider scenarios where a stream of queries is processed as opposed to a single query, similar to the real-world scenario where multiple users are using the Spark cluster at the same time. In such cases, a job competes for resources with other active jobs already in the system. We launch 50 jobs in the simulation at a rate of 1 job every 50 seconds. We report the average query execution time across all 50 jobs. We use the same query and system configuration as in Section 5.5.1.

Figure 5.6 shows our results for the multiple jobs scenario. We see that the differences are more pronounced in this case due to the build of jobs in queues and the contention for shared compute and network resources. As for the case of single job, λFlow is better than

No-Pushdown only when HDFS nodes have more than 1 core. In all cases, Net-Aware is always the optimal policy. In fact, compared to the best of λFlow and No-Pushdown, Net-Aware provides about 38% reduction in query execution time for the 1 core case, a significant improvement.

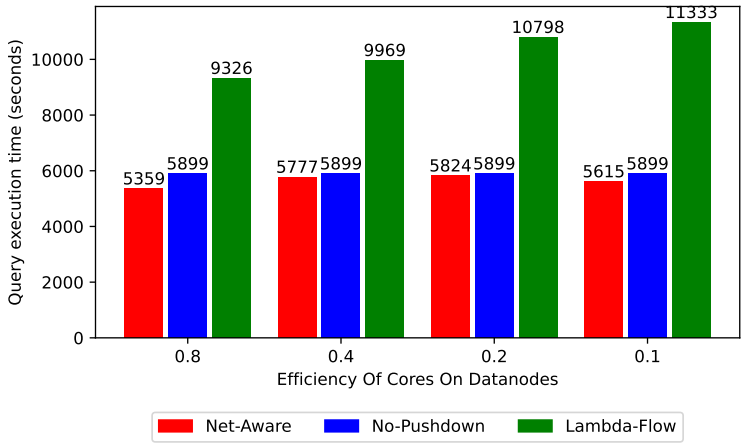
Interestingly, Net-Aware is superior to individual selective pushdown options as well. This is because, the same selective pushdown option might not be optimal for all 50 jobs in the stream. For example, while the first few jobs can benefit from (full) pushdown, subsequent jobs might be better served at the compute cluster as the storage cluster is busy processing the initial jobs. Net-Aware adapts to the current load at each cluster and can thus pick the optimal selective pushdown policy for *each* job. Compared to the best selective pushdown, the dynamic Net-Aware still provides at least 25% reduction in query execution time for all cases in Figure 5.6.

Queries with decompression under larger cluster setup

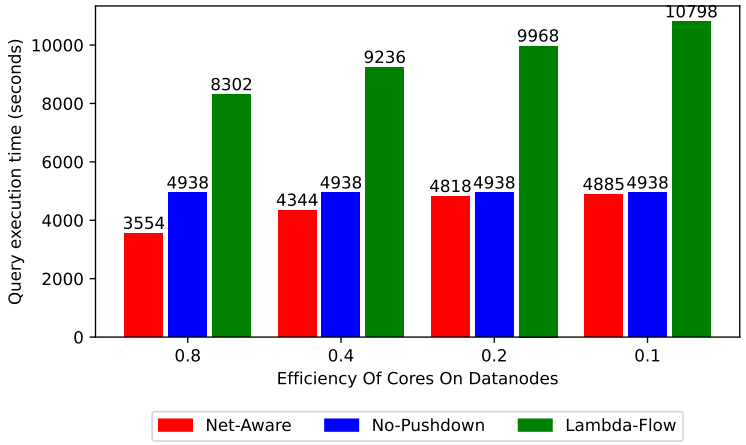
In production jobs, the input dataset is often compressed (e.g., in Parquet data format) to save space. In such cases, before executing the query, the dataset will first have to go through a compute-heavy decompression operation. The decompression operation may be better served at the compute cluster so that the larger output data size need not be transferred to compute cluster.

For this simulation, we consider queries similar to those in the previous simulations except that they initially have a decompression operation with reduction factor of 0.2 (meaning a compression factor of 5$\times$). We also consider a larger cluster setup: 150 compute nodes and 50 storage nodes. Figure 5.7 shows our simulation results for three different job sizes (using cycles/byte needed for operations as a proxy) and as a function of the efficiency of the storage nodes. This efficiency is a representation of the operation slow down caused by execution the operations at the lightweight storage cluster.

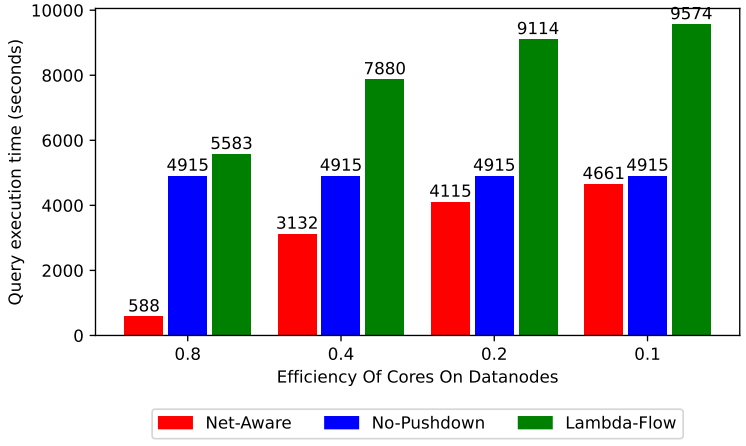
In all cases, we see that λFlow is significantly slower at processing queries due to the heavy decompression operation; as expected, λFlow does worse as the storage nodes' efficiency decreases. While the No-Pushdown option is reasonable in most cases, it does suffer performance degradation when the operations are light enough (Figure 5.7 (c)) that some of them will benefit from being pushed to storage to leverage the data transfer reduction when moving the query over to the compute cluster. In such cases, Net-Aware carefully balances the selective pushdowns to achieve as much as 8$\times$ reduction in query time over the best of No-Pushdown and λFlow.



(a) 100 cycles/byte needed for Filter and Project



(b) 50 cycles/byte needed for Filter and Project



(c) 20 cycles/byte needed for Filter and Project

Figure 5.7: Simulated query execution time when multiple jobs are processed and the input dataset is compressed.

5.5.2 Empirical Results

To evaluate SparkNDP and Net-Aware in real-world clusters (where complexity is much higher than in the simulated setup), we perform a limited evaluation in our experimental testbed.

Experimental setup

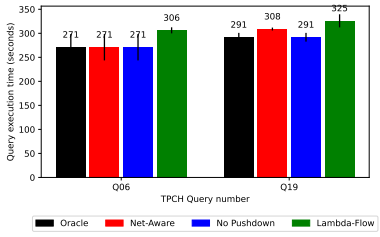
Our experimental setup consists of 10 servers. We employ 6 of them (each equipped with two Intel Xeon L5520 for a total of 16 cores) for the compute cluster. Each server runs 6 Spark containers, giving us a total of 35 Spark executors each with two cores and 0.5 GB memory. The remaining 4 servers comprise the storage cluster and each run 1 instance of HDFS datanode on a container using a maximum of 4 cores. We underclock the storage cluster CPU from 2.67 GHz to 1.6 GHz in some of our experiments using the CPUFreq governor of linux kernel [126] to experiment with different storage cluster settings. Each server is capable of achieving a network throughput of 1 Gbps, resulting in a maximum of 4Gbps across the clusters. The servers hosting Spark (compute cluster) and the servers hosting HDFS (storage cluster) are physically present on two different racks connected by a 10Gbps network. In some of our experiments, we use Traffic Control (TC) [14] and Network Emulation (NETEM) [13] modules of the Linux kernel to reduce the bandwidth between the clusters. The replication factor is set to 4 on HDFS to make the data available to all datanodes to perform NDP.

Experimental methodology

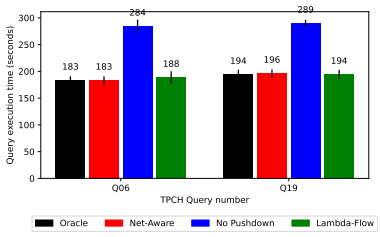
We use the prototype implementation described in Section 5.4 to evaluate SparkNDP experimentally on our aforementioned experimental setup. We use TPC-H Spark queries as our workload and run then on a 100 GB dataset generated by DBGEN. We obtain end-to-end query execution time from Spark, and use this as our metric. For comparison, we also run Spark without any pushdown (No-Pushdown) and Spark with full pushdown (λFlow [60]). Finally, we also denote the best-performing policy among all possible pushdown combinations as Oracle, which is the best offline optimal policy.

5.5.3 Experimental results

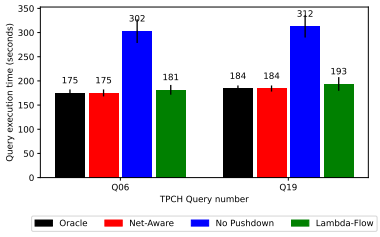
Figure 5.8 shows our empirical results for the case of a single job being processed; we show results for two different TPC-H queries, with Q19 being more computationally involved than Q06. The vertical lines at the top of each bar illustrates the standard deviation of that set of



(i) 1 core

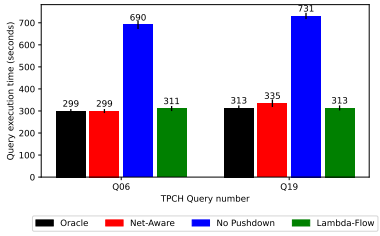


(ii) 2 cores

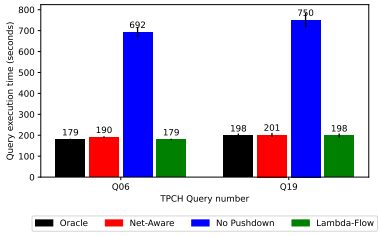


(iii) 4 cores

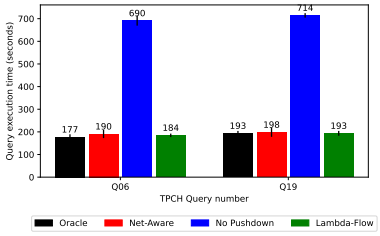
(a) Number of storage nodes = 4, storage nodes clock speed = 1.60 GHz, network bandwidth between clusters = 4 Gbps.



(i) 1 core

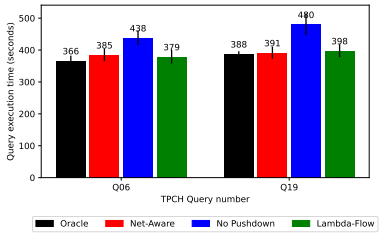


(ii) 2 cores

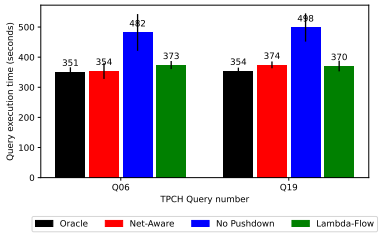


(iii) 4 cores

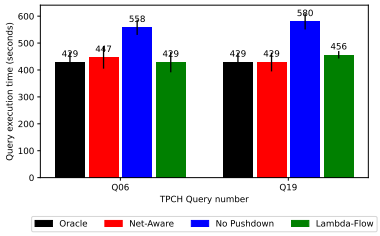
(b) Number of storage nodes = 4, storage nodes clock speed = 1.60 GHz, network bandwidth between clusters = 1 Gbps.



(i) 1 core



(ii) 2 cores



(iii) 4 cores

(c) Number of storage nodes = 2, storage nodes clock speed = 2.67 GHz, network bandwidth between clusters = 2 Gbps.

Figure 5.8: Empirical query execution time when a single job is being processed under various system configurations.

results (since each result is averaged over multiple experimental runs). Figure 5.8 (a) shows our results when we have 4 storage nodes underclocked to the lowest frequency (to emulate weaker storage cluster). We see that No-Pushdown is the best option when the storage nodes have only 1 core, but full pushdown (λFlow) is typically the best when we have more than 1 core. In some cases, selective pushdown is optimal, as in the case of Q19 when we have 4-core storage nodes. We see that, even in real experiments, Net-Aware typically is able to pick the best pushdown option, even if selective pushdown is the optimal option. Compared to No-Pushdown, λFlow, and best of No-Pushdown and λFlow, Net-Aware achieves as much as 42%, 11%, and 5% reduction in query execution time.

In Figure 5.8 (b), we reduce the bandwidth between the clusters to 1 Gbps. In this case, even single core HDFS nodes provide query execution time reduction over No-Pushdown since the network is a significant bottleneck. Net-Aware adapts to this slower network and ensures that operations are aggressively pushed down to storage; while Net-Aware is not always optimal, it does result in significant improvements over No-Pushdown.

Figure 5.8 (c) shows our results for the case of 2 HDFS nodes at full clock frequency. In this case, the effective bandwidth between the clusters is 2 Gbps since each HDFS node has a maximum bandwidth of 1 Gbps. This set of results shows that selective pushdown can indeed improve over No-Pushdown and full pushdown even in real experimental settings. In most cases, Net-Aware does recognize selective pushdown as the right policy, providing as much as 6% improvement over the best of No-Pushdown and λFlow.

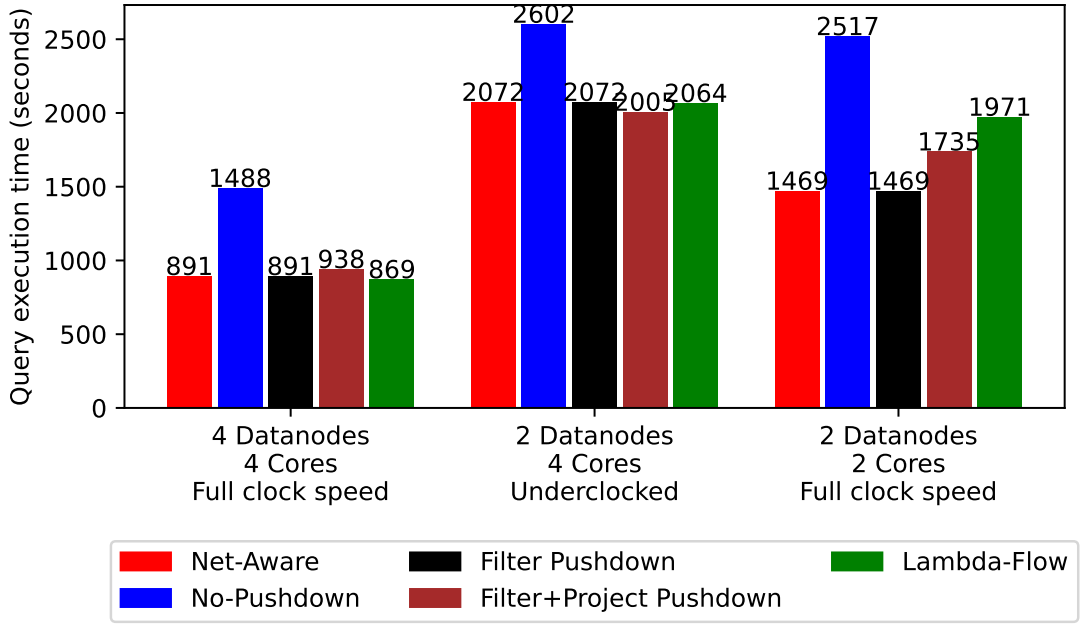


Figure 5.9: Empirical query execution time when multiple jobs are processed under various system configurations.

Finally, we also experiment with the case of multiple jobs being processed. As seen in simulations, the benefit of selective pushdown can be more pronounced in such cases. We consider Query 06 and run 10 jobs with an inter-arrival time of 50s. Figure 5.9 shows our results under three different settings (as denoted by the x-axis labels). We find that Net-Aware chooses only Filter pushdown as the right option in all three scenarios. By doing so, Net-Aware is always superior to No-Pushdown and near-optimal in all cases. For the case of 2 storage nodes with 2 cores each, Net-Aware indeed picks the optimal selective pushdown policy, providing around 25% reduction in query execution time over the best

of No-Pushdown and λFlow. This is a substantial improvement compared to the existing approaches of no NDP and full NDP.

5.6 Conclusion

Data processing and analysis is a frequently employed task in the industry, and improving the execution time of such tasks can substantially speed up analytics processes. Frameworks like Spark are routinely employed to run such data processing tasks on clusters of servers. Disaggregated infrastructure settings are being increasingly employed in practice to improve resource usage and avoid resource fragmentation; however, the network overhead in such settings poses a serious problem for data processing tasks that require data transfer from the storage cluster to the compute cluster. As such, an important problem is how to best design data processing frameworks for disaggregated infrastructure settings.

This work presents our experience and insights from designing and using an NDP-based Spark framework, SparkNDP, in a disaggregated infrastructure setting. We designed and open sourced SparkNDP, a framework built on top of Spark and HDFS that allows specific Spark operators to be pushed down to storage, thus enabling NDP for Spark, including selective push down of operations. SparkNDP works by leveraging SQLite to execute a subset of Spark operations in a lightweight manner on the storage cluster. We also constructed an analytical model to help SparkNDP decide which queries to use pushdown for and which specific operations of a query to push down, to optimize query execution time.

Our simulation results show that pushdown should be carefully employed based on the current system and network state and based on the query being executed. We find that full pushdown is only beneficial when the storage cluster has sufficient compute resources as needed for the pushed down query operators or when the initial dataset is too large to be quickly transferred over the network. We also find that *selective pushdown* is often a superior choice compared to the default no pushdown or full pushdown options, thus highlighting a novel NDP push down option that can benefit the community.

Our prototype experimental results show that there is significant query execution time reduction possible by employing the right pushdown strategy. Our implementation results also highlight the importance of taking into account the system and network state when deciding on the pushdown strategy to adopt for a given query.

Chapter 6

Model Right Sizing for Edge Devices

Machine Learning (ML) enables several edge inference applications ranging from self-driving cars to speech translation. These diverse applications have different requirements on model accuracy, latency, and energy usage. Likewise, the edge devices that host them may also have varying resource capabilities. In response, model developers typically create a handful of variants with different model sizes (e.g., number of layers). However, this approach barely covers the full space of model configurations, leading to unrealized gains.

This work presents EdgeFit, a framework to efficiently explore the size-parameter space of ML models to estimate their accuracy, latency, and energy consumption on edge devices without actually deploying each configuration. By leveraging model-agnostic kernel-level profiling and tracking the influence of device-agnostic model size-parameters on its accuracy, EdgeFit quickly discovers near-optimal model sizes. Empirical evaluation on different models across three edge devices shows that EdgeFit reduces the cost of parameter exploration by up to 77% compared to an exhaustive search, successfully finding near-optimal model configurations that meet use-case requirements while achieving 36% higher accuracy compared to existing approaches, such as pruning.

6.1 Introduction

Recent years have witnessed a significant increase in the use of machine learning (ML) applications at the edge, motivated by the need for low-latency and customized services. For instance, image classification models are crucial for scene understanding in Augmented Reality (AR) applications [82]. Similarly, language models are routinely employed for parsing unstructured documents [48]. However, each application, or use case, has very different performance requirements. Whereas AR use cases require *low latency* to match human cognitive processing for a seamless experience, document parsing can trade-off

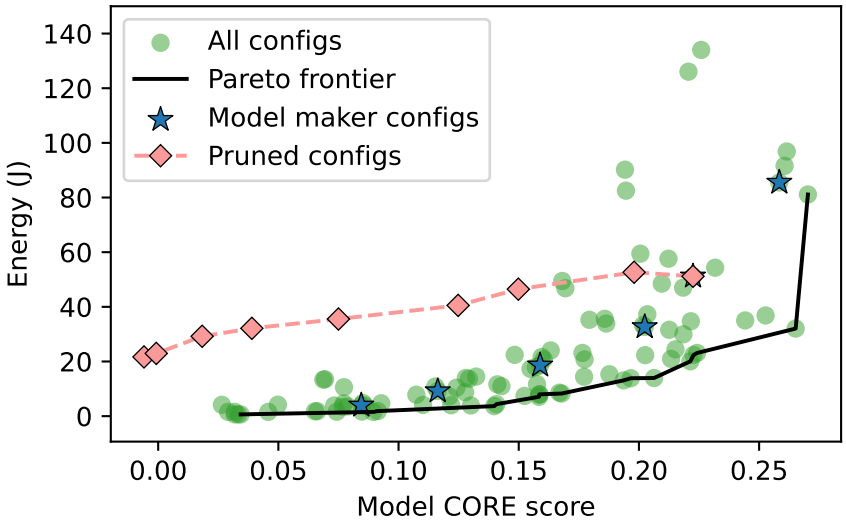


Figure 6.1: Scatter plot of energy consumed vs. CORE score for different model size-parameters of Nanochat on Orin NX; the Pareto frontier is highlighted. Also shown are the tradeoffs offered by pruning and model maker configurations.

latency for *lower energy* usage; nonetheless, both use cases benefit from increased model *accuracy*.

This ML-on-edge focus has naturally led to a commensurate rise in edge hardware development with ML acceleration capabilities, such as the NVIDIA Jetson device line with CUDA cores [107, 108]. Such edge devices differ in their computational capabilities and power consumption. An edge application provider must thus carefully choose the hosting device that meets their application requirements.

The core problem for an application provider in this context is ***how to achieve the required balance for their (latency, energy, and accuracy) metrics across edge devices***. To aid application providers, model makers often release their models in *different size offerings*. For example, Gemma 3, a popular LLM family, has differently-sized model parameter variants: 4B, 12B, and 27B [138]. The larger a model is, the more accurate it generally is, but it may also require more resources, consume more energy, and may take more time to run inference. However, a handful of model sizes is *barely sufficient* to cover the range of use case requirements across edge applications, resulting in application providers *settling for sub-optimal model variants*. While model makers can offer more variants, such offerings may not be optimal for all use cases; further, not all model sizes may be feasible, and some may be strictly Pareto-dominated by others.

To illustrate the complexity of choosing model sizes, consider the energy vs. CORE score (a model performance metric) scatter plot of model-size configurations for the Nanochat LLM on an NVIDIA Orin NX device, shown in Figure 6.1 (see Section 6.4). The explored space

of model-size configurations is shown via green dots, with the Pareto frontier highlighted by the solid line. While the model maker configuration offerings (blue stars) provide some degree of tradeoff between energy and model score, they are coarsely sampled across the entire space and are often far from the Pareto frontier, especially for the larger size variants (right of figure). We also observed similar trends for the other models (DeiT, TinyViT) we experimented with.

Prior work has explored different avenues to optimize specific ML application metrics for deployment. While techniques such as pruning and quantization and available for tuning models for inference, their effectiveness can vary widely; see, for example, the inferior trade-offs afforded by pruning for Nanochat in Figure 6.1. Other techniques, such as Dynamic Voltage and Frequency Scaling (DVFS) for energy optimization and model swapping for latency/throughput optimization, offer different point trade-offs, but do not solve the broader problem of achieving the desired balance between various metrics across different use cases.

This chapter presents the design and evaluation of *EdgeFit*, a framework that aims to find the optimal model-size variant for any model across given hardware; we refer to this approach as ***model right sizing***. EdgeFit explores the space of latency-energy-accuracy tradeoffs for a given model by using model-size parameters (such as the number of layers, the number of attention heads, the embedding dimension, etc.) as the tuning knob. To efficiently determine the right model size configuration that meets the user-specified requirements *without* exhaustive experimentation and training, EdgeFit develops accurate models to capture the relationship between model-size parameters and the various metrics.

A key enabling contribution of EdgeFit is the design of an ***estimator that efficiently estimates a given ML model's accuracy*** as a function of its size parameters. Since model accuracy can change non-linearly with various model-specific parameters, existing estimators [79] that rely only on FLOPs or number of parameters are insufficient. By carefully sampling the parameter space and leveraging active learning, our accuracy estimator allows EdgeFit to avoid the costly effort of training every model size configuration to determine its accuracy, achieving MAE as low as 1%. Further, the accuracy estimator is agnostic to the underlying hardware.

To estimate model latency and model energy consumption as a function of model size parameters, EdgeFit builds on existing kernel-level decomposition ideas [160, 87]. In particular, by developing and composing kernel-specific regression models for latency and energy, EdgeFit enables ***model-agnostic latency and energy estimation***. Our evaluations show that the latency and energy estimates are accurate, with an average MAPE of 8.34% and 10.04%, respectively, across the models and devices we experimented with.

We experimentally evaluate EdgeFit on nine diverse use cases using three Transformer-based model families on three different edge devices. Our empirical results show that the EdgeFit-recommended model configurations meet the latency and energy constraints for all use cases and provide model accuracy that is between 0.5-4.5% of the oracle solution for vision models. By contrast, the model maker and pruning configurations often fail to meet the required constraints, and on average provide 24.8% and 26.5% lower accuracy compared to EdgeFit. We envision EdgeFit being employed by model makers and device OEMs to right-size models by creating variants for diverse use cases and devices.

6.2 Related Work

Prior work related to our problem can be categorized into four groups: (i) hardware optimization, (ii) performance prediction, (iii) model scaling laws, and (iv) model optimization.

For hardware optimization, frameworks such as Zeus [155] and EcoEdgeInfer [123] focus on tuning hardware and runtime software settings, such as frequency and batch size, to maximize energy efficiency during DNN inference and training. Since these approaches operate independently of the underlying model architecture, their system-level optimizations are entirely complementary to our size-parameter optimization research. Among the literature on performance prediction, nn-meter [160], neuSight [87], LIFE [115], NeuralPower [35] and Tu et al. [142] develop predictive models to forecast the latency and energy consumption of DNN inference across different hardware platforms and model architectures. However, rather than utilizing these predictions to actively alter or right-size the model's architecture, these methods primarily apply them to select the most suitable candidate from a fixed pool of pre-trained models. Works such as Kaplan et al. [79] and Inference Scaling Laws [151] study the scaling laws of DNN training and inference, respectively, to understand the relationship between model size, dataset size, compute budget, and performance. However, these works do not explore how to leverage these scaling laws to optimize the model architecture for inference on edge devices. For model optimization, works such as IrEne [38] and DepGraph [51] optimize the architecture of DNN models by pruning unimportant parameters to reduce the model size, energy consumption, or latency. However, in our work, we show that pruning leads to significantly sub-optimal performance and accuracy trade-offs for edge inference.

By building on concepts presented in the above work, EdgeFit ***efficiently explores the size-parameter space*** to recommend a bespoke model-size configuration that satisfies user-specified constraints of the application on a device.

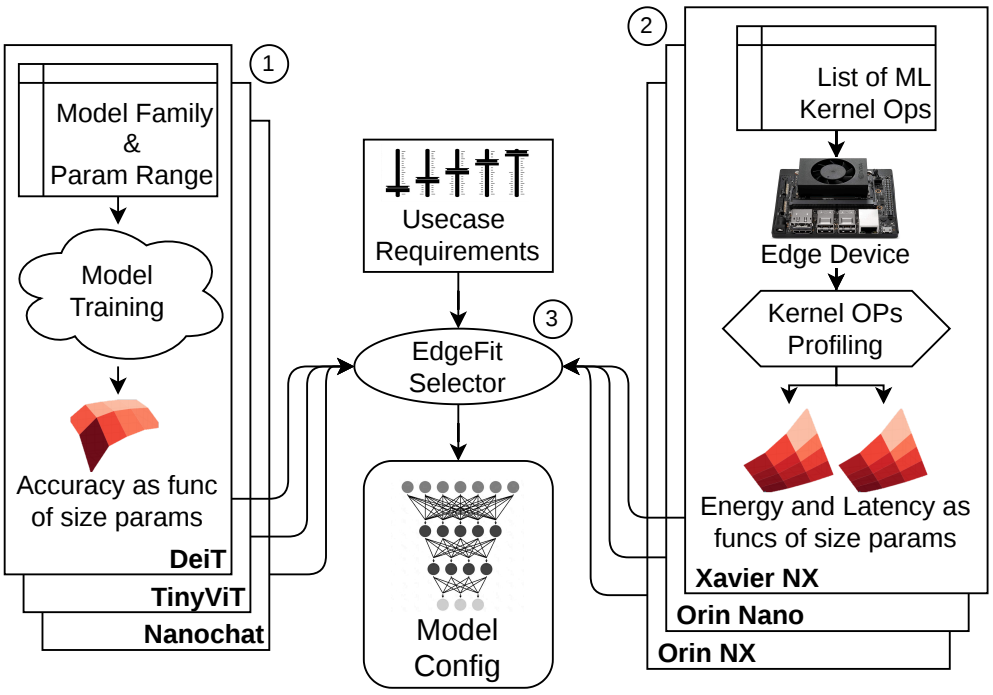


Figure 6.2: Illustration of EdgeFit's system design.

6.3 System Design

The system design of EdgeFit is shown in Figure 6.2. The framework consists of three distinct components, as numbered in Figure 6.2: (1) *Accuracy Estimator*, (2) *Hardware Performance Estimator*, and (3) *EdgeFit Selector*. We now explain the components and how they interact for model right sizing.

6.3.1 Accuracy Estimator

One key contribution of EdgeFit is efficiently estimating the accuracy of a given model as a function of its size-parameters. While one can exhaustively train all model configurations in the defined range to obtain their accuracies, this approach is computationally infeasible. Instead, we leverage the *smoothness of the accuracy curve* with respect to the size-parameters, a common observation in many model families [79]. Specifically, we train a chosen subset of model configurations and use their (post-training) accuracies to fit a curve that estimates the accuracy over the entire range of size-parameters. The choice of model configurations to train and how to fit the curve is crucial since as it affects the accuracy estimation and also the cost of training the chosen configurations.

While Bayesian optimization is a common approach for sample selection in finding minima or maxima of a function, it is not suitable for our problem since we are interested

in accurately estimating the *entire* accuracy curve rather than just finding the optimal point. (This is because the "optimal" point will vary depending on the use case.) Instead, we employ *active learning* and use ***Gaussian Process Regression (GPR)*** with Integrated Variance Reduction (IVR) to select the model configurations to train and fit the accuracy curve.

Unlike traditional curve-fitting methods like polynomial regression or spline interpolation, GPR provides both a mean accuracy estimate and *quantifies predictive uncertainty* (variance, $\sigma^2(x)$) at a given point (x). To initialize GPR, we select a preliminary set of model configurations equal in size to the number of size-parameters using Latin Hypercube Sampling (LHS) [98] and train these to obtain their accuracies. Then, to systematically minimize this uncertainty across the size-parameter domain D, we evaluate Integrated Variance as $IV(D) = \int_D \sigma^2(x) \cdot dx$. We then calculate the expected reduction in this integrated variance if a new sample x^* is added to the dataset: $IVR(x^*) = IV(D) - IV(D \cup \{x^*\})$. To select the next sample for training, we identify the candidate x_{next} that *maximizes* this variance reduction as $x_{next} = \arg\max_{x^* \in D} IVR(x^*)$. This active learning approach identifies the candidate x^* that maximally reduces the overall uncertainty of the estimated accuracy curve if added to the dataset. Consequently, we achieve accurate curve estimations with significantly fewer samples compared to randomly selecting an x_{next}. However, this approach assumes that all samples have the *same cost* of sampling, which is not the case when training different model sizes.

To address this, we introduce a ***cost-aware sampling strategy*** that incorporates the training cost of each model. Based on observations in prior literature [79], the cost of training a model is proportional to the total number of FLOPs incurred multiplied by the number of training epochs. Using FLOPs as a proxy for training cost and normalizing by it gives us the following expression for choosing x_{next}:

$$x_{next} = \arg\max_{x^* \in D} \left[IVR_{norm}(x^*) - Cost_{norm}(x^*) \right],$$

where $IVR_{norm}(x^*)$ and $Cost_{norm}(x^*)$ are the normalized integrated variance reduction and training cost, respectively. We refer to this approach as ***GPR EdgeFit***.

6.3.2 Hardware Performance Estimator

The second component of the system is responsible for estimating the latency and energy of *any* model inference on a given hardware. The model-agnostic feature of this component is crucial since model developers often lack access to deployment hardware and cannot feasibly profile every new model. To achieve this feature, we choose kernel-based profiling as our method for hardware performance estimation.

We first identify a set of representative kernel operations that capture much of the latency and energy consumption of inference across a wide range of models, as in prior works [160]. These kernels include conv2d, linear, sdpa, gelu, layernorm, and RMSnorm, which are common across model architectures. These representative kernels suffice to provide high accuracy, as evidenced by our evaluation results (see Table 6.2). Then, we profile these kernels on the target devices to create a performance database that maps kernel configurations (e.g., input size and output size) to their corresponding latency and energy usage. This database is then used to train two random forest regression (RFR) models, one that estimates the latency and one that estimates the energy use of any kernel configuration on the target hardware.

To apply the kernel-based estimators for an arbitrary model, we first decompose the model's inference process into its constituent kernel operations by running a single inference pass and recording the kernel calls using PyTorch's forward hooks, and then use the trained RFR estimators to estimate the kernels' latency and energy. Finally, we aggregate the contributions from all kernels via a simple sum operation (including multiple instances of the same kernel) to obtain the model-level estimates of latency and energy consumption on the target hardware.

6.3.3 EdgeFit Selector

The final component of the system is the EdgeFit selector, which is invoked when the application provider requests a configuration with specific requirements over latency demand, energy constraints, and accuracy expectations. While the requirements can take any arbitrary form, for evaluation, we consider constraints on allowable latency and energy consumption budget with the objective of maximizing accuracy. In response to such a request, the EdgeFit selector queries the accuracy estimator to obtain (i) the estimated accuracy curve for the model, and (ii) the hardware performance estimator to get the estimated latency and energy curves for the model on the target hardware. Then, it identifies the set of model (size-parameter) configurations that satisfy the specified latency and energy constraints and returns, from this set, the configuration with the highest estimated accuracy.

6.4 Experimental Methodology

Hardware Setup

We use three different Jetson devices for our edge evaluation: Jetson Xavier NX (8GB, Volta architecture), Jetson Orin Nano (8GB, Ampere architecture), and Jetson Orin NX

(16GB, Ampere) [108, 107]. These devices are chosen for their relevance in edge computing applications and their varying computational capabilities. All devices have APIs to collect energy consumption data.

Models and Size-Parameters

We evaluate over three models: DeiT [141], TinyViT [150], and Nanochat [81]. These Transformer-based models were chosen for their relevance in the field of computer vision and NLP. We evaluate the model efficacy using top-1 accuracy for DeiT and TinyViT. Since Nanochat is a generative language model, we use the DCLM CORE score [90] to evaluate its response quality.

For the size-parameters, we vary embedding dimension, number of attention heads, and number of layers for DeiT and TinyViT. For Nanochat, we additionally vary sequence length. We pick all combinations of parameters using grid sampling over the feasible range of size-parameter values. The range is chosen to at least include the model maker's recommended configurations and going up to the maximum size-configuration that can be feasibly trained on the given hardware. We then filter out invalid size-configurations based on architectural constraints. For example, for DeiT, the embedding dimension must be divisible by the number of attention heads. This resulted in a total of 127 size-configurations for DeiT, 101 for TinyViT, and 110 for Nanochat respectively. Since not all configurations of Nanochat could be run on Xavier NX and Orin Nano due to its large size, we skip this model on these two devices.

Model Training

DeiT and TinyViT are trained and evaluated on the ImageNet-1K dataset by distillation. For Nanochat, we use FineWeb-Edu-100B, a large-scale dataset that contains 100B GPT-2 tokens as primary pre-training dataset and a mixture dataset (SmolTalk, MMLU, GSM8K, etc.) as mid-training dataset, per the model maker's recommendation [81, 94]. Each DeiT model's training was run for 200 epochs, while TinyViT models were trained for 90 epochs. For Nanochat, we follow the model maker's recommendation of Chinchilla's scaling law [66], which suggests training tokens of 20 times the number of model parameters.

Model Right Sizing Comparison Baselines

For evaluating model right sizing, we consider the following baselines:
1. *Model Maker config* considers all size-configurations recommended by the model maker (typically 3–6 per model).

2. *Pruning config* starts by considering the largest model maker size-configuration and then iteratively pruning it in intervals of 10% pruning ratio, all the way down to 90% pruning ratio, resulting in 9 pruned configurations. For DeiT and TinyViT, we fine-tune the pruned models for 15 epochs after each step to recover the performance loss due to pruning. For Nanochat, we fine-tune for $\approx 15\%$ of the total steps in training the original model; this amounts to $\approx$ 20M tokens.

3. *Oracle config* is an offline, impractical baseline that has access to all explored (100+ per model) size-configurations.

All baselines and EdgeFit then pick the configuration, from their respective sample space (described above), that maximizes accuracy while meeting constraints on latency, say L, and energy, say E, of the use case, as discussed in Section 6.3.3:

$$x^* = \arg\max_{x \in S} \text{accuracy}(x);\ \text{latency}(x) \leq L,\ \ \text{energy}(x) \leq E$$

If no such configuration exists, we select the one closest (L2 norm) to the energy and latency constraints.

Accuracy Estimation Comparison Baselines

For evaluating our accuracy estimator, we use the following baselines:
1. *Random RF* is a random sampling method that uses a Random Forest regression model to estimate model accuracy.

2. *Random LR* also does random sampling, but uses Linear Regression to estimate the accuracy of size-configurations.

3. *Random SVR* is another random sampling method that uses a Support Vector Regression (SVR) model to estimate the accuracy of size-configurations. Random LR and Random SVR are included as baselines to evaluate the performance of Random Forest regression under simpler regression models.

4. *GRP IVR* is the active learning method that uses GPR and IVR to select configurations for training, but it does not have training-cost awareness in its selection process.

5. *GPR Greedy* is the active learning method that uses GPR and only selects low-cost configurations for training, but it does not have IVR in its selection process. It simply selects the lowest cost configuration at each step.

Use-case	Latency	Energy	Type
Industrial Fault Detection	1000 ms	0.5 J	Image
Photo Semantic Database	100 ms	0.8 J	Image
Smart Glass Gestures	33 ms	0.1 J	Image
AR Scene Understanding	33 ms	0.3 J	Image
Drone Navigation	20 ms	0.5 J	Image
Doorbell Camera	50 ms	0.06 J	Image
Self Driving Cars	30 ms	1 J	Image
Document Parsing	5000 ms	15 J	Text
Live Translation	500 ms	15 J	Text

Table 6.1: Use cases for end-to-end evaluation of EdgeFit.

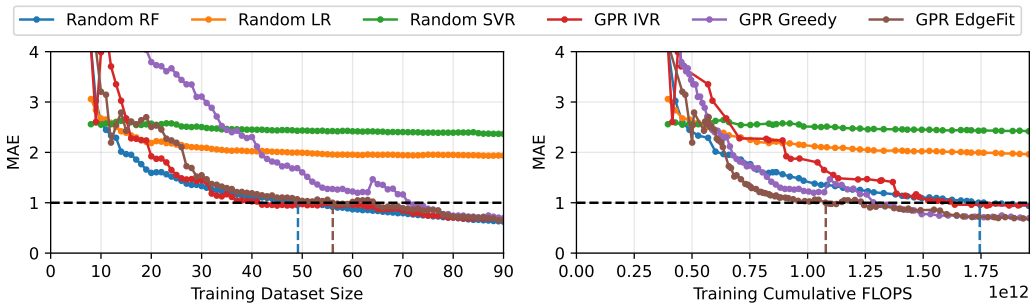


Figure 6.3: MAE of the various accuracy estimators as a function of the number of training points (left) and cumulative training cost (right) for the DeiT model.

Use cases for end-to-end evaluation

We evaluate EdgeFit end-to-end on a set of nine use cases inspired by practical edge computing applications [28, 92, 26, 52, 72, 101, 78]. These use cases cover a range of latency and energy requirements, as shown in Table 6.1, and different model types (image, text).

6.5 Evaluation

In this section, we first separately evaluate individual components of our framework, and then evaluate the overall performance and accuracy that can be achieved by EdgeFit.

6.5.1 Accuracy estimation

The accuracy estimation component of EdgeFit has two primary objectives: (1) to accurately estimate the accuracy of a given model configuration, and (2) to do so with minimal training cost. To evaluate the two objectives, we analyze how the error metrics of the accuracy estimator change as we train more samples, thereby incurring more training cost. Figure 6.3

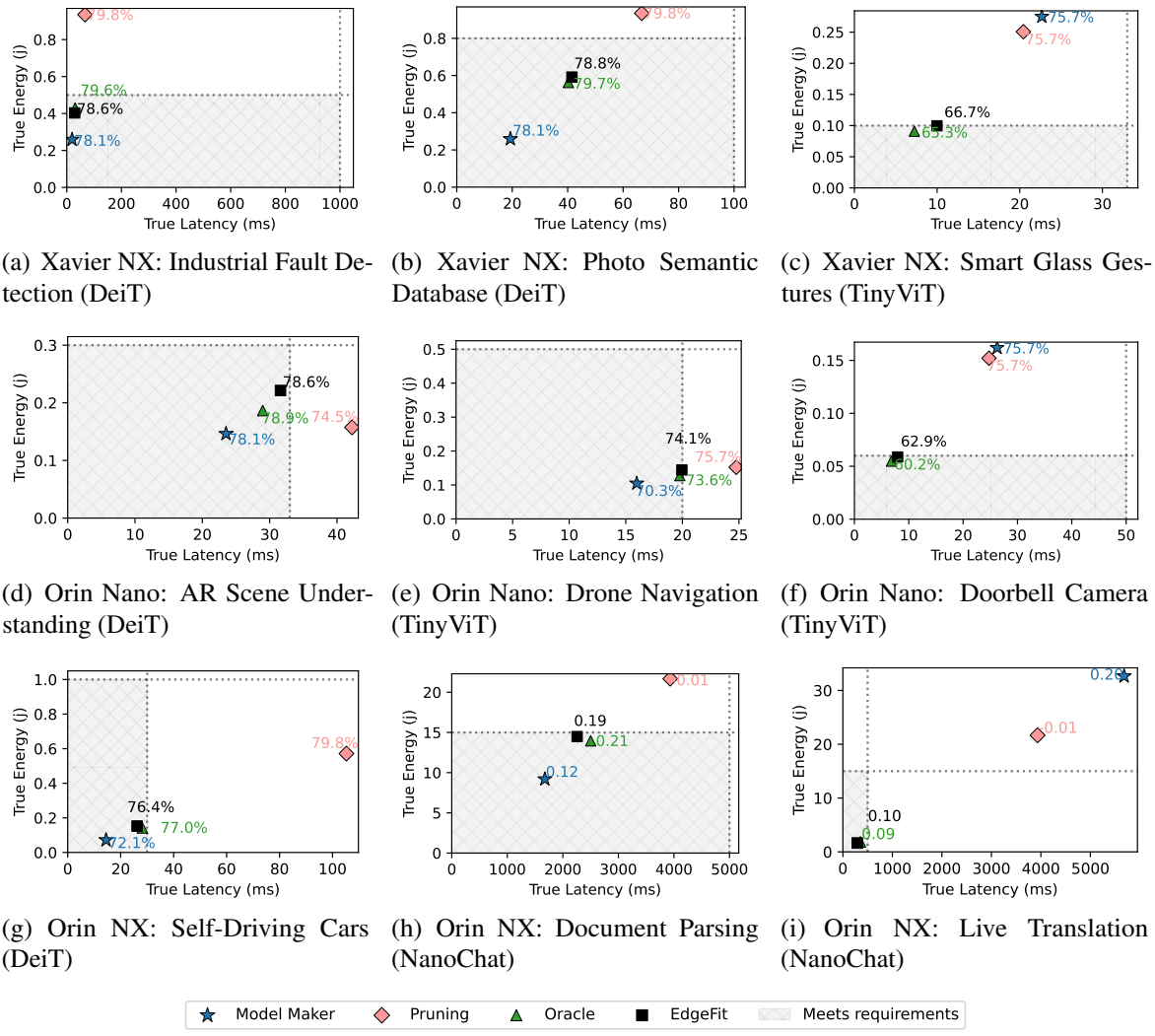
Device	Energy		Latency	
	MAE (mJ)	MAPE (%)	MAE (ms)	MAPE (%)
	DeiT			
Xavier NX	23.14	6.78	15.50	8.85
Orin Nano	13.65	10.70	15.47	11.69
Orin NX	12.82	8.32	13.40	9.49
	TinyViT			
Xavier NX	15.50	8.85	0.92	7.70
Orin Nano	15.47	11.69	1.29	6.43
Orin NX	13.40	9.49	1.91	7.79
	Nanochat			
Orin NX	3768.18	14.47	572.02	6.43

Table 6.2: Energy and Latency prediction error statistics across different edge devices and transformer architectures.

shows the progression of MAE for DeiT as a function of the number of training samples (left) and cumulative training cost (right) as different sample selection algorithms iteratively select samples and build the regressors. We see that as more samples are consumed, the MAE decreases at the cost of cumulative training cost. However, the progression of error metrics is different for different algorithms, some incurring higher training cost or requiring more samples to achieve the same error. We also note that there is ***not*** a linear relationship between the number of training samples and the cumulative training cost, as *different samples can have different training costs*. For example, as shown in the left subfigure, Random RF takes 49 samples to achieve an MAE of 1 while our proposed algorithm, GPR EdgeFit, takes 57 samples (16% higher) to achieve the same MAE. However, the cumulative training cost incurred (right subfigure) by Random RF is 59% higher than the cumulative training cost incurred by GPR EdgeFit for achieving the same MAE. For the rest of the chapter, we consider MAE=1 (in top-1 acc%) as the target accuracy threshold for DeiT and TinyViT, and MAE=0.012 (in CORE score) for Nanochat. Compared to Random RF and complete exhaustive search, respectively, GPR EdgeFit achieves MAE target with 38% and 77% lower cost for DeiT, 39% and 76% lower cost for TinyViT, and 35% and 68% lower cost for Nanochat. We observe similar trends for the other error metrics (such as MAPE and RMSE) as well.

6.5.2 Performance estimation

The performance estimation component of EdgeFit is responsible for accurately estimating the energy and latency of a given Transformer model configuration on a target edge device.



(a) Xavier NX: Industrial Fault Detection (DeiT)

(b) Xavier NX: Photo Semantic Database (DeiT)

(c) Xavier NX: Smart Glass Gestures (TinyViT)

(d) Orin Nano: AR Scene Understanding (DeiT)

(e) Orin Nano: Drone Navigation (TinyViT)

(f) Orin Nano: Doorbell Camera (TinyViT)

(g) Orin NX: Self-Driving Cars (DeiT)

(h) Orin NX: Document Parsing (NanoChat)

(i) Orin NX: Live Translation (NanoChat)

Figure 6.4: Energy vs. latency scatter plots for all use cases across three edge devices (one per row). Each subfigure shows the constraint boundary (shaded) and lists the achieved model accuracy/CORE score next to each technique's marker.

Table 6.2 shows the MAE and MAPE of the energy and latency regressors for each edge device and model. In all cases, the MAPE is below 15%, and often below 10%. The MAE of energy prediction is in 10s of mJ for DeiT and TinyViT, while the MAE of latency prediction is in 10s of ms for DeiT and in the 1–2 ms range for TinyViT. However, the MAE of energy prediction and latency prediction is higher for Nanochat, which is expected as Nanochat is a much larger model and thus has higher energy and latency values. Also, the larger Nanochat model does not fit in the memory of Xavier NX (8GB) and Orin Nano (8GB), so we only evaluate it on Orin NX (16GB). Overall, we find that the latency and energy estimator components of EdgeFit achieve reasonably low estimation error across different edge devices and models.

6.5.3 End-to-end evaluation of model right-sizing

We now evaluate the end-to-end performance of EdgeFit in right-sizing the model for all use cases listed in Table 6.1. Figure 6.4 shows the latency and energy achieved by the selected model-size configurations of EdgeFit and the baselines across different edge devices; in each subfigure, we shade the region that meets the use case's latency and energy constraints. We note that configurations in some use cases are clustered close to the energy threshold, while in others are clustered close to the latency threshold, indicating that such use cases are energy and latency constrained, respectively.

EdgeFit (square marker) successfully ***meets the specified constraints for all cases***. We also find that EdgeFit's achieved model accuracy (numbers next to markers in each subfigure) is, on average, ***within 3.5% of that of Oracle*** (triangles), highlighting its near-optimal performance.

By contrast, pruning (diamonds) does not meet the constraints for any use case. The model maker configurations (stars) perform better than pruning but are often far from the Oracle, failing to meet constraints in three of the use cases. For the six use cases where model maker configurations are within the constraints boundary, they provide worse accuracy/CORE score compared to both Oracle (by 17.9%) and EdgeFit (by 18.2%). This is because the model maker's space is restricted to a handful of configurations and is thus unable to discover superior configurations, unlike EdgeFit. On the other hand, EdgeFit efficiently navigates the space of configurations, converging close to the Oracle configuration and adhering to the constraint boundaries.

6.6 Conclusion

This chapter introduced EdgeFit, an extensible model right-sizing framework designed to meet the diverse latency, energy, and accuracy tradeoffs of ML applications on the edge. By decoupling hardware-agnostic accuracy estimation (via cost-aware active learning) from model-agnostic latency and energy estimation (via kernel-level profiling), EdgeFit successfully navigates the vast model size-parameters space without the prohibitive cost of exhaustive training and deployment. Our evaluations over three Transformer models, three edge devices, and nine use cases show that EdgeFit consistently discovers configurations that have higher accuracy (as much as 36%) while satisfying latency and energy constraints compared to baselines. Importantly, EdgeFit's performance is close to that of the Oracle, establishing it as a robust solution against future baselines.

Chapter 7

Conclusion

The rapid proliferation of AI applications has driven a critical shift towards edge computing to meet demands for low latency, high bandwidth, and enhanced privacy. However, the resource-constrained and heterogeneous nature of edge devices presents significant challenges in compute capability, memory, network connectivity, and energy consumption. This thesis confronts these challenges by developing a set of systems and algorithms to enable efficient AI inference and data preprocessing across the edge-cloud continuum.

Our research first tackled performance and energy optimization at the single-node level. We developed **EcoEdgeInfer**, a system that dynamically tunes hardware (CPU/GPU frequencies) and software (batch size) configurations in response to changing workloads. Its core algorithm, EcoGD, efficiently finds near-optimal settings, demonstrating significant energy savings of up to 31% and a 14% reduction in tail latency compared to baseline approaches.

Next, we addressed the complexities of distributed edge clusters where network performance is a critical bottleneck. For network-aware application deployment, we created **OVIDA**, a framework for disaggregated video analytics that intelligently places and replicates application components across heterogeneous nodes. OVIDA's placement policy consistently outperforms existing techniques, improving application throughput by up to 51% and accuracy by 28%. To further mitigate network bottlenecks in data-intensive edge workloads, we explored Near-Data Processing (**NDP**). Our analytical model and prototype system for Spark showed that selectively pushing down computations to storage nodes can reduce job execution times by up to 42%, highlighting the importance of data locality in disaggregated settings.

Collectively, these contributions provide a robust framework for optimizing the execution environment for AI workloads. However, they primarily operate on a fixed set of pre-existing AI models. A critical, unaddressed challenge is the inherent mismatch between generic, one-size-fits-all models and the vast diversity of edge hardware. The common practice of

selecting from a few developer-provided model variants is often suboptimal, failing to fully exploit a device's capabilities or meet specific application requirements.

To bridge this gap, we develop a framework, called **EdgeFit**, for Model Right-Sizing. It efficiently explores the size-parameter space of ML models to estimate their performance in terms of accuracy, latency, and energy consumption without actually training or deploying each configuration. When compared to baselines, EdgeFit reduces the exploration cost by up to 77% while achieving up to 36% higher accuracy, demonstrating its effectiveness in predicting hardware-aware model variants that are tailored to the specific constraints of edge devices.

In summary, this thesis proposal presents a multi-faceted approach to edge AI optimization, spanning the device, cluster, and model levels. By combining dynamic runtime optimization of the execution environment with a novel methodology for hardware-aware model synthesis, this work will provide a comprehensive solution for deploying high-performance and efficient AI on the heterogeneous edge. The outcomes will empower developers and device manufacturers to move beyond generic models and create bespoke AI solutions that unlock the full potential of resource-constrained computing platforms.

Bibliography

[1] Deploy models with Amazon SageMaker Serverless Inference - Amazon SageMaker — docs.aws.amazon.com. https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html. [Accessed 05-07-2024].

[2] Docker: Accelerated, containerized application development. Accessed: 2023-7-26.

[3] GitHub - City-of-Bellevue/TrafficVideoDataset — github.com. https://github.com/City-of-Bellevue/TrafficVideoDataset. [Accessed 05-07-2024].

[4] Internet Archive: Digital Library of Free & Borrowable Books, Movies, Music & Wayback Machine — archive.org. https://archive.org/details/twitterstream. [Accessed 05-07-2024].

[5] Lernapparat - Machine Learning — lernapparat.de. https://lernapparat.de/jit-optimization-intro/. [Accessed 05-07-2024].

[6] NVIDIA Embedded Systems for Next-Gen Autonomous Machines — nvidia.com. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. [Accessed 05-07-2024].

[7] NVIDIA Jetson. Accessed: 2024-10-18.

[8] NVIDIA TensorRT. Accessed: 2023-7-29.

[9] Production-Grade container orchestration. Accessed: 2023-7-26.

[10] Raspberry Pi. Accessed: 2024-7-3.

[11] Schedutil; The Linux Kernel documentation — docs.kernel.org. https://docs.kernel.org/scheduler/schedutil.html. [Accessed 05-07-2024].

[12] Simpy. https://simpy.readthedocs.io/en/latest/.

[13] tc-netem(8) - linux manual page. Accessed: 2023-7-26.

[14] tc(8) - linux manual page. https://man7.org/linux/man-pages/man8/tc.8.html. Accessed: 2024-7-1.

[15] TorchVision — pytorch.org. https://pytorch.org/vision/stable/index.html. [Accessed 05-07-2024].

[16] Twitter Streaming Api — developer.twitter.com. https://developer.twitter.com/en/docs/twitter-api/tweets/filtered-stream/introduction. [Accessed 05-07-2024].

[17] United states's mobile and broadband internet speeds. https://www.speedtest.net/global-index/united-states?fixed. Accessed: 2023-7-29.

[18] Video Analytics Market. Accessed: 2024-7-4.

[19] CCTV on London Underground | Freedom of Information, 2021.

[20] ADHIKARI, K., BOUCHACHIA, H., AND NAIT-CHARIF, H. Activity recognition for indoor fall detection using convolutional neural network. In *2017 Fifteenth IAPR International Conference on Machine Vision Applications (MVA)* (May 2017), IEEE, pp. 81–84.

[21] AHMAD, S., GUAN, H., FRIEDMAN, B. D., WILLIAMS, T., SITARAMAN, R. K., AND WOO, T. Proteus: A high-throughput inference-serving system with accuracy scaling. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (2024).

[22] ALABED, S., AND YONEKI, E. High-dimensional bayesian optimization with multi-task learning for rocksdb. EuroMLSys '21.

[23] ALI, A., PINCIROLI, R., YAN, F., AND SMIRNI, E. Batch: Machine learning inference serving on serverless platforms with adaptive batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), pp. 1–15.

[24] ALIPOURFARD, O., LIU, H. H., CHEN, J., VENKATARAMAN, S., YU, M., AND ZHANG, M. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (2017), NSDI'17.

[25] APPLIED INFORMATICS SOFTWARE ENGINEERING GMBH. POCO C++ Libraries - Simplify C++ Development. https://pocoproject.org/index.htmll.

[26] ASSEMBLYAI. Top tools for live transcription, 2026.

[27] BAHREINI, T., AND GROSU, D. Efficient placement of multi-component applications in edge computing systems. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing* (2017), pp. 1–11.

[28] BALA, M., CHANANA, A., CHEN, X., DONG, Q., EISZLER, T., XU, J., PILLAI, P., AND SATYANARAYANAN, M. The ooda loop of cloudlet-based autonomous drones. In *Proceedings of the 2024 IEEE/ACM Symposium on Edge Computing* (2024), SEC '24, IEEE Press, p. 178–190.

[29] BALASUBRAMONIAN, R., CHANG, J., MANNING, T., MORENO, J. H., MURPHY, R., NAIR, R., AND SWANSON, S. Near-data processing: Insights from a micro-46 workshop. *IEEE Micro 34*, 4 (2014), 36–42.

[30] BALLER, S., JINDAL, A., CHADHA, M., AND GERNDT, M. Deepedgebench: Benchmarking deep neural networks on edge devices. In *2021 IEEE International Conference on Cloud Engineering (IC2E)* (Los Alamitos, CA, USA, oct 2021), IEEE Computer Society, pp. 20–30.

[31] BARTOLOMEO, G., CAO, J., SU, X., AND MOHAN, N. Characterizing distributed mobile augmented reality applications at the edge. In *Companion of the 19th International Conference on emerging Networking EXperiments and Technologies* (2023), vol. 5, pp. 9–18.

[32] BARTOLOMEO, G., YOSOFIE, M., BÄURLE, S., HALUSZCZYNSKI, O., MOHAN, N., AND OTT, J. Oakestra: A lightweight hierarchical orchestration framework for edge computing. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (2023), pp. 215–231.

[33] BHARGAVA, P., DROZD, A., AND ROGERS, A. Generalization in nli: Ways (not) to go beyond simple heuristics, 2021.

[34] BOCHKOVSKIY, A., WANG, C., AND LIAO, H. M. Yolov4: Optimal speed and accuracy of object detection. *CoRR abs/2004.10934* (2020).

[35] CAI, E., JUAN, D.-C., STAMOULIS, D., AND MARCULESCU, D. Neuralpower: Predict and deploy energy-efficient convolutional neural networks, 2017.

[36] CANEL, C., KIM, T., ZHOU, G., LI, C., LIM, H., ANDERSEN, D., KAMINSKY, M., AND DULLOOR, S. R. Scaling video analytics on constrained edge nodes. *MLSys abs/1905.13536* (2019), 406–417.

[37] CAO, B. B., SHARMA, A., SINGH, M., GANDHI, A., DAS, S., AND JAIN, S. Representation similarity: A better guidance of dnn layer sharing for edge computing without training. *arXiv preprint arXiv:2410.11233* (2024).

[38] CAO, Q., LAL, Y. K., TRIVEDI, H., BALASUBRAMANIAN, A., AND BALASUBRAMANIAN, N. Irene: Interpretable energy prediction for transformers, 2021.

[39] CHEN, Y., XU, C., RAO, W., MIN, H., AND SU, G. Octopus: Hybrid big data integration engine. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)* (2015), pp. 462–466.

[40] CHO, B., AND SEYFE, E. Taking advantage of a disaggregated storage and compute architecture. *Spark+ AI Summit* (2019).

[41] CHOWDHURY, M., AND STOICA, I. Efficient coflow scheduling without prior knowledge. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (2015), SIGCOMM '15, pp. 393–406.

[42] CRYSTAL. Crystal: Software-defined storage for openstack swift. http://cloudlab.urv.cat/crystal/.

[43] DALIBARD, V., SCHAARSCHMIDT, M., AND YONEKI, E. BOAT: Building Auto-Tuners with Structured Bayesian Optimization. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia, 2017), WWW '17, p. 479–488.

[44] DE OLIVEIRA, I. O., LAROCA, R., MENOTTI, D., FONSECA, K. V. O., AND MINETTO, R. Vehicle-Rear: A new dataset to explore feature fusion for vehicle identification using convolutional neural networks. *IEEE Access 9* (2021), 101065–101077.

[45] DEAN, J., AND BARROSO, L. A. The Tail at Scale. *Communicatios of ACM 56*, 2 (2013), 74–80.

[46] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA, 2007), SOSP '07, pp. 205–220.

[47] DENG, J., DONG, W., SOCHER, R., LI, L.-J., LI, K., AND FEI-FEI, L. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition* (2009), pp. 248–255.

[48] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)* (2019), pp. 4171–4186.

[49] DIEGUES, N., AND ROMANO, P. Self-Tuning intel transactional synchronization extensions. In *11th International Conference on Autonomic Computing (ICAC 14)* (Philadelphia, PA, June 2014), USENIX Association, pp. 209–219.

[50] DUTT, A., RACHURI, S. P., LOBO, A., SHAIK, N., GANDHI, A., AND LIU, Z. Evaluating the energy impact of device parameters for dnn inference on edge. In *Proceedings of the 14th International Green and Sustainable Computing Conference* (New York, NY, USA, 2024), IGSC '23, Association for Computing Machinery, p. 52–55.

[51] FANG, G., MA, X., SONG, M., MI, M. B., AND WANG, X. Depgraph: Towards any structural pruning, 2023.

[52] FARRA, N., RAFFA, G., NACHMAN, L., AND HAJJ, H. Energy-efficient mobile gesture recognition with computation offloading. In *2011 International Conference on Energy Aware Computing* (2011), pp. 1–6.

[53] FORTUNE BUSINESS INSIGHTS. CCTV Camera Market Size, Growth | Global Report.

[54] FOUNDATION, L. Sharpening the edge: Overview of the lf edge taxonomy and framework, Jul 2020.

[55] GAO, P. X., NARAYAN, A., KARANDIKAR, S., CARREIRA, J., HAN, S., AGARWAL, R., RATNASAMY, S., AND SHENKER, S. Network requirements for resource disaggregation. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (Savannah, GA, Nov. 2016), USENIX Association, pp. 249–264.

[56] GHASEMI, M., KLEISARCHAKI, S., CALMANT, T., GÜRGEN, L., GHADERI, J., KOSTIC, Z., AND ZUSSMAN, G. Real-time camera analytics for enhancing traffic intersection safety. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services* (2022), p. 630–631.

[57] GITHUB. GitHub Copilot. https://copilot.github.com/. [Accessed 05-07-2024].

[58] GLOVER, F. Tabu search - part i. In *ORSA Journal on Computing* (1989), vol. 1, pp. 190–206.

[59] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[60] GRACIA-TINEDO, R., SANCHEZ-ARTIGAS, M., GARCIA-LOPEZ, P., MOATTI, Y., AND GLUSZAK, F. Lamda-flow: Automatic pushdown of dataflow operators close to the data. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)* (2019), pp. 112–121.

[61] GU, D., XIE, X., HUANG, G., JIN, X., AND LIU, X. Energy-efficient gpu clusters scheduling for deep learning, 2023.

[62] HAIDAR, A., TOMOV, S., DONGARRA, J., AND HIGHAM, N. J. Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (2018), pp. 603–613.

[63] HAIGHT, R., HAENSCH, W., AND FRIEDMAN, D. Solar-powering the internet of things. *Science 353*, 6295 (2016), 124–125.

[64] HARCHOL-BALTER, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.

[65] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition, 2015.

[66] HOFFMANN, J., BORGEAUD, S., MENSCH, A., BUCHATSKAYA, E., CAI, T., RUTHERFORD, E., DE LAS CASAS, D., HENDRICKS, L. A., WELBL, J., CLARK, A., HENNIGAN, T., NOLAND, E., MILLICAN, K., VAN DEN DRIESSCHE, G., DAMOC, B., GUY, A., OSINDERO, S., SIMONYAN, K., ELSEN, E., RAE, J. W., VINYALS, O., AND SIFRE, L. Training compute-optimal large language models, 2022.

[67] HSIEH, K., HARLAP, A., VIJAYKUMAR, N., KONOMIS, D., GANGER, G. R., GIBBONS, P. B., AND MUTLU, O. Gaia: Geo-Distributed machine learning approaching LAN speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, Mar. 2017), USENIX Association, pp. 629–647.

[68] HU, Z., YE, N., PHILLIPS, C., CAPES, T., AND MOHOMED, I. mmFilter: Language-guided video analytics at the edge. In *Proceedings of the 21st International Middleware Conference Industrial Track* (2020), Middleware '20, p. 1–7.

[69] HUANG, J., MAJUMDER, P., KIM, S., FULTON, T., PULI, R. R., YUM, K. H., AND KIM, E. J. Computing en-route for near-data processing. *IEEE Transactions on Computers 70*, 6 (2021), 906–921.

[70] HUANG, Z., LIU, Y., FANG, Y., AND HORN, B. K. P. Video-based fall detection for seniors with human pose estimation. In *2018 4th International Conference on Universal Village (UV)* (2018), pp. 1–4.

[71] HUNG, C.-C., ANANTHANARAYANAN, G., BODIK, P., GOLUBCHIK, L., YU, M., BAHL, P., AND PHILIPOSE, M. VideoEdge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)* (2018), pp. 115–131.

[72] IMMICH. Searching, 2026.

[73] JALAPARTI, V., DOUGLAS, C., GHOSH, M., AGRAWAL, A., FLORATOU, A., KANDULA, S., MENACHE, I., NAOR, J. S., AND RAO, S. Netco: Cache and i/o management for analytics over disaggregated stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA, 2018), SoCC '18, p. 186–198.

[74] JEONG, E., KIM, J., AND HA, S. Tensorrt-based framework and optimization methodology for deep learning inference on jetson boards. *ACM Trans. Embed. Comput. Syst. 21*, 5 (oct 2022).

[75] JIANG, J., ANANTHANARAYANAN, G., BODIK, P., SEN, S., AND STOICA, I. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 253–266.

[76] JOCHER, G. YOLOv5 by Ultralytics, May 2020.

[77] JOCHER, G., CHAURASIA, A., AND QIU, J. Ultralytics yolov8, 2023.

[78] JOUINI, O., SETHOM, K., NAMOUN, A., ALJOHANI, N., ALANAZI, M. H., AND ALANAZI, M. N. A survey of machine learning in edge computing: Techniques, frameworks, applications, issues, and research directions. *Technologies 12*, 6 (2024).

[79] KAPLAN, J., MCCANDLISH, S., HENIGHAN, T., BROWN, T. B., CHESS, B., CHILD, R., GRAY, S., RADFORD, A., WU, J., AND AMODEI, D. Scaling laws for neural language models, 2020.

[80] KARAKUS, C., HUILGOL, R., WU, F., SUBRAMANIAN, A., DANIEL, C., ÇAVDAR, D., XU, T., CHEN, H., RAHNAMA, A., AND QUINTELA, L. Amazon sagemaker model parallelism: A general and flexible framework for large model training. *CoRR abs/2111.05972* (2021).

[81] KARPATHY, A. nanochat: The best chatgpt that $100 can buy, 2025.

[82] KIM, B.-S., KOHLI, P., AND SAVARESE, S. 3D scene understanding by voxel-CRF. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)* (2013), pp. 1425–1432.

[83] KLIMOVIC, A., LITZ, H., AND KOZYRAKIS, C. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, USA, 2018), pp. 759–773.

[84] KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (2012), F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Eds., vol. 25, Curran Associates, Inc.

[85] KU, Y.-J., AND DEY, S. Sustainable vehicular edge computing using local and solar-powered roadside unit resources. In *2019 IEEE 90th Vehicular Technology Conference (VTC2019-Fall)* (2019), pp. 1–7.

[86] KUHN, H. W. The hungarian method for the assignment problem. *Nav. Res. Logist. Q. 2*, 1-2 (Mar. 1955), 83–97.

[87] LEE, S., PHANISHAYEE, A., AND MAHAJAN, D. Forecasting gpu performance for deep learning training and inference. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Mar. 2025), ASPLOS '25, ACM, p. 493–508.

[88] LETHAM, B., AND BAKSHY, E. Bayesian optimization for policy search via online-offline experimentation, 2019.

[89] LI, D., CHEN, X., BECCHI, M., AND ZONG, Z. Evaluating the energy efficiency of deep convolutional neural networks on cpus and gpus. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)* (2016), pp. 477–484.

[90] LI, J., FANG, A., SMYRNIS, G., AND *et al.*. Datacomp-lm: In search of the next generation of training sets for language models. *arXiv preprint arXiv:2406.11794* (2024).

[91] LIN, T.-Y., MAIRE, M., BELONGIE, S., BOURDEV, L., GIRSHICK, R., HAYS, J., PERONA, P., RAMANAN, D., ZITNICK, C. L., AND DOLLÁR, P. Microsoft COCO: Common objects in context.

[92] LIU, L., DONG, Z., WANG, Y., AND SHI, W. Prophet: Realizing a predictable real-time perception pipeline for autonomous vehicles. In *2022 IEEE Real-Time Systems Symposium (RTSS)* (2022), pp. 305–317.

[93] LIU, P., QI, B., AND BANERJEE, S. EdgeEye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking* (2018), pp. 1–6.

[94] LOZHKOV, A., BEN ALLAL, L., VON WERRA, L., AND WOLF, T. Fineweb-edu: the finest collection of educational content, 2024.

[95] LUCCIONI, S., JERNITE, Y., AND STRUBELL, E. Power hungry processing: Watts driving the cost of ai deployment? In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency* (New York, NY, USA, 2024), FAccT '24, Association for Computing Machinery, p. 85–99.

[96] LUCHKO, I. Digital camera autodetect. Accessed: 2024-6-22.

[97] MARCO, V. S., TAYLOR, B., WANG, Z., AND ELKHATIB, Y. Optimizing deep learning inference on embedded systems through adaptive model selection. *ACM Trans. Embed. Comput. Syst. 19*, 1 (2020), 1–28.

[98] MCKAY, M. D., CONOVER, W. J., AND BECKMAN, R. J. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics 21*, 2 (1979), 239–245.

[99] MEISNER, D., SADLER, C. M., BARROSO, L. A., WEBER, W.-D., AND WENISCH, T. F. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, CA, USA, 2011), ISCA '11, pp. 319–330.

[100] MISRA, S., RACHURI, S. P., DEB, P. K., AND MUKHERJEE, A. Multiarmed-bandit-based decentralized computation offloading in fog-enabled iot. *IEEE Internet of Things Journal 8*, 12 (2021), 10010–10017.

[101] NABIYOUNI, M., SCERBO, S., BOWMAN, D. A., AND HÖLLERER, T. Relative effects of real-world and virtual-world latency on an augmented reality training task: An ar simulation experiment. *Frontiers in ICT Volume 3 - 2016* (2017).

[102] NGUYEN, Q.-M., PHAN, L.-A., AND KIM, T. Load-Balancing of Kubernetes-Based edge computing infrastructure using resource adaptive proxy. *Sensors 22*, 8 (2022).

[103] NVIDIA. MLPerf Benchmarks. https://www.nvidia.com/en-us/data-center/resources/mlperf-benchmarks.

[104] NVIDIA. NVIDIA Jetson Nano System-on-Module. https://developer.download.nvidia.com/assets/embedded/secure/jetson/Nano/docs.

[105] NVIDIA. NVIDIA Jetson Xavier NX System-on-Module. https://en.miivii.com/uploads/file/20200511/NV_Jetson_Xavier_NX_DataSheet_v0.1.pdf.

[106] NVIDIA. Software-based power consumption modeling. https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3275/index.html. [Accessed 05-07-2024].

[107] NVIDIA. Nvidia jetson orin series, 2024.

[108] NVIDIA. Nvidia jetson xavier series, 2024.

[109] NYIKOSA, F. M., OSBORNE, M. A., AND ROBERTS, S. J. Bayesian optimization for dynamic problems, 2018.

[110] OPENAI. ChatGPT. https://openai.com/chatgpt/. [Accessed 05-07-2024].

[111] ORACLE. Java class InputStream. https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html.

[112] PADMANABHAN, A., AGARWAL, N., IYER, A., ANANTHANARAYANAN, G., SHU, Y., KARIANAKIS, N., XU, G. H., AND NETRAVALI, R. Gemel: Model merging for memory-efficient, real-time video analytics at the edge. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 973–994.

[113] PANOPOULOS, I., NIKOLAIDIS, S., VENIERIS, S. I., AND VENIERIS, I. S. Exploring the performance and efficiency of transformer models for nlp on mobile devices, 2023.

[114] PASZKE, A., GROSS, S., MASSA, F., LERER, A., BRADBURY, J., CHANAN, G., KILLEEN, T., LIN, Z., GIMELSHEIN, N., ANTIGA, L., DESMAISON, A., KOPF, A., YANG, E., DEVITO, Z., RAISON, M., TEJANI, A., CHILAMKURTHY, S., STEINER, B., FANG, L., BAI, J., AND CHINTALA, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems* (2019), H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc.

[115] PATWARI, R., SIRASAO, A., AND DAS, D. Forecasting llm inference performance via hardware-agnostic analytical modeling, 2025.

[116] PECH-PACHECO, J. L., CRISTOBAL, G., CHAMORRO-MARTINEZ, J., AND FERNANDEZ-VALDIVIA, J. Diatom autofocusing in brightfield microscopy: a comparative study. In *Proceedings 15th International Conference on Pattern Recognition. ICPR-2000* (2002), vol. 3, pp. 314–317.

[117] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[118] PUGLIESE, R., REGONDI, S., AND MARINI, R. Machine learning-based approach: global trends, research directions, and regulatory standpoints. *Data Science and Management 4* (2021), 19–29.

[119] PYTORCH. Models and pre-trained weights - torchvision 0.15 documentation.

[120] RABKIN, A., ARYE, M., SEN, S., PAI, V. S., AND FREEDMAN, M. J. Aggregation and degradation in JetStream: Streaming analytics in the wide area. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Seattle, WA, Apr. 2014), USENIX Association, pp. 275–288.

[121] RACHURI, S. P., BRONZINO, F., AND JAIN, S. Decentralized modular architecture for live video analytics at the edge. In *Proceedings of the 3rd ACM Workshop on Hot Topics in Video Analytics and Intelligent Edges* (New York, NY, USA, 2021), HotEdgeVideo '21, Association for Computing Machinery, p. 13–18.

[122] RACHURI, S. P., GANTASALA, A., EMANUEL, P., GANDHI, A., FOLEY, R., PUHOV, P., GKOUNTOUVAS, T., AND LEI, H. Optimizing near-data processing for spark. In *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)* (2022), pp. 636–646.

[123] RACHURI, S. P., SHAIK, N., CHOKSI, M., AND GANDHI, A. Ecoedgeinfer: Dynamically optimizing latency and sustainability for inference on edge devices. In *2024 IEEE/ACM Symposium on Edge Computing (SEC)* (2024), pp. 191–205.

[124] RADOVANOVIĆ, A., KONINGSTEIN, R., SCHNEIDER, I., CHEN, B., DUARTE, A., ROY, B., XIAO, D., HARIDASAN, M., HUNG, P., CARE, N., TALUKDAR, S., MULLEN, E., SMITH, K., COTTMAN, M., AND CIRNE, W. Carbon-aware computing for datacenters. *IEEE Transactions on Power Systems 38*, 2 (2023), 1270–1280.

[125] RAHMANIAN, A., ALI-ELDIN, A., TESFATSION, S. K., SKUBIC, B., GUSTAFSSON, H., SHENOY, P., AND ELMROTH, E. RAVAS: Interference-Aware model selection and resource allocation for live edge video analytics. In *2023 IEEE/ACM Symposium on Edge Computing (SEC)* (2023), pp. 27–39.

[126] LINUX KERNEL ORGANIZATION INC. Cpufreq governors. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[127] OPEN INFRASTRUCTURE LABS. Caerus dike. https://github.com/open-infrastructure-labs/caerus-dike/tree/v1.

[128] ROMERO, F., LI, Q., YADWADKAR, N. J., AND KOZYRAKIS, C. INFaaS: Automated model-less inference serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 397–411.

[129] SAMIKWA, E., MAIO, A. D., AND BRAUN, T. Disnet: Distributed micro-split deep learning in heterogeneous dynamic iot. *IEEE Internet of Things Journal 11*, 4 (2024), 6199–6216.

[130] SANDLER, M., HOWARD, A., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[131] SANH, V., DEBUT, L., CHAUMOND, J., AND WOLF, T. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[132] SHAN, Y., HUANG, Y., CHEN, Y., AND ZHANG, Y. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, USA, 2018), pp. 69–87.

[133] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Incline Village, NV, USA, 2010), pp. 1–10.

[134] SINGH, M., RACHURI, S. P., CAO, B. B., SHARMA, A., BHUMIREDDY, V., BRONZINO, F., DAS, S. R., GANDHI, A., AND JAIN, S. Ovida: Orchestrator for video analytics on disaggregated architecture. In *Proceedings of the 2024 IEEE/ACM Symposium on Edge Computing* (2025), SEC '24, IEEE Press, p. 135–148.

[135] S.K, P., KESANAPALLI, S. A., AND SIMMHAN, Y. Characterizing the performance of accelerated jetson edge devices for training deep learning models. *Proc. ACM Meas. Anal. Comput. Syst. 6*, 3 (dec 2022).

[136] SOMASHEKAR, G., TANDON, K., KINI, A., CHANG, C.-C., HUSAK, P., BHAGWAN, R., DAS, M., GANDHI, A., AND NATARAJAN, N. OPPerTune: Post-Deployment Configuration Tuning of Services Made Easy. In *NSDI 2024* (Santa Clara, CA, USA, 2024), USENIX.

[137] SUN, H., YU, Y., SHA, K., AND LOU, B. mVideo: Edge computing based mobile video processing systems. *IEEE Access 8* (2020), 11615–11623.

[138] TEAM, G., KAMATH, A., FERRET, J., PATHAK, S., AND *et al.*. Gemma 3 technical report, 2025.

[139] THE APACHE SOFTWARE FOUNDATION. Apache Spark. http://spark.apache.org/.

[140] THE APACHE SOFTWARE FOUNDATION. WebHDFS REST API. https://hadoop.apache.org/docs/r1.0.4/webhdfs.html.

[141] TOUVRON, H., CORD, M., DOUZE, M., MASSA, F., SABLAYROLLES, A., AND JÉGOU, H. Training data-efficient image transformers & distillation through attention, 2021.

[142] TU, X., MALLIK, A., CHEN, D., HAN, K., ALTINTAS, O., WANG, H., AND XIE, J. Unveiling energy efficiency in deep learning: Measurement, prediction, and scoring across edge devices. In *Proceedings of the Eighth ACM/IEEE Symposium on Edge Computing* (Dec. 2023), SEC '23, ACM, p. 80–93.

[143] TURC, I., CHANG, M., LEE, K., AND TOUTANOVA, K. Well-read students learn better: The impact of student initialization on knowledge distillation. *CoRR abs/1908.08962* (2019).

[144] TURC, I., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Well-read students learn better: On the importance of pre-training compact models, 2019.

[145] VANHOY, G., LICHTMAN, M., HOARE, R. R., AND BREVIK, C. Rapid prototyping framework for intelligent arrays with heterogeneous computing. In *2022 IEEE International Symposium on Phased Array Systems & Technology (PAST)* (2022), pp. 01–06.

[146] VINÇON, T., BERNHARDT, A., PETROV, I., WEBER, L., AND KOCH, A. Nkv: Near-data processing with kv-stores on native computational storage. In *Proceedings of the 16th International Workshop on Data Management on New Hardware* (New York, NY, USA, 2020), DaMoN '20, Association for Computing Machinery.

[147] VINCENT, J. How much electricity does AI consume? — theverge.com. https://www.theverge.com/24066646/ai-electricity-energy-watts-generative-consumption. [Accessed 05-07-2024].

[148] WANG, Q., AND CHU, X. Gpgpu performance estimation with core and memory frequency scaling. *IEEE Transactions on Parallel and Distributed Systems 31*, 12 (2020), 2865–2881.

[149] WENG, Q., XIAO, W., YU, Y., WANG, W., WANG, C., HE, J., LI, Y., ZHANG, L., LIN, W., AND DING, Y. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)* (Renton, WA, Apr. 2022), USENIX Association, pp. 945–960.

[150] WU, K., ZHANG, J., PENG, H., LIU, M., XIAO, B., FU, J., AND YUAN, L. Tinyvit: Fast pretraining distillation for small vision transformers. In *European conference on computer vision (ECCV)* (2022).

[151] WU, Y., SUN, Z., LI, S., WELLECK, S., AND YANG, Y. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models, 2025.

[152] XIN, R. World record set for 100 tb sort by open source and public cloud team. https://opensource.com/business/15/1/apache-spark-new-world-record.

[153] XU, R., LEE, J., WANG, P., BAGCHI, S., LI, Y., AND CHATERJI, S. LiteReconfig: cost and content aware reconfiguration of video object detection systems for mobile GPUs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 334–351.

[154] XU, Z., ZHAO, L., LIANG, W., RANA, O. F., ZHOU, P., XIA, Q., XU, W., AND WU, G. Energy-aware inference offloading for dnn-driven applications in mobile edge clouds. *IEEE Transactions on Parallel and Distributed Systems 32*, 4 (2021), 799–814.

[155] YOU, J., CHUNG, J.-W., AND CHOWDHURY, M. Zeus: Understanding and optimizing GPU energy consumption of DNN training. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (Boston, MA, Apr. 2023), USENIX Association, pp. 119–139.

[156] YU, X., YOUILL, M., WOICIK, M., GHANEM, A., SERAFINI, M., ABOULNAGA, A., AND STONEBRAKER, M. Pushdowndb: Accelerating a dbms using s3 computation, 2020.

[157] YUN, J., KANG, B., RAMEAU, F., AND FU, Z. In defense of pure 16-bit floating-point neural networks, 2023.

[158] ZAHARIA, M., CHOWDHURY, M., DAS, T., DAVE, A., MA, J., MCCAULEY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA, USA, 2012), NSDI'12, pp. 15–28.

[159] ZENG, X., FANG, B., SHEN, H., AND ZHANG, M. Distream: scaling live video analytics with workload-adaptive distributed edge intelligence. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems* (2020), pp. 409–421.

[160] ZHANG, L. L., HAN, S., WEI, J., ZHENG, N., CAO, T., YANG, Y., AND LIU, Y. nn-meter: towards accurate latency prediction of deep-learning model inference on diverse edge devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2021), MobiSys '21, Association for Computing Machinery, p. 81–93.

[161] ZHANG, M., KRINTZ, C., AND WOLSKI, R. Edge-adaptable serverless acceleration for machine learning internet of things applications. *Software: Practice and Experience 51* (12 2020).

[162] ZHANG, M., WANG, F., ZHU, Y., LIU, J., AND WANG, Z. Towards cloud-edge collaborative online video analytics with fine-grained serverless pipelines. In *Proceedings of the 12th ACM Multimedia Systems Conference* (2021), pp. 80–93.

[163] ZHANG, W., LI, S., LIU, L., JIA, Z., ZHANG, Y., AND RAYCHAUDHURI, D. HetroEdge: Orchestration of real-time vision applications on heterogeneous edge clouds. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications* (Apr. 2019), pp. 1270–1278.

[164] ZHANG, Y., GOIRI, I. N., CHAUDHRY, G. I., FONSECA, R., ELNIKETY, S., DELIMITROU, C., AND BIANCHINI, R. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 724–739.

[165] ZHANG, Y., LIU, J.-H., WANG, C.-Y., AND WEI, H.-Y. Decomposable intelligence on cloud-edge iot framework for live video analytics. *IEEE Internet of Things Journal 7*, 9 (2020), 8860–8873.

[166] ZHAO, F., WANG, X., LIN, P., AND CHEN, Y. Comprehensive analysis of the heterogeneous computing performance of dnns under typical frameworks on cloud and edge computing platforms. *Expert Systems with Applications 229* (2023), 120475.