System Design for Intelligent Web Services

by

Johann-Alexander Hauswald

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2017

Johann-Alexander Hauswald

jahausw@umich.edu

ORCID iD: 0000-0001-6969-0992

To my family and all those who wished me well.

ACKNOWLEDGEMENTS

It is a series of ambitious decisions informed by a team of exceptional mentors that have brought me to where I am today. Jason and Lingjia, no words can express my gratitude to you both, you are an incredible team. Jason, you showed me what it means to be truly immersed, dedicated, and passionate about my work, unlocked my zeal for mentoring, and trained me to unconditionally take the path less traveled. Lingjia, your bar for quality drives my desire for excellence across all my endeavors, you taught me to love and defend my research and convictions, and to pursue the difficult questions knowing the answer may not be readily available. Trev, I am grateful and fortunate to have started in your group, been given the chance to find my devotion to research, and continue as a graduate student. My dissertation committee, I thank you for your wisdom and guidance in the final steps of my studies. I made lifelong friends as part of Tron and Clarity labs, the culture fostered was an environment conducive to success as a group at the forefront of research.

To my parents, Robert and Isabelle, to whom I am grateful beyond belief for allowing me to follow my passions and take the calculated risks to progress in life, knowing I have your unconditional support. To my brother Philippe, whose footsteps I have been following for as long as I can remember, thank you for being a big brother and a sanctuary of support in trying times. I would be remiss if I did not conclude and acknowledge my family, closest friends, and all those I met along the way that have wished me well, your support means everything.

Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

System Design for Intelligent Web Services

by

Johann-Alexander Hauswald

Chairs: Jason Mars and Lingjia Tang

The devices and software systems we interact with on a daily basis are more intelligent than ever. The computing required to deliver these experiences for end-users is hosted in Warehouse Scale Computers (WSC) where *intelligent web services* are employed to process user images, speech, and text. These intelligent web services are emerging as one of the fastest growing class of web services. Given the expectation of users moving forward is an experience that uses intelligent web services, the demand for this type of processing is only going to increase. However, today's cloud infrastructures, tuned for traditional workloads such as Web Search and social networks, are not adequately equipped to sustain this increase in demand.

This dissertation shows that applications that use intelligent web service processing on the path of a single query require orders of magnitude more computational resources than traditional Web Search. Intelligent web services use large pretrained machine learning models to process image, speech, and text based inputs and generate a prediction. As this dissertation investigates, we find that hosting intelligent web services in today's infrastructures exposes three critical problems: 1) current infrastructures are computationally inadequate to host this new class of services, 2) system designers are unaware of the bottlenecks exposed by these services and the implications on future designs, 3) the rapid algorithmic churn of these intelligent services deprecates current designs at an even faster rate.

This dissertation investigates and addresses each of these problems. After building a representative workload to show the computational resources required by an application composed of three intelligent web services, this dissertation first argues that hardware acceleration is required on the path of a query to sustain demand moving forward. We show that GPU- and FPGA-accelerated servers can improve the query latency on average by 10× and 16×. Leveraging the latency reduction, GPU- and FPGA-accelerated servers reduce the Total Cost of Ownership (TCO) by 2.6× and 1.4×, respectively. Second, we focus on Deep Neural Networks (DNN), a state-of-the-art algorithm for intelligent web services and design a DNN-as-a-Service infrastructure enabling application-agnostic acceleration and single-point of optimization. We identify compute bottlenecks that inform the design of a Graphics Processing Unit (GPU) based system; addressing the compute bottlenecks translates to a throughput improvement of 133× across seven DNN based applications. GPU-enabled datacenters show a TCO improvement over CPU-only designs by 4-20×. Finally, we design a runtime system based on a GPU equipped server that improves current systems accounting for recent advances in intelligent web service algorithms. Specifically, we identify asynchronous processing key for accelerating dynamically configured intelligent services. We achieve on average 7.6× throughput improvements over an optimized CPU baseline and 2.8× over the current GPU system.

By thoroughly addressing these problems, we produce designs for WSCs that are equipped to handle the future demand for intelligent web services. The investigations in this thesis address significant computational bottlenecks and lead to system designs that are more efficient and cost-effective for this new class of web services.

CHAPTER I

Introduction

As computing devices become evermore present in our lives, we rely on an expanding set of features powered by Artificial Intelligence (AI) to support and accomplish tasks on a daily basis. We can search the web using speech and natural, messy language and receive personalized, intelligently curated results because the system is able to understand our intent [1]. We can receive intelligent suggestions for responding to email, interact with personal assistants to set reminders and make orders online, intelligently sort pictures; these are just a few examples among a growing number of applications [3, 5, 16, 70], delivered across of range of devices [2, 14, 22]. Such systems are increasingly prevalent in today's devices, and this growth is expected to further increase with the rise in wearable devices where natural language is the primary medium of interaction [6]. ABI Research predicts there will be 485 million annual wearable device shipments by 2018 [99] showing the looming increase in demand for this type of processing.

These capabilities are made possible by *intelligent web services* deployed in Warehouse Scale Computers (WSC) that use large machine learning models to process image, speech, and text based inputs. However, the processing required to support the aforementioned use cases is orders of magnitude larger than traditional web service applications currently deployed in the cloud. Cloud deployment of intelligent web

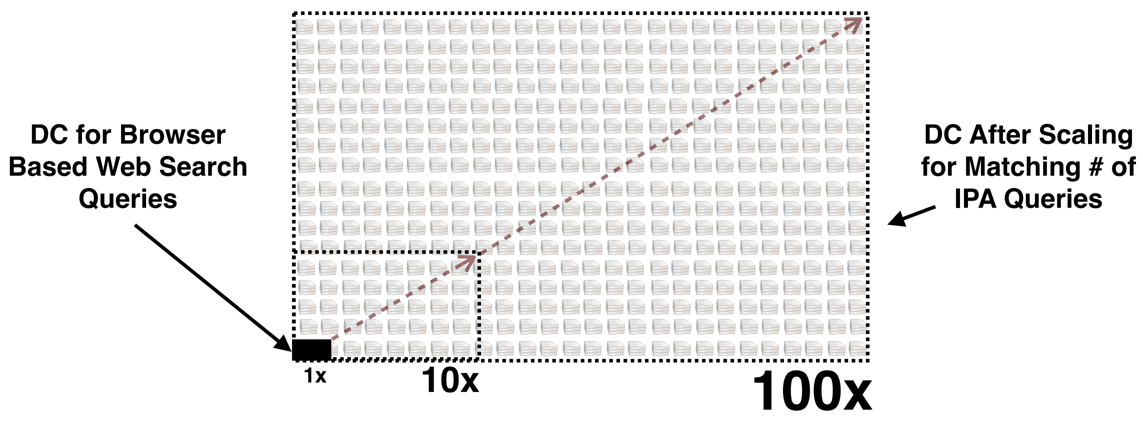

DC for Browser Based Web Search Queries
DC After Scaling for Matching # of IPA Queries
1x
10x
100x

Figure 1.1: Impact of Higher Computational Requirements for Intelligent Personal Assistant (IPA) Queries on Datacenters (DCs)

services has emerged as a significant challenge system architects have been tasked with solving. Figure 1.1 depicts the scaling that would be required of current cloud infrastructures to sustain the demand of an intelligent web service, in this case an intelligent personal assistant [28]. Building increasingly larger datacenters is not a feasible solution to addressing the large demand for intelligent web service backed applications.

1.1 Complex Intelligent Web Service Pipelines

Siri [7], Allo [15], and Cortana [23] represent a class of emerging intelligent web service pipelines known as Intelligent Personal Assistants (IPAs). An IPA is an application that uses inputs such as the user's voice, vision (images), and contextual information to provide assistance by answering questions in natural language, making recommendations, and performing actions.

IPAs differ from many of the web service workloads currently present in modern WSCs. In contrast to the queries of traditional browsercentric services, IPA queries stream through software components that leverage recent advances in speech recognition, natural language processing, and computer vision to provide users a speech-driven and/or image-driven contextually-based question-and-answer system [61]. Due to the computational intensity of these components and the large data-driven models they use, service providers house the required computation in massive datacenter platforms in lieu of performing the computation on the mobile devices themselves. This offloading approach is used by both Siri and Google Now as they send compressed recordings of voice command/queries to datacenters for speech recognition and semantic extraction [105]. However, datacenters have been designed and tuned for traditional web services such as Web Search and questions arise as to whether the current design employed by modern datacenters, composed of general-purpose servers, is suitable for emerging IPA workloads.

1.2 Advances in State-of-the-art Machine Learning

Significant machine learning problems must be solved across various query types to support the range of use cases that can benefit from intelligent web services, including classifying images, recognizing faces, decoding speech, and analyzing text. These are challenging machine learning problems that require powerful algorithms to provide a satisfactory experience for users. One such machine learning algorithm, Deep Neural Network (DNN), has recently gained popularity in solving this wide range of challenges. Using a DNN model trained on a large corpus of data has been shown in the last few years to significantly outperform traditional machine learning techniques in a number of domains [65]. Numerous internet service companies (Apple, Google, Microsoft, Facebook) have been reported to use DNN as their core machine learning algorithm for a wide range of applications [12,17,21,31].

Considering the amount of computation dedicated to DNN inference at the query level, there is opportunity to accelerate a centralized DNN service. However, DNNs require large pretrained machine learning models and significant acceleration is required to provide them as a user-facing low-latency web service, as prior work has also noted [41,42,83]. Additionally, the churn of new DNN architectures and models

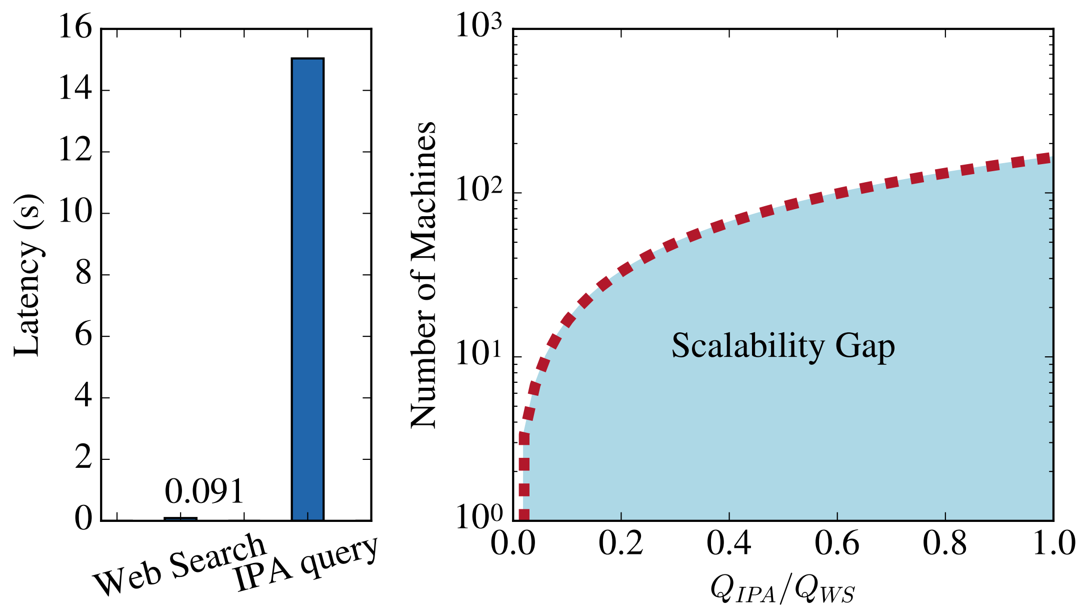


Figure 1.2: Scalability Gap

make it difficult for system designers to pace the progress and design systems accounting for the progress in the field of intelligent web services. Consequently, questions emerge as to the design of systems for state-of-the-art algorithms backing intelligent web services and how to pace the rapid progress in this space.

1.3 Three Challenges

When considering the future of intelligent web services hosted in WSC, there are a number of challenges that emerge.

1.3.1 Scalability Gap

To gain insights on the required resource scaling for IPA queries in modern datacenters, we juxtapose the computational demand of an intelligent personal assistant [28] query with that of a Web Search query. Figure 1.2 (left) presents the average latency of both Web Search using open-source Apache Nutch [4, 55] and Sirius [28] queries. As shown in the figure, the average Nutch-based Web Search query

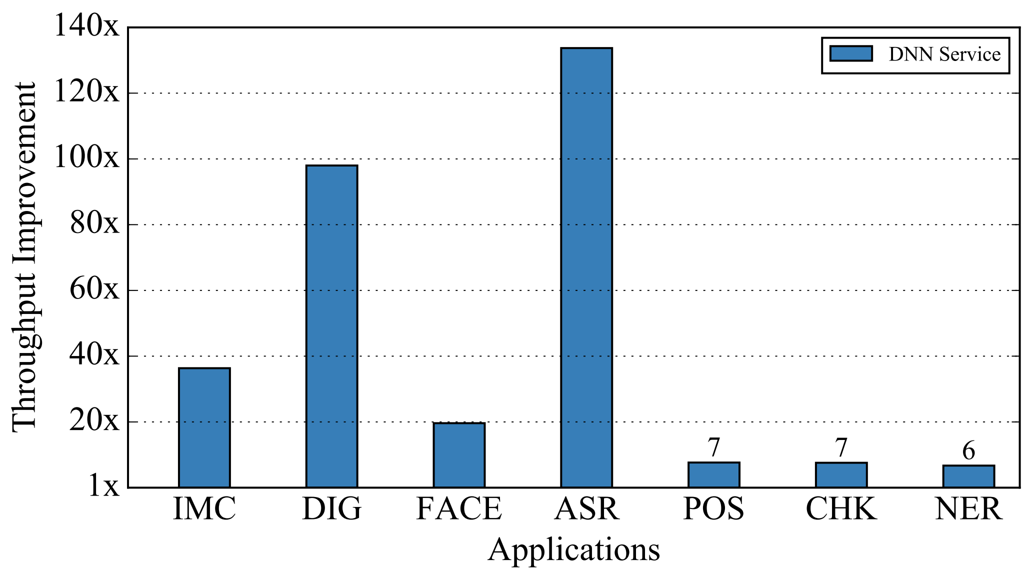


Figure 1.3: Achieved Throughput Improvement (GPU over Single-thread CPU)

latency is 91ms on a Haswell based server. In contrast, the latency of a Sirius query is significantly longer, averaging around 15s.

Based on this significant difference in the computational demand, we perform a back-of-the-envelope calculation of how the compute resources (machines) in current datacenters must scale to match the throughput in queries for IPAs and Web Search. Figure 1.2 (right) presents the number of machines needed to support IPA queries as the number of these queries increases. Current datacenter infrastructures will need to scale their compute resources to 165× their current size when the number of IPA queries scale to match the number of Web Search queries. This throughput difference is referred to as the *scalability gap*.

1.3.2 Intelligence Bottlenecks

To gain insights on how state-of-the-art Deep Neural Networks (DNNs) behave on today's systems, we perform a real-system analysis on a set of DNN based intelligent web services and their performance on a state-of-the-art server grade GPU.

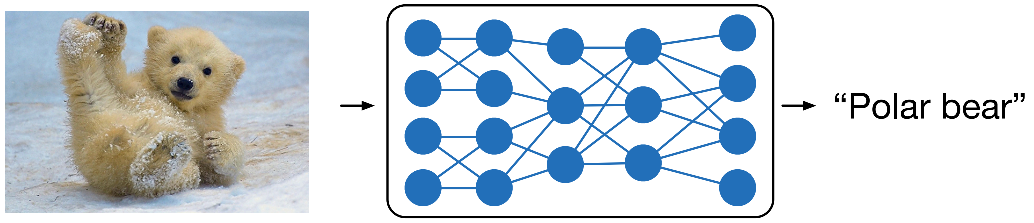


Figure 1.4: Conventional DNN Processing

We focus on the GPU as prior work has shown that DNNs are amenable to GPU acceleration [45]. Figure 1.3 presents the throughput improvement achieved on the GPU over a Xeon CPU core. As shown in the figure, the throughput improvement across image, speech, and natural language processing applications varies greatly. Large networks (ASR) achieve large improvement from computing large matrix multiplications on the GPU. On the other hand, Natural Language Processing (NLP) applications (POS, CHK, NER) have small networks and thus the size of the matrix multiplications in the neural network forward pass is relatively small. This limits the resulting improvement achieved by the GPU. These bottlenecks suggests there are significant challenges in designing a high throughput, low latency system for DNN based intelligent web services.

1.3.3 Intelligent Web Service Churn

The *intelligent web service churn* is the rapid pace at which new techniques and algorithms are being developed that continuously increase the accuracy of intelligent services. In surveying the landscape of deep learning based intelligent web services, we first illustrate the fundamental difference between conventional (static) DNN based applications and recent state-of-the-art Natural Language Processing (NLP) applications. Figure 1.4 shows a conventional image processing pipeline made up of a fixed size input (an image) and a preconfigured neural network architecture where the constituent layers are statically defined (size and number of parameters). At in-

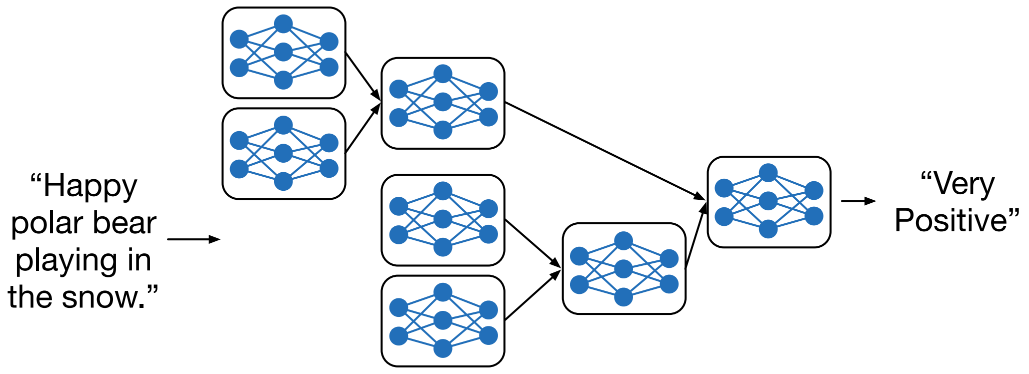


Figure 1.5: Dynamic DNN Processing

ference, the neural network is executed once to provide the classification of the image. Figure 1.5 illustrates the neural network topology of a tree-structured LSTM [110]. Conversely to the image processing pipeline, the topology of this network is dynamically defined meaning the number of neural network invocations (blue boxes) is only known at inference time (as opposed to being statically defined). This illustrates a fundamental difference between the static and dynamic applications. Current state-of-the-art systems are designed for the traditional type of processing and may not be able to handle this dynamism.

1.4 Summary of Contributions

This dissertation investigates the design and deployment of large scale intelligent web services addressing each of the challenges set forth and posits that, to sustain demand moving forward, accelerator based WSCs are critical. Through this investigation, we design tools allowing us to investigate a set of hitherto unexplored workloads. We show that by understanding the underlying computational characteristics of the algorithms and optimizing their computational footprints, significant performance improvements and Total Cost of Ownership (TCO) reductions are to be had. We design new techniques and propose novel insights into designing scalable systems for intelligent web services. With these intelligent web services in hand, this dissertation performs an in-depth investigation of the viability of various acceleration strategies, and provides insights on future server designs.

1.4.1 Sirius: an End-to-end Personal Assistant

In Chapter III, we construct Sirius, an open end-to-end intelligent personal assistant system with both speech and image front-ends (Section 3.3). We then characterize Sirius on commodity hardware and investigate the sources of the *scalability gap* for this workload and confirm there is a limited speedup potential for this workload on general-purpose processors and acceleration is indeed needed to address the scalability gap (Section 3.4). We extract 7 computational bottlenecks comprising 92% of the cycles consumed by Sirius' queries to compose a C/C++ benchmark suite (**Sirius Suite**) for acceleration (Section 3.5). We then port these workloads and conduct a thorough performance evaluation on a spectrum of accelerator platforms proposing future server designs based on these accelerators (Section 3.6).

1.4.2 DjiNN and Tonic: DNN as a Service

In Chapter IV, we present the design and implementation of DjiNN, a DNN service infrastructure that supports a spectrum of applications and neural network architectures (Section 4.2). After introducing 7 end-to-end applications that use the DNN service, we identify performance bottlenecks in the DNN service (Section 4.3) and evaluate strategies to mitigate them, achieving high throughput and GPU scalability without diminishing query latency beyond a certain threshold on a GPU accelerator platform (Section 4.4). We evaluate various configurations including the number of GPUs, PCIe and network configurations, as well as disaggregated and integrated server design options. We identify cost-efficient server designs and system architectures that achieve maximal throughput and maximal throughput per dollar while satisfying the latency constraints based on the above investigations (Section 4.5).

1.4.3 Fine-Grained Cross-Input Batching for NLP

In Chapter V, we identify a set of recently published state-of-the-art DNN based NLP applications (Section 5.1) and perform an in-depth characterization of these applications showing key computational differences compared to previously studied DNN based applications (Section 5.2). We show how current GPU based systems for DNN applications would lead to suboptimal performance, present evidence as to why current systems are unsuitable for NLP, and develop a new taxonomy to show the differences across the landscape of DNN based applications (Section 5.3). We outline the design and implementation of a novel system using *fine-grained cross-input batching* focused on providing high throughput for NLP applications (Section 5.4) and show significant throughput gains over state-of-the-art systems (Section 5.5).

CHAPTER II

Background and Related Work

This chapter presents background on Deep Neural Networks (DNN) as well as recent related work that explores acceleration for intelligent applications at scale.

2.1 Deep Neural Networks

A neural network is a directed graph of *neurons*, where each neuron is a processing element that applies a function to its input(s) to generate an output. The structure of the network is defined by a set of connections between different groups of neurons that perform the same function, known as the *layers* of the network. As illustrated in Figure 2.1, a layer can be of type input, hidden, or output. A neural network can have multiple hidden layers, where the number of such layers defines the *depth* of the network. Common to all neural networks is a classifier layer that produces the final output(s) of the network. This layer has as many outputs as there are classes to predict by the network.

Deep Neural Network (DNN) A DNN is a neural network with many hidden layers. Typically, each neuron is exhaustively connected to the neurons of the subsequent layer, in a configuration also known as a fully connected network. Each neuron computes a weighted sum of its inputs to form an output that is sent to the next

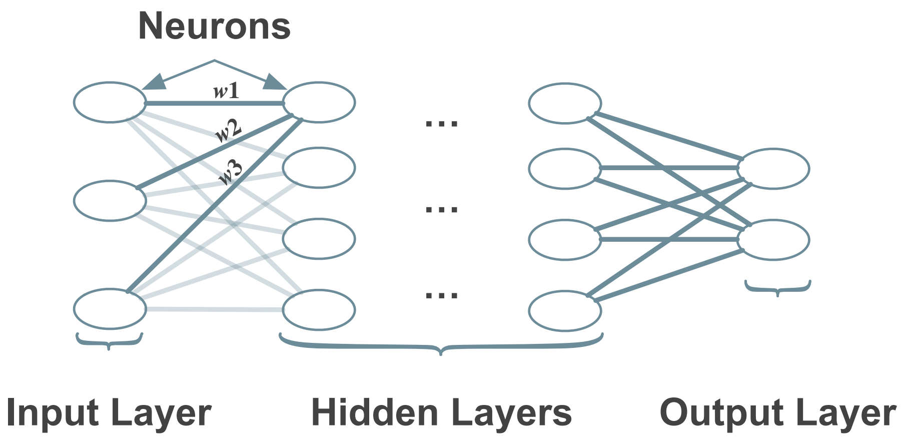


Figure 2.1: Deep Neural Network (DNN) Architecture

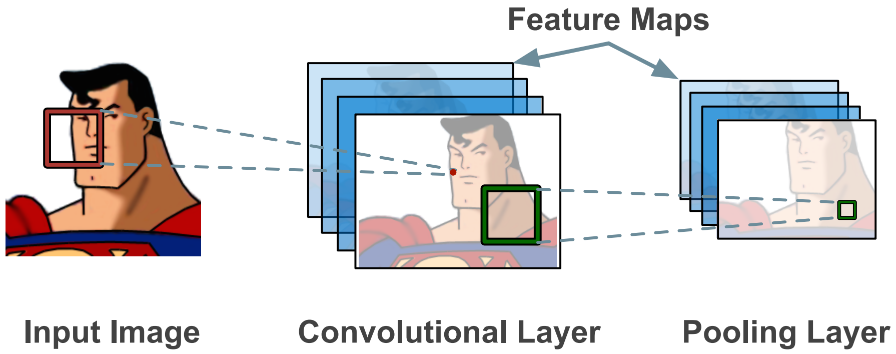


Figure 2.2: Convolutional Neural Network (CNN) Architecture

layer. The weights applied to the inputs are learned during training and stored in a pretrained model describing the entire network. The structure of a DNN is depicted in Figure 2.1, where the weights ($w1$, $w2$, $w3$) are applied to the neuron's inputs to produce the output; this process is analogous for all the network's neurons.

Convolutional Neural Network (CNN) CNNs, a special case of DNNs, have a similar structure to DNNs except they are specialized for image-related tasks. Two important types of layers in CNNs include convolutional and pooling layers, used to extract features from input images. In these layers, each neuron is mapped to a region of the image to which the neuron applies a convolution or pooling operation. Because of this segmentation into regions, the network is not fully connected. These are also

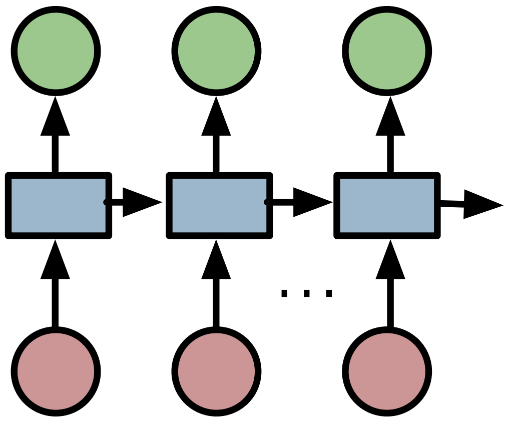


Figure 2.3: Long-Short Term Memory (LSTM) Architecture

called sparsely connected networks. In convolutional layers, the learned weights are *kernels* that are convolved with the image to extract features. Figure 2.2 shows the kernel (red box) applied to the image generating a feature map. At each layer, there are multiple learned kernels each producing a distinct feature map (shaded feature maps in the figure). The pooling layers downsample each feature map to retain only “interesting” features (green boxes). This convolution-and-downsample process is repeated multiple times in a CNN to produce high quality features describing the input image. These features are used in the fully connected classifier layer to predict the content of the image.

Long-Short Term Memory (LSTM) Networks LSTMs [63] are another class of neural network architectures, heavily based on Recurrent Neural Networks (RNNs). Their primary characteristic, compared to DNNs and CNNs, is they retain state and process their inputs as a sequence (as opposed to all at once like an image for a CNN). These traits make them particularly well suited for Natural Language Processing (NLP) applications where the input is a sequence of words that need to be read sequentially to preserve the semantic structure of the input. Commonly, the first layer in an RNN or LSTM is a *word embeddings* layer that translates each word in the input into a vector allowing mathematical operation on the input [91]. Figure 2.3 shows the computational pattern of an LSTM where the input (bottom red circles) is processed in sequence where a state is retained (blue box) and used as a weighted input to the next input.

2.2 Application Specific Acceleration

In addition to prior work focusing on datacenter efficiency [64, 78, 86–88, 93, 112, 113, 119, 121], recent work proposes a heterogeneous server design [68] for speech and image recognition, where the Gaussian Mixture Model (GMM) scoring and image matching algorithms were ported to hardware accelerators. However their work does not address the acceleration of NLP algorithms or DNN-based speech recognition. Custom accelerators for specific cloud applications have also been proposed, for example for memcached [80] and database systems [73] showing the growing need for specialized hardware in server applications. The Catapult project [97] at Microsoft Research has ported key components of Bing's page ranking to FPGAs. In this work, we focus on accelerating the components that make up an intelligent personal assistant focusing on their impact in the end-to-end system.

Prior work has also investigated acceleration of individual components of an intelligent personal assistant on various platforms. For speech recognition systems that use a combination of Gaussian Mixture Models and Hidden Markov Models (HMM), prior work characterizes and accelerates the workload in hardware [74,89]. In the past, GPUs have been successful in accelerating speech recognition's GMM [52] and more recently Automatic Speech Recognition (ASR) was ported using a hybrid CPU-GPU approach [71]. The Carnegie Mellon *In Silicon Vox* [81] project has implemented an FPGA based GMM/HMM speech recognizer with a relatively small vocabulary. Image processing algorithms have been shown to map well to accelerators [51,60,100]. Key natural language processing techniques also show promising results when ported to hardware [85, 108]. Low-power accelerators for deep neural networks [41, 53] have garnered the interest of researchers as DNNs can be parallelized easily but have better accuracy compared to conventional machine learning techniques [59].

In investigating workload characterization and acceleration, this dissertation differs from prior work in that it takes a holistic approach in understanding the design of an entire system and considers the entire suite of services when designing a system for intelligent web services. As we will show, systems need to be considered and investigated end-to-end because of the inherent complexity and computational requirements of the service.

2.3 Accelerating DNNs

Deep learning techniques are outperforming state-of-the-art traditional machine learning methods in speech and image tasks [75]. There is growing interest both in implementing software for deep learning methods within open source libraries [34, 58, 69] and in improving hardware designs for DNNs via CPU optimizations [117] and ASICs [41,42,83,98,114]. In this work, we focus on leveraging commodity GPU accelerators to optimize the throughput of DNN and on relieving bandwidth bottlenecks in the network and interconnect to sustain high throughput across DNN-based services.

The Catapult project [97] at Microsoft Research ported key components of Bing's page ranking to FPGAs, showing the ongoing need for specialized hardware for datacenter applications. Microsoft also studied reducing the total amount of machines needed in a datacenter to train an image classification network increasing the efficiency of the datacenter [43]. DistBelief [84] investigated distributing deep learning tasks across large systems efficiently, and Coates et al. [45] investigated designing a large network of GPUs connected with high-speed interconnects specialized for deep learning and show they are able to effectively distribute computation in a WSC. These systems investigate training deep learning networks while this work focuses on the inference task of DNNs in online applications.

CHAPTER III

Sirius: an End-to-End Voice and Vision Personal Assistant

This chapter presents the design and study of **Sirius**, an end-to-end Intelligent Personal Assistant (IPA) developed to study how future Warehouse Scale Computers (WSC) should evolve for this new type of application. The insight made in this chapter is that current datacenter infrastructure are improperly equipped to handle the amount of compute required on the path of a single IPA query. In building a representative, end-to-end system, this chapter shows that in order to sustain demand moving forward, accelerator-equipped WSCs are critical. After presenting the end-to-end design of the system, we decompose Sirius into its algorithmic components and investigate the design space of accelerators and perform a Total Cost of Ownership (TCO) analysis of the datacenter hosting the intelligent personal assistant workload. As our findings show, two accelerator platforms emerge as viable candidates informing future accelerator-based server designs.

3.1 Designing an Intelligent Personal Assistant

We construct an end-to-end standalone IPA service, **Sirius**, that implements the core functionalities of an IPA such as speech recognition, image matching, natural language processing, and a question-and-answer system. Sirius takes as input user dictated speech and/or image(s) captured by a camera. There are three pathways of varying complexity through the Sirius back-end based on the nature of the input query. In designing Sirius, we focus on the following design objectives:

1. **Completeness** - Sirius should provide a complete IPA service that takes the input of human voice and images and provide a response to the user's question with natural language.

2. **Representativeness** - The computational techniques used by Sirius to provide this response should be representative of state-of-the-art approaches used in commercial domains.

3. **Deployability** - Sirius should be deployable and fully functional on real systems.

We have constructed Sirius by integrating three services built using well-established open source projects that include techniques and algorithms representative of those found in commercial systems. These open-source projects include CMU's Sphinx [67], representing the widely-used Gaussian Mixture Model based speech recognition, Kaldi [96] and RWTH's RASR [103], representing industry's recent trend toward Deep Neural Network based speech recognition, OpenEphyra [104] representing the-state-of-the-art question-and-answer system based on IBM's Watson [56], and SURF [35] implemented using OpenCV [39] representing state-of-the-art image matching algorithms widely used in various production applications.

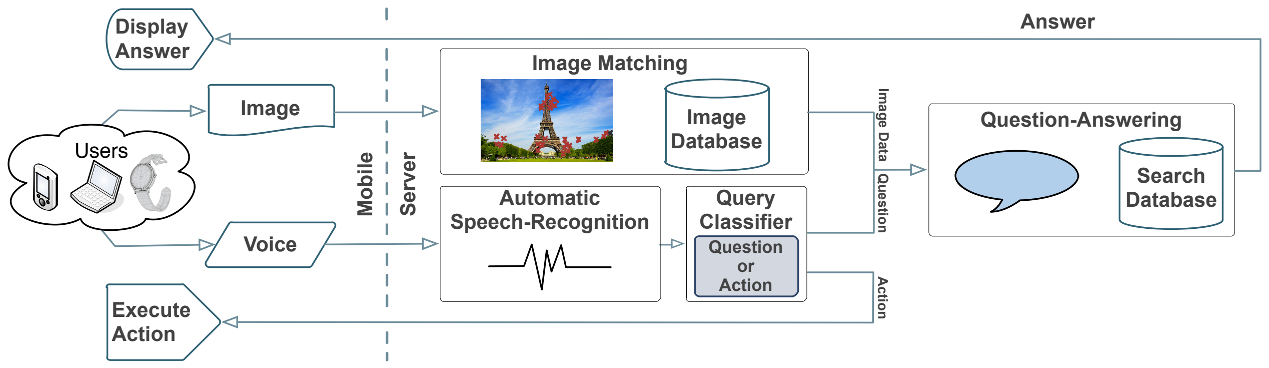


Figure 3.1: End-to-end Diagram of the Sirius Pipeline

3.2 Sirius Overview: Life of an IPA Query

Figure 3.1 presents a high-level diagram of the end-to-end Sirius query pipeline. The life of a query begins with a user's voice and/or image input through a mobile device. Compressed versions of the voice recording and image(s) are sent to a server housing Sirius. The user's voice is then processed by an Automatic Speech Recognition (ASR) front-end that translates the user's speech question into its text equivalent using statistical models. The translated speech then goes through a Query Classifier that decides if the speech is an action or a question. If it is an action, the command is sent back to the mobile device for execution. Otherwise, the Sirius back-end receives the question in plain text. Using Natural Language Processing (NLP) techniques, the Question-Answering (QA) service extracts information from the input, searches its database, and chooses the best answer to return to the user. If an image accompanies the speech input, Sirius uses computer vision to match the input image to the closest resembling image in its image database and return relevant information about the matched image using the Image Matching (IMM) service. For example, a user can ask *"What time does this restaurant close?"* using image(s) of the restaurant captured from a mobile device. Sirius can then return an answer to the query based not only on the speech, but also information from the image.

As shown in Figure 3.1, there are a number of pathways a single query can take

Table 3.1: Query Taxonomy

Query Type	**Example**	**Service**	**Result**
Voice Command (VC)	“Set my alarm for 8am.”	ASR	Action on device
Voice Query (VQ)	“Who was elected 44th president?”	ASR & QA	Answer from QA
Voice-Image Query (VIQ)	“When does this restaurant close?”	ASR, QA & IMM	Answer from IMM and QA

based on the type of directive, whether it be question or action, and the type of input, speech only or accompanied by images. In order to design the input set used with Sirius, this work identifies a query taxonomy of three classes that covers these pathways. Table 3.1 summarizes these query classes providing an example for each, the Sirius services they exercise, the resulting behavior of Sirius.

3.3 Services and Algorithmic Components

As shown in Figure 3.2, Sirius is composed of three IPA services: Automatic Speech Recognition (ASR), Question-Answering (QA), and Image Matching (IMM). These services can be further decoupled into their individual algorithmic components. In order to design Sirius to be representative of production grade systems, Sirius leverages well-known open-source infrastructures that use the same algorithms as commercial applications. Speech recognition in Google Voice, for example, has used speaker-independent Gaussian Mixture Model (GMM) and Hidden Markov Model (HMM) and is adopting Deep Neural Networks (DNNs) [49,62]. The OpenEphyra framework used for question-answering is an open-source release from CMU’s prior research collaboration with IBM on the Watson system [56]. OpenEphyra’s NLP techniques, including conditional random fields (CRF), have been recognized as state-of-the-art and are used at Google and in other industry question-answering systems [116]. The image matching pipeline is based on the SURF algorithm, which is widely used in industry [20, 27, 35]. SURF is implemented using the open-source computer vision OpenCV library [39], which is employed in commercial products from companies like Google, IBM, and Microsoft. The design of these services are described in the re-

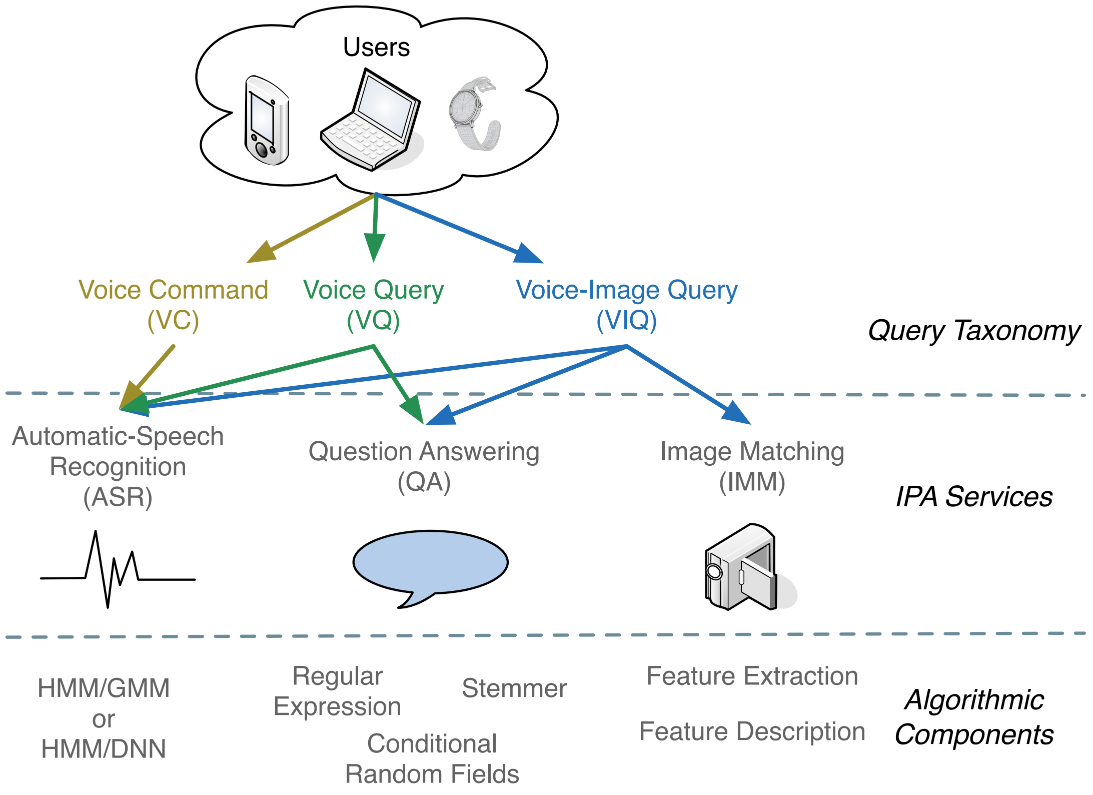


Figure 3.2: Tier-level View of Sirius

mainder of this section.

3.3.1 Automatic Speech Recognition (ASR)

The inputs to the ASR are feature vectors representing the speech segment, generated by fast pre-processing and feature extraction of the speech. The ASR component relies on a combination of a Hidden Markov Model (HMM) and either a Gaussian Mixture Model (GMM) or a Deep Neural Network (DNN). Sirius' GMM-based ASR uses CMU's Sphinx [67], while the DNN-based ASR includes Kaldi [96] and RWTH's RASR [103].

As shown in Figure 3.3, the HMM builds a tree of states for the current speech frame using input feature vectors. The GMM or DNN scores the probability of the state transitions in the tree, and the Viterbi algorithm [57] then searches for the most

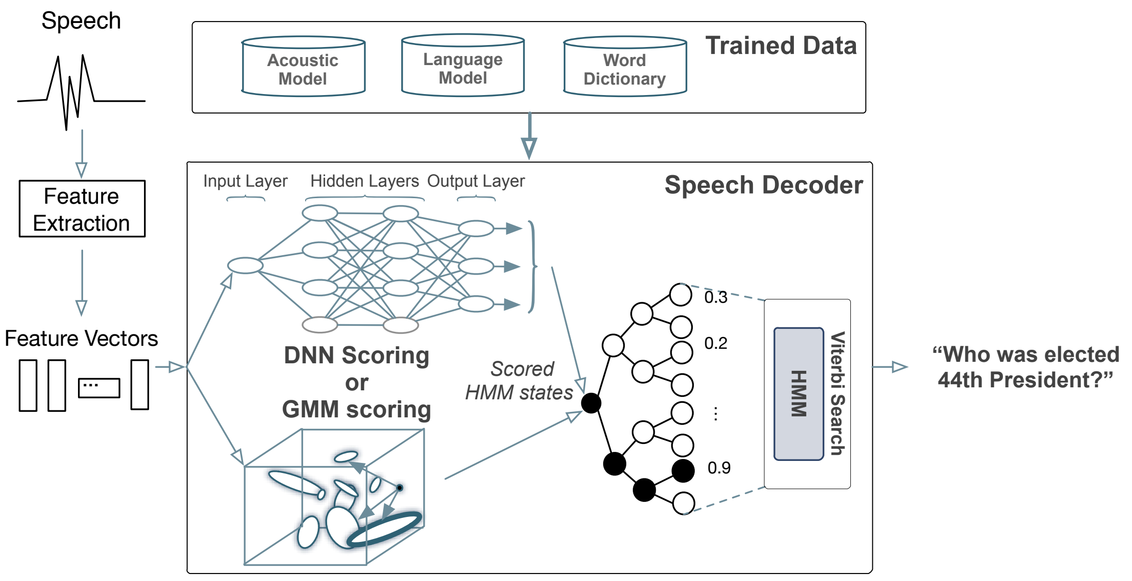


Figure 3.3: Automatic Speech Recognition Pipeline

likely path based on these scores. The path with the highest probability represents the final translated text output. The GMM scores HMM state transitions by mapping an input feature vector into a multi-dimensional coordinate system and iteratively scores the features against the trained acoustic model.

On the other hand, the DNN based implementation scores the transition probabilities using the output from a neural network. The depth of a DNN is defined by the number of hidden layers where scoring amounts to one forward pass through the network. In recent years, industry and academia have moved towards DNNs over GMMs due to their higher accuracy [48, 66].

3.3.2 Image Matching (IMM)

The image matching pipeline receives an input image, attempts to match it against images in a pre-processed image database, and returns information about the matched images. The database used in Sirius is the Mobile Visual Search [40] database, which is a database of objects taken from a mobile device.

Figure 3.4 details the steps described in the following section in performing per-

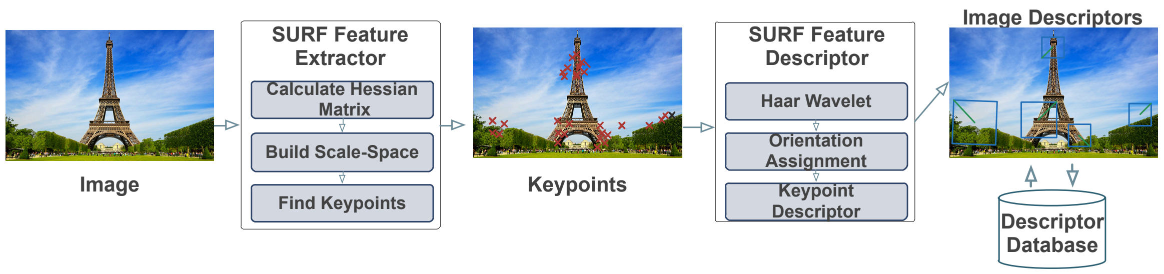


Figure 3.4: Image Matching Pipeline

forming image matching on a single image. Image keypoints are first extracted from the input image using the SURF algorithm [35]. Specifically, in Feature Extraction (FE), the image is downsampled and convolved multiple times to find interesting points at different scales. After thresholding the convolution responses, the local maxima responses are stored as the image keypoints, which represent interesting regions of the image. The keypoints are then passed to the Feature Descriptor (FD) component where they are assigned an orientation vector, and similarly oriented keypoints are grouped into feature descriptors. This process reduces variability across input images, increasing chances of finding the correct match. The descriptors from the input image are matched to pre-clustered descriptors representing the database images by using an approximate nearest neighbor (ANN) search. The database image with the highest number of matches is returned.

3.3.3 Question-Answering (QA)

The text output from ASR is passed to OpenEphyra (OE) [104], which uses three natural language processing techniques to extract textual information: word stemming, regular expression matching, and part-of-speech tagging. Figure 3.5 shows a diagram of the OE engine incorporating these components, generating Web Search queries and filtering the returned results. The Porter Stemming [95] algorithm (stemmer) exposes the root of a word by matching and truncating common word endings.

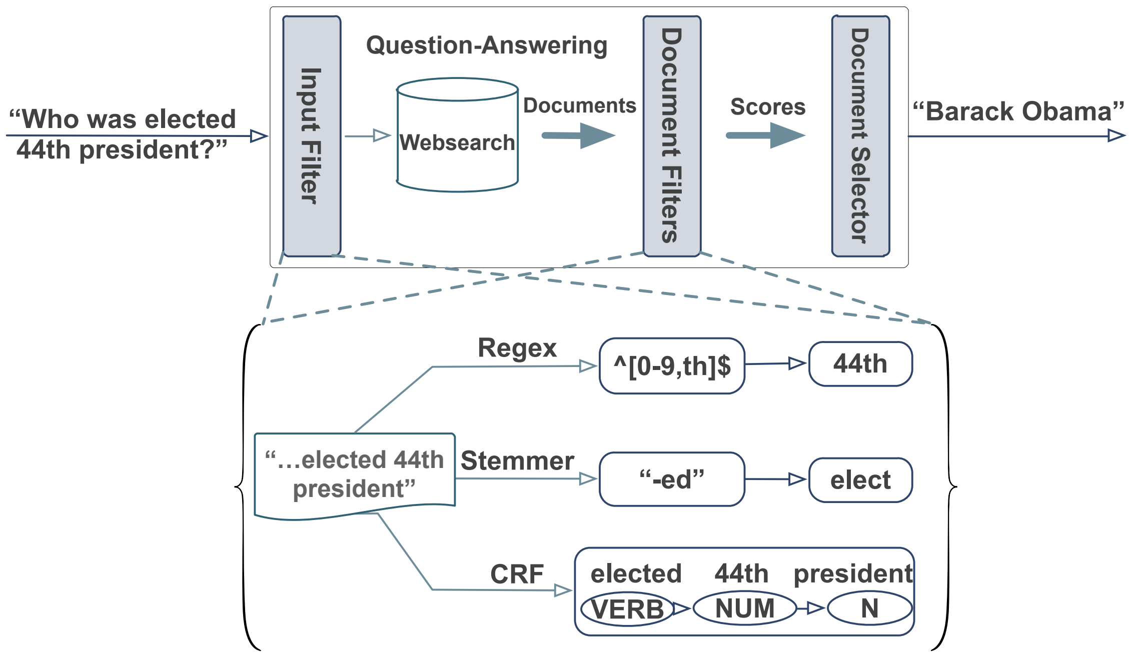


Figure 3.5: OpenEphyra Question-Answering Pipeline

OE uses a suite of regular-expression patterns to match common query words (what, where, etc) and filter any special characters in the input. The Conditional Random Field (CRF) classifier [77] takes a sentence, the position of each word in the sentence, and the label of the current and previous word as input to makes predictions on the part-of-speech for each word of an input query. Each input query is parsed using the aforementioned components to generate queries to the Web Search engine. Next, filters using the same techniques are used to extract information from the returned documents. The document with the highest overall score after score aggregation is returned as the best answer.

3.4 Real System Analysis for Sirius

In this section, we present a real-system analysis of Sirius. The experiments in this section are performed using an Intel Haswell server (details in Table 3.3).

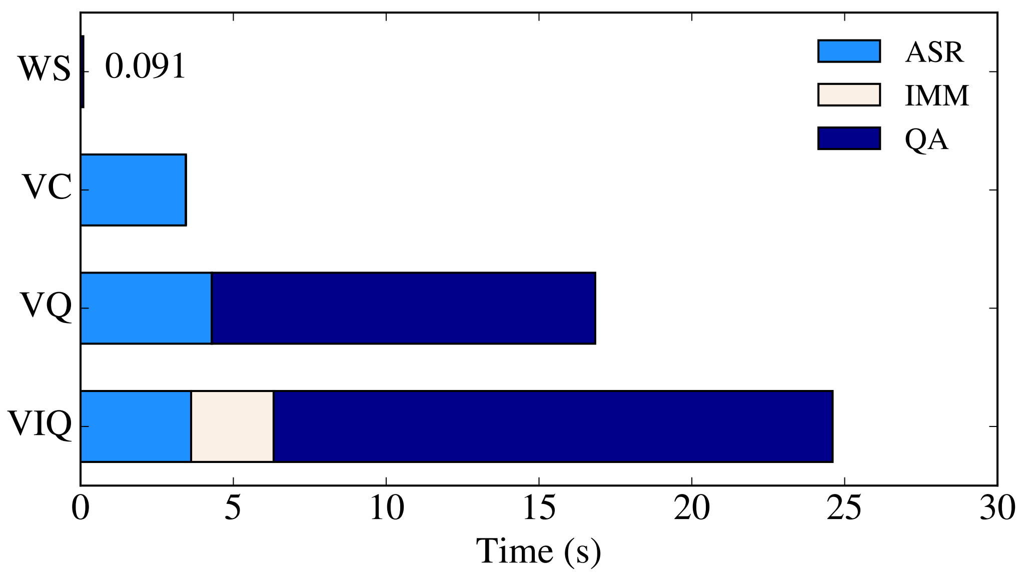


Figure 3.6: Latency Across the Different Query Types

3.4.1 Sirius Query Deep Dive

To better understand the IPA query characteristics, we further investigate the average latency and latency distributions of various query types for Sirius. Figure 3.6 presents the average latency across query types including traditional Web Search (WS), Voice Command (VC), Voice Query (VQ) and Voice Image Query (VIQ). As shown in the figure, the latency of all three Sirius query types are significantly higher than that of Web Search queries. The shortest query type is VC, which only uses the ASR service. Yet it still requires orders of magnitude more computation than Web Search. The longest query type is VIQ, which uses all three services including ASR, IMM, and QA. Among all three services, QA consistently consumes the most compute cycles.

Figure 3.7 presents the latency distribution for each Sirius service. As shown in the figure, QA has the highest variability in latency, ranging from 1.7s to 35s depending on the input query. Figure 3.8 further presents the breakdown of execution time among QA's hot components (described later in this section) across the complete

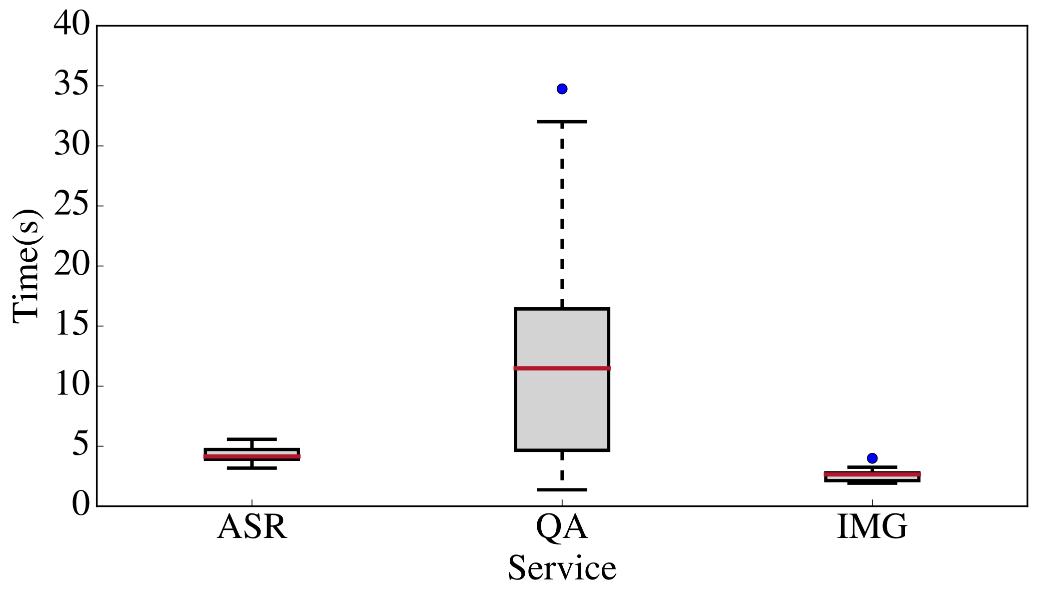


Figure 3.7: Latency Variability Across Services

Table 3.2: Voice Query Input Set

Q#	**Query**
q1	"Where is Las Vegas?"
q2	"What is the capital of Italy?"
q3	"Who is the author of Harry Potter?"
...	...
q15	"What is the capital of Cuba?"
q16	"Who is the current president of the United States?"

VQ query input set (shown in Table 3.2). The reason for this high latency variability is not immediately clear from inspecting the query input set, especially when considering the small difference between Q2 and Q15 in Table 3.2. However, after further investigation, this work identifies that the high variance is primarily due to the runtime variability of various document filters in the NLP component used to select the most fitting answer for a given query. Figure 3.9 demonstrates the correlation between latency and the number of hits in the document filters. The other services, ASR and IMM, have very low query to query variability. Next, we investigate the cycle breakdown of the algorithmic components that comprise each service.

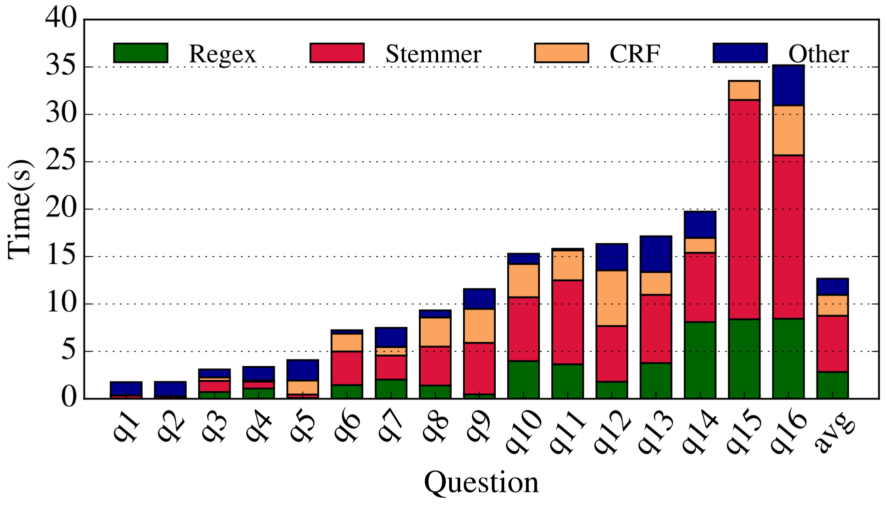


Figure 3.8: OpenEphyra Breakdown

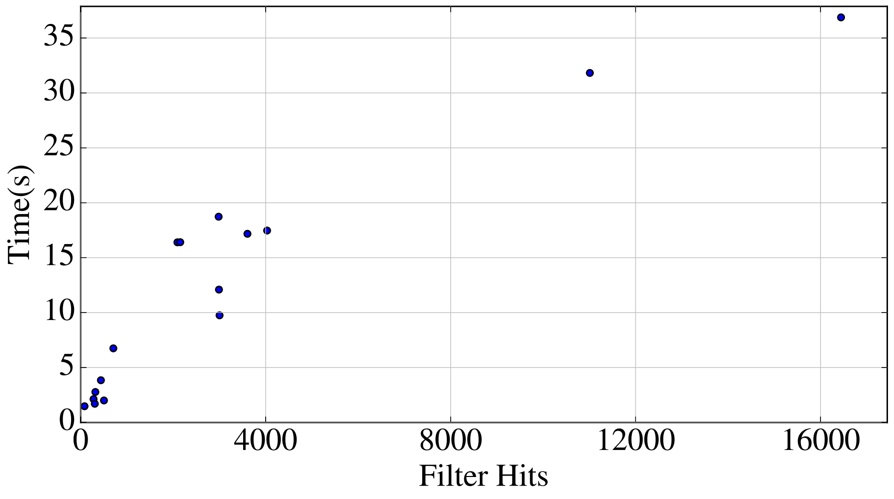


Figure 3.9: Latency and Filter Hits in OpenEphyra

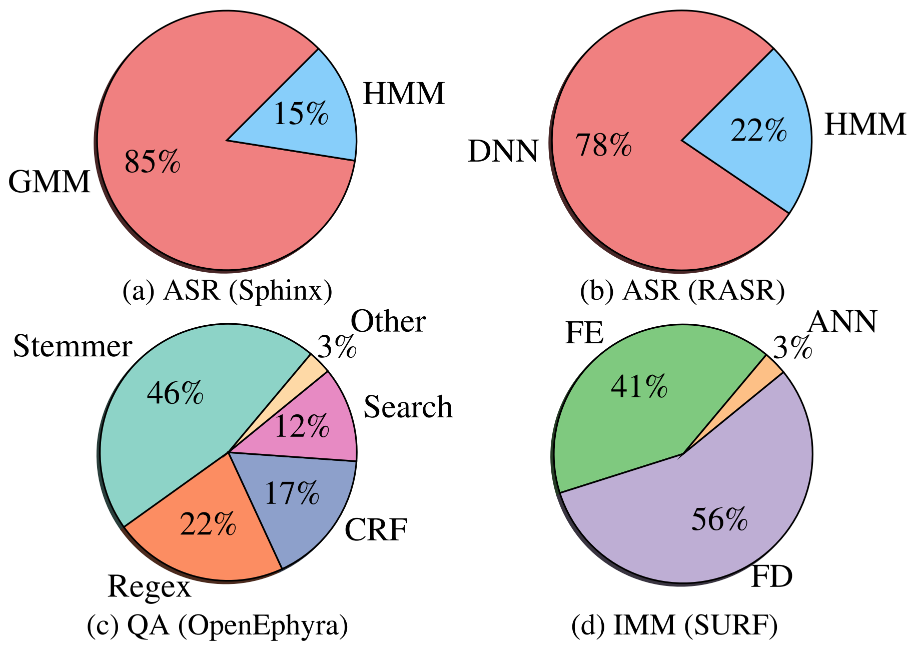


Figure 3.10: Cycle Breakdown per Service

3.4.2 Cycle Breakdown of Sirius Services

To identify the computational bottlenecks of each service, we perform a top-down profiling of the hot algorithmic components for each service, shown in Figure 3.2, using Intel VTune [19]. Figure 3.10 presents the average cycle breakdown results. Across services, a few hot components emerge as good candidates for acceleration. For example, a high percentage of the execution for ASR is spent on scoring using either GMM or DNN. For QA, on average 85% of the cycles are spent in three components including stemming, regular expression pattern matching and CRF, and for IMM, the majority of cycles are spent either performing feature extraction or description using the SURF algorithm.

We then identify the architectural bottlenecks for these hot components to investigate the performance improvement potential for a general-purpose processor. Figure 3.11 presents the instructions per cycle (IPC) and potential architectural bottlenecks (including front-end, speculation and back-end) for each component, identi-

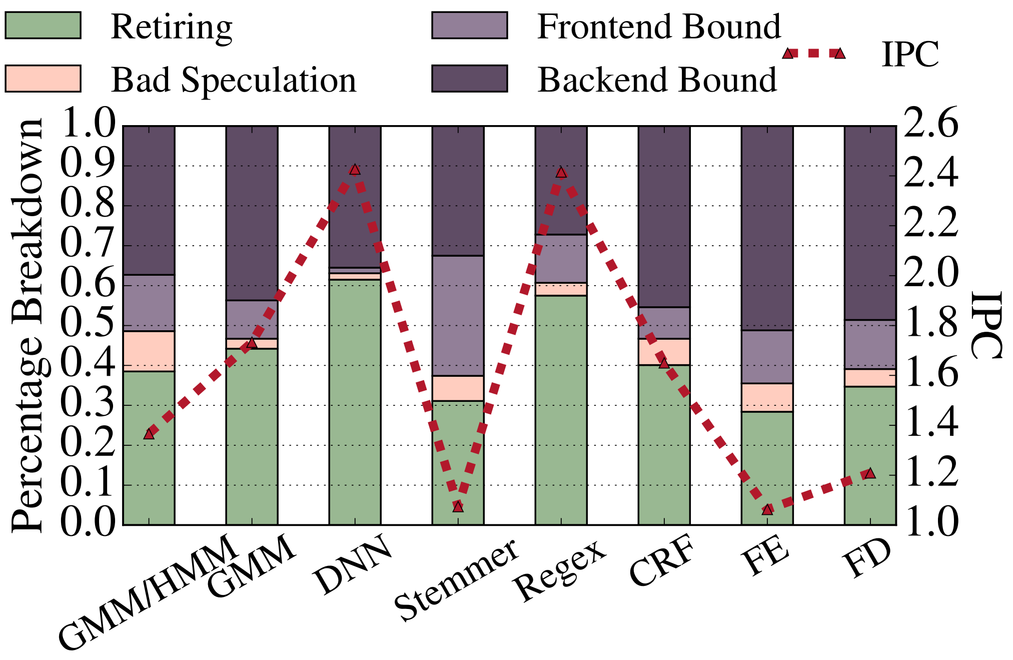


Figure 3.11: IPC and Bottleneck Breakdown

fied using VTune. A few of the service components including DNN and Regex execute relatively efficiently on Xeon cores. This graph indicates that even with all stall cycles removed (i.e., perfect branch prediction, infinite cache, etc) the maximum speed-up is bound by around 3×. Considering the orders of magnitude difference indicated by the scalability gap, further acceleration is needed to bridge the gap.

3.5 Accelerating Sirius

In this section, we describe the platforms and methodology used to accelerate the key components of Sirius. We also present and discuss the results of accelerating each of these components across 4 different accelerator platforms.

Table 3.3: Platform Specifications

Model	Multicore Intel Xeon E3-1240 V3	GPU NVIDIA GTX 770	Phi Intel Xeon Phi 5110P	FPGA Xilinx Virtex-6 ML605
Frequency	3.40 GHz	1.05 GHz	1.05 GHz	400 MHz
# Cores	4	8*	60	N/A
# HW Threads	8	12288	240	N/A
Memory	12 GB	2 GB	8 GB	512 MB
Memory BW	25.6 GB/s	224 GB/s	320 GB/s	6.40 GB/s
Peak TFLOPS	0.5	3.2	2.1	0.5

** Core = SM (Streaming Multiprocessor), 2048 threads/SM*

3.5.1 Accelerator Platforms

This paper uses a total of four platforms, summarized in Table 3.3, to accelerate Sirius. The baseline platform is an Intel Xeon Haswell CPU running single-threaded kernels. The advantages and disadvantages of each accelerator platform are summarized below.

- **Multicore CPU** - *Advantages:* High clock frequency, not limited by branch divergence. *Disadvantages:* Least amount of threads available.

- **GPU** - *Advantages:* Massively parallel. *Disadvantages:* Power hungry, custom ISA, hard to program, large data transfer overheads, limited branch divergence handling.

- **Intel Phi** - *Advantages:* Many core, standard programming model (same ISA), manual porting (optional compiler help), handles branch divergence, high bandwidth. *Disadvantages:* Data transfer overheads, relies on compiler. *Note:* 1 core is used for the operating system running on the device itself.

- **FPGA** - *Advantages:* Can be tailored to implement very efficient computation and data layout for the workload. *Disadvantages:* Runs at a much lower clock frequency, expensive, hard to develop for and maintain with software updates.

Table 3.4: Sirius Suite and Granularity of Parallelism

Service	Benchmark	Baseline	Input Set	Data Granularity
ASR	*Gaussian Mixture Model (GMM)*	CMU Sphinx [67]	HMM states	HMM state
	Deep Neural Network (DNN)	RWTH RASR [103]	HMM states	Matrix mult.
QA	*Porter Stemming (Stemmer)*	Porter [95]	4M word list	Individual word
	Regular-Expression (Regex)	SLRE [29]	100 expr./400 sent.	Expr-sentence pair
	Conditional Random Fields (CRF)	CRFsuite [92]	CoNLL Task [115]	Sentence
IMM	*Feature Extraction (FE)*	SURF [35]	JPEG Image	Image tile
	Feature Description (FD)	SURF [35]	Vector of Keypoints	Keypoint

3.5.2 Sirius Suite: A Collection of IPA Compute Bottlenecks

To investigate the viability and trade-offs of accelerating IPAs, this work extracts the key computational bottlenecks of Sirius (described in Section 3.4) to construct a suite of benchmarks called **Sirius Suite**. Sirius Suite as well as its implementations across the described accelerator platforms are available alongside the end-to-end Sirius application [28]. As a basis for Sirius Suite, we port existing open-source C/C++ implementations available for each algorithmic component to our target platforms. We additionally implemented standalone C/C++ benchmarks based on the source code of Sirius where none were currently available. The baseline implementations are summarized in column 3 of Table 3.4. For each Sirius Suite benchmark, we built an input set representative of IPA queries. Table 3.4 shows the granularity at which each thread performs the computation on the accelerators. For example, both GMM and DNN kernels receive input feature vectors from the HMM search, which are all scored in parallel but at different levels of abstraction, respectively, based on each implementation.

3.5.3 Porting Methodology

The common porting methodology used across all platforms is to exploit the large amount of data-level parallelism available throughout the processing of a single IPA query. The following subsections describe the platform-specific highlights of the porting efforts.

Multicore CPU

The Pthread library is used to accelerate the kernels on the multicore platform by dividing the size of the data. Each thread is responsible for a range of data over a fixed number of iterations. This approach allows each thread to run concurrently and independently, synchronizing only at the end of the execution.

For the image matching kernels, the images are pre-processed for feature extraction by tiling the images. Each thread of the CPU is assigned one or more tiles of the input image (depending on the size of each tile). This allows to spawn threads once at the beginning of execution and synchronize threads at the end, instead of parallelizing at a smaller granularity within the SURF algorithm, which would require multiple synchronizations between loops. However, as the tile size decreases, the number of “good” keypoints decreases, so the tile size is fixed to a minimum of 50×50 per thread.

GPU

Sirius Suite use NVIDIA’s CUDA library to port the Sirius components to the NVIDIA GPU. To implement each CUDA kernel, we varied and configured the GPU block and grid sizes to achieve high resource utilization, matching the input data to the best thread layout. Additional string manipulation functions currently not supported in CUDA for the stemmer kernel.

Intel Phi

We port our Pthread versions to the Intel Phi platform, leveraging the ability of the target compiler to parallelize the loops on the target platform. For this, we use Intel’s ICC cross-compiler. The Phi kernel is built and run directly on the target device allowing for rapid prototyping and debugging. On the Phi platform, we sweep the total amount of threads spawned in increments of 60, increasing the number of hardware threads per core. For some kernels, the maximum number of threads (with enough input data) did not always yield the highest performance. To investigate the potential of this platform to facilitate ease of programming, we use the standard

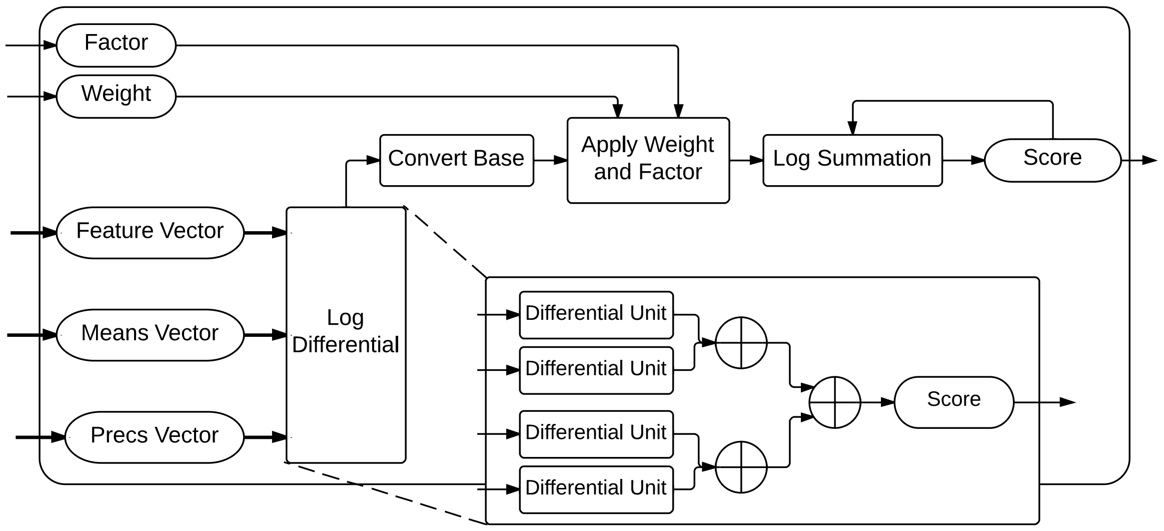


Figure 3.12: FPGA GMM Diagram

programming model and custom compiler to extract performance from the platform. As such, the results represent what can be accomplished with minimal programmer effort.

FPGA

We use previously published details of FPGA implementations for a number of our Sirius Benchmarks in this work. However, due to limited published details for two of our workloads and to gain further insights, we design our own FPGA implementations for both GMM and Stemmer and evaluate them on a Xilinx FPGA.

GMM - The major computation of the algorithm lies in three nested loops that iteratively score the feature vector against the training data. This training data comes from an acoustic model, a language model, and a dictionary in the forms of a means vector, a pre-calculated (precs) vector, a weight vector, and a factor vector. All of this data is used to generate a score for the probability of an HMM

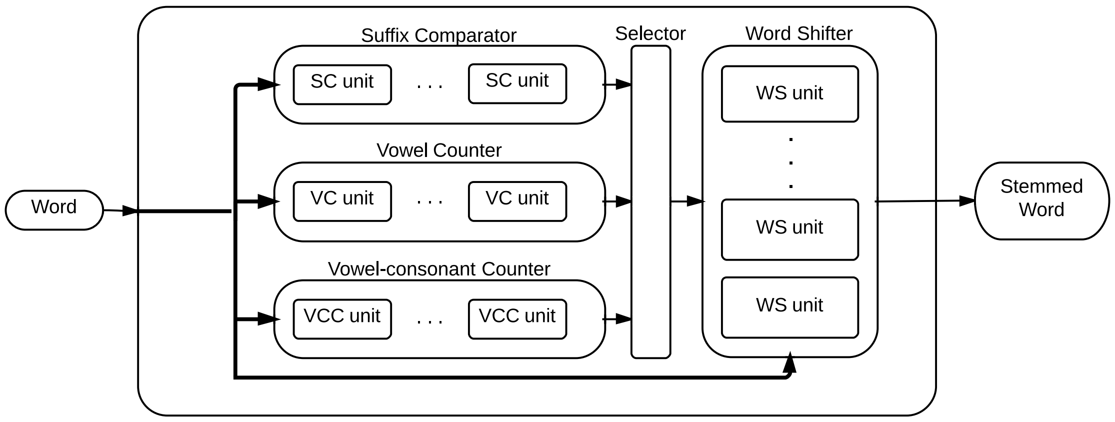


Figure 3.13: FPGA Stemmer Diagram

state transition. The focus when implementing the algorithm on the FPGA is to maximize parallelization and pipeline utilization, which leads to the design presented in Figure 3.12. This figure depicts both a core that computes the score of a single iteration of the outermost loop and a callout of a log differential unit. The log differential unit is used to fully parallelize the innermost loop, while the entire core can be instantiated multiple times to parallelize the outermost loop. Because of this, the design is highly scalable as multiple cores can be used to fill the FPGA fabric. The middle loop of the algorithm is not parallelizable and is represented by the Log Summation unit. This design is able to create a high throughput device with a linear pipeline.

Stemmer - The Stemmer algorithm computes the root of a word by checking for multiple conditions, such as the word's suffixes or roots. Figure 3.13 summarizes a single step for the stemmer implementation. By taking advantage of the mutual exclusivity of test conditions, we are able to parallelize these comparisons, which allowed the FPGA to achieve a much lower latency than the original Porter algorithm. This implementation performs multiple vector operations simultaneously to count vowels,

Table 3.5: Speedup of Sirius Suite Across Platforms

Service	Benchmark	CMP	GPU	Phi	FPGA
ASR	*GMM*	3.5	70.0	1.1	169.0
	DNN	6.0*	54.7*	11.2	110.5 [54]
QA	*Stemmer*	4.0	6.2	5.6	30.0
	Regex	3.9	48.0 [118]	1.1	168.2 [120]
	CRF	3.7	3.8 [94]	4.7	7.5 [109]
IMM	*FE*	5.2	10.5	2.5	34.6 [38]
	FD	5.9	120.5	12.7	75.5 [38]

* This includes DNN and HMM combined.

vowel-consonant pairs, and compare suffixes. Together, these operations select the correct word shift for the specific step. This forms a single pipelined core based upon six steps dealing with the different possibilities of suffixes. We instantiate multiple cores to fill the FPGA fabric to deliver maximum performance.

3.5.4 Accelerator Results

Table 3.5 and Figure 3.14 present the performance speedup achieved by the Sirius kernels running on each accelerator platform, organized by service type. For the numbers from prior literature, we scale the FPGA speedup number to match the FPGA platform based on fabric usage and area reported in prior work. We also use numbers from literature for kernels (Regex and CRF) that were already ported to the GPU architecture and yielded better speedups than our implementations.

ASR

The GMM implementation, extracted from CMU Sphinx's acoustic scoring, had the best performance on the GPU (70×) after optimizations. These custom optimizations on the GPU achieved an order of magnitude improvement by optimizing the data structure layout to ensure coalesced global memory accesses. This leveraged concurrent reads to sequential memory positions for a warp (32 threads). In addition, it was possible to store the entire data required for the GMM in the GPU memory

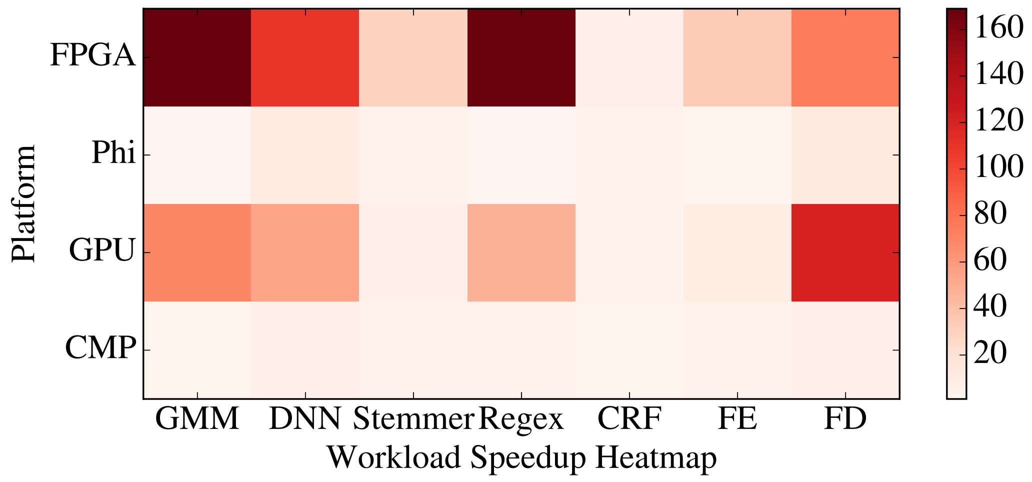


Figure 3.14: Heat Map of Acceleration Results

(2GB) during the deployment time reducing communication between the host and device. The Phi platform did not perform as well as the GPU, indicating that the custom compiler may not have achieved the optimal data layout. The FPGA implementation using a single GMM core achieved a speedup of 56×; when fully utilizing the FPGA fabric it achieved a 169× speedup using 3 GMM cores. RWTH's DNN includes both multithreaded and GPU versions out-of-the-box. The RWTH's DNN parallelizes the entire framework (both HMM search and DNN scoring) and achieves good speedup in both cases. In the cases where a custom kernel is used or cite literature, we assume a 3.7× speedup for the HMM [44] as a reasonable lower bound.

QA

The NLP algorithms as a whole have very similar performance across platforms because of the nature of the workload: high input variability with many test statements causes high branch divergence. Fine tuning the stemming algorithm on the Phi to spawn 120 threads instead of the maximum and switching from allocating a range of data per thread to interlaced array accesses yields a better performance given the lower number of threads used. The FPGA stemmer implementation achieved a

6× speedup over the baseline with a single core using only 17% of the FPGA fabric. Scaling the number of cores to fully utilize the resources of the FPGA yielded a 30× speedup over the baseline. The stemmer algorithm contains many test statements and is not well suited for SIMD operations. Improving the initial stemmer implementation for the GPU by replacing most of the conditional branches with efficient XOR operations [106] did not yield benefits in our experiments. The fine-grained XOR-based implementation performed worse than the initial version due to additional synchronization between threads.

IMM

The image processing kernels achieved the best speedup on the GPU which uses heavily optimized OpenCV [39] SURF implementations yielding speedups of 10.5× and 120.5× for FE and FD, respectively. Prior work shows that the FPGA yields better FE speedups but does not show similar increases for FD. The tiled multicore version yields good speedup but the performance does not scale as well on the Phi because the number of tiles is fixed, which means there is little advantage to having more threads available. The GPU version has better performance because it uses a data layout explicitly optimized for a larger number of threads.

3.6 Implications for Future Server Design

In this section, we investigate the performance, power, and cost-efficiency trade-offs when configuring servers with different accelerator platforms for Sirius.

3.6.1 Server Level Design

We first investigate the end-to-end latency reduction and the power efficiency achieved across server configurations for Sirius' services including ASR, QA and IMM.

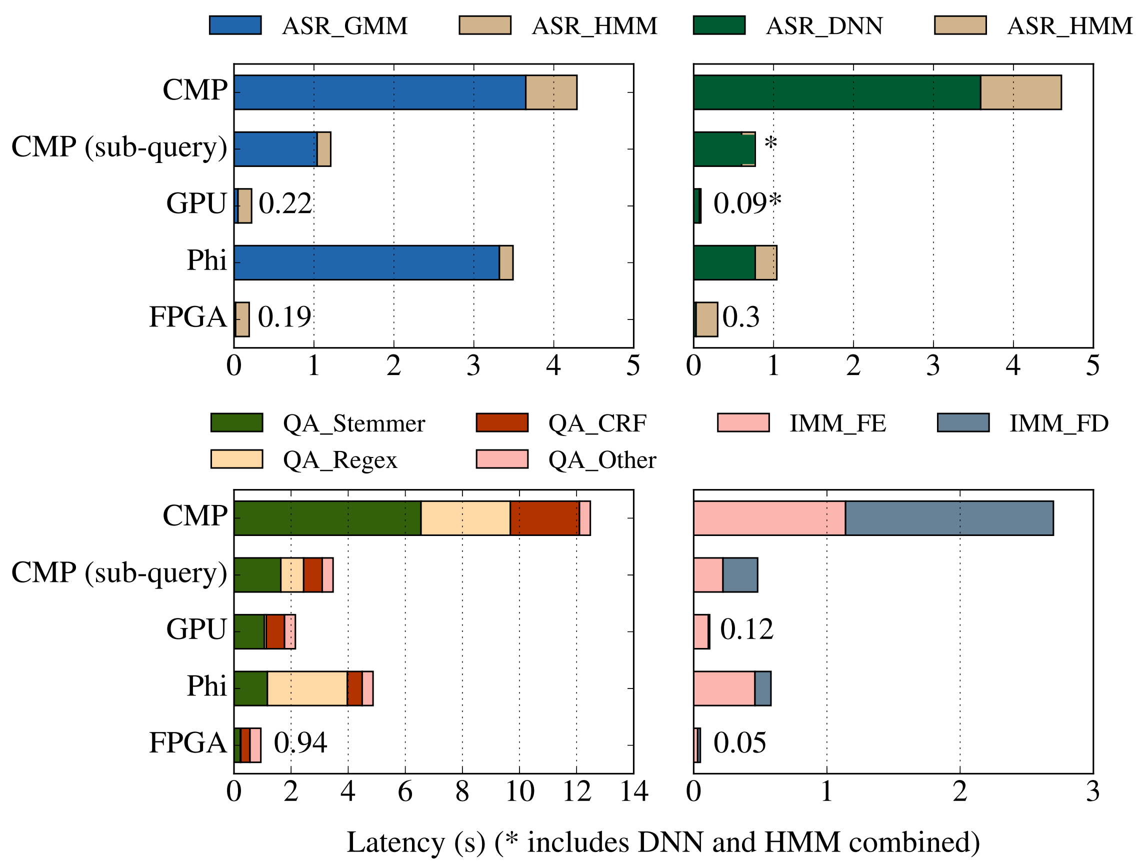


Figure 3.15: Latency Across Platforms for Each Service

Latency Improvement

Figure 3.15 presents the end-to-end query latency across Sirius' services on a single leaf node configured with each accelerator. Presented are both results for ASRs that use GMM/HMM and DNN/HMM as key algorithms. The latency breakdown for all hot components within a service is also presented in the figure. QA is focused on the NLP components comprising 88% of the cycles of QA as search has already been well studied [55].

The baseline in this figure, CMP, is the latency of the original algorithm implementations of Sirius running on a single core of an Intel Haswell server, described in Table 3.3. CMP (sub-query) is the Pthreaded implementation of each service exploiting parallelism within a single query, thus reducing the single query latency. This is executed on 4 cores (8 hardware threads) of the Intel Haswell server. CMP (sub-query) in general achieves a 25% latency reduction over the baseline. Across all services, the GPU and FPGA significantly reduce the query latency. For example, the FPGA implementation of ASR (GMM/HMM) reduces the speech recognition query latency from 4.2s to only 0.19s. The FPGA outperforms the GPU for most of the services except ASR (DNN/HMM). Although the Phi can reduce the latency over the single core baseline (CMP), the Phi is generally slower than the Pthreaded multicore baseline.

Energy Efficiency

Figure 3.16 presents the energy efficiency (performance/watt) for each accelerator platform across four services of the Sirius pipeline, normalized to the performance/watt achieved by using all cores on a multicore CPU by query-level parallelism. Here performance is defined as $1/latency$. Table 3.6 presents the power (TDP) for each accelerator platform. The FPGA has the best performance/watt, exceeding every other platform by a significant margin, with more than 12× energy efficiency over

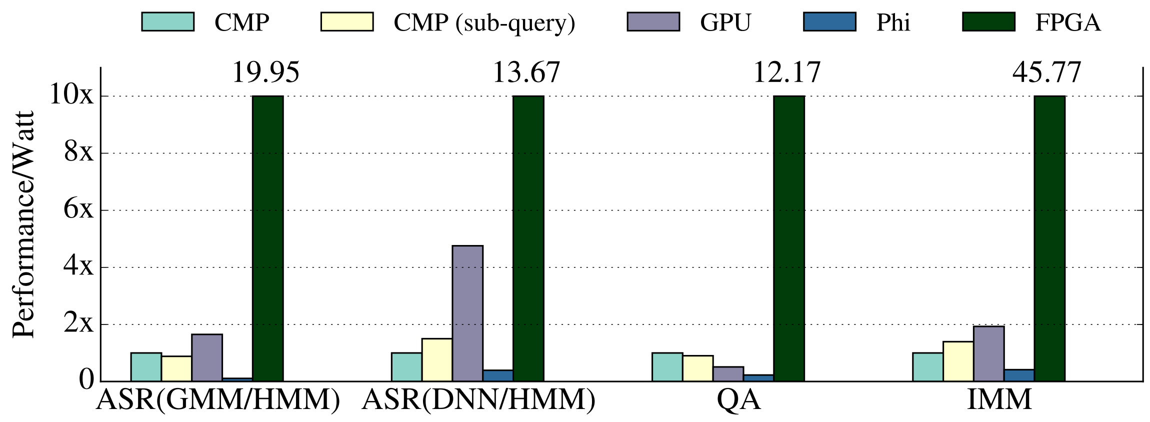


Figure 3.16: Performance per Watt

Table 3.6: Platform Power and Cost

Platform	Power TDP (W)	Cost ($)
Intel Xeon CPU E3-1240	80	250
NVIDIA GPU GTX 770	230	399
Intel Xeon Phi 5110P	225	2,437
Xilinx Virtex-6 FPGA	22	1,795

the baseline multicore. The GPU's performance/watt is also higher than the baseline for 3 of 4 services. Its performance/watt is worse than the baseline for QA, mainly due to its moderate performance improvement for this service.

3.6.2 Datacenter Design

Based on the latency and energy efficiency trade-offs for server platforms discussed in the previous section, this section evaluates multiple design choices for datacenters composed of accelerated servers to improve performance (throughput) and reduce the total cost of ownership (TCO).

Throughput Improvement

The latency reduction shown in Figure 3.15 can translate to significant throughput improvements. Figure 3.17 presents the throughput improvement achieved using various acceleration platforms without degrading latency beyond the baseline. Similar to

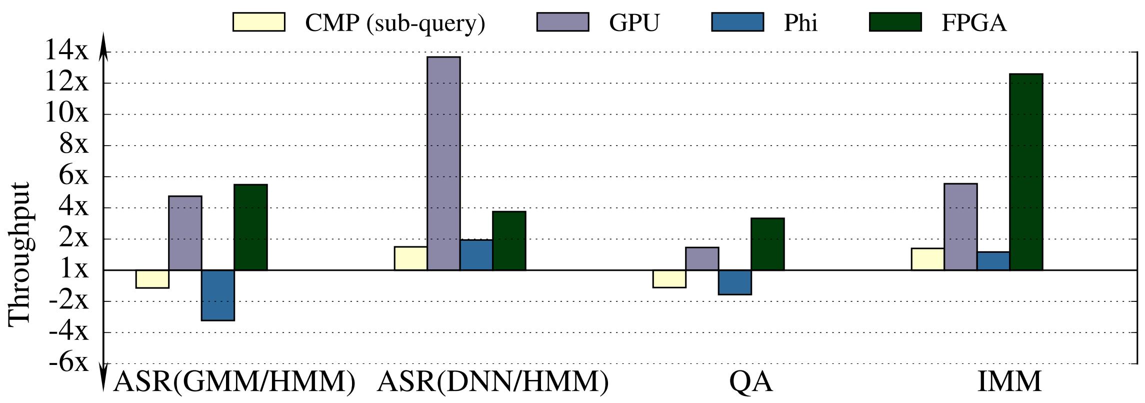


Figure 3.17: Throughput Across Services

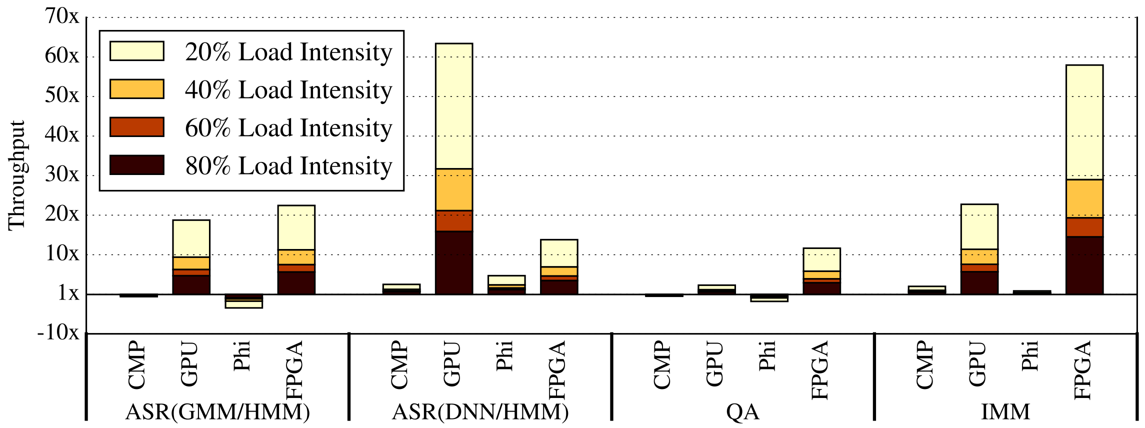


Figure 3.18: Throughput Improvement at Various Load Intensities

Figures 3.15 and 3.16, the CMP baseline executes the original Sirius workload on the Intel Haswell platform, where all four cores are utilized to serve queries, thus achieving similar throughput as CMP (sub-query level). Note that CMP's query latency is however significantly longer because CMP (sub-query level) exploits parallelism within a single query. Figure 3.17 demonstrates that significant latency reductions achieved by the GPU and FPGA translate to significant throughput improvement. For example, the GPU provides 13.7× throughput improvement over the baseline CMP for ASR (DNN/HMM), while the FPGA achieves 12.6× throughput for IMM. For QA, the throughput improvement across the platforms is generally more limited than other services.

Table 3.7: TCO Model Parameters [33]

Parameter	Value
DC Depreciation Time	12 years
Server Depreciation Time	3 years
Average Server Utilization	45%
Electricity Cost	$0.067/kWh
Datacenter Price	$10/W
Datacenter Opex	$0.04/W
Server Opex	5% of Capex / year
Server Price (baseline)	$2,102 [30]
Server Power (baseline)	163.6 W [30]
PUE	1.1

Figure 3.18 presents the throughput improvement achieved using each acceleration platform at various load levels (the server is modeled as an M/M/1 queue). Compared to Figure 3.17, which presents the throughput improvement at 100% load, when considering queuing effect, the lower the server load, the bigger impact latency reduction would have on throughput improvement. In other words, Figure 3.17 demonstrates a lower bound of throughput improvement for a queuing system. Since datacenter servers often operate at medium-to-low load, as shown in Figure 3.18, significantly higher throughput improvement can be expected.

TCO Analysis

Improving throughput allows reduction in the amount of computing resources (servers) needed to serve a given load. However, reducing the number of servers may or may not lead to reduction in the total cost of ownership of a datacenter (DC). Although reducing the machines leads to reduction on DC construction cost and power/cooling infrastructure cost, it may increase the per server capital or operational expenditure cost either by additional accelerator purchase cost or the energy cost. Here we present a cost analysis to evaluate the implication on the datacenter cost when using each accelerated server platform. The TCO analysis is performed using the TCO model recently proposed by Google [33]. The parameters used in the TCO

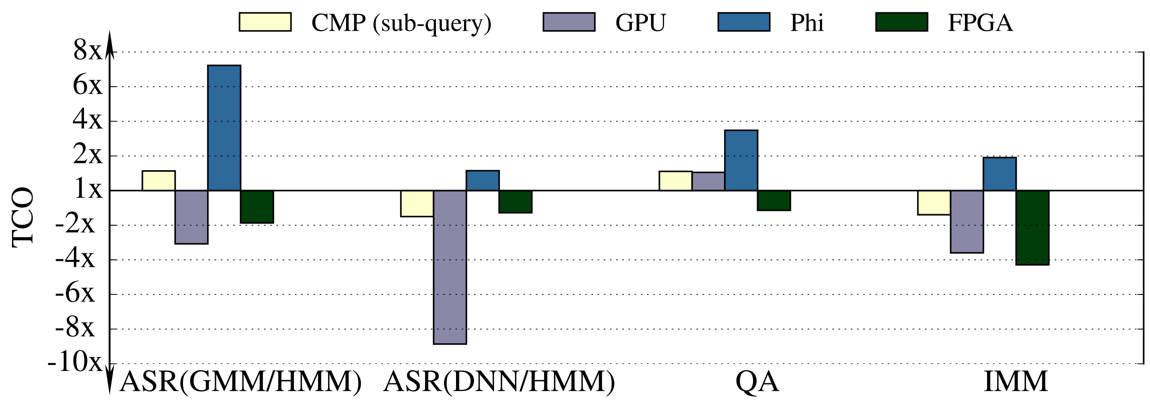


Figure 3.19: TCO Across Platforms for Each Service

model are described in Table 3.7. The server price and power usage are based on the following server configuration based on the OpenCompute Project: 1 CPU Intel Xeon E3-1240 V3 3.4 GHz, 32 GB of RAM, and two 4TB disks [30].

Figure 3.19 presents the datacenter TCOs with various acceleration options, normalized to the TCO achieved by a datacenter that uses only CMPs. Overall, the FPGA and GPU provide high TCO reduction. For example, the GPU achieves over 8× TCO reduction for ASR (DNN) and the FPGA achieves over 4× TCO reduction for IMM. The next section further discussed the TCO results and use them to derive the DC designs.

Homogeneous Datacenter Design

Based on latency results from Figure 3.15 and TCO results from Figure 3.19, we first investigate the trade-offs when designing a homogeneous datacenter, that is, all servers in the datacenter have the same configuration. Homogeneous datacenters are often desirable as they minimize the management and maintenance overhead [86].

When designing a datacenter, it would be ideal to maximize performance (e.g., minimize query latency or improve throughput for a given latency constraint) and minimize the total cost of ownership. However, trade-offs may need to be made as to which objective should be prioritized if both cannot be optimized by the same

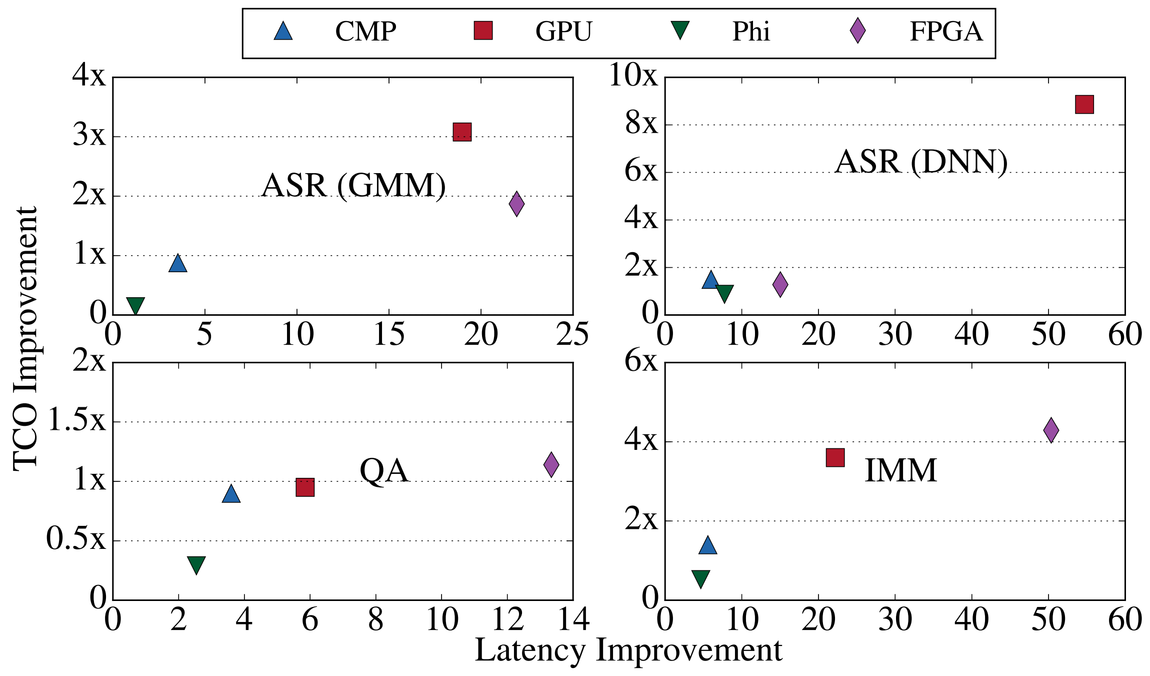


Figure 3.20: Trade-off Between TCO and Latency

design. Figure 3.20 presents the trade-offs between the query latency improvement and the TCO improvement for each server option across four Sirius services. The x-axis presents latency improvement and the y-axis shows the TCO improvement.

As shown in the figure, the FPGA achieves the lowest latency (highest latency improvement) among all accelerating platforms for 3 out of 4 services studied. However, the FPGA's relatively high purchase cost allows GPUs to achieve similar or higher TCO savings with smaller latency reduction as the FPGAs. When the FPGA is not considered an option, the GPU achieves the optimal latency and TCO for all services. Even with the FPGA as an accelerator candidate, a GPU-accelerated datacenter provides the best latency and TCO for ASR using DNN.

Table 3.8: Homogeneous DC (GMM and DNN are ASR services)

	With FPGA				Without FPGA				Without {FPGA, GPU}			
	GMM	DNN	QA	IMM	GMM	DNN	QA	IMM	GMM	DNN	QA	IMM
Hmg-latency	FPGA				GPU				CMP			
Hmg-TCO (w/ L constraint)	GPU											
Hmg-power eff. (w/L constraint)	FPGA											

Table 3.8 summarizes the homogeneous datacenter design for each of the main Sirius services under different conditions and optimization objectives. Presented are three first-order design objectives: minimizing latency, minimizing TCO with a latency constraint, and maximizing energy efficiency with a latency constraint, shown as three rows of the table. The latency constraint here is CMP (sub-query) using the latency shown in Figure 3.15. The first row (with FPGA, without FPGA, without FPGA or GPU) also shows the design constraints for the accelerator candidates.

Key Observation - *In conclusion, FPGAs and GPUs are the top 2 candidates for homogeneous accelerated datacenter designs across all three design objectives. An FPGA-accelerated datacenter allows DCs to minimize latency and maximize energy efficiency for most of the services and is the best homogeneous design option for those objectives. Its power efficiency is desirable for datacenters with power constraints, especially for augmenting existing filled datacenters that are equipped with capped power infrastructure support. It also improves TCO for all four services. On the other hand, FPGA-accelerated datacenters incur higher engineering cost than the rest of the platforms. For DCs where engineering cost needs to be under a certain constraint, GPU-accelerated homogeneous datacenters achieve relatively low latency and high throughput. They also achieve similar or higher TCO reduction than FPGA due to its low purchase cost. GPUs could be a desirable option over FPGAs when the high engineering overhead of FGPA implementation is a concern, especially given the quick workload churn (e.g., binaries are updated on the monthly basis) in modern datacenters.*

Table 3.9: Heterogeneous DC (GMM and DNN are ASR services)

	With FPGA				Without FPGA				Without {FPGA, GPU}			
	GMM	DNN	QA	IMM	GMM	DNN	QA	IMM	GMM	DNN	QA	IMM
Het-latency	FPGA	GPU (3.6x)	FPGA									
Het-TCO (w/ L constraint)	GPU		FPGA (20%)	FPGA (19%)	GPU				CMP			
Het-power eff. (w/L constraint)	FPGA											

Heterogeneous (Partitioned) Datacenter Design

Next, we explore the design options for partitioned heterogeneous datacenters. Because each service can run on its most suitable platform in a partitioned heterogeneous datacenter, this strategy may provide additional opportunities for further latency reduction or TCO reduction. Table 3.9 shows various DC design choices for different design objectives (rows), accelerator candidate sets (with FPGA, without FPGA, and without FPGA and GPU) and services (columns). The numbers in parenthesis show the improvement on the metric of the specific design objective of that row when the DC design switches from a homogeneous baseline to a heterogeneous partitioned design.

As shown in the first row of the table, when designing a partitioned heterogeneous DC for ASR, QA and IMM services, if all accelerators are considered viable candidates, GPUs can be used to optimize the latency for ASR (DNN) and achieves 3.6× latency reduction for that service compared to the homogeneous DC using FPGA across all services. Similarly, using FPGAs for QA and IMM achieves 20% and 19% TCO improvement, respectively.

Key Observation - *In conclusion, the partitioned heterogeneity in the study does not provide much benefit over the homogeneous design. The amount of benefit is certainly dependant on the workload partition across services. However, overall, most of the algorithms and services in the Sirius workload exhibit a similar trend in terms of preferences for accelerators for FPGA and GPU. There is also additional cost associated with managing a heterogeneous/partitioned datacenter that needs to be justifiable by the performance gain.*

Query-level Results for DC designs

In previous sections, we focused on latency, energy-efficiency, and TCO trade-offs for various acceleration options across three services in Sirius. In this section, we focus on these trade-offs across three query types supported by Sirius, namely, Voice Com-

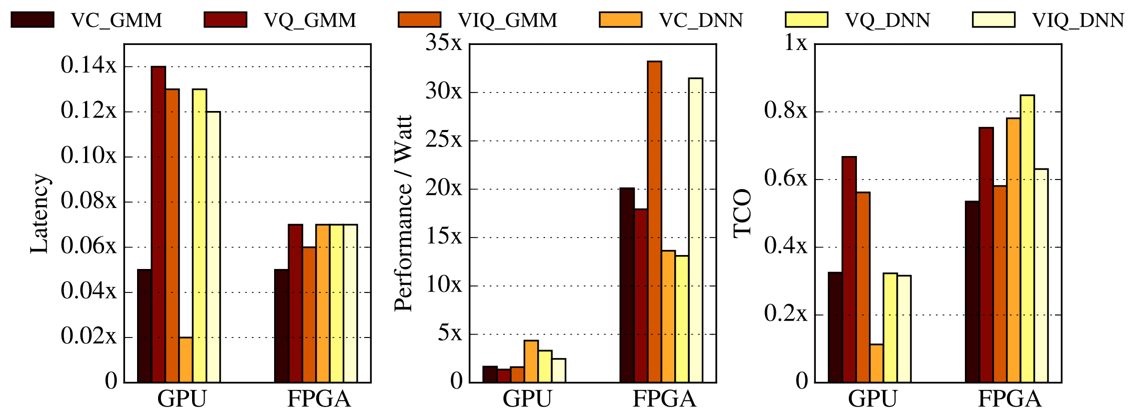


Figure 3.21: Latency, Energy Efficiency, and TCO of GPU and FPGA Datacenters

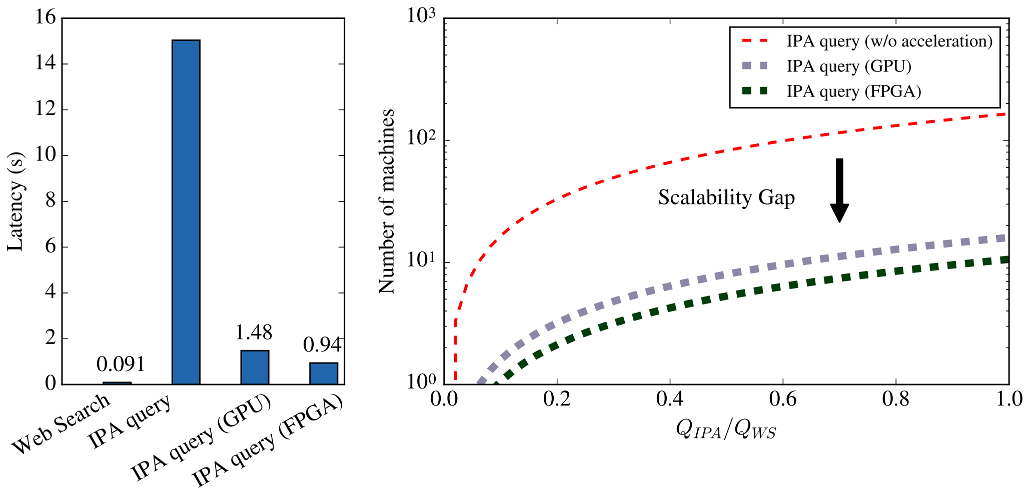


Figure 3.22: Bridging the Scalability Gap

mand (VC), Voice Query (VQ) and Voice Image Query (VIQ). Figure 3.21 presents the query latency of three query types achieved by the best two homogeneous datacenters, composed of GPU- and FPGA-accelerated servers, respectively. In addition to query latency, energy efficiency of the servers and the TCO of the datacenters to support these query types are also presented. GPU-accelerated homogeneous datacenters achieve on average 10× latency reduction, and FPGA-accelerated datacenters achieve a 16× reduction. The accelerated datacenters also reduce the TCO on average by 2.6× and 1.4×, respectively.

Figure 3.22 presents the latency reduction of these two accelerated datacenters and how homogeneous accelerated datacenters can significantly reduce the scalability gap for datacenters, from the current 165× resource scaling (previously shown in Figure 1.2) down to 16× and 10× for GPU- and FPGA-accelerated datacenters, respectively.

3.7 Summary

This chapter introduced Sirius, an open end-to-end intelligent personal assistant application, modeled after popular IPA services such as Apple's Siri. Sirius leverages well-established open infrastructures for speech recognition, computer vision, and question-answering systems. We use Sirius to investigate the performance, power, and cost implications of hardware accelerator-based server architectures for future datacenter designs. We show that GPU- and FPGA-accelerated servers can improve the query latency on average by 10× and 16×. Leveraging the latency reduction, GPU- and FPGA-accelerated servers can reduce the TCO by 2.6× and 1.4×, respectively.

CHAPTER IV

DjiNN and Tonic: DNN as a Service

This chapter presents the design and study of **DjiNN** and **Tonic Suite**, a Deep Neural Network (DNN) as a service infrastructure to study how future Warehouse Scale Computers (WSC) should evolve for DNN processing. The insight made in this chapter is that if intelligent web services can all leverage a common machine learning algorithm and infrastructure, the optimizations applied can be focused on the common case benefiting a range of intelligent web services. DjiNN is a centralized DNN service infrastructure that supports a diverse set of DNN-based applications in WSCs. Tonic Suite is a suite of 7 DNN-based applications from a wide range of domains including image classification, facial recognition, speech recognition and natural language processing. After characterizing the 7 Tonic Suite applications, our findings will show there are significant bottlenecks inhibiting an efficient system for DNN processing. The techniques set forth in this chapter show significant gains can be had in performance that translate into Total Cost of Ownership (TCO) reduction for a datacenter equipped with Graphics Processing Units (GPUs).

4.1 Designing a DNN Web Service

We present the design of **DjiNN**, a general DNN-as-a-Service infrastructure that supports a spectrum of emerging IPA applications, and **Tonic Suite**, a set of 7 end-to-end applications built on the DjiNN service. DjiNN is a centralized DNN service infrastructure that supports a diverse set of DNN-based applications in WSCs. Tonic Suite is a suite of 7 DNN-based applications from a wide range of domains including image classification, facial recognition, speech recognition and natural language processing. We extract the underlying DNN computation from each individual Tonic application. We create the generalized and configurable DjiNN service with a common interface to process the DNN computation for each application. In designing the DNN as a service, we target the following objectives:

1. **Decoupled Architecture** - The DjiNN web service needs to be a standalone service accepting and processing requests coming over the network.

2. **Diverse Applications** - A general DNN service must be capable of processing requests from a wide range of applications.

3. **Request Processing** - The DNN web service must be able to process multiple incoming requests with low overhead.

4.2 DjiNN and Tonic

Figure 4.1 presents an overview of the system. Tonic Suite applications make requests to the DjiNN Service. DjiNN houses the trained DNN network architecture and configuration in-memory for each Tonic Suite application. To process each application's requests, DjiNN executes the DNN inference pass, which generates a prediction using the pre-processed input from the application, and returns the prediction result to the application.

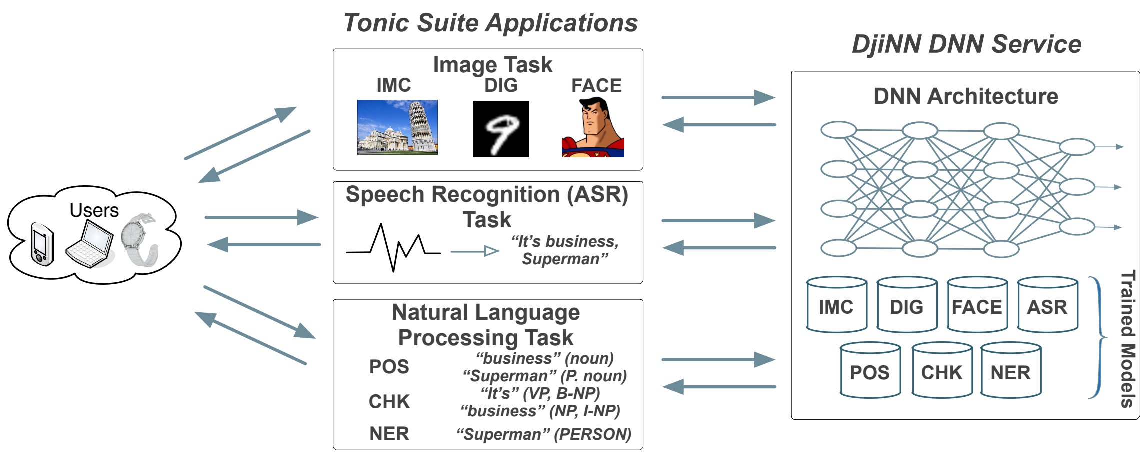


Figure 4.1: DjiNN Architecture

4.2.1 DjiNN Service

The goal of the DjiNN service is to provide a unified service that executes the DNN portion of the Tonic Suite applications. In our design, we target the following objectives:

- **Decoupled Architecture** – DjiNN needs to be a standalone service accepting and processing external requests. The DjiNN service is designed to accept requests using a custom socket protocol over TCP/IP. We use Caffe [69] for DNN computation, an open-source actively developed DNN library widely used in both academia and industry. For each incoming request, DjiNN spawns a worker thread, executes the DNN computation, and sends the prediction back to the application.

- **Diverse Applications** – A general DNN service must be capable of processing requests from a wide range of applications. Caffe's general framework supports various types of neural network layers. This enables flexible neural network configurations using Caffe. Caffe is extended to support DNN architectures from various applications representative of emerging WSC workloads including image processing, speech recognition, and natural language processing. Figure 4.1 shows the design, where DjiNN receives image, speech, and text based requests. DjiNN currently supports 7 DNN based Tonic applications. Supporting more applications simply requires providing DjiNN a pretrained neural network model.

- **Request Processing** – DjiNN must be able to process multiple incoming requests with limited overhead. At initialization, DjiNN loads the pre-trained model associated with each application into memory, giving all worker threads read-only access to this data. Consequently, incoming requests using the same model are accepted without needing to load their own copy of the model into memory.

4.2.2 Tonic Suite

Table 4.1: Tonic Suite Neural Network Architectures

Type	Application	Network	NN Type	Layers	Params
Image Service	Image Classification (IMC)	AlexNet [50]	CNN	22	60M
	Digit Recognition (DIG)	MNIST [79]	CNN	7	60K
	Facial Recognition (FACE)	DeepFace [111]	CNN	8	120M
Speech Service	Automatic Speech Recognition (ASR)	Kaldi [96]	DNN	13	30M
NLP Service	Part-of-Speech Tagging (POS)	SENNA [47]	DNN	3	180K
	Chunking (CHK)	SENNA [47]	DNN	3	180K
	Name Entity Recognition (NER)	SENNA [47]	DNN	3	180K

The DNN applications used in Tonic Suite are the bread and butter of the DjiNN service. They are based on recently published neural networks that achieve state-of-the-art accuracy in their target domains, which are summarized in Table 4.1. The suite of applications bundled with the neural network configurations, the trained models, and the server infrastructure to run the end-to-end applications have been released [10].

Image Task

Tonic Suite's image tasks encompass three applications: image classification, digit recognition, and facial recognition. The image tasks do not have pre or postprocessing steps; the service sends the most likely prediction about the image back to the application. Each of the three image applications is described below.

Image Classification (IMC) - Image classification sends an image to the DjiNN service and a prediction of what the image contains is sent to the application. This prediction is made by a model trained on 1.4M images from ImageNet [50], which can predict 1000 unique classes. AlexNet, a neural network architecture developed by Krizhevsky et al. [75], achieves very high accuracy and outperforms other methods in large scale image classification competitions [102].

Digit Recognition (DIG) - Digit recognition sends an image of a hand-written digit to the service and a prediction of the most likely digit (between 0-9) is returned to the application. The network architecture is based on MNIST [79], a widely used neural network for this task that achieves over 98% accuracy. A sample image is included in Figure 4.1.

Facial Recognition (FACE) - The facial recognition application predicts the identity of faces using the DjiNN webservice. The neural network architecture used in Tonic Suite was recently published by Facebook. DeepFace [111] is a facial recognition network that achieves near human-accuracy. This network is replicated into Tonic Suite and trained on a publicly available dataset of celebrity faces from PubFig83+LFW [36]. Using this dataset, DjiNN service classifies the input from 83 candidate celebrity faces.

Automatic Speech Recognition (ASR) Task

Included in Tonic Suite is a DNN based speech-to-text decoder adapted from Kaldi [96], a state-of-the-art speech recognition toolbox actively developed by researchers from Microsoft and academia. Kaldi's speech processing techniques have been demonstrated to achieve very low word error rates (WER) on standard decoding benchmarks. The speech recognition application requires preprocessing to generate feature vectors describing the speech input that are sent to the DjiNN webservice.

The service returns predictions for each feature vector that are postprocessed to find the most likely sequence of text to produce the final result.

Natural Language Processing (NLP) Task

Included in Tonic Suite are NLP tasks designed to glean semantic information from input text. These tasks include part-of-speech (POS) tagging, word chunking (CHK), and name entity recognition (NER). For these applications, the text is preprocessed into word vector representations before being sent to DjiNN. After receiving the word predictions from the DNN service, the postprocessing step searches for the most likely sequence of tagged words. The neural networks are based on Senna [47], a natural language processing toolbox developed by NEC Labs. The pretrained models are trained on Wikipedia for over 2 months and achieve over 89% accuracy for these applications.

Part-of-Speech Tagging (POS) Part-of-speech tagging assigns each word with a part of speech, for example if it is a noun or a verb.

Word Chunking (CHK) Word chunking tags each segment of a sentence as a noun or verb phrase where each word is labeled as a begin-chunk (B-NP) or an inside-chunk (I-NP). First, this application internally makes a POS service request, updates the tags for its input, and then makes its own DNN service request.

Name Entity Recognition (NER) Name entity recognition labels each word in the sentence with a category, for example whether it is a location or a person.

4.3 Identifying Bottlenecks for a DNN Service

This section presents a real-system analysis of the DNN service and evaluates the baseline DNN service performance on a state-of-the-art GPU. This section also compares the GPU performance with the performance achieved on an Intel Xeon processor. It then conducts a performance analysis to identify bottlenecks to guide

Table 4.2: Platform Specifications

Hardware	**Specifications**	**Quantity**
System	4U Intel Dual CPU Chassis, 8$\times$ PCIe 3.0 $\times$ 16 slots	1
CPU	Intel Xeon E5-2620 V2, 6C, 2.10 GHz	2
HDD	1TB 2.5" HDD	1
Memory	16GB DDR3 1866 MHz ECC/Server Memory	16
GPU	NVIDIA Tesla K40 M-Class 12 GB PCIe	8

further throughput optimizations in the following sections. The configuration of the experimental platform is summarized in Table 4.2. We use 1 GPU for all the experiments in this section.

4.3.1 DNN vs. non-DNN Components

First each DNN application is profiled on the Intel Xeon to characterize the amount of computation the back-end DNN service constitutes for each application. Figure 4.2 presents the average execution cycle breakdown for each application between its DNN portion and the rest of the computation (made up of query pre- and postprocessing). For IMC, DIG, and FACE, the input images are directly fed into the DNN. Consequently, almost all of the cycles for the image services are spent on DNN computation. ASR requires substantial pre- and postprocessing to translate a voice recording into the final text. Nevertheless, the DNN service still consumes almost half of the execution cycles for ASR. For the NLP tasks, which also have pre- and postprocessing, more than two thirds of the total execution time is DNN computation. This result demonstrates that DNN computation consumes a high percentage of the total execution time for almost all applications, motivating the need to design a common efficient DNN service in datacenters.

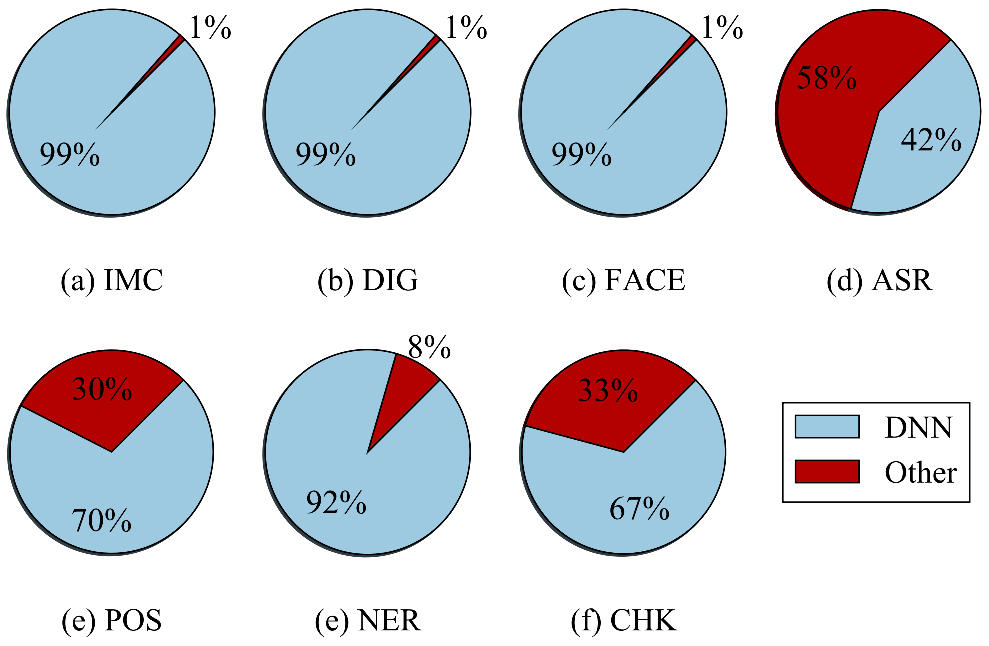


Figure 4.2: Cycle Breakdown for each DNN Application

4.3.2 Performance Bottlenecks

To guide the throughput optimizations, each DNN service is profiled using the NVIDIA Profiler [9] and the NVIDIA Visual Profiler [26] to conduct performance analysis. Figure 4.3 presents the profiling information of several hardware performance counters for each application. The metrics are collected at the kernel level for each application, and are weighted by each kernel's execution time to calculate the average performance of the entire application. As shown in the figure, the ratio of the IPC to the peak IPC (IPC/Peak IPC) is relatively low for NLP tasks. All applications exhibit low memory bandwidth utilization (low L1, shared memory, and L2 bandwidth utilization) relative to the peak bandwidth utilization, indicating that the low IPC is not caused by a memory bandwidth limit. On the other hand, the IPC is roughly correlated to the GPU *occupancy*, the ratio of the number of active warps to the theoretical peak number of active warps. All three NLP tasks have under 20% occupancy, while ASR achieves above 90% occupancy. Low occupancy indicates that

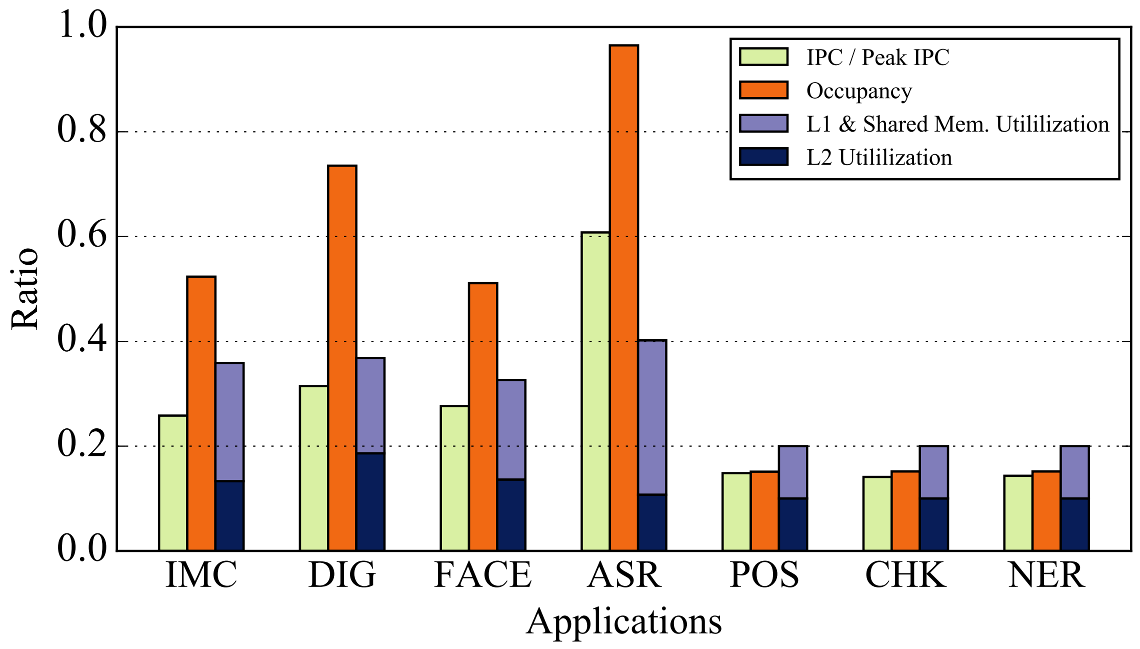


Figure 4.3: Performance Bottleneck Analysis

the GPU is not fully utilized for the NLP tasks. The kernels of these applications do not have enough thread blocks to hide the operation latency.

4.4 Designing a High Throughput System

As observed in the previous section, the throughput improvement achieved by a GPU is substantially different across the DNN service component of all applications. This is due to the different neural network architectures of each application and the resulting varying degrees of GPU occupancy.

In this section, we investigate and design techniques aiming to achieve the maximal throughput for the DNN service on GPUs. We investigate three throughput improving techniques: 1) batching multiple queries into a combined query to increase occupancy on the GPU; 2) executing concurrent kernels to achieve better GPU resource efficiency; and 3) scaling the number of GPUs in a server. In addition to designing and evaluating techniques for throughput improvement, this investigation also allows us to gain insights on the throughput capability of state-of-the-art GPUs for the DNN service.

Table 4.3: DjiNN Service Applications

App	Input	Data Size (KB)	Output	Batch size
IMC	1 image	604	1 classification	16
DIG	100 images	307	100 classifications	16
FACE	1 image	271	1 classification	2
ASR	548 feature vectors	4594	548 probability vectors	2
POS	28 word sentence	38	28 probability vectors	64
CHK	28 word sentence	75	28 probability vectors	64
NER	28 word sentence	43	28 probability vectors	64

4.4.1 Batching DNN Inputs to Improve Throughput

We first investigate techniques to increase GPU occupancy and DNN service throughput by batching multiple DNN inputs into a single query. The application name, input type, input size, and output data for the DNN service of each application is summarized in the first 4 columns of Table 4.3. To batch multiple inputs into a larger query the query input size is increased by stacking multiple inputs into a larger matrix. Consequently, this increases the dimensions of the matrix multiplication executed in the DNN's forward pass on the GPU. The increased computation achieved by batching increases the occupancy on the GPU and the system throughput.

For each application, we vary the batch size and study the impact on the throughput and latency achieved by the GPU. Figure 4.4 presents how throughput is affected with varying input batch sizes. As shown in the figure, all applications exhibit a similar trend: the throughput first increases then plateaus as the batch size continues to increase. The throughput saturation point for each application is at a different batch size. In addition, the throughput benefits from batching are different across applications. Automatic Speech Recognition (ASR), which already achieves a considerable

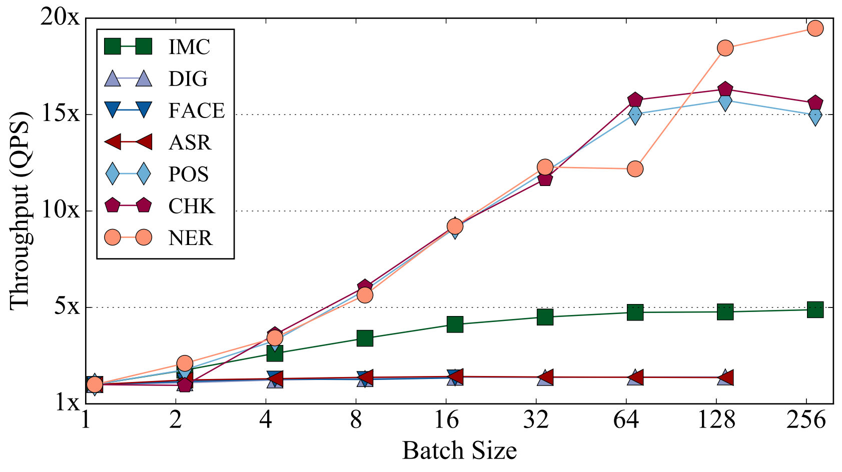


Figure 4.4: Throughput as Batch Size Increases

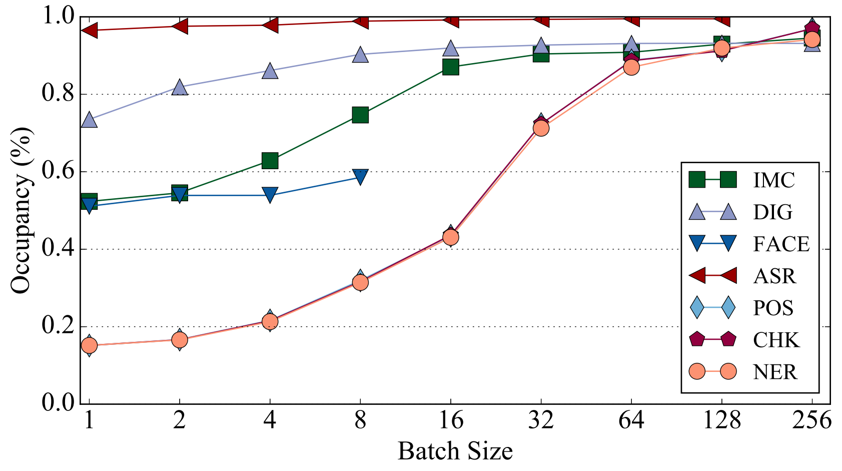


Figure 4.5: GPU Occupancy as Batch Size Increases

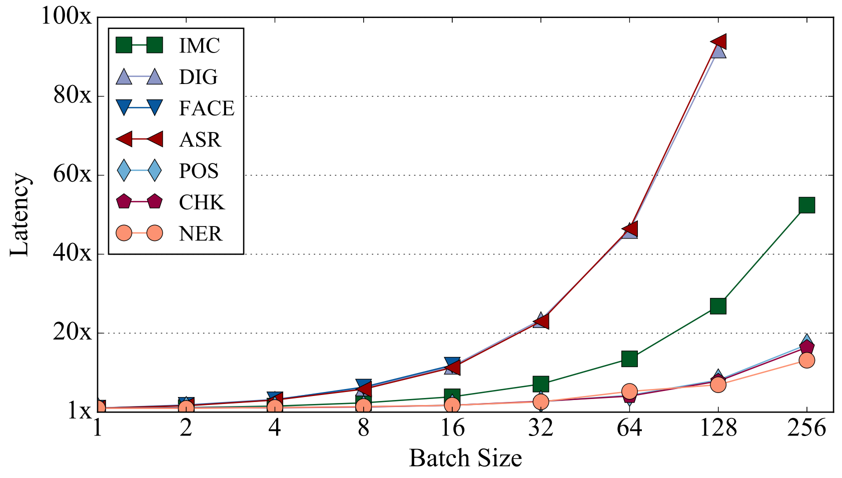


Figure 4.6: Latency as Batch Size Increases

(120×) throughput improvement over a Xeon core (Figure 1.3) and near 100% GPU occupancy without batching (Figure 4.3), has a small throughput gain with larger batch sizes. On the other hand, some applications achieve very high throughput improvement from batching. For example, NLP tasks achieve over a 15× throughput improvement.

This throughput improvement is from improving the GPU occupancy by batching queries, as shown in Figure 4.5. For NLP tasks, the baseline (batch size of 1) involves too little computation to fully occupy the GPU's resources, achieving only 20% occupancy. Increasing the batch size increases the amount of computation required. Consequently, the neural network computation uses more resources and the GPU occupancy significantly increases, achieving above 80% occupancy at a batch size of 64. There is no data for FACE beyond a batch size of 8 in this figure because of the large size of the neural network and the high profiling overhead incurred.

Figure 4.6 presents the query latency for each DNN service. All inputs in a batched query are combined in an aggregated large matrix computation and thus share the same query latency across inputs within a batch. As shown in the figure, the query latency for each DNN service increases slightly at first. As the throughput plateaus, the latency starts to increase sharply. At this point, the GPU is saturated and the queuing delay starts to dominate the latency.

Based on Figures 4.4 and 4.6, we identify the batch size for each application to achieve high throughput while limiting query latency impact. These final values are summarized in the last column of Table 4.3. Overall, with the selected batch size, we achieve 15× throughput improvement for NLP tasks and 5× for IMC with limited latency increases in both cases.

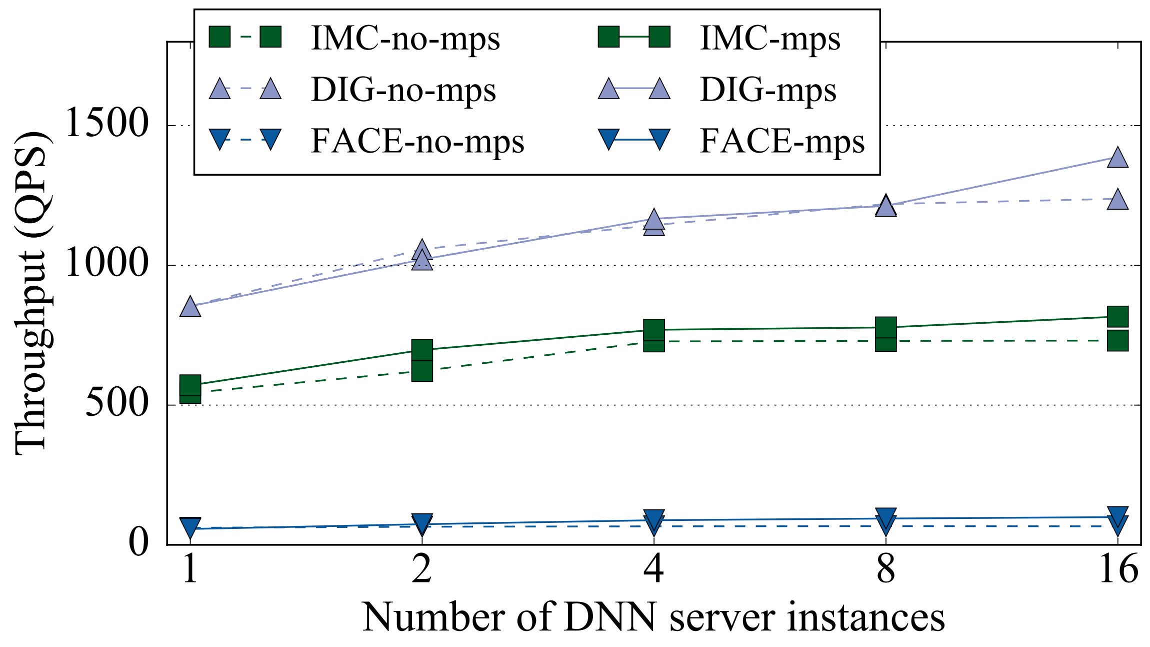


Figure 4.7: IMG Service Throughput as the Number of DNN Server Instances Increases

4.4.2 Supporting Multiple DNN Services on a GPU

Next, we use NVIDIA's Multi-Process Service (MPS) [25], which allows kernels from different processes to execute concurrently on the GPU. Without MPS, each CUDA process allocates separate scheduling and storage resources on the GPU. Each time a different process executes, the GPU must context switch before resuming execution; all processes must timeshare the GPU. MPS allocates a shared pool of scheduling and storage resources for independent processes. As a result, the GPU can schedule multiple kernels concurrently from the same pool of resources without the need to context switch.

Figures 4.7, 4.8, and 4.9 present the throughput improvement as the number of concurrent DNN services on the GPU increases from 1 to 16 (the maximum number of simultaneous processes that MPS supports). Throughput is measured as queries per second (QPS). Using MPS, DNN service instances can concurrently execute on

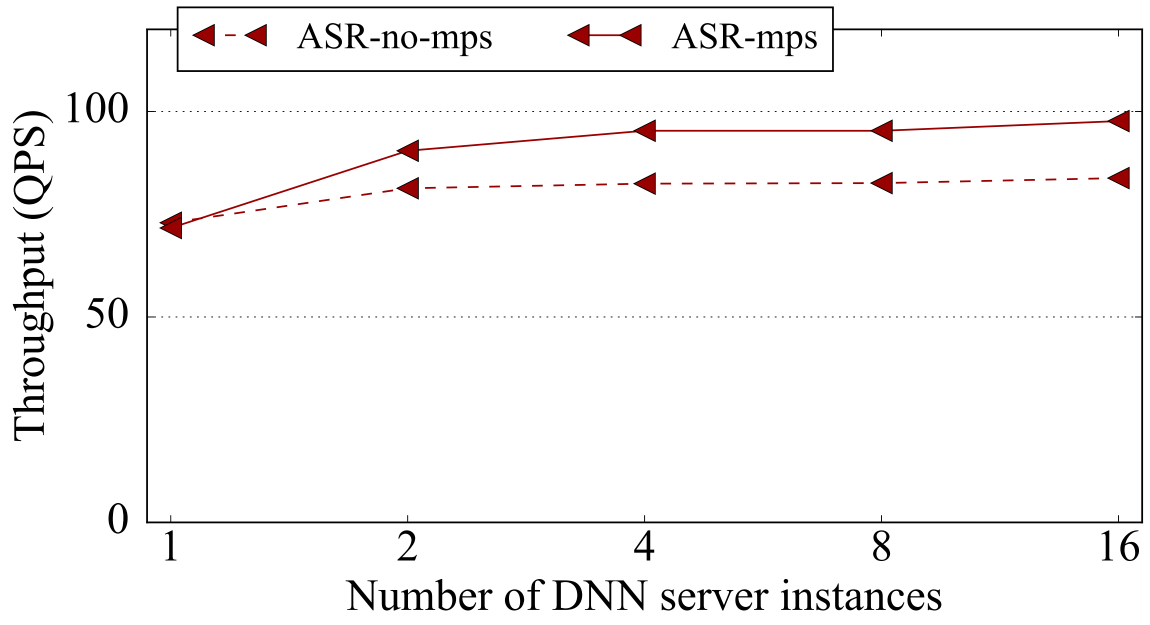


Figure 4.8: ASR Service Throughput as the Number of DNN Server Instances Increases

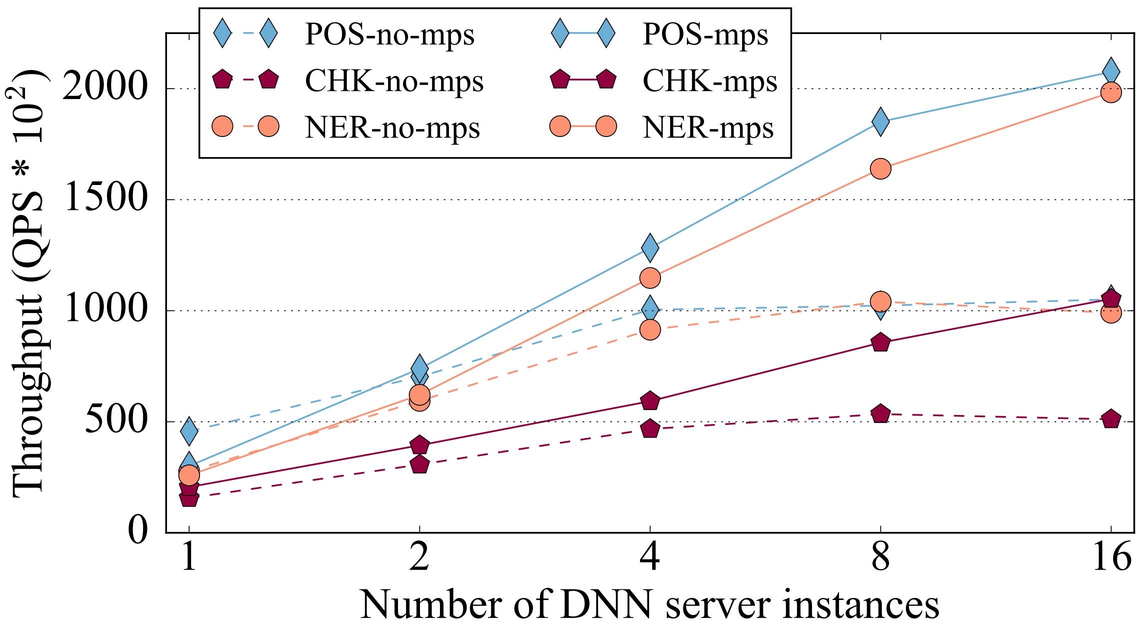


Figure 4.9: NLP Service Throughput as the Number of DNN Server Instances Increases

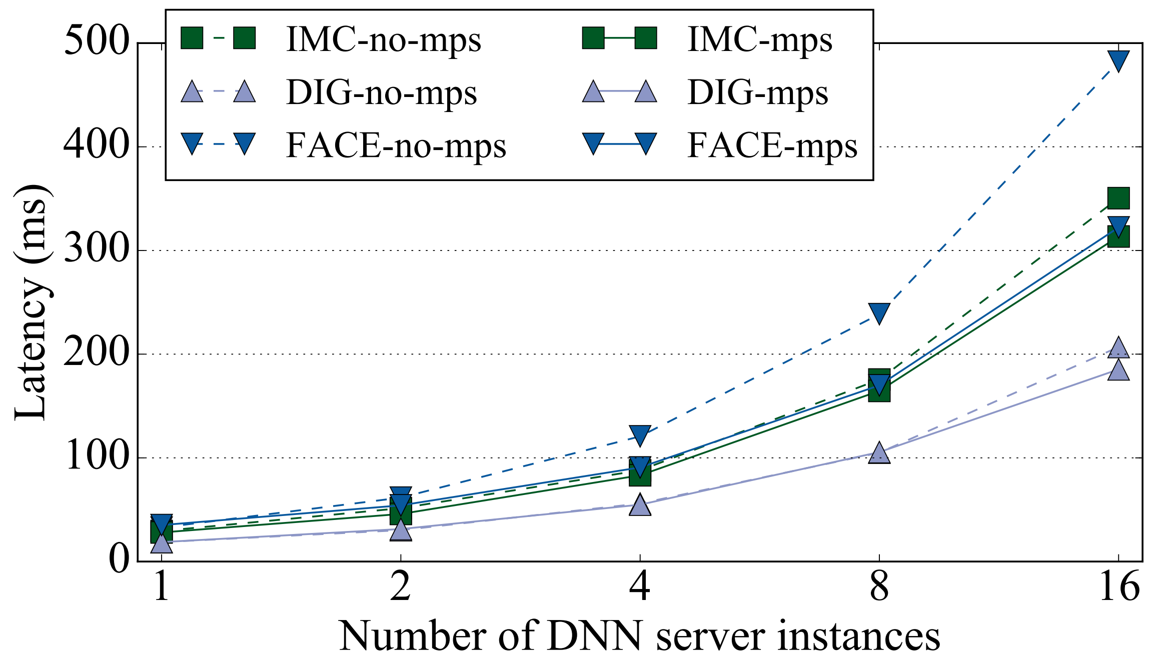


Figure 4.10: IMG Service Latency as the Number of DNN Server Instances Increases

the GPU. For comparison, the non-MPS cases are also presented, where queries from multiple DNN service instances are time sharing the GPU. The batch sizes summarized in the last column of Table 4.3 are used for each application. As shown in the figure, the achieved throughput increases as the number of DNN services on the GPU increases. With MPS, increasing concurrent kernels further improves throughput beyond what batching achieves (shown when the number of DNN instances is equal to 1). As previously described, without MPS, CUDA kernels launched by different processes timeshare the GPU's resources and have limited concurrency. With MPS, CUDA kernels launched by different processes can be executed concurrently. Because of this concurrency, the server queuing time for the next available time slice on the GPU is reduced and throughput increases. The throughput plateaus as the number of concurrent DNN services on the GPU further increases. Overall, the DNN service achieves up to a 6× throughput improvement with concurrent service execution on

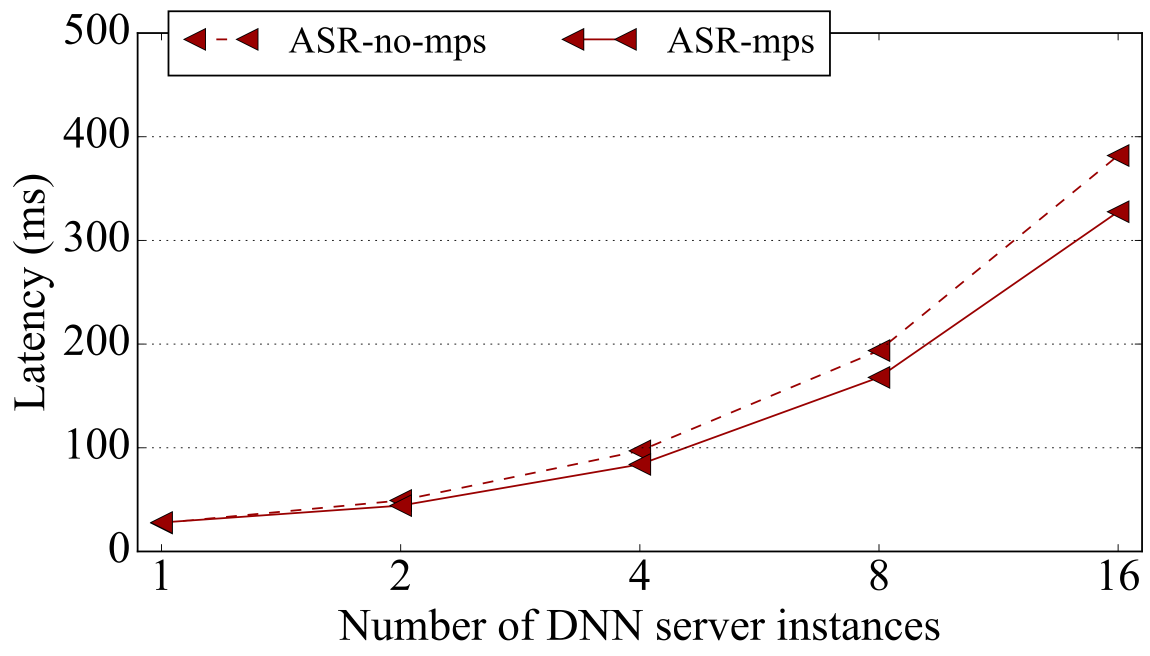


Figure 4.11: ASR Service Latency as the Number of DNN Server Instances Increases

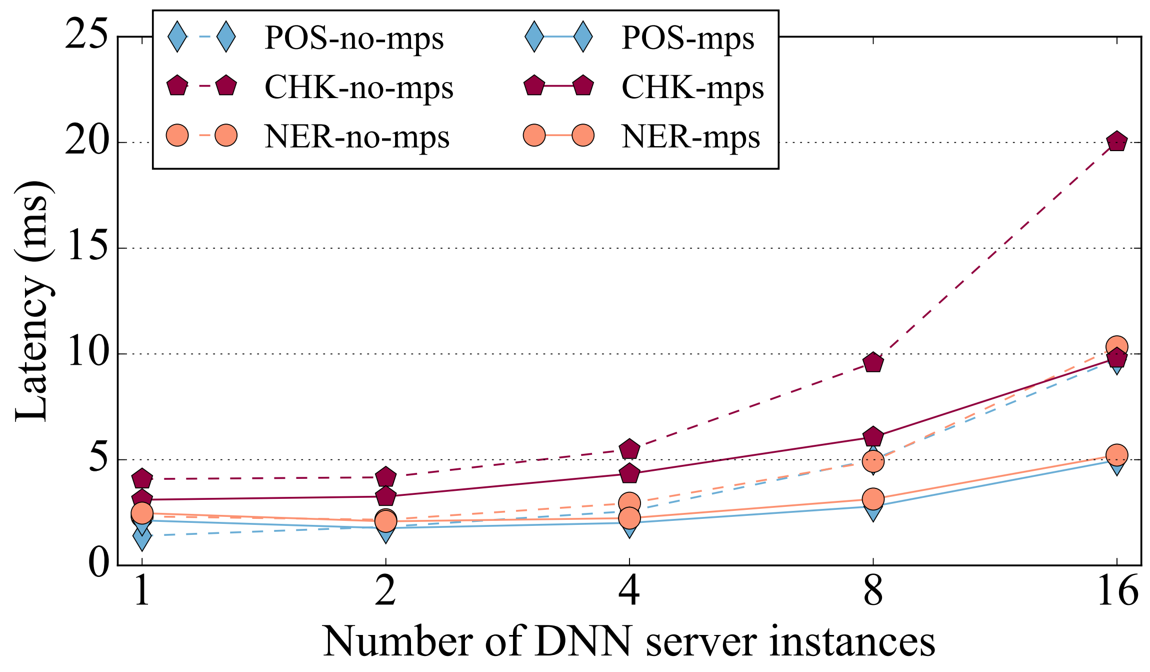


Figure 4.12: NLP Service Latency as the Number of DNN Server Instances Increases

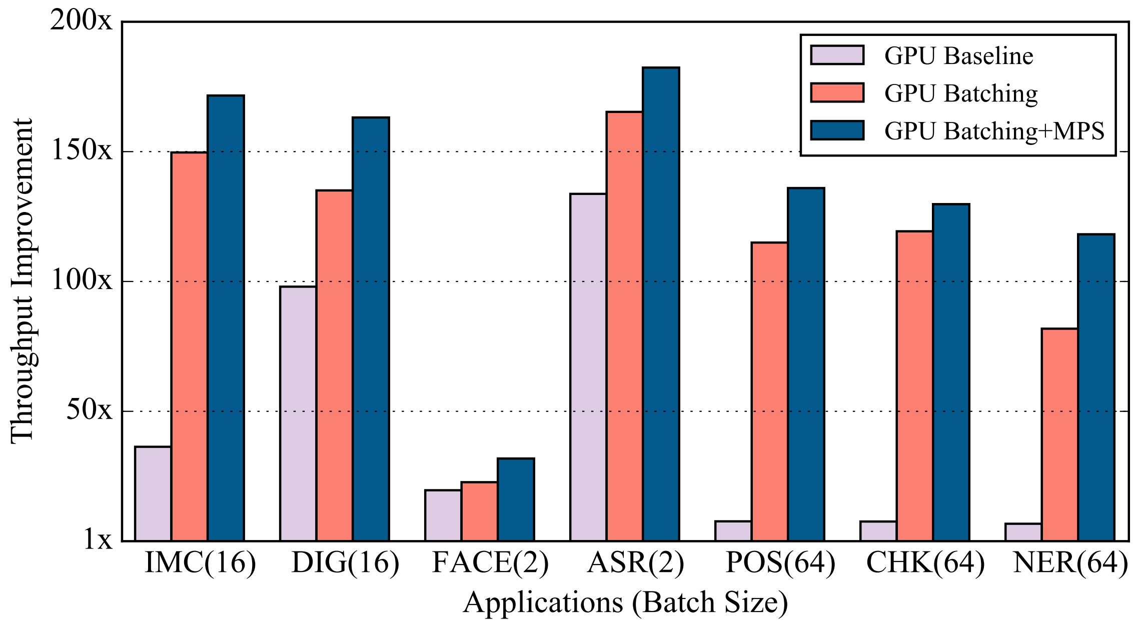


Figure 4.13: Throughput Improvement after Optimizations (GPU over Single-thread CPU)

the GPU.

Figures 4.10, 4.11, and 4.12 present the query latency as the number of concurrent DNN services on the GPU increases. The query latency is relatively small when the number of concurrent DNN services is under 4 but increases sharply as the number of DNN services grows. MPS successfully limits the latency increase when compared to experiments without. As discussed earlier, MPS reduces the queuing and thus, as shown in the figures, reduces the query latency up to 3×, compared to the non-MPS configuration. Compared to the baseline configuration of executing a single service at a time on the GPU, the DNN service applications benefit from concurrent DNN services, which improves both throughput and latency.

Combining Figures 4.7, 4.8, and 4.9, with Figures 4.10, 4.11, and 4.12 four MPS concurrent DNN servers on one GPU achieves high throughput gain with limited latency impact. For the DNN portion of most applications, more than 4 concurrent DNN services would have to trade high latency increase for low throughput improve-

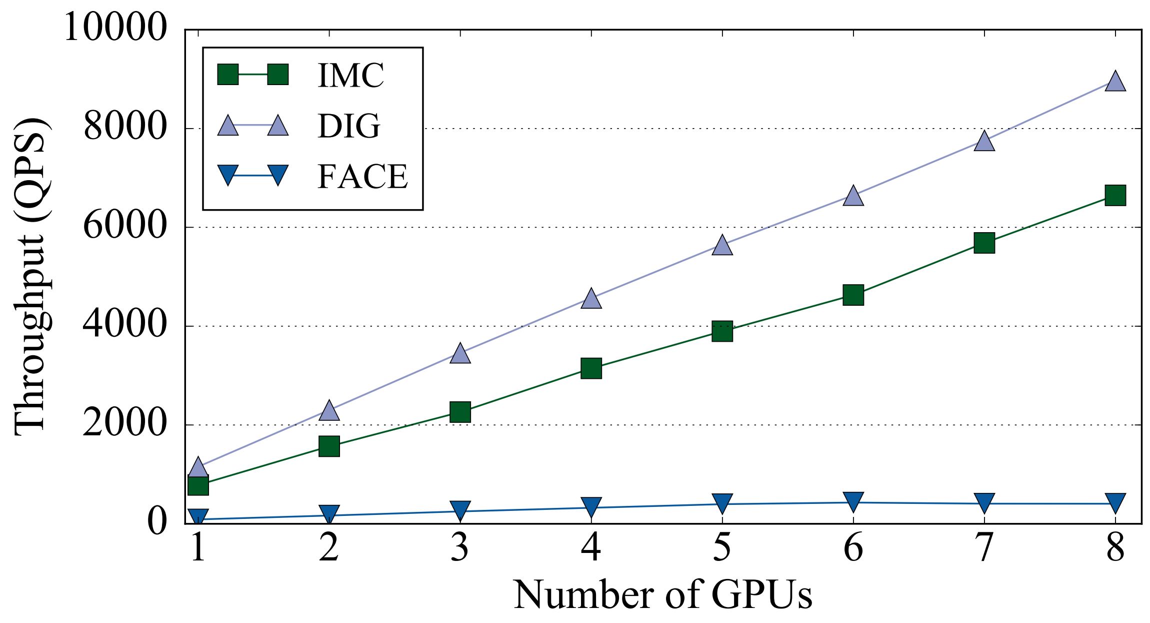


Figure 4.14: IMG Service Throughput as Number of GPUs Increases

ment. Note the latency achieved using 4 concurrent DNN services on the GPU is smaller than the single query service time on the CPU.

Figure 4.13 summarizes the final throughput improvements on a K40 GPU after applying input batching (with the best batch size next to each application in the figure) and MPS. We achieve significant throughput benefits across the applications through these two optimizations. For NLP applications, batching and MPS together improve the GPU throughput gain from 7× to over 120×. The DNN service components achieve over 100× throughput improvement on the GPU for all but the FACE application, which achieves a 40× improvement.

4.4.3 GPU Scalability

To further improve the system throughput, we scale the number of GPUs in a server and measure the system throughput of our optimized DNN service. The results are presented in Figures 4.14, 4.15, and 4.16, with each application configured

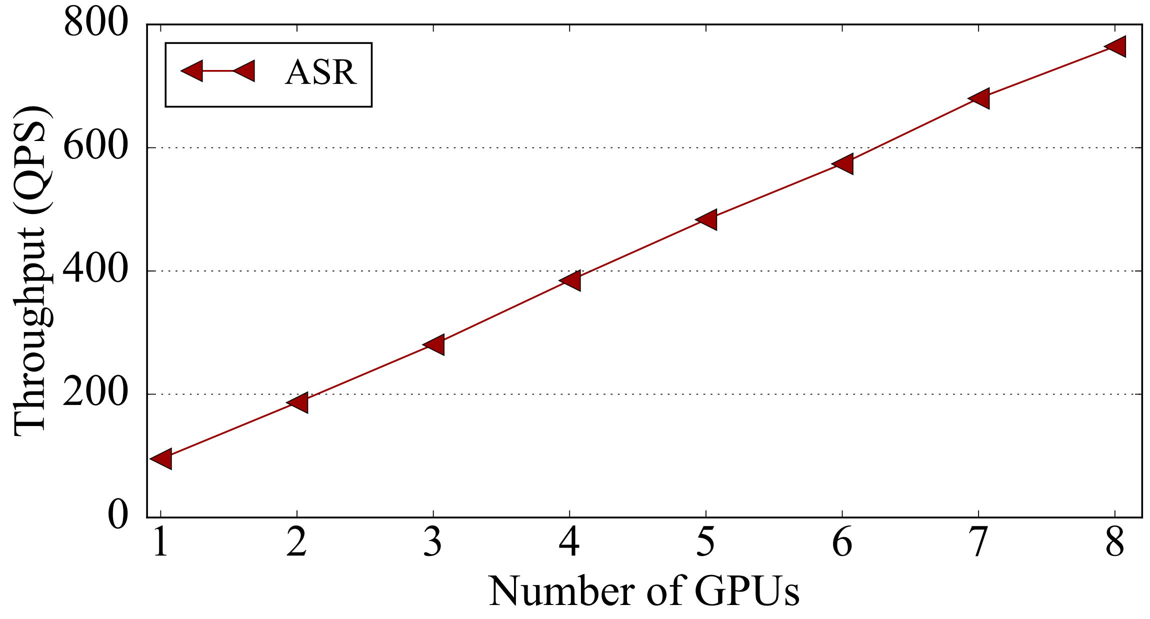


Figure 4.15: ASR Service Throughput as Number of GPUs Increases

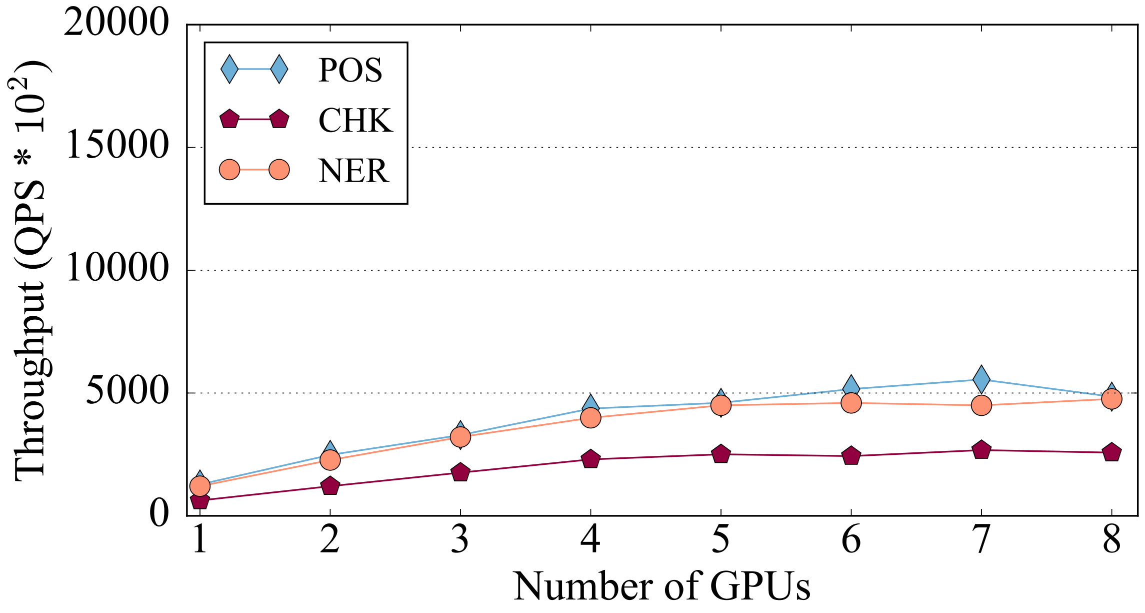


Figure 4.16: NLP Service Throughput as Number of GPUs Increases

to use the optimal batch size and 4 MPS processes per GPU. As shown in the figure, both image services and the speech recognition service achieve near-linear scaling as the number of GPUs increases. There is no communication between GPUs and the PCIe bandwidth between the CPU and each GPU is sufficient for these services. However, for the NLP tasks, which have relatively small neural networks, the throughput plateaus as the number of GPUs reaches 4. For NLP tasks, each query requires less computation and the throughput (QPS) is several orders of magnitude higher than the other two services. The throughput plateau is due to the PCIe bandwidth limitation.

In conclusion, the GPU scalability is dependent on the DNN characteristics for each application. For 3 out of 7 applications, by combining the optimizations and scaling the number of GPUs, 1000$\times$ throughput improvement is achieved on the 8 GPU system over a CPU core.

4.5 Implications for Future WSC Designs

Based on the insights gained from our throughput investigations in prior sections, we discuss the design of cost-efficient servers and the WSC systems necessary to provide a centralized DNN service for a wide range of applications. We first characterize the bandwidth requirements of the DNN service, identifying bandwidth to the GPUs as the performance bottleneck for NLP applications. Then consider three WSC design strategies for housing the DNN service and develop a TCO model to investigate the tradeoffs between the three designs, identifying the bandwidth constraint as a limiting factor for the TCO improvement of certain classes of DNN-based services. Finally, we describe and evaluate several network and interconnect architectures that can address the bandwidth limitation.

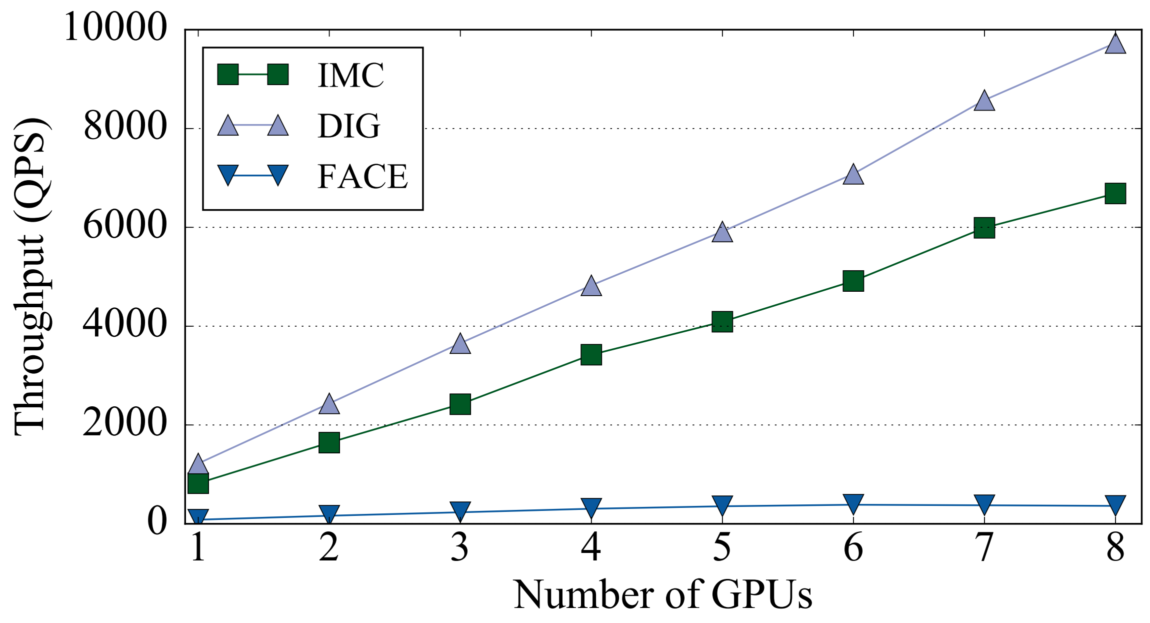


Figure 4.17: IMG Service Throughput as Number of GPUs Increases (no PCIe bandwidth limits)

4.5.1 Bandwidth Requirements for Peak Throughput

To design the network configurations for the DNN servers in datacenters, we examine the bandwidth requirements of the DNN service. The peak throughput gain is measured to guage what can be achieved without bandwidth constraints. To do so, communication is avoided by pinning the input of the DNN service to the GPU memory, which eliminates any data transfer (including transferring the final result). We stress-test the system to measure the throughput of a system with no PCIe bandwidth limit. Repeating the experiment of scaling out the number of GPUs using this PCIe-bypassing setup, we measure the theoretical throughput improvement, presented in Figures 4.17, 4.18, and 4.19. Without the PCIe bandwidth limit, all applications exhibit near-linear throughput improvement as the number of GPUs increases. This is expected because the computational capabilities are increasing

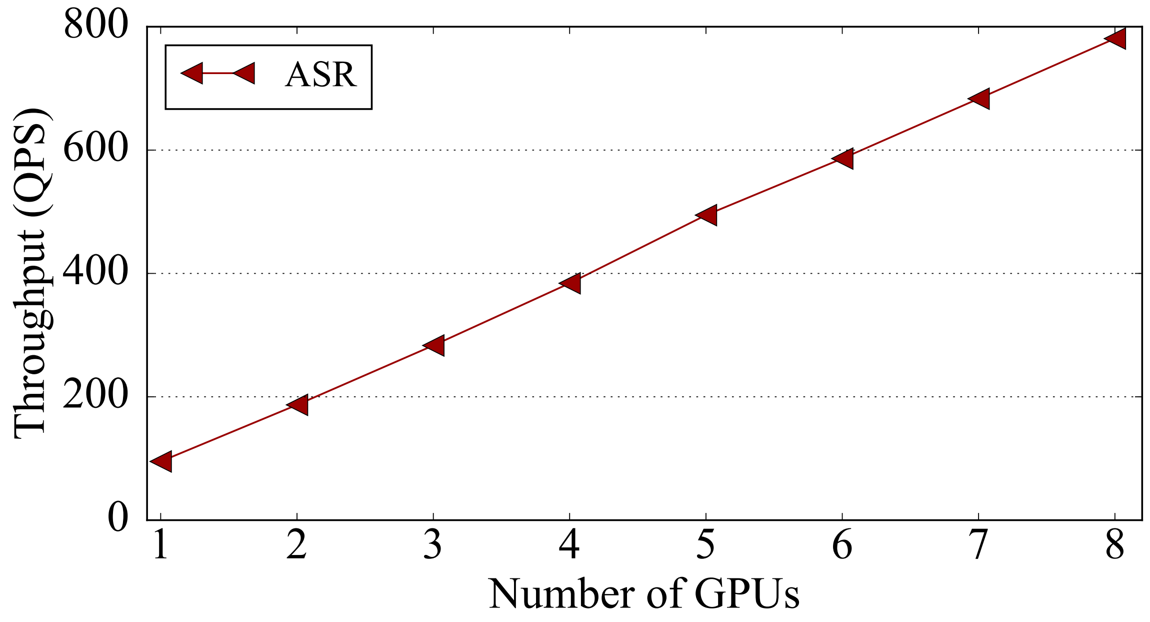


Figure 4.18: ASR Service Throughput as Number of GPUs Increases (no PCIe bandwidth limits)

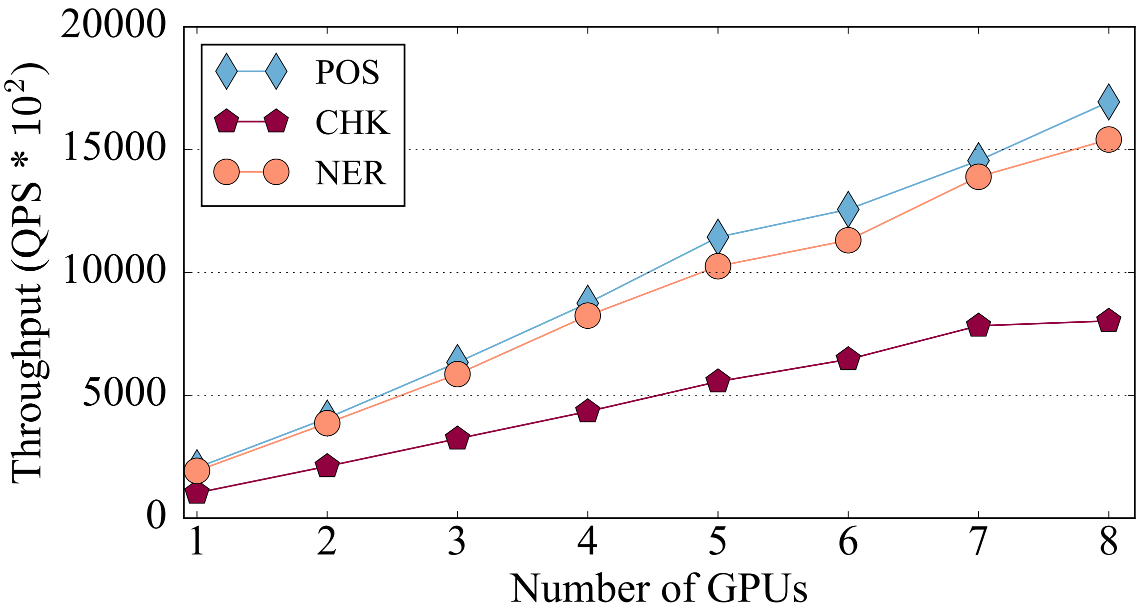


Figure 4.19: NLP Service Throughput as Number of GPUs Increases (no PCIe bandwidth limits)

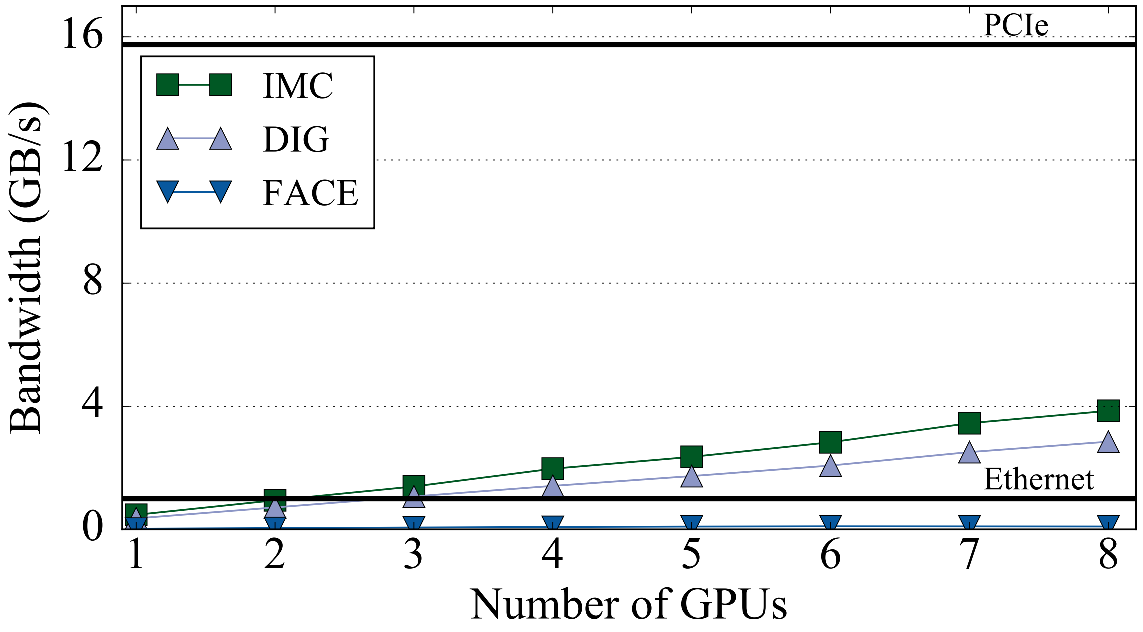


Figure 4.20: IMG Service Bandwidth Requirement as Number of GPUs increases

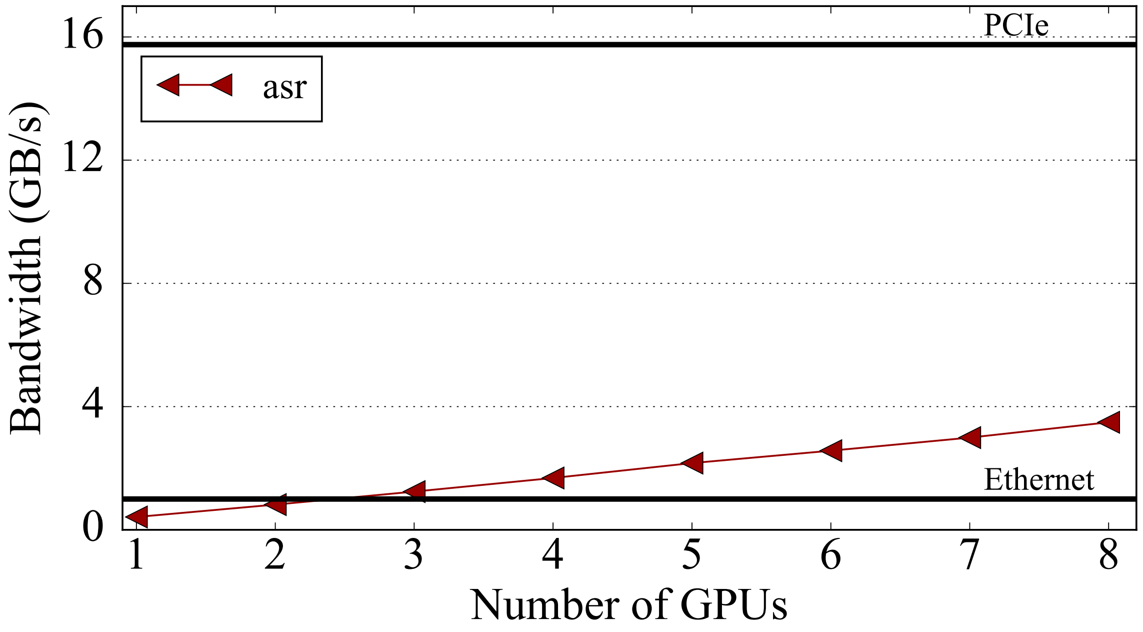


Figure 4.21: ASR Service Bandwidth Requirement as Number of GPUs Increases

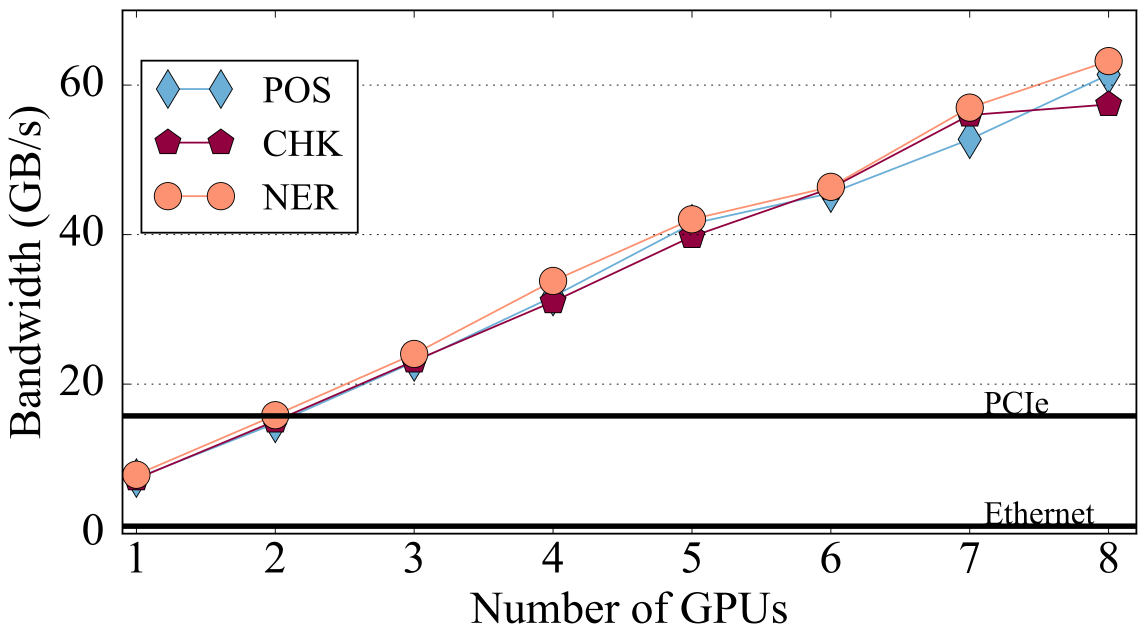


Figure 4.22: NLP Service Bandwidth Requirement as Number of GPUs Increases

without any bandwidth contention.

Based on the throughput improvement without the bandwidth constraint, we calculate the network bandwidth requirement for each application to achieve the maximum throughput. Figures 4.20, 4.21, and 4.22 present the network bandwidth requirements as the number of GPUs increases. As a point of reference, the peak bandwidth of several existing technologies, PCIe v3 and 10Gb ethernet (10GbE), are shown on the graph. For the computation-heavy tasks (IMC, DIG, FACE, ASR), the system is not bound by the PCIe bandwidth and the theoretical throughput can be achieved by a network with a bandwidth of at least 4GB/s. On the other hand, the light-computation tasks (NLP) require far higher bandwidth to sustain the near-linear throughput scaling. Later, these bandwidth requirements will be used as a guide to designing WSCs that are provisioned with sufficient bandwidth to overcome these bottlenecks.

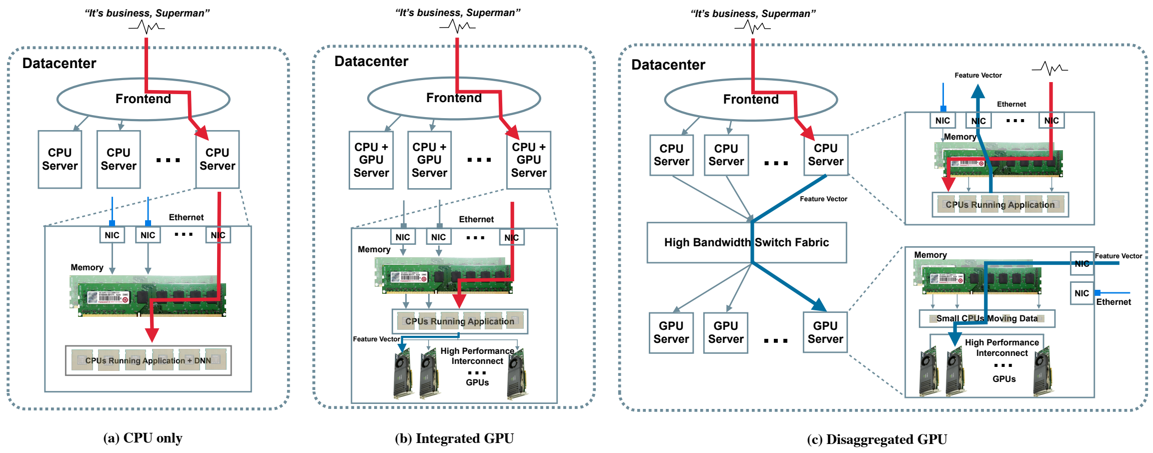


Figure 4.23: Three WSC Designs Considered

4.5.2 WSC Architectures for a DNN Service

We next describe three design points for WSCs that can be used to house the DjiNN service as illustrated in Figure 4.23.

CPU Only Design As a baseline, we describe a CPU only datacenter that has no GPU capability. This design, presented in Figure 4.23a, includes homogeneous servers and contain beefy *CPU servers* that service all of the workloads in the datacenter, including non-DNN applications, DNN applications, and the DjiNN service. Each DNN query that hits the datacenter passes through a front-end (e.g., a load balancer) to one of the CPU servers. The path taken by each query is illustrated by a red arrow in Figure 4.23a. After the query hits the NIC, it is placed in memory for the CPU to process in full.

Integrated GPU Design Second, Figure 4.23b presents the design of a datacenter with *Integrated GPUs*, containing a single server type of beefy CPUs and GPUs. In this design, the work of processing a query is handled within one server. However, unlike the *CPU Only* design, the work of processing the query is split between the CPU and the GPU. The path of the query to the CPU is shown as a red arrow in Figure 4.23b and upon receiving the query, the CPU performs (if necessary) preprocessing on the query. The result of the preprocessing is passed to the GPU hosting the DjiNN service via the PCIe bus (blue arrow in Figure 4.23b), where the GPU processes the request. By offloading DNN inference to the GPU, this model offers substantially higher throughput over the *CPU Only* model. However, by joining the GPU and CPU within the same box, along with the overwhelming preference in WSC design for homogeneous server configurations [32], GPUs have to be apportioned to servers to accommodate the homogeneous case. This study assumes 12 GPUs per server based on the latest available number of PCIe $\times 16$ slots available today on commodity high performance motherboards.

Disaggregated GPU Design To address the lack of flexibility of the integrated design, this work considers a design that has *Disaggregated GPUs.* In this design, two types of servers coexist in the datacenter. Beefy CPU servers, resembling those described for the *CPU Only* model, handle all non-DNN workloads as well as pre- and postprocessing for DNN queries. In this design, illustrated in Figure 4.23c, each DNN-based query is first preprocessed on the CPU server, then the result is sent over the network to a GPU server hosting the DjiNN service. The GPU server is designed as a multicore system with wimpy CPU cores whose purpose is to pass query data to the GPUs.

The advantage of this approach over the *Integrated GPU* is it decouples the GPUs and beefy CPUs. Such a decoupling can be critical in WSCs where designers are motivated to use a limited number of server configurations to simplify hardware and software maintenance and insure against overspecializing servers in the presence of ever-evolving workloads. By decoupling CPUs and GPUs, the amount of GPU compute can be provisioned to handle the amount of GPU work available in the datacenter without adding GPUs to each server. However, a major challenge in this model is to provision sufficient bandwidth between the CPU and GPU servers. To provide the necessary bandwidth between the two, 16 dedicated 10GbE NICs[1] are aggregated

[1]PCIe $\times 16$ supports up to 15.875GB/s. 10GbE can theoretically sustain 1.25GB/s, but may have significant protocol overheads. Assuming 80% of theoretical peak can be obtained, $16\times$ 1.25GB/s

Table 4.4: TCO Parameters

Component	Cost Factor
300W GPU-capable server	\$6864
High-end 240W GPU	\$3314
75W wimpy server	\$1716
Networking equipment	\$750/10GbE NIC
WSC capital expenditures	\$10/Watt
Operational expenditures	\$0.04/Watt/month
Power Usage Efficiency (PUE)	1.1
Electricity	\$0.067 per kWh
Interest rate on capital expenditures	8%
Server lifetime	3 years
Loan amortization period	3 years
Server maintenance/operations	5%/month

on each device and employ a high performance network fabric to sustain sufficient bandwidth.

4.5.3 Total Cost of Ownership

To assess the tradeoffs between these three designs, we compute the Total Cost of Ownership (TCO) for WSCs constructed to house DNN-based webservices using a methodology inspired by Barroso et al. [32]. The methodology for computing TCO includes upfront hardware capital expenditures (e.g., purchasing servers, CPUs, memory, GPUs, networking equipment, facilities, etc.), operating costs (operations, maintenance and power), as well as financing costs. The GPU and CPU failure rate differences are not explicitly modelled. Cost factors are summarized in Table 4.4. Power is measured on the GPU-enabled system to supply power draw estimates. In characterizing the price of the servers and GPUs, competitive market prices are used for the components at the time of this writing. For the GPU-capable server and GPU parts, the configurations priced are reflective of the high-end server used throughout this paper. To characterize the costs of networks in the approach, 500 server leaf

connection yields 16GB/s.

Table 4.5: DNN Service Workloads

Type	Description
MIXED	Mix (IMC, DIG, FACE, ASR, POS, CHK, NER)
IMAGE	Image processing (IMC, DIG, FACE)
NLP	Natural language processing (POS, CHK, NER)

nodes are assumed and connected to a hierarchical 10GbE network containing a mix of core and edge switches. Then the cost of those switches is averaged out across the 10GbE NICs installed in the servers to arrive at a cost estimate of $750 per NIC.

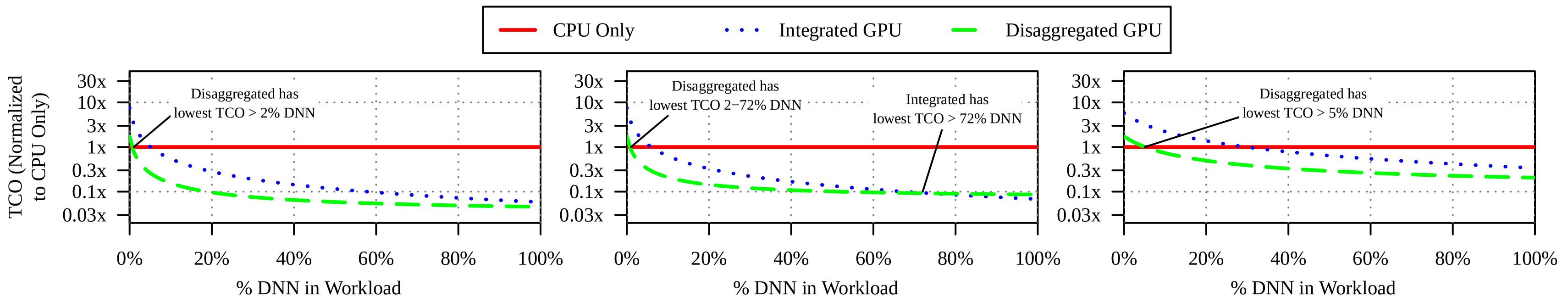
To characterize each WSC design, we first assume a workload composed in part by one of the DNN service mixes described in Table 4.5 and in part by non-DNN webservices. For this mix of webservices, we provision enough compute for the *CPU Only* design point to characterize its TCO and obtain a series of performance targets for each service. For example, given a workload composed of 70% from the MIXED DNN workload along with 30% non-DNN services, we provision 30% of the servers to non-DNN services and 10% to each of the DNN services (the MIXED workload is composed of 7 services). We then build out the *Integrated GPU* and *Disaggregated GPU* designs, each matching the throughput obtained by the *CPU Only* design, finally applying the model described above to characterize their TCO.

DNN's Implications for WSC Design The results of the TCO analysis are presented in Figure 4.24 for (a) the MIXED workload, (b) the IMAGE workload and (c) the NLP workload. Each plot presents the TCO of the three WSC designs across a range of assumptions about the mix of DNN and non-DNN services (x-axis), where the presented TCO is normalized to the *CPU Only* case and presented on a log scale (y-axis).

For the MIXED workload presented in Figure 4.24a, both GPU-based designs show substantial improvements over the *CPU Only* design, except when the workload is composed almost entirely of non-DNN services. This demonstrates that there are potentially sizable cost savings available by accelerating DNN-based services (up to 20× for *Disaggregated GPU* design) as these services consume an increasing volume of cycles in WSCs. The *Disaggregated GPU* design also improves upon the *Integrated GPU* design by between 10% and 2×, which can be attributed to the relatively inefficient use of GPUs by some of the DNN services in the *Integrated GPU* design. In particular, each server in the *Integrated GPU* design utilizes the same number of GPUs, while the NLP services can saturate only a subset of those available GPUs because they are bandwidth-limited by the PCIe interface. This inefficiency is alleviated by the *Disaggregated GPU* design, which decouples CPUs and GPUs and allows for fewer GPUs to be employed in the WSC.

The IMAGE workload, presented in Figure 4.24b, behaves similar to the MIXED workload, except there is a crossover point when the number of DNN services exceeds 72% of the workload. After this point, the *Integrated GPU* design has lower TCO than the *Disaggregated GPU* design. Because the TCO benefits in the *Disaggregated GPU* design over the *Integrated GPU* design arise from over-provisioning GPUs in the *Integrated GPU* design, those benefits slowly disappear as the workload running in the WSC is comprised of more DNN-based services that utilize all of the GPUs in the server (i.e. IMC, FACE and DIG).

The NLP case, presented in Figure 4.24c has a similar trend to 4.24a: the *Disaggregated GPU* model has the lowest TCO over most of the workload mixes and is a modest improvement over the *Integrated GPU* design over that entire range. However, the TCO for the NLP case is much closer to the TCO of the *CPU Only* design, showing a maximum improvement of 4×, as opposed to the 20× for the MIXED case. This difference occurs because, instead of being partially composed of NLP services as in the MIXED workload, the NLP workload is composed entirely of NLP services. Because the performance of the NLP applications is bound by the bandwidth of the PCIe, the available GPUs cannot be fully utilized.



(a) MIXED workload **(b) IMAGE workload** **(c) NLP workload**

Figure 4.24: TCO of WSCs Housing Proportions of DNN and non-DNN Webservices, Normalized to CPU Only Design (lower is better)

Table 4.6: Interconnect and Network Configurations. The networks are designed to use bonded ethernet connections numerous enough to saturate the CPU/GPU interconnect, assuming an additional protocol overhead of 20% on ethernet. Prices are phrased as the purchase cost over the PCIeV3/10GbE design point

	Interconnect			**Ethernet**	
	Architecture	Bandwidth (GB/s)	Price ($)	Bandwidth (GB/s)	Price ($/NIC)
PCIeV3/10GbE	1× PCIe v3 bus shared by GPUs	15.87	+$0	1.25 per NIC, up to 16 NICs	+$0
PCIeV4/40GbE	1× PCIe v4 bus shared by GPUs	31.75	+$2000	5 per NIC, up to 9 NICs	+$1250
QPI/400GbE	1 QPI link between GPU and CPU socket 6 links/GPUs per socket	307.2 (25.6 per link)	+$4000	50 per NIC, up to 8 NICs	+$4250

4.5.4 Addressing the Bandwidth Bottleneck

To address this bandwidth limitation, we consider two alternative designs to the typical configuration comprised of GPUs supplied by PCIe v3 and a 10GbE network. First, representative of cutting edge technology available today, we describes a design that connects the GPUs with PCIe v4, which doubles the bandwidth of PCIe v3 to 31.75GB/s. Accordingly, the network is provisioned to also have more bandwidth by using a 40GbE network with teamed connections at the server level. Assuming a 20% protocol overhead for ethernet, the PCIe v4 bus can be saturated by 9 teamed 40GbE connections. Second, representative of a more aggressively designed system that uses near-future technology, a design that employs Quick Path Interconnect (QPI) [76] is considered to connect CPUs to GPUs inside the server. Assuming 12 GPUs inside a 2-socket server, 6 point-to-point QPI links would be needed in each socket. Standard QPI links available at the time of this writing yield 25.6 GB/s, which is a total of 307.2 GB/s across all 12 links. To provision enough bandwidth in the network to feed the GPUs, and again assuming a 20% protocol overhead for ethernet, 8 teamed 400GbE connections are sufficient to saturate the QPI links.

We summarize these alternative design points in Table 4.6. Included in the table are the assumptions about the cost of these alternative designs, which are developed using a similar methodology described for the PCIe v3/10GbE design point, along with projections of the unit costs for PCIe v4, QPI, 40GbE NICs/switches and

400GbE NICs/switches.

Network Impact on Performance and TCO The impact of these design points are characterized with improved bandwidth by scaling up the networking equipment in the *Disaggregated GPU* model. The assumption is made that bandwidth-constrained DNN services (NLP) bypass the bandwidth limitations demonstrated in Figure 4.22 and continue to scale up in throughput beyond the throughput measured on the GPU-enabled server. In the *Disaggregated GPU* design, we model this performance improvement due to scaling up the network then introduce designs for the *CPU Only* and *Integrated GPU* cases that match the performance improvement. Note that we model *CPU Only* designs as having PCIe v3 and 10GbE, as improving the network does little to improve performance of the *CPU Only* design.

The results of this exercise are presented in Figure 4.25, applying it to workloads comprised entirely of either the MIXED DNN service (a) or of the NLP DNN service (b) in Figures 4.25 (the IMAGE workload is not bandwidth constrained, so it is not considered here). The figure shows the performance improvement achieved by introducing the improved network into the *Disaggregated GPU* design as black lines with "x" marks. Each group of bars shows the growth in various components of TCO that are associated with growing the WSC to improve performance.

Several interesting conclusions can be drawn from these experiments. First, improving the bandwidth provisioning in the network is an essential step to unlocking the full potential of GPUs for bandwidth-heavy NLP services. Large performance improvements can be realized while minimally impacting TCO in GPU-enabled WSCs. As the figure shows, the growth in TCO for the *Disaggregated GPU* design stems primarily from increased networking costs because the approach relies heavily on the network to pass large amounts of data from CPU-based compute servers to GPU-centric servers. In the *Integrated GPU* design the cost increases are slight, showing

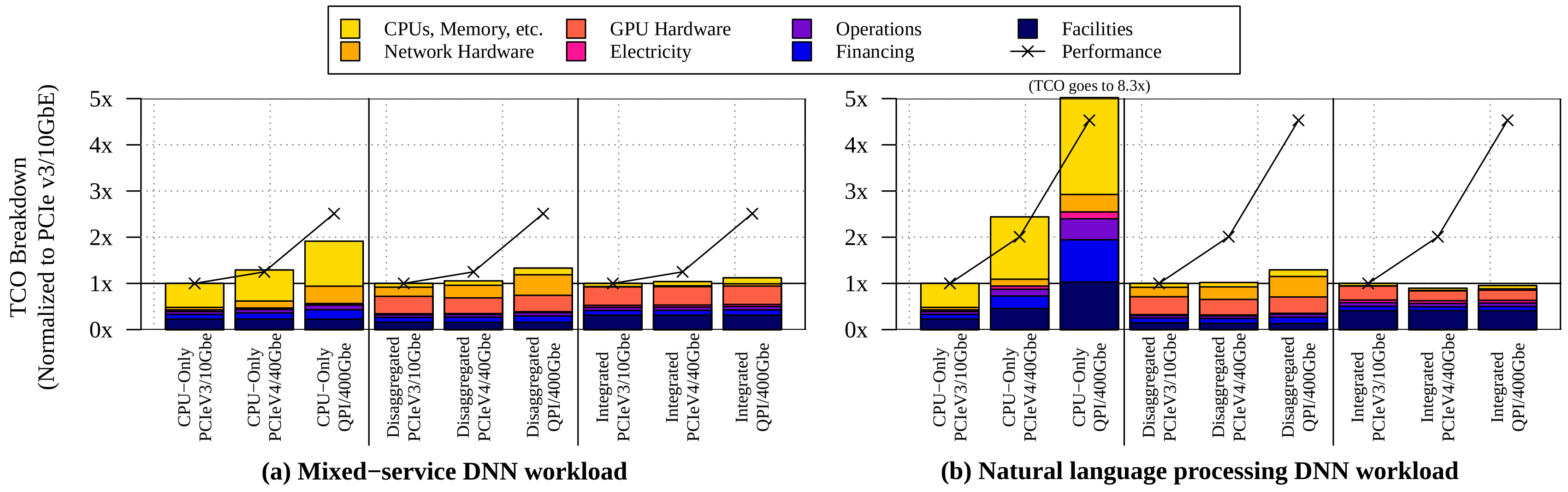


Figure 4.25: TCO Breakdown Showing Impact of Future Networking Technologies on GPU-enabled WSCs Housing DNN Services

up primarily in the MIXED workload as increases in the server cost (PCIe and QPI costs appear as part of the server costs). For the NLP workload, improving the bandwidth actually reduces TCO slightly for both improved network designs. This occurs because the increased utilization of GPUs allows the design to use fewer GPUs while still improving performance significantly. Second, scaling up the performance of DNN-based services is extremely difficult to do without accelerating them. For both the MIXED and NLP workloads, scaling up throughput requires scaling up the number of servers in the *CPU Only* design roughly in proportion to that increase. Given current CPU and GPU designs, this identifies GPUs as being the more promising direction for scaling up DNN-based webservices.

4.6 Summary

This work introduces **DjiNN**, an open source deep neural network service and **Tonic Suite**, a suite consisting of 7 end-to-end DNN-based applications in the vision, speech, and natural language processing domains. Using DjiNN, we design a high-throughput DNN system based on massive GPU server designs. In most cases, our final server design achieves over a 100× throughput gain on a single GPU compared to the CPU baseline, and achieves almost linear scaling with the number of GPUs. We study the total cost of ownership to provide insights into designing future warehouse scale computer architectures for DNN services. In terms of total cost of ownership, GPU-enabled datacenters show an improvement over CPU-only designs by 4-20×. In the case of bandwidth-heavy NLP applications, we show that leveraging improved GPU interconnect and network components to alleviate bandwidth constraints is one of the keys to achieving the aforementioned improvements.

CHAPTER V

Fine-Grained Cross-Input Batching for Natural Language Processing

This chapter introduces *fine-grained cross-input batching* as a technique to address scalability issues for deep learning based Natural Language Processing (NLP) applications. As the underlying Deep Neural Network (DNN) based NLP algorithms change, hitherto designed systems must equally adapt. In this chapter, we investigate a set of three distinct NLP applications and show, not just how varied the algorithmic landscape is for deep learning based applications, but how varied NLP applications are with respect to each other. The key finding is that NLP applications have computational characteristics making current systems perform suboptimally. The technique set forth in this chapter addresses the iterative and dependent computation patterns involved in executing an end-to-end NLP application and shows substantial improvements over other systems.

5.1 Natural Language Processing Applications

Recent advances in machine learning techniques has prompted the emergence of applications where users interact with their personal computing devices using natural language rather than a constrained set of buttons and fields. The category of machine learning tasks facilitating this transition, Natural Language Processing (NLP), has become critical to the evolution of modern user interfaces. In this work, we aim to answer research questions as system designers building datacenter systems hosting state-of-the-art NLP applications. We aim to study NLP applications that are 1) representative of complete applications designed to service user queries and 2) achieve the state-of-the-art accuracy in solving their respective tasks. Based on these criteria, we surveyed recent publications and select 3 applications solving two of the most prominent problems among the NLP community: *sentiment analysis* and *automatic text summarization*.

Sentiment Analysis - This application (SA) analyzes the emotions and attitudes in natural language, an application that plays a pivotal role in business planning, political campaigns, and social media analysis [82]. We investigate a Convolutional Neural Network (CNN) based implementation [72] (SA-CNN) and a tree-structured long short-term memory neural network based implementation [110] (SA-LSTM). SA-CNN and SA-LSTM achieve state-of-the-art accuracy on binary and 5-class sentiment analysis, respectively. To achieve state-of-the-art accuracy on 5-class classification, SA-LSTM uses a parser that generates a constituency tree [123]. This constituency tree describes the semantic relationships of the words in the sentence the application is analyzing.

Summarization - Automatic Text Summarization extracts the crux from a body of text, allowing users and higher-level algorithms to ignore extraneous information. Automatic summarizations are widely used in news and content delivery services, for

Table 5.1: Application Specifications

Application	**Network**	**Input**	**Input Length**	**Description**
SA-LSTM [110]	LSTM	Movie Reviews [24]	2 - 67 Words	Sentiment Analysis
NAMAS [101]	DNN	News Articles [11]	1 - 20 Words	Text Summarization
SA-CNN [72]	CNN	Movie Reviews [24]	2 - 67 Words	Sentiment Analysis

example by news agency and websites to automatically generate synopses, keywords and titles of news articles [8,13]. In this work, we study the abstractive summarization application, NAMAS [101] which is designed at Facebook to generate news titles based on the first sentence of a news article.

5.2 Characterization

In this section, we characterize three state-of-the-art NLP applications and juxtapose their computational characteristics with previously studied deep learning based applications.

5.2.1 Variable and Dependent Invocations

To understand the dynamism of the SA-LSTM application and the rest of the NLP applications studied, we begin by studying the nature of the inputs to these applications and how they affect the variability in computation.

Table 5.1 shows the three NLP applications studied where the second column shows the different types of neural network architectures. The network type defines how the input is processed strongly suggesting there will be large differences between the three applications. We use the entire dataset (training and testing) supplied with each open-source implementation as a representative dataset of the variability in input these applications have in deployed environments. The fourth column shows the range of input sizes in each dataset where for example the SA-LSTM application has input sentences ranging from 2 to 67 words.

Figures 5.1, 5.2, and 5.3 show the result of our experiment where we plot the probability mass function (PMF) of each application. The x-axis is the number of invocations to the neural network (NN) computation. SA-LSTM (Figure 5.1) and NAMAS (Figure 5.2) have large variance in the number of NN invocations. Before processing an input sentence, SA-LSTM preprocesses its input using a constituency parser that extracts the semantic relationships of the words in the sentence [123] and the resulting input to the LSTM is a parse tree. The number of NN invocations is the number of nodes in the parse tree. Additionally, there is a direct dependence between the invocations of the NNs since there is an explicit hierarchical dependence between each NN (leaf node). The input to the NAMAS application is a news article headline and a desired length for the output summary. The number of NN invocations for NAMAS is the number of words in the output summary. There is a dependence between each NN invocation because the output word in the previous step impacts the word generated at the next timestep to generate a grammatically correct summary. While the dataset for SA-CNN has a range between 2 and 67 words, the application pads the input to the network to the longest sentence in the training data (in this case 67 words). Consequently, there is no variance in the number of NN invocations as the CNN is executed once. For the rest of this work, we use this application as a representative application of static neural network processing.

We can conclude that two of three applications investigated show high input variability that is directly correlated with the input to the application. Additionally, the algorithmic structure of SA-LSTM (tree-structured input) and NAMAS (linear dependency between NN invocations) creates a dependency between NN invocations that, as we will see later in this work, expose new challenges in designing efficient systems for deep learning based applications.

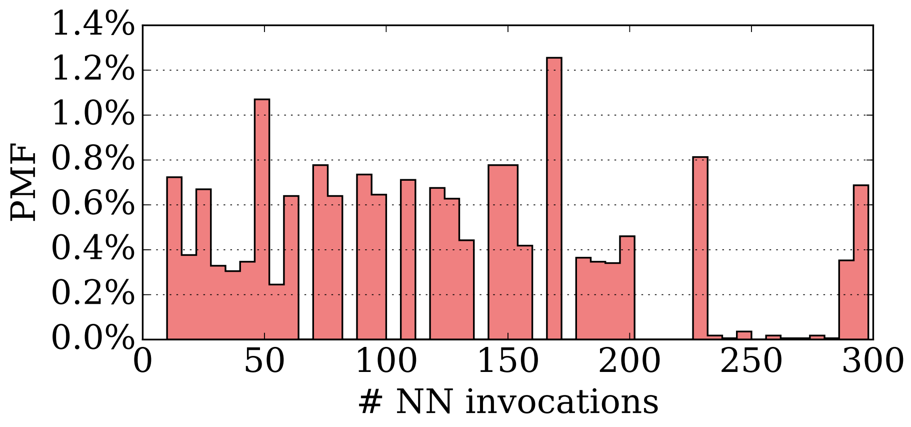


Figure 5.1: NN Invocation Variability for SA-LSTM

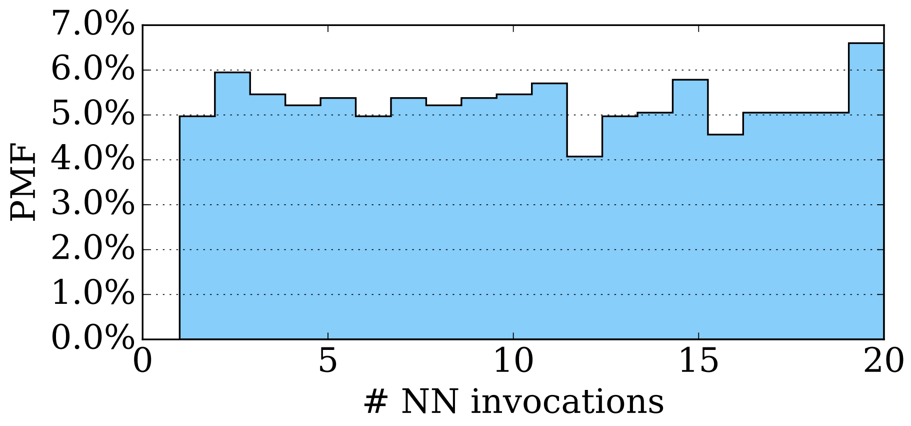


Figure 5.2: NN Invocation Variability for NAMAS

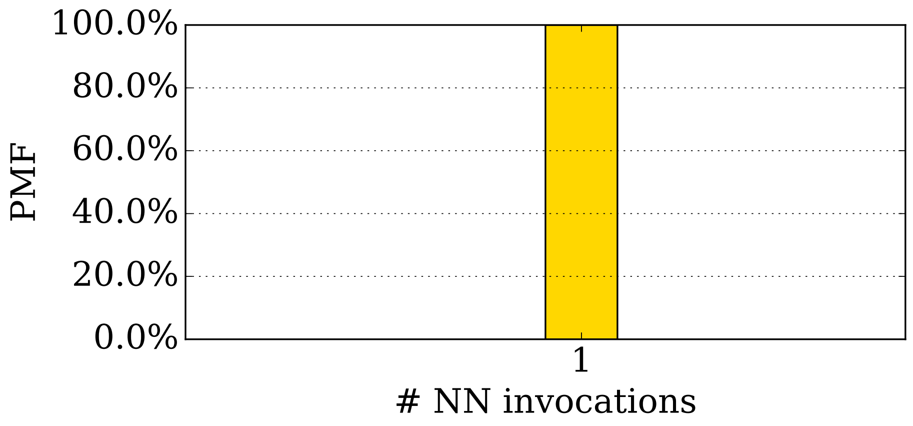


Figure 5.3: NN Invocation Variability for SA-CNN

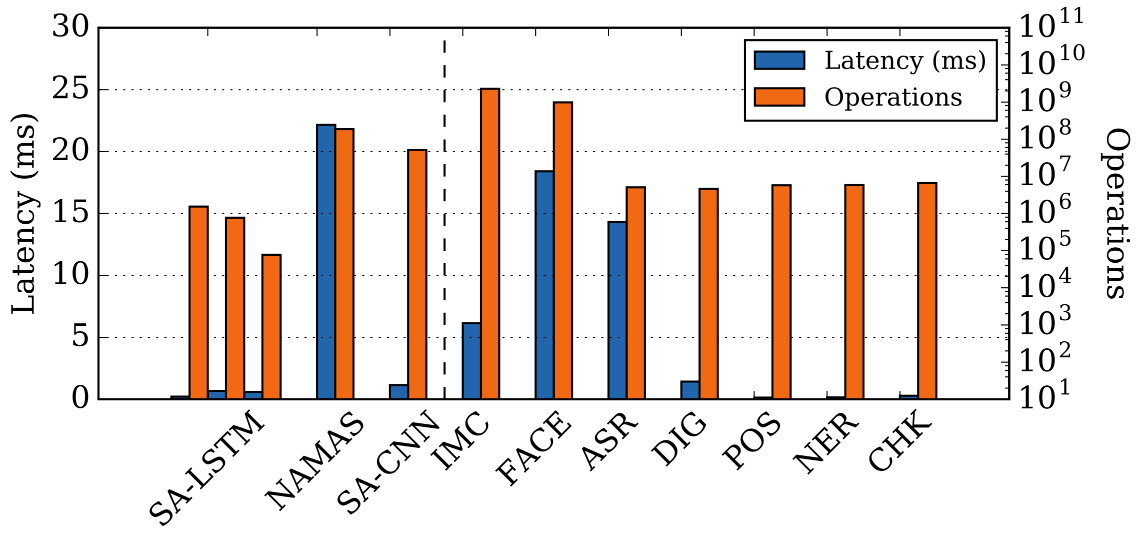


Figure 5.4: Latency and FLOPS of DNN Applications on the GPU

5.2.2 Kernel Computation

Intuitively, the iterative nature of processing natural language lends itself to smaller Neural Network (NN) kernels since an NN invocation processes a single word, compared to, for example, an entire image.

We characterize this difference in Figure 5.4, which shows the number of floating-point operations per NN invocation on the GPU and the corresponding GPU latency. These applications include the NLP applications (left) as well as those from Tonic Suite [10] (right).

Generally, the NLP applications have a lower number of operations when compared to their most similar counterparts in Tonic Suite. SA-LSTM is most similar in its network architecture as the three NLP applications in Tonic Suite (POS, CHK, NER) and have the lowest number of operations from the applications studied. NAMAS is the most similar to the ASR workload of Tonic Suite in that it has large fully connected layers that execute for each NN invocation and has the highest number of operations executed. SA-CNN is a CNN that is most similar to IMC and FACE and has almost two orders of magnitude less operations than the Tonic Suite applica-

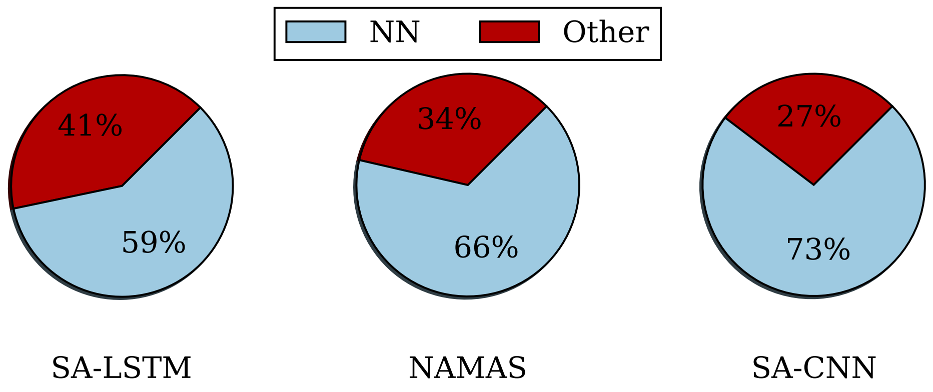


Figure 5.5: NLP Cycle Breakdowns

tions. On average, the NLP applications take 3.6ms to execute on the GPU where the SA-LSTM invocations to the NN are in the sub-millisecond range. Conversely, the Tonic Suite applications have an average execution time on the GPU of more than 10ms. While NAMAS stands apart as an application that has the highest latency on the GPU across all the applications studied, as we saw in the previous section its dependent and iterative computational pattern still makes it drastically different than the Tonic Suite applications.

From this we can take away that, as expected from the nature of the input and computation, NLP applications have small per NN invocation latencies making it more difficult to offload a large portion of work to an accelerator for processing.

5.2.3 Cycle Breakdown

We next look at the breakdown of cycles spent doing NN computation versus the rest of the application. As shown in the previous chapter, NN computation is amenable to GPU acceleration so we investigate portions of the workload that we can potentially accelerate. Figure 5.5 shows the breakdown of the execution of all three applications where the NN executes on the GPU and rest of the applications executes on the CPU. 60% of the cycles in SA-LSTM are consumed by the NN portion while the rest is outside the NN. SA-LSTM has a large preprocessing step where the input is sent to a parser that generates a constituency tree, representing the semantic relationship of the words in the sentence. The application makes multiple calls to the NN and requires the result of the previous NN call before processing the next input. After each leaf node is processed, there is also an NN call that *composes* the output of two leaf nodes. Simply put, the application is extracting useful information from each word, combining the two results into a vector representation, and using it as input to the next NN. This ensures that information is propagated up the tree. NAMAS consumes 66% of its cycles inside the NN. The output of a single NN invocation in the application is a list of potential next words in the summary and a probability associated with each. A Viterbi search is applied before the next invocation of the NN to prune the list of candidate words that could make up the summary. SA-CNN consumes the most cycles inside the CNN. This is expected because after a preprocessing step of padding the input and generating the vector representation of all the words, this is used as the input to a CNN with multiple layers whose output is the resulting sentiment. As a result, there is no iterative computation.

Given the iterative nature of the workload and the smaller per NN computation, these breakdowns are expected since the workload either consumes cycles traversing the parse tree (SA-LSTM) or processing the intermediate results of the NNs output (NAMAS). This differs significantly from the computation breakdown seen in Section 4.3 where on average across all 7 applications over 80% of the computation is spent inside the NN. In fact, for the image workloads the NN portion consumes 99% of the cycles (minimal pre- and post-processing).

From this characterization, we can conclude that NLP applications have three distinct characteristics: 1) they have input dependent and variable NN computation, 2) the compute per NN call is relatively small, and 3) they spend a large fraction of their execution time outside the NN, iteratively calling the NN engine. Next, we will investigate the behavior of these applications and their characteristics when deployed using state-of-the-art systems for DNN based applications.

5.3 Applicability of the Current State-of-the-art

In this section, we investigate how current techniques of characterizing DNN based applications apply and show the shortcomings of using the DjiNN web service with the NLP applications.

5.3.1 Batching to Increase Occupancy

Prior work uses occupancy and batching as a way to quantify how effectively the GPU is utilized and increase system throughput. We investigate applying the same principles to the NLP applications studied and juxtapose the DjiNN applications for comparison. Figure 5.6 shows the DjiNN applications juxtaposed with the three applications studied in this chapter (SA-LSTM, NAMAS, SA-CNN). The y-axis is the throughput gain achieved by each application at a batch size of 4 on the GPU normalized to a batch size of 1 (no batching). The x-axis is the occupancy of each application. Occupancy is a metric to quantify how effectively the resources of the GPU are being utilized by the application. Specifically, it is the ratio between the number of active warps and the theoretical number of warps this application could spawn on the device. The occupancy is collected using the NVIDIA Profiler [9] and it is weighted by each kernel's execution time to calculate the average performance of the entire application.

A few interesting insights can be drawn from this graph. First looking at the occupancy, the NLP applications all exhibit relatively low occupancy (at a batch size of 4) when compared to the DjiNN applications. Naturally, they are more similar to the cluster of the NLP applications that have low occupancy from DjiNN. However,

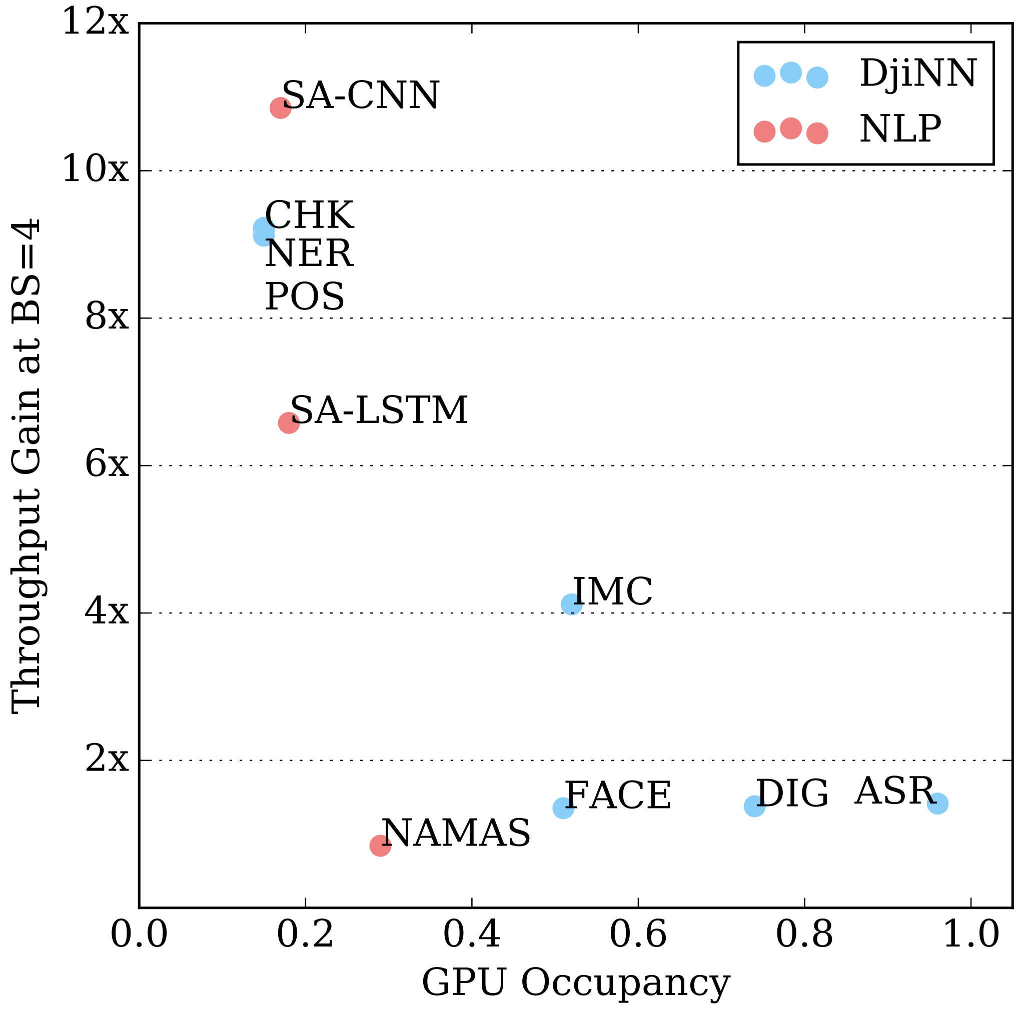


Figure 5.6: Occupancy and Throughput Gain using DjiNN Batching

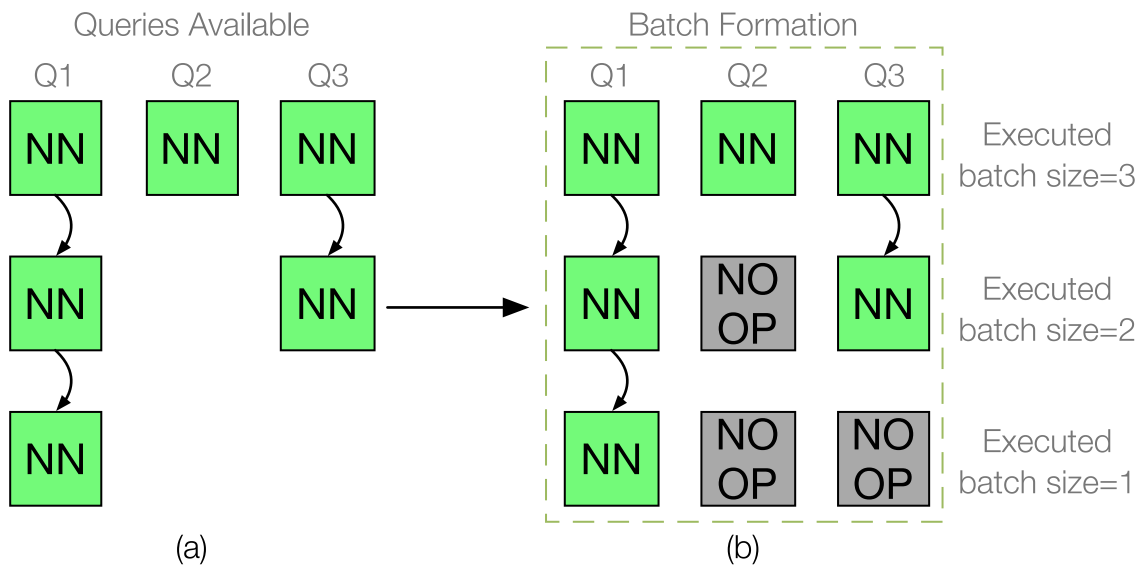


Figure 5.7: Padding to Batch

this graph also tells us that the NLP applications don't benefit as much from larger batch sizes as the DjiNN applications do (correlates with the low occupancy of the application at that batch size). NAMAS and SA-LSTM have low occupancy and relatively low throughput gains, when compared to the DjiNN applications.

5.3.2 Applying DjiNN Style Batching

Current systems providing a high throughput DNN service, namely DjiNN (detailed in Chapter IV), rely on a fixed DNN topology to batch inputs together into a larger matrix to execute on the GPU.

Padding to Batch The dynamic structure of the DNNs present in these NLP applications poses a significant challenge for DjiNN as the system currently assumes batches are formed application-side and the batches are perfectly formed. Figure 5.7a shows 3 queries incoming to the DjiNN service with variable length inputs (boxes show different number of NN invocations for each query). For DjiNN to be able to process these queries, it would need to pad the queries to the longest query of the batch before

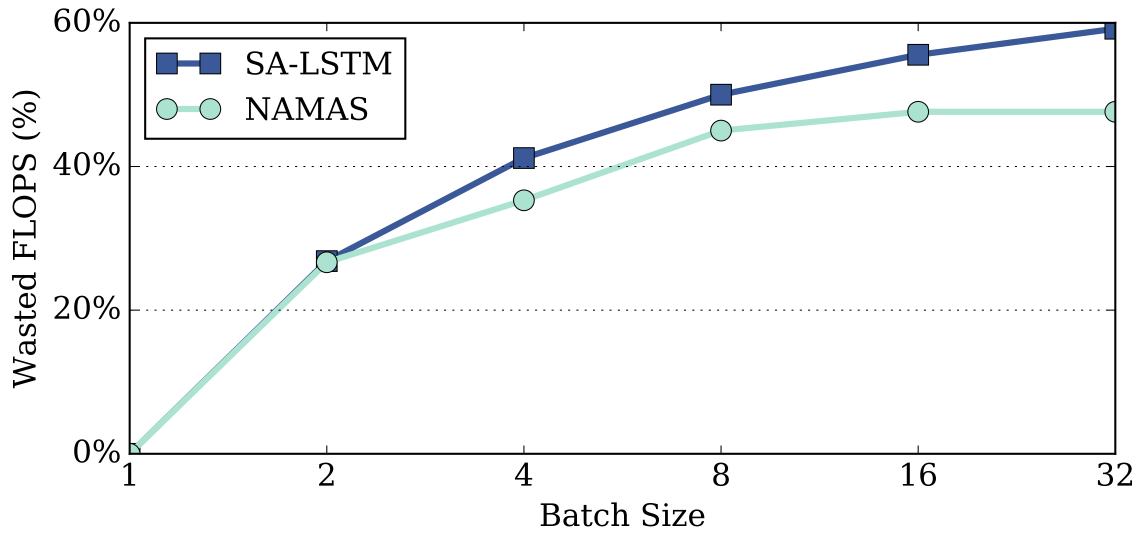


Figure 5.8: Percentage of FLOPS Wasted from Padding

being sent to the DjiNN web service (Figure 5.7b). DjiNN would execute the batch computation in lockstep since there is a sequential dependency of the NNs within a single query but NNs of independent queries can execute in the same batch. As Figure 5.7b shows as the queries execute, less meaningful computation is executed.

Wasting Compute Figure 5.8 shows the amount of computation wasted as the batch sizes increases. We use a trace of randomly generated queries that have variable input lengths for each application. As soon as there are enough queries to form a batch, all queries will be padded to the longest batch. As batch size increases, the range of query lengths within a single batch increases meaning more queries must be padded. At batch size of 32, up to 60% of the computation is unnecessary computation, significantly wasting computation on the GPU. At a certain batch size, the computation wasted plateaus because the dataset does not have infinitely long inputs to continue illustrating the problem.

The previous two sections show that GPU occupancy is not sufficient to derive throughput gain from batching and DjiNN's batching technique is not suitable for these NLP applications. This strongly suggests we need to find a better indicator of the potential benefits that can be had from designing a system for these NLP applications that also encompasses the DjiNN (static) applications.

5.3.3 Taxonomizing Dynamic DNNs

The analysis in the previous two sections suggests we need a new way to assess what systems to use when designing systems for dynamically defined neural network computation. We showed that generally all DNN based applications can benefit from batching but it is not a good indicator of how well a system for DNN computation will perform for a given application. In this section, we propose a new taxonomy that uses the characteristics of the NLP applications to differentiate them compared to the statically defined networks previously studied.

Figure 5.9 presents the three NLP applications and the seven DjiNN applications. On the x-axis is the occupancy at a batch size of 1 and the y-axis is the coefficient of variation of NN computation for each of the applications. This is a metric used to quantify variability in the NN computation a given application calculated as the standard deviation of the length of the input dataset over the mean.

This graph exposes two clusters of applications. The first is the set of applications that are amenable to DjiNN batching, these are the applications that have low variability in their NN computation. It is expected that SA-CNN would fall into this category because it does not have variability in its computation (all queries are padded to a fixed length). The second cluster, SA-LSTM and NAMAS, have high variability in their computation and are clustered relatively close to each other.

As our experimental results will show, the higher the variability in the computation (highest is SA-LSTM), the more an application can benefit from a precisely designed system to address the challenges in providing high throughput for this class of applications. We have now shown that two applications sit apart and are not amenable to current systems for DNN processing. We next design a system to ad-

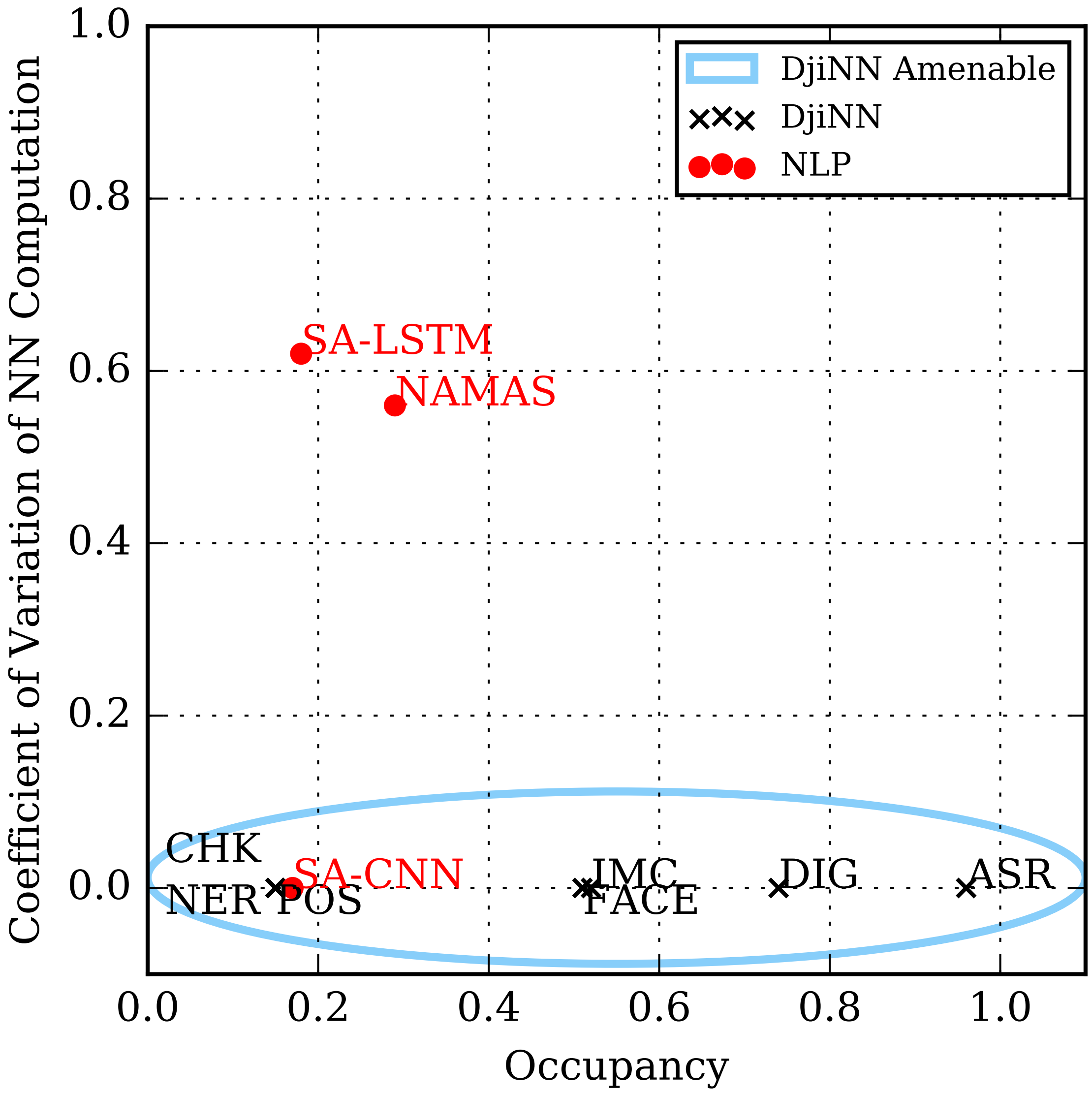


Figure 5.9: NN Application Taxonomy

dress the challenges in increasing system performance for these applications that is also applicable to the DjiNN amenable applications.

5.4 System Design for NLP

To investigate the design of a system specialized for dynamic NLP applications, we design an infrastructure to support *fine-grained cross-input batching*, a novel technique to address the new challenges emergent of large scale system design for these applications. We first outline the requirements of such a system and describe the implementation addressing each of the requirements.

5.4.1 Requirements

NLP applications have three core characteristics: they have dependent NN calls rendering intra-query batching impossible (Section 5.2.1), they have iterative and small NN computation making batching even more critical (Section 5.2.2), and they have intermediate non-NN processing making the computation iterative (Section 5.2.3). We design a runtime system that accounts for these characteristics while providing benefits for applications that use traditional batching techniques. We target the following objectives:

1. **Dependency and Input Length Agnostic Batching** - The system must be able to batch NN computation, irrespective of dependencies between NN calls and the variable length of the incoming queries.

2. **High Throughput Web Service** - The system must be able to deliver and sustain high throughput, accept queries over the network as a web service, and handle concurrent requests.

3. **Scalable Design** - The system must be designed in a manner that can scale

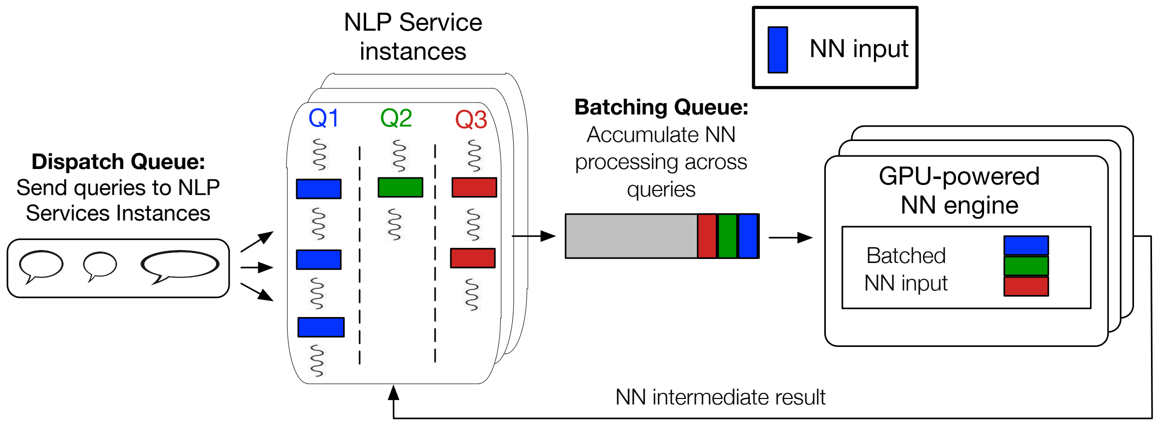

NLP Service instances
NN input
Q1
Q2
Q3
Batching Queue: Accumulate NN processing across queries
Dispatch Queue: Send queries to NLP Services Instances
GPU-powered NN engine
Batched NN input
NN intermediate result

Figure 5.10: System Design Overview

to fully utilize the underlying resources available.

5.4.2 System Design

Next, we detail the specific design of the system targeting each of the design goals previously mentioned.

Fine-Grained Cross-Input Batching (FGCIB) - We design fine-grained cross-input batching to allow NN batching across multiple queries. The system collects NN computation from multiple inflight queries to form a batch of NN computation that has independent NNs within a single batch. As shown in Figure 5.10, a thread processing the query will execute the CPU portion of the query until it meets NN computation (colored box in the diagram), at which point it will place the NN computation in a work queue, save the progress of that query, and suspend its execution. The thread is now free to service new incoming queries and repeat the process. At a given batch size, the NN engine will pull the NN computation from the queue, batch the input, execute the batched NN computation, and make a callback to the NLP service signaling the NLP service the queries can resume their execution.

Single Instance, Multiple Workers - The web service is designed using Thrift [107]. Thrift provides a flexible cross-language interface for designing web services capable of accepting concurrent requests over a network connection. To further increase the throughput, we decompose the system into asynchronous pipeline stages where the overall throughput of the system is dominated by the stage with the lowest throughput. As we will discuss in the evaluation, we use multiple workers pushing work to a common work queue to increase the throughput of the non-NN portion (labeled as "Other" in Figure 5.5) of the workload thereby making the NN processing stage the bottleneck.

Multiple Instances, Multiple Workers - We replicate the number of NLP service instances with a tunable number of workers to fully utilize the resources of the system. Each instance has its own work queue meaning there are now multiple instances of the DNN engine executing work on the GPU, further increasing the system utilization. The front-end dispatch queue round-robins queries to all the service instances.

5.5 Evaluation

We next evaluate fine-grained cross-input batching, documenting our observations and its efficacy in accelerating NLP applications with irregular computational structures. We first evaluate the system for a single NLP service instance then scale the system up consuming the full resources of the experimental platform.

5.5.1 Methodology

Our experimental setup uses a client-server architecture. Acting as the client is Treadmill [122], an open-source load generator deployed at Facebook, to send queries to our server over the network. The server uses the FGCIB technique described in Section 5.4 to process the queries. Queries are sent following an exponentially distributed inter-arrival rate, as prior research shows such a distribution accurately models production query arrival times [90]. The queries are dispatched from the front-end dispatch queue to each service instance on the server hosting the NLP services for processing.

The applications, available as open-source projects, are using highly optimized open-source libraries for their underlying NN processing. SA-LSTM and NAMAS are using Torch [46] and SA-CNN is using Theano [37]. For the CPU baseline, we link the libraries to Intel's MKL [18] while the GPU implementations are linked to cuBLAS for the matrix multiplication portion of the NN workloads. For all experiments, the machine learning model is pinned to the GPU so the only data transferred is the NN input. The platform used is a dual-socket Intel Xeon CPU E5-2630v3 running at 2.40GHz with 8-cores, 2-way HyperThreading and an NVIDIA Titan X GPU. One socket of the machine is dedicated to running the parser that is used by SA-LSTM to generate the tree before the LSTM is executed and one socket for the applications.

5.5.2 Single Service Instance

Throughput Improvement Figures 5.11, 5.12, and 5.13 show the throughput of using a single NLP service instance for SA-LSTM, NAMAS, and SA-CNN, respectively. On the x-axis of each graph, we show the CPU baseline, the GPU baseline, and FGCIB. The bars are the throughput achieved for each baseline as well as the throughput as we scale the batch size for FGCIB. We also plot the occupancy of the system to show, as we increase the batch size, the increased utilization of the GPU using the technique. In this experiment, FGCIB is configured to use a single instance and a single worker pushing work into the batching queue.

Interestingly, the CPU baseline for SA-LSTM (Figure 5.11) achieves higher throughput than the GPU baseline. From our characterization, this is expected because given the application makes multiple calls to the GPU, there is substantial overhead in kernel launch and data transfer (input to the NN) to the GPU. As previously mentioned, given the processing for a single NN is relatively small and the CPU is using a highly

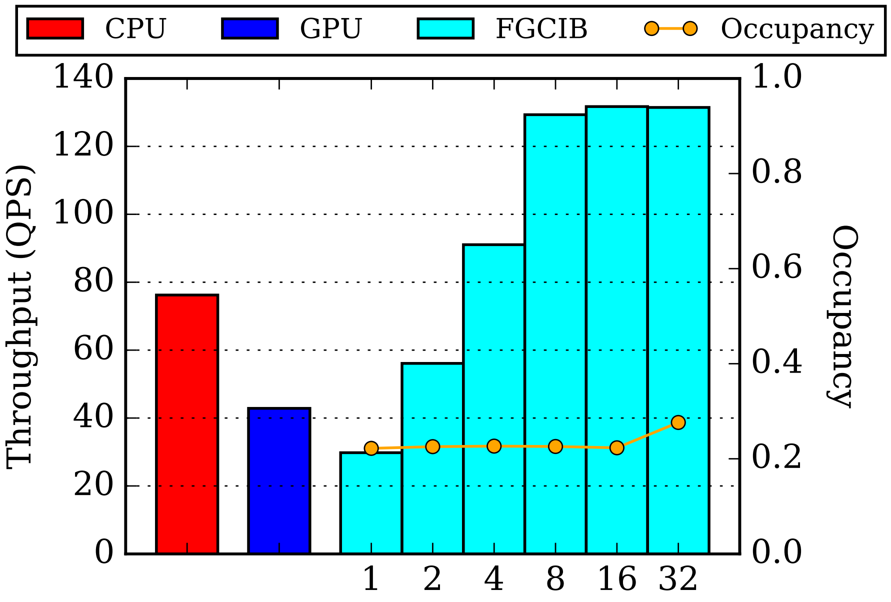


Figure 5.11: Throughput of FGCIB for SA-LSTM

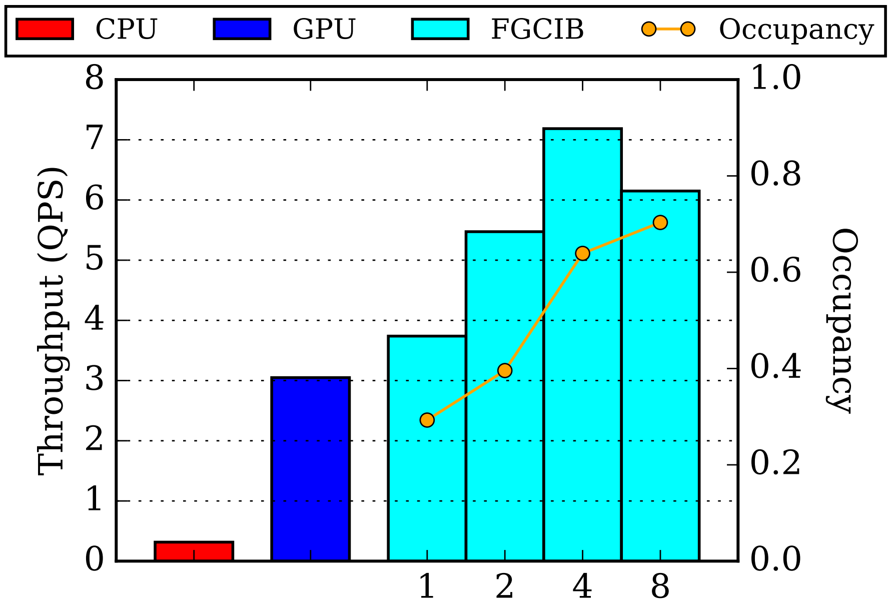


Figure 5.12: Throughput of FGCIB for NAMAS

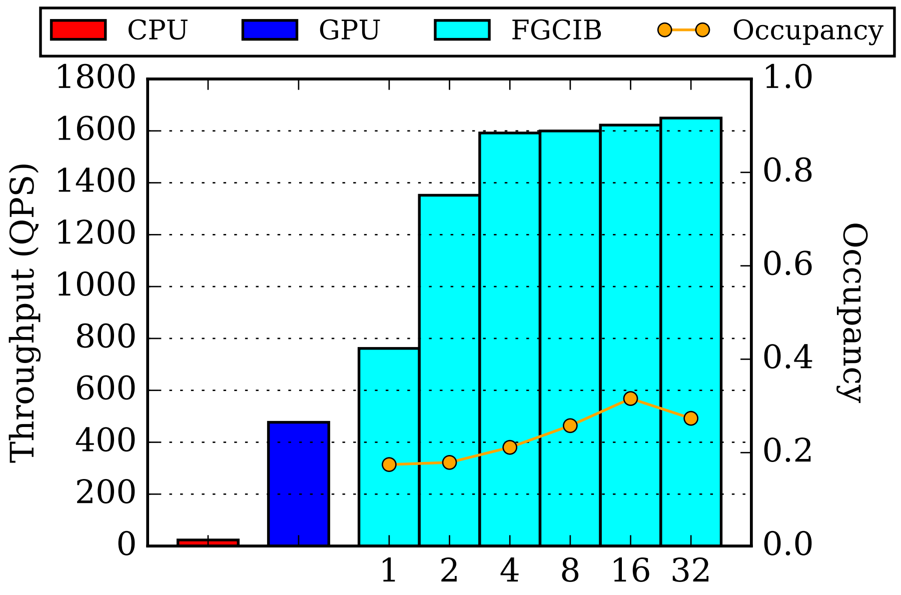


Figure 5.13: Throughput of FGCIB for SA-CNN

optimized matrix multiplication library, the CPU baseline achieves almost 2× the throughput of the GPU baseline. At a batch size of 1, FGCIB achieves lower throughput than the GPU baseline and achieves its lowest occupancy. This overhead is due to the extra logic and structures (work queues) required to support the technique. However as we increase the batch size, the system throughput increases and the overhead is amortized as FGCIB is able to fill its pipeline stages and more effectively use the GPU resources available, efficiently pushing batched work to the GPU. At a batch size of 4, the technique outperforms the CPU baseline. After a batch size of 8, the throughput plateaus because the bottleneck is now in feeding the GPU work. With more CPU workers pushing work to the queue, we can expect to further increase the throughput. Given the relatively small size of the NN computation, the occupancy only starts to increase at a relatively large batch size. For NAMAS (Figure 5.12), the GPU baseline outperforms the CPU baseline. At a batch size of 1, FGCIB achieves marginal benefits over the GPU baseline because of pipelining (more queries can be inflight). The highest throughput is achieved at a batch size of 4 before the occupancy and throughput plateau. There are two explanations for this: 1) the GPU's

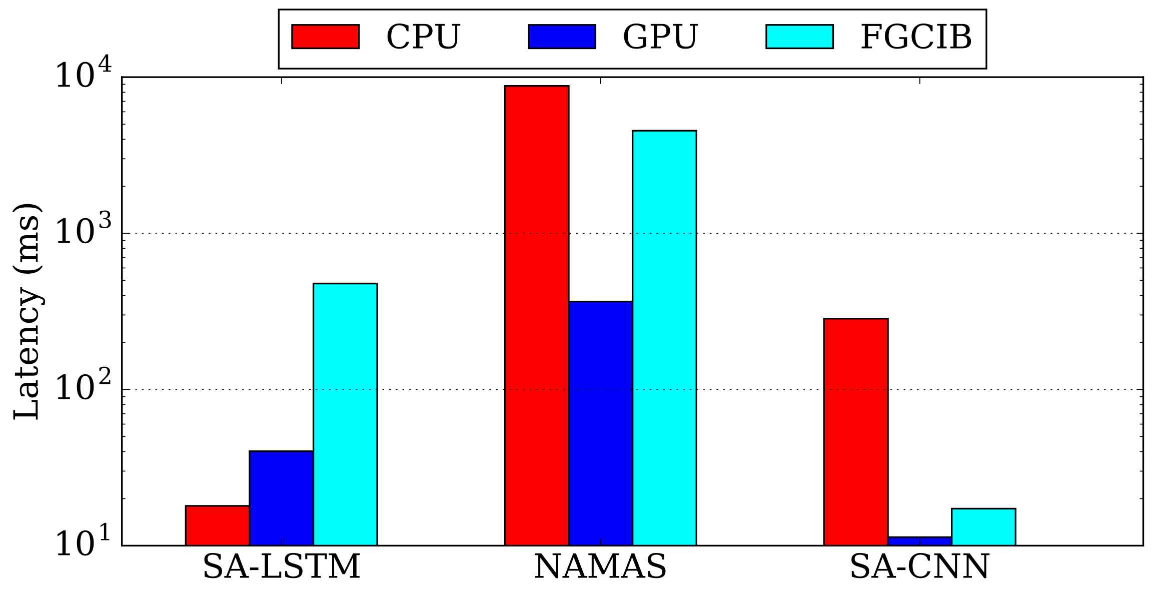


Figure 5.14: NLP Service Latency

occupancy at this point is near maximum so batching beyond 4 does not provide additional gains, and 2) the CPU is now the bottleneck since there is a substantial CPU portion required to process each NN call, involving a large data transfer from the GPU back to the host. Overall, NAMAS achieves over a 2× throughput benefit over the next best baseline. SA-CNN (Figure 5.13) achieves significant throughput benefits from using the system as well. As previously noted, there is no iterative or dependent computation for this workload so the benefits are entirely from batching inputs and using an asynchronous, pipelined system.

Single Query Latency Figure 5.14 shows the average latency of a single query for the CPU, GPU, and FGCIB. For each configuration, the latency is shown at the highest throughput achieved by that configuration with minimal queueing delay, meaning the latency collected is at low load for the baselines. For FGCIB, we select the smallest batch size yielding the highest throughput. This is a batch size of 16 for SA-LSTM, 4 for NAMAS, and 4 for SA-CNN. For two of the three applications, the latency of FGCIB is higher than both of the baselines. We have identified several optimizations that can be applied to improve the single query latency. For example, SA-LSTM is tree structured meaning there is leaf node parallelism allowing batching NN computation within a single query. This would reduce the latency of a single query given the tree would accomplish its computation earlier and would not be preempted in the batching queue by new queries. Persistent kernel launch would also reduce latency. Given we are launching the same kernel (layer) on the GPU at a fixed batch size multiple times (more so than static applications), amortizing the kernel launch time would reduce the latency of communicating with the GPU. The Titan X GPU also has two copy engines (host-to-device and device-to-host), which would allow FGCIB to overlap large data transfers incurred from batching multiple queries together. These copy engines allow for bidirectional, simultaneous communication. This would be beneficial for NAMAS which generates a large tree of states between each NN invocation, which translates to a large communication overhead.

All three applications achieve significant throughput benefits over to the CPU and GPU baselines. From this experiment, we can takeaway the following: 1) only at larger batch size does FGCIB begin to see benefits and amortize the overhead of pushing work to the GPU; 2) an asynchronous and pipelined design allows applications that do not have iterative and dependent computation to still benefit from the infrastructure put in place for FGCIB. The single query latency of FGCIB is one order of magnitude larger than the baseline system. We have identified several key components at the algorithmic and system level that can be optimized that would further reduce the latency.

The results also suggest the following: 1) using more workers could further improve system throughput, and 2) the occupancy of the GPU is still relatively low meaning there is still performance left on the table because the GPU is not fully loaded. We next investigate scaling up the system to fully utilize the experimental platform.

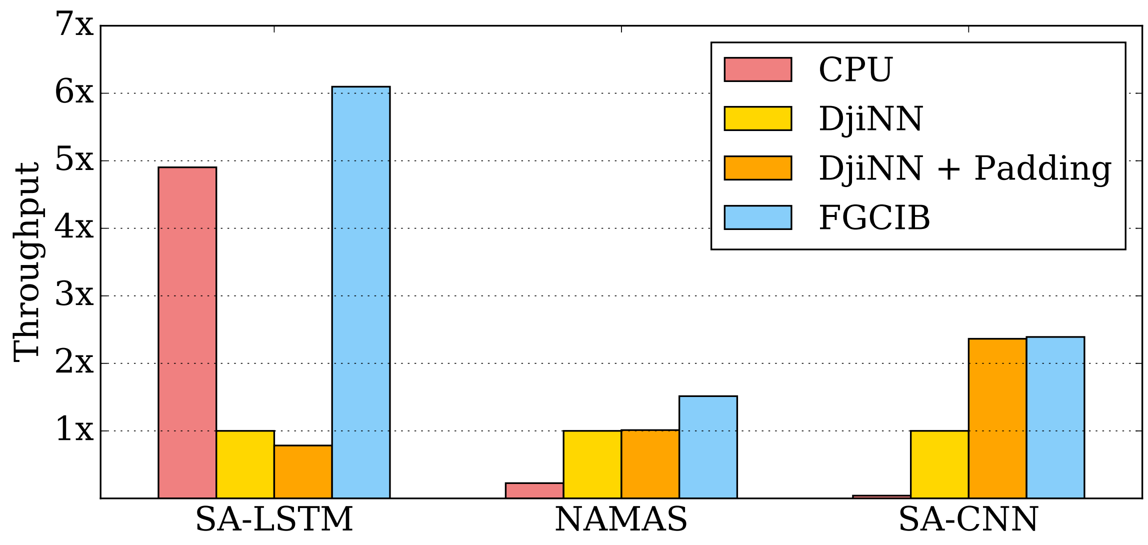


Figure 5.15: Throughput Improvement over CPU and DjiNN

5.5.3 Scaling to Multiple Service Instances

In the previous section, we investigated the throughput of a single instance of the NLP service running on our experimental platform. The results strongly suggested the need to scale up the system to increase utilization and further increase system throughput. In this section, we investigate scaling up the number of NLP service instances across the baselines and our system. We compare FGCIB to the DjiNN service infrastructure out of the box and to a batching baseline of padding all the queries in a batch to the length of the longest friend.

Figure 5.15 shows the results of this study across all three applications investigated. The throughput is normalized to the throughput achieved by the DjiNN service out of the box (multiple service instances, no batching). For each configuration, we empirically select the best configuration amongst service instances, number of workers, and batch size (when applicable). For the baselines (CPU, DjiNN, DjiNN + Padding), the number of workers is set to 1. For FGCIB and DjiNN + Padding, we select the smallest batch size with the highest throughput by sweeping the configurations available.

We empirically found the best configuration of FGCIB for SA-LSTM to be 4 instances, 4 workers, and a batch size of 32. SA-LSTM achieves the highest throughput improvements over the DjiNN baseline. This is because this workload has the strongest prevalence of the NLP characteristics across the three applications studied and so benefits the most from FGCIB (iterative, dependent, small NN computation). NAMAS achieves small benefits over the DjiNN baselines with the best configuration to be 1 instance, 1 worker, and a batch size of 4. Scaling the number of instances or workers did not further increase the throughput because the GPU portion is the bottleneck and already achieves relatively high occupancy at a batch size of 4. Finally, for SA-CNN the best configuration is 4 instances, 4 workers, and a batch size of 16. Given the queries are already padded for the application, the DjiNN + Padding configuration achieves very similar throughput to FGCIB. The marginal gains of FGCIB over that baseline can be attributed to the pipelining of the infrastructure.

Generally, SA-LSTM and NAMAS achieve significant throughput gains from the technique. NAMAS achieves a 6.7× throughput improvement over the CPU. On average, FGCIB achieves 7.8× throughput improvement over the CPU. As shown in Figure 5.15, our system achieves 2.8× higher throughput than the GPU baseline (maximum number of instances sharing the GPU). When compared to the state-of-the-art acceleration technique, the system on average achieves 2.3× higher throughput. Specifically, our system achieves on average 4.5× higher throughput for SA-LSTM and NAMAS while achieving slightly higher throughput for CNN. These results demonstrate that our system is more effective than state-of-the-art at handling deep learning applications with dynamically defined computation and performs slightly better (no worse) compared to the state-of-the-art for traditional statically-defined deep learning applications.

5.6 Summary

Natural Language Processing (NLP) applications represent the next, relatively unexplored set of applications that system architects need to rethink their systems for. The departure from statically defined NN based applications is inherent in the nature of the inputs to the NLP applications that require not only analysis of the individual words but also of their semantic position in the sentence. We identify three representative NLP applications that seemingly use the same algorithmic components (neural networks) but have drastically different computational characteristics. Through our in-depth characterization, we show that NLP applications have three main characteristics: 1) iterative and dependent NN computation, 2) the computation per NN call is small, and 3) a significant fraction of the time is spent outside the NN for intermediate processing. These characteristics lead current systems for high throughput DNN inference systems to perform suboptimally. We propose a design to address the limitations of current systems while also supporting statically defined NN workloads. Our system allows batching NN computation across queries to break the dependencies introduced within a query and allow queries of different length to be batched. We achieve on average 7.6× throughput improvements over an optimized CPU baseline and 2.8× over the current state-of-the-art GPU system.

CHAPTER VI

Conclusion and Future Directions

As cloud providers are building increasingly larger WSCs to accommodate the growing demand for a variety of web services, the type of applications running in these WSCs is changing. Traditional workloads, like web search and social networks, while still prevalent and widely used are beginning to share infrastructure with a new class of applications, namely *intelligent web services*. As this dissertation shows, these new web services are computationally very different from what is running in current WSCs. These intelligent web services are not only sharing resources in current WSCs, they are prompting new accelerator based designs given their large compute footprints. This dissertation investigates the design of an end-to-end application composed of three distinct intelligent web services to build an *intelligent personal assistant*. As intelligent web services began to mature in the last few years, DNNs became the algorithm of choice underlying the computation. This work then investigated providing a unified *DNN as a service* infrastructure arguing that a common, optimized engine can benefit a suite of intelligent web services of which large-scale applications like intelligent personal assistants are composed of. This thesis then concludes its investigation by focusing on *natural language processing* as the broad applicability and interest has spawned accelerated progress in this domain of intelligent web services and design a novel system for large-scale deployment of these web services.

6.1 Summary of Themes and Results

The design of end-to-end intelligent web service applications - Through our investigations in studying intelligent web services, we found it was critical to design applications composed of intelligent web services representative of those used in production systems by the large cloud service providers.

- We designed the first open-source end-to-end voice and vision based personal assistant based on investigative research to construct a system using the same algorithmic components deployed in production systems.
- We designed and open-sourced a DNN as a service infrastructure to study how DNN based intelligent web services can be deployed at scale.
- Alongside our open-source artifacts, we composed benchmark and application suites to evaluate the end-to-end systems, paving the way for studying intelligent web services in the future.
- We found it was critical to leverage load testing frameworks to study the rapidly evolving and changing landscape of intelligent web services to expose critical bottlenecks in current designs.

Accelerator rich WSCs are critical for intelligent web services - With representative end-to-end workloads in hand, we found that current CPU-only WSCs are inadequately equipped to sustain the demand needed of cloud infrastructures hosting intelligent web services. Acceleration on the path of a query using intelligent services is critical.

- We observed that the compute resources needed to sustain intelligent web service workloads is orders of magnitude higher than traditional datacenter workloads.

- We used the application suites at hand and ported them across spectrum of accelerator platforms, and found GPU and FPGA based design are the most promising moving forward.

- Focusing on DNN based intelligent web services, we aggressively optimized the DNN as a service infrastructure using batching and concurrent service instances on a system using 8 server grade GPUs.

- As the landscape of intelligent web services evolves, we developed *fine-grained cross-input batching* targeting dynamically defined neural network architectures.

Accelerator based WSC show significant TCO improvements - Acceleration on the path of a query shows significant promise and after performing a TCO analysis, we saw significant cost reductions can be had across different designs and optimization targets.

- Accelerator rich WSCs hosting intelligent web services showed significant TCO benefits over current designs when equipped with GPUs and FPGAs.

- Not all deep learning applications achieved performance benefits equally which means the workload composition informs the design choices of the WSCs.

- Addressing PCIe bandwidth bottlenecks for certain DNN based applications allows system designers to fully unlock the potential of a GPU rich WSC hosting DNN based intelligent web services.

6.2 Future Directions

The investigation of intelligent web services and their impact on the future of WSCs has just begun. This dissertation, while answering a number of questions in this space, also incites a set of new questions spurring exciting research in this direction. This section outlines a few of those directions.

Intelligent Personal Assistant Design

This dissertation is the first to design an end-to-end intelligent personal assistant the research community can use and further develop. Sirius has evolved from a research project into a platform allowing researchers, developers, and industry professionals to investigate how best to design an end-to-end application leveraging a series of intelligent web services on the path of a query. The idea of a platform to compose intelligent services into a pipeline is an exciting future direction as it opens multiple research directions, to name a few: (i) comparing the accuracy of an intelligent service in isolation versus in a larger pipeline and how errors propagate, (ii) allows building large scale applications that can scale beyond a single server, and (iii) proving out new algorithms given a target application.

Single Service Intelligent Application

The current working assumption is that end-to-end applications that require intelligence on the path of a query leverage a multitude of intelligent web services accomplishing disparate tasks along the way. This has informed the design until now where designers are able to decompose the task into microservices, easily spreading the computation across multiple machines. Recent advances in the deep learning community have shown promise in designing DNNs capable of accomplishing larger scale tasks within a single DNN outperforming the accuracy of designs using multiple DNNs for the same task (especially true in the NLP domain). As this evolves, this may pose a set of new challenges as the microservice architecture may fall out of favor for designs of large networks that may not fit in the compute budget of a single server.

Spreading Computation beyond the Datacenter

While this dissertation focuses on designing and optimizing intelligent web services in datacenter infrastructures, today's mobile devices are becoming powerful enough to share or even accomplish the compute required on the path of an intelligent query. Cloud providers can leverage the cycles in our pockets or homes further increasing the efficiency of their datacenters, delivering even lower latency for end users, or even increasing the scope of use-cases the application can accomplish. The task of intelligently partitioning work between mobile and cloud is an interesting one, fraught with challenges in data transmission, network variability, and security and privacy implications.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Ai is transforming google search. the rest of the web is next. https://www.wired.com/2016/02/ai-is-changing-the-technology-behind-google-searches.

[2] Amazon echo. www.amazon.com/echo.

[3] Amazon echo can now order your pizza. https://techcrunch.com/2016/02/03/amazon-echo-can-now-order-your-pizza/.

[4] Apache nutch. http://nutch.apache.org.

[5] Apple ios 10 uses ai to help you find photos and type faster. https://www.engadget.com/2016/06/13/ios-10-ai.

[6] Apple watch. www.apple.com/watch/.

[7] Apple's Siri. https://www.apple.com/ios/siri/.

[8] Clipped summarizes anything into bullet points and infographics through the power of ai. http://clipped.me/. Accessed: 2016-11-18.

[9] Cuda toolkit documentation. http://docs.nvidia.com/cuda/profiler-users-guide/.

[10] DjiNN and Tonic: DNN as a Service. http://djinn.clarity-lab.org.

[11] Duc-2003. http://duc.nist.gov/data.html.

[12] Facebook's quest to build an artificial brain depends on this guy. www.wired.com/2014/08/deep-learning-yann-lecun.

[13] Flipbord's approach to automatic summarization. http://engineering.flipboard.com/2014/10/summarization/. Accessed: 2016-11-18.

[14] Google home: a speaker to finally take on the amazon echo. http://www.theverge.com/2016/5/18/11688376/google-home-speaker-announced-virtual-assistant-io-2016.

[15] Google's Google Now. http://www.google.com/landing/now/.

[16] Google's new artificial intelligence maps the london underground. http://www.sciencemag.org/news/sifter/google-s-new-artificial-intelligence-maps-london-underground.

[17] Inside the artificial brain that's remaking the google empire. www.wired.com/2014/07/google_brain.

[18] Intel math kernel library. https://software.intel.com/en-us/intel-mkl.

[19] Intel vtune. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[20] Kooaba, Inc. http://www.vision.ee.ethz.ch/ surf/download.html.

[21] Microsoft corp to challenge apple inc with siri alternative: More intelligent and fast enough! www.dazeinfo.com/2013/06/18/microsoft-corp-to-challenge-apple-inc-with-siri-alternative/-more-intelligent-and-fast-enough.

[22] Microsoft to take on amazon echo, google home with home hub and cortana. http://www.phonearena.com/news/Microsoft-to-take-on-Amazon-Echo-Google-Home-with-Home-Hub-and-Cortana_id88632.

[23] Microsoft's Cortana. http://www.windowsphone.com/en-us/features-8-1.

[24] Movie review data. https://www.cs.cornell.edu/people/pabo/movie-review-data/.

[25] Multi-process service. https://docs.nvidia.com/deploy.

[26] Nvidia visual profiler. https://developer.nvidia.com/NVIDIA-visual-profiler.

[27] Qualcomm Acquires Kooaba Visual Recognition Company. http://mobilemarketingmagazine.com/qualcomm-acquires-kooaba-visual-recognition-company/.

[28] Sirius: An Open End-to-End Voice and Vision Personal Assistant. http://sirius.clarity-lab.org.

[29] SLRE: Super Light Regular Expression Library. http://cesanta.com/.

[30] Thinkmate high performance computing. http://www.thinkmate.com/system/rax-xf2-1130v3-sh.

[31] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[32] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: an introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 2013.

[33] L. A. Barroso, J. Clidaras, and U. Holzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. 2013.

[34] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio. Theano: new features and speed improvements. *Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop*, 2012.

[35] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *Computer Vision–ECCV 2006*, pages 404–417. Springer, 2006.

[36] B. C. Becker and E. G. Ortiz. Evaluating open-universe face identification on the web. In *Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2013.

[37] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, pages 1–7, 2010.

[38] D. Bouris, A. Nikitakis, and I. Papaefstathiou. Fast and Efficient FPGA-Based Feature Detection Employing the SURF Algorithm. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 3–10, Washington, DC, USA, 2010. IEEE Computer Society.

[39] G. Bradski. *Dr. Dobb's Journal of Software Tools*, 2000.

[40] V. R. Chandrasekhar, D. M. Chen, S. S. Tsai, N.-M. Cheung, H. Chen, G. Takacs, Y. Reznik, R. Vedantham, R. Grzeszczuk, J. Bach, and B. Girod. The stanford mobile visual search data set. In *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, MMSys '11, pages 117–122, New York, NY, USA, 2011. ACM.

[41] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[42] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 609–622, Washington, DC, USA, 2014. IEEE Computer Society.

[43] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman. Project adam: building an efficient and scalable deep learning training system. In *Operating Systems Design and Implementation(OSDI)*, 2014.

[44] J. Chong, E. Gonina, and K. Keutzer. Efficient automatic speech recognition on the gpu. *Chapter in GPU Computing Gems Emerald Edition, Morgan Kaufmann*, 1, 2011.

[45] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *International Conference on Machine Learning(ICML)*, 2013.

[46] R. Collobert, S. Bengio, and J. Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[47] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, 2011.

[48] G. E. Dahl, D. Yu, L. Deng, and A. Acero. Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(1):30–42, 2012.

[49] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. A. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[50] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition (CVPR)*, 2009.

[51] T. H. Dinh, D. Q. Vu, V.-D. Ngo, N. P. Ngoc, and V. T. Truong. High throughput fpga architecture for corner detection in traffic images. In *Communications and Electronics (ICCE), 2014 IEEE Fifth International Conference on*, pages 297–302. IEEE, 2014.

[52] P. R. Dixon, T. Oonishi, and S. Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Comput. Speech Lang.*, 23(4):510–526, Oct. 2009.

[53] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 449–460, Washington, DC, USA, 2012. IEEE Computer Society.

[54] C. Farabet, Y. LeCun, K. Kavukcuoglu, E. Culurciello, B. Martini, P. Akselrod, and S. Talay. Large-scale FPGA-based convolutional networks. *Machine Learning on Very Large Data Sets*, 2011.

[55] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. In *Proceedings*

of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 37–48, New York, NY, USA, 2012. ACM.

[56] D. Ferrucci, E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. A. Kalyanpur, A. Lally, J. W. Murdock, E. Nyberg, J. Prager, N. Schlaefer, and C. Welty. Building Watson: An Overview of the DeepQA Project — Ferrucci — AI Magazine. *AI MAGAZINE*, 31(3):59–79, Sept. 2010.

[57] G. D. Forney Jr. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, 1973.

[58] I. J. Goodfellow, D. Warde-Farley, P. Lamblin, V. Dumoulin, M. Mirza, R. Pascanu, J. Bergstra, F. Bastien, and Y. Bengio. Pylearn2: a machine learning research library. *arXiv preprint arXiv:1308.4214*, 2013.

[59] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[60] J. Hauswald, T. Manville, Q. Zheng, R. Dreslinski, C. Chakrabarti, and T. Mudge. A hybrid approach to offloading mobile image classification. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 8375–8379. IEEE, 2014.

[61] M. A. Hearst. 'Natural' Search User Interfaces. *Commun. ACM*, 54(11):60–67, Nov. 2011.

[62] G. Hinton, L. Deng, D. Yu, G. Dahl, A. rahman Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. Sainath, and B. Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[63] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[64] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski. Adrenaline: Pinpointing and reigning in tail queries with quick voltage boosting. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '15, Washington, DC, USA, 2015. IEEE Computer Society.

[65] X. Huang, J. Baker, and R. Reddy. A historical perspective of speech recognition. *Commun. ACM*, 2014.

[66] X. Huang, J. Baker, and R. Reddy. A historical perspective of speech recognition. *Commun. ACM*, 57(1):94–103, Jan. 2014.

[67] D. Huggins-Daines, M. Kumar, A. Chan, A. W. Black, M. Ravishankar, and A. I. Rudnicky. Pocketsphinx: A free, real-time continuous speech recognition system for hand-held devices. In *Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on*, volume 1, pages I–I. IEEE, 2006.

[68] R. Iyer, S. Srinivasan, O. Tickoo, Z. Fang, R. Illikkal, S. Zhang, V. Chadha, P. M. S. Jr., and S. E. Lee. Cogniserve: Heterogeneous server architecture for large-scale recognition. *IEEE Micro*, 31(3):20–31, 2011.

[69] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[70] A. Kannan, K. Kurach, S. Ravi, T. Kaufmann, A. Tomkins, B. Miklos, G. Corrado, L. Lukacs, M. Ganea, P. Young, et al. Smart reply automated response suggestion for email. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*, volume 36, pages 495–503, 2016.

[71] J. Kim, J. Chong, and I. R. Lane. Efficient On-The-Fly Hypothesis Rescoring in a Hybrid GPU/CPU-based Large Vocabulary Continuous Speech Recognition Engine. In *INTERSPEECH*. ISCA, 2012.

[72] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[73] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 468–479, New York, NY, USA, 2013. ACM.

[74] R. Krishna, S. Mahlke, and T. Austin. Architectural optimizations for low-power, real-time speech recognition. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 220–231, New York, NY, USA, 2003. ACM.

[75] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[76] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar. Next generation intel® micro-architecture (nehalem) clocking architecture. In *VLSI Circuits, 2008 IEEE Symposium on*, pages 62–63. IEEE, 2008.

[77] J. Lafferty, A. McCallum, and F. C. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[78] M. Laurenzano, Y. Zhang, L. Tang, and J. Mars. Protean code: Achieving near-free online code transformations for warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO-47, New York, NY, USA, 2014. ACM.

[79] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[80] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: Designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 36–47, New York, NY, USA, 2013. ACM.

[81] E. C. Lin, K. Yu, R. A. Rutenbar, and T. Chen. A 1000-word Vocabulary, Speaker-independent, Continuous Live-mode Speech Recognizer Implemented in a Single FPGA. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, FPGA '07, pages 60–68, New York, NY, USA, 2007. ACM.

[82] B. Liu. Sentiment analysis and opinion mining. *Synthesis lectures on human language technologies*, 5(1):1–167, 2012.

[83] D. Liu, T. Chen, S. Liu, J. Zhou, S. Zhou, O. Teman, X. Feng, X. Zhou, and Y. Chen. Pudiannao: A polyvalent machine learning accelerator. In *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2015.

[84] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment(PVLDB)*, 2012.

[85] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu. Designing a programmable wire-speed regular-expression matching accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 461–472, Washington, DC, USA, 2012. IEEE Computer Society.

[86] J. Mars and L. Tang. Whare-map: Heterogeneity in homogeneous warehouse-scale computers. In *ISCA '13: Proceedings of the 40th annual International Symposium on Computer Architecture*. IEEE/ACM, 2013.

[87] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[88] J. Mars, L. Tang, K. Skadron, M. L. Soffa, and R. Hundt. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro*, 32(3):88–99, May 2012.

[89] B. Mathew, A. Davis, and Z. Fang. A low-power accelerator for the sphinx 3 speech recognition system. In *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '03, pages 210–219, New York, NY, USA, 2003. ACM.

[90] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[91] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[92] N. Okazaki. CRFsuite: a fast implementation of Conditional Random Fields (CRFs), 2007. http://www.chokkan.org/software/crfsuite/.

[93] V. Petrucci, M. A. Laurenzano, Y. Zhang, J. Doherty, D. Mosse, J. Mars, and L. Tang. Octopus-man: Qos-driven task management for heterogeneous multicore in warehouse scale computers. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '15, Washington, DC, USA, 2015. IEEE Computer Society.

[94] N. Piatkowski. Linear-Chain CRF@GPU, 2011. http://sfb876.tu-dortmund.de/crfgpu/linear_crf_cuda.html.

[95] M. F. Porter. An algorithm for suffix stripping. *Program: electronic library and information systems*, 14(3):130–137, 1980.

[96] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, et al. The kaldi speech recognition toolkit. In *Proc. ASRU*, 2011.

[97] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *41st Annual International Symposium on Computer Architecture (ISCA)*, June 2014.

[98] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz. Convolution engine: balancing efficiency & flexibility in specialized computing. In *International Symposium on Computer Architecture(ISCA)*, 2013.

[99] A. Research. Wearable Computing Devices, Like Apple iWatch, Will Exceed 485 Million Annual Shipments by 2018. 2013. https://www.abiresearch.com/press/wearable-computing-devices-like-apples-iwatch-will.

[100] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski. Orb: an efficient alternative to sift or surf. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 2564–2571. IEEE, 2011.

[101] A. M. Rush, S. Chopra, and J. Weston. A neural attention model for abstractive sentence summarization. *arXiv preprint arXiv:1509.00685*, 2015.

[102] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge(ILSVRC), 2014.

[103] D. Rybach, S. Hahn, P. Lehnen, D. Nolden, M. Sundermeyer, Z. Tüske, S. Wiesler, R. Schlüter, and H. Ney. RASR - the RWTH Aachen University Open Source Speech Recognition Toolkit, 2011.

[104] F. Seide, G. Li, and D. Yu. Conversational speech transcription using context-dependent deep neural networks. In *Interspeech*, pages 437–440, 2011.

[105] M. G. Siegler. Apple's Massive New Data Center Set To Host Nuance Tech. http://techcrunch.com/2011/05/09/apple-nuance-data-center-deal/.

[106] A. Singh, N. Kumar, S. Gera, and A. Mittal. Achieving magnitude order improvement in porter stemmer algorithm over multi-core architecture. In *Informatics and Systems (INFOS), 2010 The 7th International Conference on*, pages 1–8, March 2010.

[107] M. Slee, A. Agarwal, and M. Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5(8), 2007.

[108] Y. Sun, Z. Wang, S. Huang, L. Wang, Y. Wang, R. Luo, and H. Yang. Accelerating frequent item counting with fpga. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 109–112, New York, NY, USA, 2014. ACM.

[109] S. Swaminathan, R. Tessier, D. Goeckel, and W. Burleson. A dynamically reconfigurable adaptive viterbi decoder. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 227–236, New York, NY, USA, 2002. ACM.

[110] K. S. Tai, R. Socher, and C. D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

[111] Y. Taigman, M. Yang, M. Ranzato, and L. Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Computer Vision and Pattern Recognition (CVPR)*, 2014.

[112] L. Tang, J. Mars, W. Wang, T. Dey, and M. L. Soffa. Reqos: Reactive static/dynamic compilation for qos in warehouse scale computers. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '13, pages 89–100, New York, NY, USA, 2013. ACM.

[113] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune. Optimizing google's warehouse scale computers: The numa experience. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '13, pages 188–197, Washington, DC, USA, 2013. IEEE Computer Society.

[114] O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *ACM SIGARCH Computer Architecture News*, 2012.

[115] E. F. Tjong Kim Sang and S. Buchholz. Introduction to the conll-2000 shared task: Chunking. In *Proceedings of the 2Nd Workshop on Learning Language in Logic and the 4th Conference on Computational Natural Language Learning - Volume 7*, ConLL '00, pages 127–132, Stroudsburg, PA, USA, 2000. Association for Computational Linguistics.

[116] O. Tckstrm, D. Das, S. Petrov, R. McDonald, and J. Nivre. Token and type constraints for cross-lingual part-of-speech tagging. *Transactions of the Association for Computational Linguistics*, 1:1–12, 2013.

[117] V. Vanhoucke, A. Senior, and M. Z. Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, 2011.

[118] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 265–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[119] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[120] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna. Compact Architecture for High-throughput Regular Expression Matching on FPGA. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 30–39, New York, NY, USA, 2008. ACM.

[121] Y. Zhang, M. Laurenzano, J. Mars, and L. Tang. Smite: Precise qos prediction on real system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, MICRO-47, New York, NY, USA, 2014. ACM.

[122] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference.

[123] M. Zhu, Y. Zhang, W. Chen, M. Zhang, and J. Zhu. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, pages 434–443, 2013.