AI Tax: The Hidden Cost of AI Data Center Applications

DANIEL RICHINS, The University of Texas at Austin, USA and Intel, USA
DHARMISHA DOSHI, MATTHEW BLACKMORE,
ASWATHY THULASEEDHARAN NAIR, NEHA PATHAPATI, ANKIT PATEL, and
BRAINARD DAGUMAN, Intel USA
DANIEL DOBRIJALOWSKI, Intel Poland
RAMESH ILLIKKAL, KEVIN LONG, and DAVID ZIMMERMAN, Intel, USA
VIJAY JANAPA REDDI, Harvard University, USA and The University of Texas at Austin, USA

Artificial intelligence and machine learning are experiencing widespread adoption in industry and academia. This has been driven by rapid advances in the applications and accuracy of AI through increasingly complex algorithms and models; this, in turn, has spurred research into specialized hardware AI accelerators. Given the rapid pace of advances, it is easy to forget that they are often developed and evaluated in a vacuum without considering the full application environment. This article emphasizes the need for a holistic, end-to-end analysis of artificial intelligence (AI) workloads and reveals the "AI tax." We deploy and characterize *Face Recognition* in an edge data center. The application is an AI-centric edge video analytics application built using popular open source infrastructure and machine learning (ML) tools. Despite using state-of-the-art AI and ML algorithms, the application relies heavily on pre- and post-processing code. As AI-centric applications benefit from the acceleration promised by accelerators, we find they impose stresses on the hardware and software infrastructure: storage and network bandwidth become major bottlenecks with increasing AI acceleration. By specializing for AI applications, we show that a purpose-built edge data center can be designed for the stresses of accelerated AI at 15% lower TCO than one derived from homogeneous servers and infrastructure.

CCS Concepts: • **Computing methodologies → Machine learning**; **Artificial intelligence**; • **Software and its engineering → Software performance**;

Additional Key Words and Phrases: AI tax, end-to-end AI application

This work extends a previous article published in the HPCA 2020 Industry Session: *Missing the Forest for the Trees: End-to-End AI Application Performance in Edge Data Centers* [70]. This article adds (1) a discussion on the need for end-to-end AI benchmarks; (2) additional data and explanation for the deployment choices made in *Face Recognition*; (3) examination of one of the causes of broker waiting time; and (4) AI tax and acceleration analysis of a second edge data center workload.

Authors' addresses: D. Richins, The University of Texas at Austin, and Intel; D. Doshi, M. Blackmore, A. T. Nair, N. Pathapati, A. Patel, B. Daguman, D. Dobrijalowski, R. Illikkal, K. Long, and D. Zimmerman, Intel, 1900 Prairie City Rd, Folsom, CA 95630, USA; V. Janapa Reddi, Harvard University and The University of Texas at Austin.
Authors current address: D. Richins, Maxwell Dworkin, 33 Oxford St, Cambridge, MA 02138; M. Blackmore, 2111 NE 25th Ave, Hillsboro, OR 97124, USA; D. Dobrijalowski, Jana z Kolna 11, Tryton Building, Intel, 80-864, Gdansk, Poland; V. Janapa Reddi, Maxwell Dworkin, 33 Oxford St, Cambridge, MA 02138.

0734-2071/2021/03-ART3 $15.00
https://doi.org/10.1145/3440689

ACM Reference format:
Daniel Richins, Dharmisha Doshi, Matthew Blackmore, Aswathy Thulaseedharan Nair, Neha Pathapati, Ankit Patel, Brainard Daguman, Daniel Dobrijalowski, Ramesh Illikkal, Kevin Long, David Zimmerman, and Vijay Janapa Reddi. 2021. AI Tax: The Hidden Cost of AI Data Center Applications. *ACM Trans. Comput. Syst.* 37, 1–4, Article 3 (March 2021), 32 pages.
https://doi.org/10.1145/3440689

1 INTRODUCTION

Artificial intelligence (AI), especially the field of machine learning (ML), is transforming the marketplace. Sparked by advances in computer system design, enterprises are leveraging AI in every possible manner to provide unprecedented new services to their end users, ranging from recommendation-based online shopping and personalized social network services to virtual personal assistants and better health care.

To enable ML, there has been a flurry of work at two extremes. At one extreme is the effort that focuses on hardware acceleration of ML kernels [11, 17, 18, 22, 43]. At the other extreme is the effort that focuses on engineering the system and its supporting infrastructure, such as the associated networking and storage. The former is essential for enabling microprocessor advancements, while the latter is essential for allowing cloud-scale deployment.

But recent years have seen a shift in the needs of the industry. While much research has been dedicated to maximizing and accelerating ML performance, recent industry perspectives have urged for a more holistic understanding of ML development and performance. Facebook, for example, has discussed some of the challenges it has faced running AI at scale and encouraged research on mitigating those challenges [54]. Instead of focusing solely on the AI kernel computation time, there is a need to look at the bigger picture. Enabling AI applications involves several stages: ingesting the data, pre-processing the data, offloading the data to an AI accelerator, waiting for data, post-processing the result, and so forth, all of which affect the requests' end-to-end latency and total system throughput.

At the same time, the industry is witnessing AI services migrate from warehouse-scale systems to smaller purpose-built data centers located at the edge, closer to end users [48]. These edge data centers complement existing cloud- or large-scale services by being physically closer to the data source, which enables faster responses to latency-sensitive or bandwidth-hungry application services [51]. There are also data sovereignty and regulatory compliance rules to safeguard data privacy that are addressed with edge data centers [52]. Moreover, many mid-size organizations find it more economical to invest in on-premise data centers that are purpose-built for executing a particular type of task [34]. So despite the continued growth in public cloud solutions, spending for edge data centers is predicted to increase [30].

In this work, we study the intersection of user-facing AI computing—the inference side of ML—and smaller, edge data centers to reveal the often overlooked "AI tax": the additional compute cycles, infrastructure, and latency required to support the AI at the application's heart. In the context of a data center, execution of a fully developed, deployment-ready AI-centric application relies on more than just AI algorithms. End-users' requests demand pre-processing to ready them for the pipeline; intermediate data must be communicated between stages, often over a network using custom protocols; the communication framework often has built-in data reliability safeguards that impose overheads on data movement; and each stage faces its own overhead for moving data. All of these components together add to the overhead of executing AI.

We study a full deployment of *Face Recognition*, an end-to-end video analytics AI-centric application at the edge. Our setup is an industry deployment of the Google FaceNet [71] architecture in an edge data center. Our application is entirely focused on the inference side of ML, where

it is exposed to end users and faces associated latency constraints. As an AI-centric application, *Face Recognition* is a good choice as it employs three distinct artificial intelligence algorithms, including two neural networks and a classification algorithm. Furthermore, it is representative of the reality of a considerable portion of AI and ML applications: many AI applications exist as streaming services, deployed in data centers, serving real-time needs of consumers. Coordinating the many activities required to transform raw data into useful, easily consumable conclusions requires the intricacies and nuances of any distributed application: networking equipment, storage devices, coordination, data durability, power distribution, cooling, communication protocols, data compression, and so forth [58].

We find that in today's edge data centers, already the communication framework can constitute over 33% on the latency of the application. *Face Recognition* is built on top of Apache Kafka [5], which is widely adopted both directly and as a fabric upon which advanced streaming frameworks are built [47, 59, 65, 66, 75, 77]. Kafka is also representative of alternative frameworks that utilize communication hot spots. The simplicity and impressive performance of Apache Kafka have established it as a common denominator for many industry-quality projects. Despite this, requests can spend substantial time passing through the framework.

Moreover, we show that as accelerator technologies advance and integrate into production environments, the supporting portions of the pipeline will soon supplant AI as the primary determinant of performance. We measure the implications of greater AI inference acceleration. Apache Kafka becomes increasingly stressed to move the vastly increased volume of data ingested by the application. Even at relatively low acceleration factors, the added stress will quickly overwhelm Kafka's current capabilities. We demonstrate that at a very modest 8× acceleration factor, Kafka overwhelms the capabilities of its underlying storage.

These findings present a unique opportunity on the compute research spectrum: rather than neglecting the execution context of AI and without moving into the realm of cloud compute where resources must be generic and homogeneous enough to handle all kinds of workloads, we show a proof-of-concept for the economic value of edge data centers. We demonstrate how a data center that is custom-built for the needs of a streaming AI workload can accommodate the anticipated requirements of accelerated AI without over-provisioning, thereby realizing an overall decrease in the total cost of ownership (TCO) in excess of 15% over a homogeneous edge data center.

Although our deep-dive analysis primarily focuses on edge video analytics with *Face Recognition* (as the poster child application), we also conduct a basic analysis of a second application, *Object Detection*, deployed similarly to *Face Recognition*, and show that it faces comparable AI tax challenges. In other words, our analyses and conclusions are not specific to one application; we further discuss how the underlying infrastructure of an end-to-end AI application will present mainly the same bottlenecks regardless of the AI application.

In summary, our main contributions and insights are as follows:

(1) Where much focus is devoted to tuning and accelerating AI inference to enable faster compute, we instead evaluate the larger **system-level implications of end-to-end AI applications and expose the AI tax**.
(2) We show that the **general-purpose CPU performance remains a significant determinant of overall request performance** because processing an end-user request requires more than just AI kernel computation.
(3) The **communication layer of an AI application imposes a large overhead** on the latency of processing.
(4) The **increased throughput from AI acceleration will overwhelm the communication substrate**.

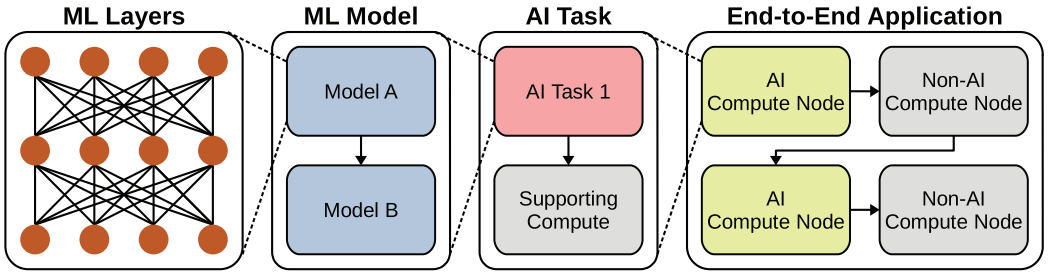


Fig. 1. Increasing levels of abstraction in understanding AI applications. At its most basic level, a machine learning AI application is computation of layers through common operations like matrix multiplication. Layers are used in combination to create an ML model. One or more models combine with supporting compute to compose AI tasks. And finally, AI tasks and additional supporting infrastructure is distributed in a data center to create a complete application.

(5) A **purpose-built data center can adapt to the upcoming challenges of accelerated AI at a lower TCO** than a generic, homogeneous data center.

There are, of course, innumerable ways to deploy a data center AI application. In this research, we focus on a scheme that concentrates communication in a few data brokers. We expect that our main takeaways will be applicable to any deployment that utilizes some sort of data broker. Namely, with the increasing data throughput of increasingly effective AI accelerators, the brokers will become a point of failure. The amount of data moving through the application can quickly overwhelm the capabilities of the brokers.

The proper solutions to the AI tax may reside in a customized edge data center, as we suggest, or they may depend on a different deployment scheme. In any case, our work clearly demonstrates the criticality of understanding AI applications from an end-to-end viewpoint. Without that perspective, we would have no comprehension of the AI tax or any reason to suspect that it would become the limiting factor to performance.

The remainder of this article is structured as follows. In Section 2, we motivate study of AI applications in a full, end-to-end, data center context. In Section 3, we introduce our primary application, *Face Recognition*, and explain its deployment and optimization in our edge data center. In Section 4, we elucidate the AI tax, characterizing the performance and limitations of the end-to-end AI application. In Section 5, we conduct a forward-looking analysis of *Face Recognition* under accelerated AI compute and identify significant impediments to improving performance. In Section 6, we show that the AI tax exposition is not unique to *Face Recognition* by similarly deploying and evaluating a second application. In Section 7, we show that an edge data center can be purpose-built to address the upcoming challenges of AI while reducing TCO. We then distinguish our work from prior art in Section 8 and conclude in Section 9.

2 AI IN EDGE DATA CENTERS

With the explosive growth of AI applications, much work has been done to try to characterize and understand them through numerous proposed benchmark suites. Ranging from device-level computations to complete applications, these benchmarking efforts cover a range of abstraction levels, each recognizing the need to understand AI applications from a variety of abstraction levels (Figure 1). In this section, we explore the range of abstractions and explain the need for the next level of abstraction: edge data center, end-to-end AI applications. In Section 2.1, we look at benchmarking efforts aimed at understanding ML layers. Section 2.2 looks at complete ML models built from potentially many layers of compute. Section 2.3 considers entire AI tasks composed of

potentially multiple ML models operating in tandem. Finally, Section 2.4 takes the next logical step and motivates the study of a complete AI application as it exists in an industry-quality edge data center.

2.1 ML Layers

At its heart, an AI application is a set of computations done on some piece of hardware, and, though this seems simplistic, it is critical to understand what computations compose the application and how they fit together (Figure 1, "ML Layers"). DeepBench [14] enables researchers to benchmark the primitive layers of ML models (matrix multiplies, convolutions, and recurrent operations) at the lowest level—on CPU and GPU using primitive ML libraries. It exists below any ML framework (such as TensorFlow [39] and Caffe [8]). Its goal is to elucidate the performance of the most common operations on various hardware devices.

Taken in isolation, though, DeepBench is incomplete. Basic operations like matrix multiplication are essential in ML, but in practice they occur only in certain model layers and within specific data access patterns that can vary from one ML framework to another. These details, available only in the next level of abstraction, give important context for the basic operations of benchmarks like DeepBench.

2.2 ML Model

The next level of abstraction is the ML model, built from fundamental operations and layers (Figure 1, "ML Model"). A complete model will typically operate on its input (such as an image) to produce a useful evaluation (such as identifying the objects in the image). Benchmarking complete models is critical because it determines precisely which operations are to be performed, how common each operation is, and the computational intensity of operating.

There are numerous benchmark suites that address this level of abstraction. AI Matrix [19, 79] introduces numerous complete ML models, covering such diverse application domains as image classification and neural machine translation, and also provides for synthetic models that are generated to mimic desired workload characteristics. MLMark, an EEMBC benchmark, is focused on benchmarking ML inference on embedded or edge devices [73]. AI-Benchmark [1] is designed specifically to target smartphone performance [55, 56]. AIIA-Benchmark targets accelerator devices and aims to provide a useful means of comparison.

While these benchmark suites are indispensable, they are also incomplete. ML models do not exist or execute in a vacuum but rely on supporting compute. In practice, this often means that, unlike these benchmarks, ML model execution cannot operate uninterrupted but must be supported by additional models or even non-ML compute.

2.3 AI Task

For ML to be useful, it needs the help of supporting compute, i.e., pre- and post-processing. Additionally, a model is often only one piece of a series of models working as a unit to produce a result. Together, these compose an AI task (Figure 1, "AI Task"). For example, before a model can operate on an input image, the image must be transformed into the proper size, with the proper color encoding, in the proper layout, to match the requirements of the model; the model output must similarly undergo transformation to be prepared for the next stage of processing or to be returned to the user. At this level of abstraction, the real behavior of an AI application starts to become apparent.

Without the pre- and post-processing steps, the AI kernel is basically useless. Even so, we are aware of no benchmarking suite that captures this more complete, context-aware, AI task-level view.

Table 1. Comparison of Various ML Benchmarks Showing How Much of the End-to-End AI Application is Captured by Each

Benchmark	ML Layers	ML Model	AI Task	Orchestration Software	Pre-Processing	Post-Processing	Network	Storage
DeepBench	✓							
AI Matrix	✓	✓						
MLMark	✓	✓						
AI-Benchmark	✓	✓						
AIIA-Benchmark	✓	✓						
DAWNBench	✓	✓						
MLPerf	✓	✓						
End-to-End	?	?	?	?	?	?	?	?

While most benchmarks thoroughly cover the ML layers and model space, a complete application is far larger than these computational kernels, and none of the existing benchmarks capture the whole picture.

While an AI task may represent a complete application, if it is executing on a single device, much of the AI compute in practice takes place as real-time services provided by industry players for end users. In such a scenario, an AI task is itself incomplete, as an AI task will be part of a larger application that spans multiple servers and interacts over the network both internally and with end users.

2.4 End-to-End Application

The highest level of abstraction is the data center level, where we finally see the end-to-end application (Figure 1, "End-to-End Application"). The various AI tasks are deployed to different nodes throughout the data center along with their supporting compute. Additionally, some nodes may be entirely dedicated to non-AI, supporting compute. *Of critical importance, at the data center level, the networking equipment, storage devices, communication protocols, data center management software, and application coordination software all become part of the AI application.*

The numerous past ML benchmarking efforts are invaluable for understanding the heart of AI applications; however, they all fall short of this highest level of abstraction. Table 1 illustrates, for a small sampling of benchmarks, how much of an AI application is *not captured* and *not understood* as a result of failing to rise to this level of abstraction. All the benchmarks we have mentioned in this section do a good job of benchmarking ML compute details, but they all leave gaping holes in the understanding of end-to-end application performance. DAWNBench [38] and MLPerf [27] are noteworthy for trying to introduce greater realism into benchmarking ML models; unfortunately, they do not go far enough. DAWNBench recognizes the importance of batch sizing [46], which is widely known and adopted in realistic deployments of AI applications to maximize performance. MLPerf takes this further and introduces a number of "scenarios" under which ML models can be evaluated. These scenarios incorporate the essential concept of latency-bounded throughput—maximizing throughput while honoring latency constraints. They try to mimic a setup that would exist in a real-world data center where AI applications are deployed [57].

Ultimately, however, even though these benchmarks acknowledge that ML models are executed in a server-like scenario, they sidestep the issue. MLPerf, for example, attempts to mimic a server setup by having requests arrive at random intervals according to a Poisson distribution, but this ignores the portion of the pipeline that actually provides and pre-processes the requests.

AIBench is an ambitious undertaking that recognizes the need for end-to-end benchmarks [49]. Designed to be easily extensible by building from a common framework, AIBench has implemented

two scenarios: e-commerce searching and online language translation. Both scenarios operate in a data center, utilize multiple stages of compute and inference coordinated by orchestration software, rely on network and storage, and utilize pre- and post-processing. However, as it stands, AIBench is insufficient to cover the breadth and depth of AI applications and has not undergone the extensive and holistic evaluation for which we argue in this article. For these reasons, we consider it more of a standalone application than a true benchmark.

To fully understand an AI application requires taking a holistic view at this highest level of abstraction. We are not aware of any effort to capture this complete understanding. While we do not present a new benchmark suite, in this work we undertake to present a thorough, forward-looking, end-to-end evaluation of a complete AI application.

3 END-TO-END VIDEO ANALYTICS APPLICATION

Given the importance of understanding the end-to-end AI application as deployed in a data center, we describe in this section the AI-centric *Face Recognition* application we developed and how we deployed it. It is based on Google's FaceNet algorithm [71]. However, *Face Recognition* is much more than just FaceNet—it is a full data center application. Going from the algorithm to the full workload requires algorithm partitioning, containerization, work coordination, and communication management. We explain how the logical flow of *Face Recognition* is transformed into a functional streaming data center application.

We chose a video analytics application for our studies of AI tax due to the rising importance of this domain. The global market for video analytics is expected to hit US$25 billion by 2026 due to rapid adoption of video technologies across industries such as retail, manufacturing, and smart cities [40]. Video analytics uses AI to provide cost-efficient business intelligence insights to its users. The domain is slated for deployment in edge data centers, as opposed to traditional cloud- or warehouse-scale systems due to latency constraints, network bandwidth, and privacy regulations.

In Section 3.1, we introduce and describe *Face Recognition*. Section 3.2 summarizes our edge data center setup. In Section 3.3, we explore different ways of deploying it to the data center nodes. In Section 3.4, we introduce the Apache Kafka framework, which we use to coordinate the application steps and manage communication between them. Section 3.5 explores the design space for individual containers while considering both latency and throughput constraints. In Section 3.6, we explain how *Face Recognition* is deployed at scale.

We detail the deployment of a second data center–level application in Section 6.

3.1 Video Analytics Pipeline

Video analytics is the automatic analysis of video data. For our analysis, we developed and deployed a video analytics application for edge usage called *Face Recognition* (Figure 2). Though it uses ML, this application is strictly user-facing, i.e., it uses inference rather than training.

Our implementation of *Face Recognition* relies heavily on artificial intelligence and ML algorithms implemented in TensorFlow [41]. Given a number of input video streams, the application parses the videos into individual frames, locates faces within the frames, and identifies each face as a certain individual. The video streams could represent a surveillance system's cameras [68], offline processing of recorded videos, a transaction\less shopping environment [2], or many other applications where multiple streams are concurrently being fed into the system.

The *Face Recognition* application consists of four primary processing stages (Figure 2) and is built from MT-CNNs (multi-task cascaded convolutional networks) [78] along with Google's FaceNet [16, 71]. Like many real-world use cases, the application involves multiple inferences per query.

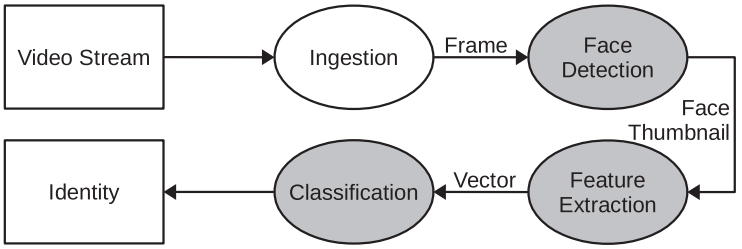


Fig. 2. Algorithmic flow of *Face Recognition*. A video stream enters *ingestion* for separation into individual frames. *Face detection* finds any faces within a frame and produces a thumbnail for each. *Feature extraction* generates identifying features for each face. Finally, *classification* finds a nearest match to known faces to produce an identity. The shaded ovals indicate the stages where AI compute takes place.

(1) **Ingestion** is a pre-processing stage that ingests a video stream and parses it into individual frames. This stage is critical as FaceNet cannot operate directly on video.
(2) **Face detection (AI)** relies on MT-CNNs to detect any faces within a frame without making any effort to identify them. It determines bounding boxes for and produces a `160x160` thumbnail of each face in a frame.
(3) **Feature extraction (AI)** is built using the Inception-Resnet [72] architecture and produces a 128-byte vector of essential features that describe each face.
(4) **Classification (AI)** compares the feature vector for a face against a set of known face vectors to find the best match by means of a support vector machine (SVM), yielding an identification.

Face Detection is designed as a pipelined streaming application—it ingests video streams at or near their native frame rate, injects them into the pipeline, and yields facial identities for frames after some delay. Even though the overall latency may exceed the time between adjacent frames in a video stream, because the application is pipelined, the throughput is at or near the native frame rate.

3.2 Edge Data Center

For our experiments, we utilize a small edge data center built from high-performance servers (see Table 2). We use over 2,200 processor cores spread across 40+ nodes to ensure that we have a realistic deployment whose characteristics can scale to larger setups. Each node is equipped with 56 physical cores spread across two sockets, 384 GB of RAM, high-speed local NVMe storage, and 100 Gbps Ethernet. The nodes are connected in a fat tree topology [60].

We rely on industry-standard, open source tools for our deployment. After dividing the application algorithm into steps, we deploy the different steps in lightweight Docker [12] containers. Depending on the resource requirements of a container, it may be deployed in isolation on a data center node or it may be deployed alongside other containers on the same node. The deployment of the various containers is managed using Kubernetes [32]. As the application steps are separated from each other, data has to be communicated between them; for this, we rely on Apache Kafka [5]. We explore these and other details in the remainder of this section.

3.3 Stage Count

As an application, *Face Recognition* separates its algorithmic steps into discrete stages that coordinate with one another while running independently on separate nodes to produce a legitimate data center application. Separating the algorithm into multiple coordinated steps allows different stages of the application to adapt to the speed and requirements of other stages.

Table 2. Server Details

Component	Details
CPU	2× Intel Xeon Platinum 8,176 [25]
Cores	*28*
Base Frequency	*2.10 GHz*
Max Turbo Frequency	*3.80 GHz*
SMT	*2-way*
LLC	*38.5 MB*
Memory	384 GB DDR4-2666 [24]
Storage	Intel SSD P4510
Read BW	*2.85 GB/s*
Write BW	*1.1 GB/s*
Read Latency	*77 us*
Write Latency	*18 us*
Network	Full duplex 100 Gbps Ethernet

Each server in our data center is well-equipped, using leading-edge technology. Our nodes are powered by Intel Xeon Platinum 8,176 or comparable CPUs.

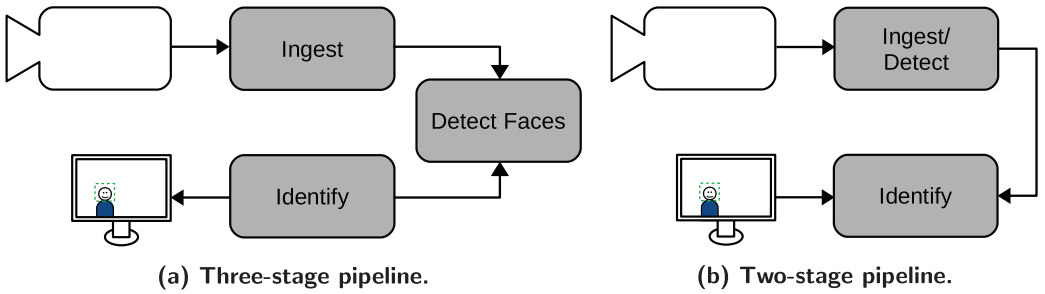


Fig. 3. Two possibilities for deploying *Face Recognition* as a data center application.

Even though there are four logical steps in the *Face Recognition* algorithm (ingestion, face detection, feature extraction, and classification), in practice this reduces to three. Feature extraction and classification are tightly coupled and their code is hard to separate. We term the combined steps, "identification." This simplification results in a reduced design exploration space.

In optimizing *Face Recognition*, we explored two designs for separating the algorithm into stages, shown in Figure 3. In Figure 3(a), the application is separated into its three logical components: ingestion, face detection, and identification. The ingestion stage, like the algorithmic step of the same name, is fed a video stream that is parsed into separated video frames. Because the ingestion and face detection stages exist in separate containers, likely on physically distinct nodes within the data center, the separated video frames must be transferred between them over the network. Similarly, the cropped face thumbnails produced by the face detection stage are sent over the network to the identification stage.

The alternative deployment for *Face Recognition* is shown in Figure 3(b): ingestion and face detection are combined into a single *ingest/detect* stage that operates along with identification. In this design, ingestion and face detection processes operate within the same container, so frames are transferred directly between them, leaving only the face thumbnails to be sent over the network to the identification stage.

Beyond the obvious difference between the two- and three-stage designs (the three-stage design imposes greater demands on the network), there is a subtler and more profound difference that must be considered. In the three-stage design, the junction between the ingestion and face detection stages is very simple: ingestion always produces one frame at a time at a particular rate and each frame must run through face detection exactly once. In contrast, the junction between face detection and identification transfers a variable number of faces and this face count determines the amount of work to be done in identification. Hence, the compute demands placed on the identification stage will vary based on the nature of the video streams—a video stream that captures many faces will demand greater identification processing power. Thus, by decoupling face detection from identification, we create a point of flexibility where a single face detection (or ingest/detect) container can be serviced by potentially many identification containers—i.e., load balancing. The junction between ingestion and face detection has no such requirement.

Both in order to reduce the demands on the network and to combine processing steps where it makes sense to do so, we adopt the two-stage design (Figure 3(b)). We also utilize one additional container, the *broker*, that we discuss in Section 3.4.

The *ingest/detect* container runs two processes internally, one for ingestion and the other for face detection. Ingestion processes a video stream (in our experiments, we use a `1920x1080` video file for deterministic operation) and parses the stream into frames. It resizes the frames to `960x540` before passing them to the face detection process. Face detection produces a thumbnail for each face in a frame, if any (our video yields zero to five faces and averages 0.64 faces per frame, with face thumbnails averaging 37 kB each). If no faces are found, identification is not needed; otherwise, the faces are transferred to the *identification* container. It consists of a single process (the combined feature extraction and classification, as mentioned earlier) that processes face thumbnails to yield an identity.

In accordance with industry practice, we execute all inference directly on the CPU [54]. This yields the lowest latency, which is critical in a user-facing application.

3.4 Apache Kafka

Communication between containers running on separate nodes within a data center requires intelligence and elegance. We rely on Apache Kafka [5] to manage the communication between ingest/detect and identification, allowing for load balancing and offering rapid adaptation in the presence node failures. Though there exists a variety of open source tools for building and managing streaming applications [3, 4, 6, 7, 13, 35], we note that these tools tend to rely on a separate framework for enabling communication between containers. It is common in practice to rely on Apache Kafka to serve this purpose, and each of these projects has proponents extolling the benefits of using Kafka [47, 59, 65, 66, 75, 77]. We therefore use Kafka directly to coordinate communication between our containers.

Apache Kafka implements the publish-subscribe pattern of communication [42]. This pattern operates by relying on an intermediate staging area for data, instead of data producers sending data directly to data consumers. The intermediate staging area is divided into *topics* to distinguish different kinds of data. Data *producers* publish data (send it to the intermediate staging area) without any knowledge of the data *consumers* or even a guarantee that any consumers exist. They simply publish the data as a *topic*. Similarly, consumers subscribe to a topic, oblivious to all details about the producers. As producers publish data to a given topic, the data become visible to the consumers subscribed to the same topic; consumers are then free to process the data.

In Kafka, the intermediate staging area where topic data is stored is implemented in *brokers* (these are the brokers mentioned earlier being deployed in their own containers). A topic is implemented by creating partitions—open file handles—typically spread across multiple brokers. When

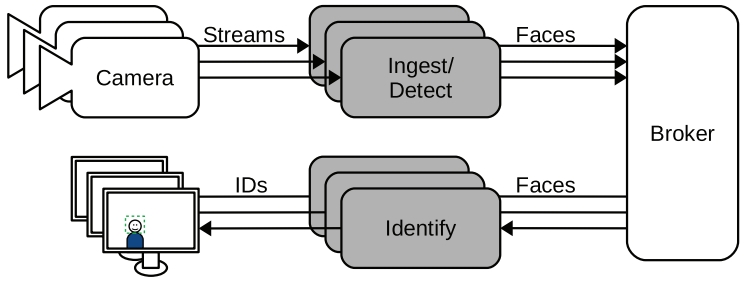


Fig. 4. Data center deployment of *Face Recognition*. Algorithms run as standalone processes in lightweight Docker containers deployed on separate nodes throughout the data center. *Ingestion* and *face detection* both run in the *ingest/detect* container while *feature extraction* and *classification* are combined in *identify*. Communication between steps within a container happens internally while communication between containers is coordinated through Apache Kafka *brokers*.

a producer publishes data to a topic, it may send that data to any of the partitions, which the corresponding broker receives and writes to the open file. When a consumer requests data from a partition, the broker reads it from the same file. In contrast to producers, partitions may have a maximum of one consumer. Thus, an application should divide a topic into at least as many partitions as there are consumers in order to maximize parallelism.

The topic partition also serves as the basic unit of replication. Kafka assumes and encourages data replication for reliability should a broker go down. Each partition has a "leader" and, in the presence of replication, some number of followers. After new data is written to a leader partition, it is replicated to the followers. Producers and consumers interact with the broker that holds the leader partition, while the follower partitions are spread among the remaining brokers. In the event of a broker failure, one of the follower partitions will become the new leader partition. Unlike partitions, there are no "leader brokers" or "follower brokers"; both leader and follower partitions are spread among all available brokers; thus, no one broker is more important or heavily utilized than any other.

Our data center deployment of *Face Recognition* is depicted in Figure 4. The ingest/detect containers function as producers, sending face thumbnails extracted from each frame to brokers as the "faces" topic. The identification containers are the corresponding consumers, subscribing to the "faces" topic. The placement of brokers between ingest/detect and identification containers was chosen to provide load balancing. As we will show in Section 4, the two containers have different latencies; we thus instantiate more identification than ingest/detect containers. By placing brokers between them, Kafka ensures that the work is spread among the consumers evenly. Each of the three container images is deployed a set number of times and distributed throughout the data center. Because of the extremely low network utilization relative to capacity (Section 5.4), the placement of containers relative to one another in the data center is unimportant. We use a minimum of three broker nodes in all cases to allow for three-way data replication, reflecting common practice in industry-quality deployments.

When we experimented with the three-stage (four containers: ingestion, face detection, identification, and brokers) setup, the communication of video frames between ingestion and face detection was also passed through the brokers. We simply created an additional topic—frames—within the same set of brokers.

3.5 Container Resources

An important consideration in deploying containers is determining what resources each should be allocated. Figure 5 shows how the ingest/detect and identification containers perform with

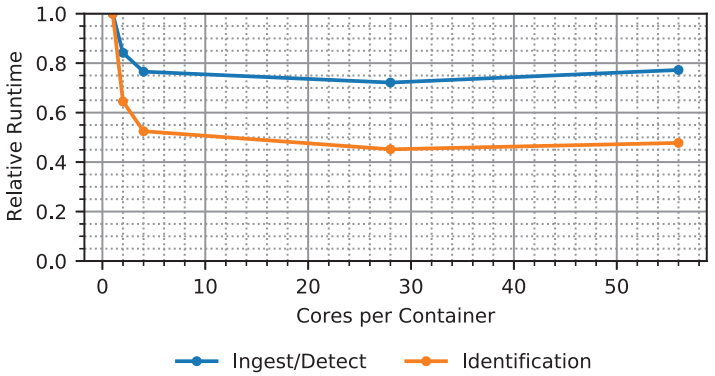


Fig. 5. Relative computational latency of *Face Recognition* containers with core scaling. Core scaling shows very limited effectiveness in decreasing the latency, particularly in identification containers.

increasing core counts. With more cores, both containers complete their operations at lower latency, but they do not scale linearly. Doubling the core count from one to two yields only a 16% reduction in latency in ingest/detect and a 36% reduction in identification. At larger core counts, the computational latency actually increases for both containers.

The "correct" allocation of cores to containers depends on the workload requirements. Often in a data center application the key metric is latency-bounded throughput: maximize the throughput of the application as long as the latency of individual queries remains below some upper bound. This is a prominent metric used in MLPerf Inference [57].

For *Face Recognition*, where we lack a clear latency bound and given the very poor core-scaling behavior, we optimize for throughput by assigning a single core to each container and arbitrarily declare that the resulting latency is acceptable. This may not be the case in other applications and deployments.

3.6 Container Ratios

As discussed in Section 3.3, identification can take a variable amount of time, depending on the number of faces that need processing. Furthermore, we will see in Section 4 that identification takes significantly longer to perform its calculations than ingest/detect, even when identifying only a single face. To ensure that identification can keep up with ingest/detect, we allocate many more identification than ingest/detect containers.

The precise ratio of identification to ingest/detect containers is dependent upon the characteristics of the video streams and the latency bound. If video streams never contain more than one face at a time, fewer identification containers are needed. However, even if video streams tend to show low face counts on average but have large spikes where there are many faces at once, this can temporarily overwhelm the identification containers and so requires that more such containers are instantiated. We will see this in our experiments (Section 4.2).

4 AI TAX

We start our exposition of the AI tax by evaluating the end-to-end performance of *Face Recognition*. We aim to understand what fraction of the cycles in an AI application go to AI processing versus the non-AI components. To this end, we examine the lifetime of a frame as it flows through the AI-centric *Face Recognition* application. In Section 4.1, we explain how we measure frame progress. Section 4.2 breaks down the end-to-end progress of a frame in each stage of the pipeline and shows that AI computation is not so central as one would expect in an AI application. In Section 4.3, we

```
1  from kafka import KafkaProducer
2  from multiprocessing import Queue
3  ...
4
5  def detect_faces(frame):
6      """Do the work of resizing faces using an ML model."""
7      ...
8
9  def produce_faces():
10     producer = KafkaProducer()
11     while True:
12         # frame_queue is a multiprocessing.Queue.  It is filled with frames
13         # from the ingestion process.
14         frame = frame_queue.get()
15         start_time = time.time()
16         faces = detect_faces(frame)
17         end_time = time.time()
18         if len(faces) > 0:
19             producer.send(faces)
20
21         # Send high-level event information to the logger.
22         logging.info("Face Detection", extra={
23             "compute_time": end_time - start_time,
24             "face_count": len(faces),
25             "data_size": sys.getsizeof(faces)
26         })
27
28 ...
29 produce_faces()
30 ...
```

Listing. 1. Simplified code showing the basic operation and event gathering of face detection within an ingest/detect container. The function `produce_faces` (line 9) operates in an infinite loop (line 11). `frame_queue` (line 14) is a `multiprocessing` data structure to pass frames between the ingestion process and this process, which operate in the same container. `detect_faces` is called on line 16 and does the heavy-lifting to extract faces from frames. Interaction with Kafka brokers (line 19) is extremely simple. We capture high-level event information, such as the execution time of `detect_faces` (lines 15 and 17), the number of faces found (line 24), and the size of the face data (line 25). These events are logged for later processing (line 22).

break down the application behavior in each container and reveal how much supporting compute is needed to enable AI processing. We show that it is vitally important to view AI application performance holistically, as it involves much more than just AI processing, and the supporting code and infrastructure tax have a profound impact on latency.

4.1 Instrumenting the End-to-End Execution

To really understand an AI application deployed in even an edge data center, we must raise the level of abstraction from how applications are traditionally evaluated. While we do not claim to have the right level of abstraction for all end-to-end workloads, for *Face Recognition*, we believe that we have identified a good level of abstraction for tracking and measuring application progress without perturbing the application's original behavior.

Application progress is a sequence of unit steps that are necessary for a frame to progress through the application. We term the units of application progress "events"; these are high-level steps in the application and correspond to the stages described in Section 3: video ingestion, face detection, broker waiting time, and identification. Listing 1 shows simplified Python code

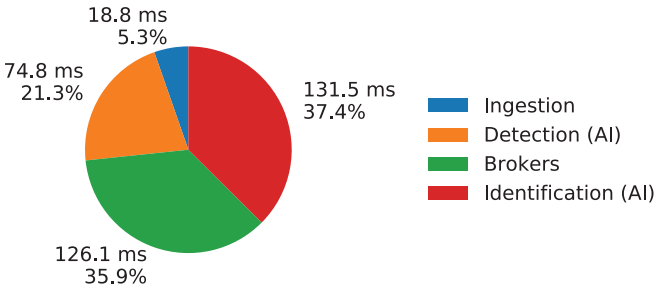


Fig. 6. Breakdown of end-to-end frame latency. With inference steps in detection and identification stages, less than 60% of the latency arises from AI stages. Over a third of a frame's lifetime is spent in brokers.

demonstrating the operation of the face detection process with event-logging code inserted. The event-logging code in the listing is only slightly simplified from our actual code. Event-based logging lets us track end-to-end application progress. This higher level of abstraction is critical in enabling engineers to architect at a cluster level, where the complete application executes, instead of just at a node level.

We log all the events during execution of the application using Elasticsearch [31] and Logstash [26] running on a separate server. We measure the execution time of each step as well as the sizes of data that are transferred between stages. This is done using timestamps around the major regions of interest, e.g., the time to do the face identification *excluding* the supporting code (e.g., iteration management). In essence, events capture the major steps that a frame goes through from beginning to end. We use built-in language functions to measure the size of the data that are transferred through the brokers. Due to the infeasibility of instrumenting a complex program such as Kafka, we approximate the broker waiting time event by calculating the time delta between the end of face detection and the beginning of identification.

Our instrumentation method has negligible overhead and resource requirements, since we are only logging events. It has minimal impact on the application (see Figure 8).

4.2 AI Applications Are More than Just AI

One of the end user's primary concerns is latency. In *Face Recognition*, this is the total time of a frame progressing serially from ingestion through identification; the latency of any stage that is not performing AI contributes to the AI tax.

We conduct experiments with 840 ingest/detect processes (producers) executing on 15 nodes (56 processes per node), 1,680 identification processes (consumers) executing on 30 nodes (56 processes per node), and 3 brokers (each given its own node). We require the brokers to maintain 3× data replication, which is standard practice for disaster recovery.

We measure an average face size of 37.3 kB and an end-to-end latency of 351 ms. While this latency may seem large, there are two points to remember. First, the throughput per stream is around 10 frames per second (FPS)—and a single stream could be divided among three ingest/detect instances for 30 FPS operation—regardless of the latency, since the application is pipelined; the output video still displays smoothly, just with a small delay. Second, there are multiple inferences per frame, performed sequentially, with the inference stages located on different nodes to improve performance [50]; the communication between the stages imposes additional latency.

Figure 6 summarizes the average latency for each stage. Ingestion operates quickly, taking only 18.8 ms, while the AI stages, face detection and identification, take 74.8 and 131.5 ms, respectively. Remarkably, over a third of the end-to-end latency is spent waiting between stages, at 126.1 ms. As in any real-time application, tail latency is an important factor to consider. We measure a 99th

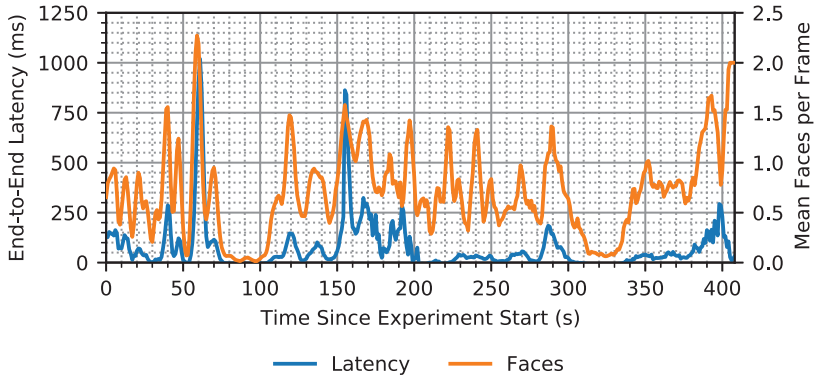


Fig. 7. Latency tracks the total number of faces in the system. As the experiment runs, average end-to-end latency is clearly correlated to the number of average faces per frame; with higher average faces per frame, the total number of faces in the system is pushed upward, causing more delay through the communication system.

percentile tail latency of 2.21 s, with the standalone 99th percentile tail latencies of ingestion, detection, waiting time, and identification at 27 ms, 116 ms, 1.84 s, and 380 ms, respectively. We remind the reader that these latencies can be improved somewhat by allocating additional cores to each stage of processing; in our implementation, however, we have chosen to optimize for throughput over latency.

From the tail latency breakdowns, we see that the end-to-end tail latency derives almost entirely from the waiting time in the brokers. This waiting time in turn results, at least in part, from congestion in the application. As shown in Figure 7, when ingest/detect processes collectively produce a surplus of faces, identification has a hard time keeping up, leaving the faces in the brokers for a longer time. When there are almost no faces detected, identification containers are almost idle and so are able to fetch the few faces quickly.

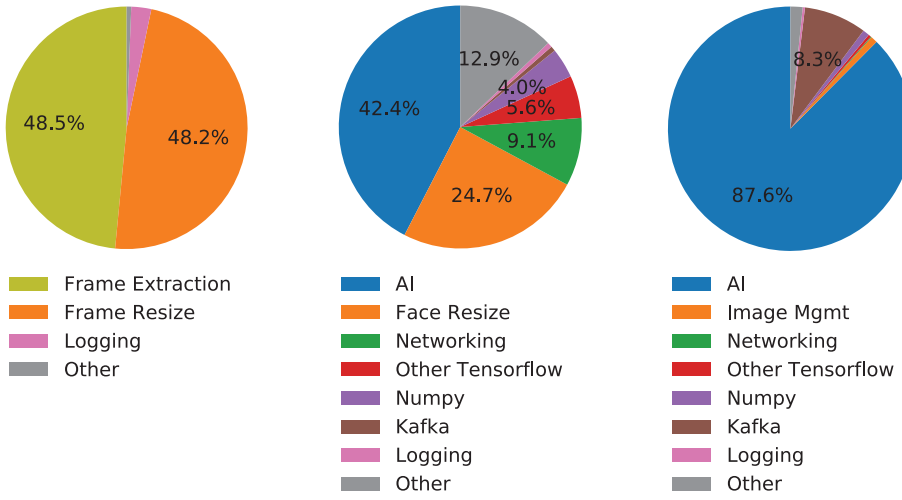
In summary, deep learning inference performance is more than just the performance of an individual node in the system. Even with a well-balanced system, there is a substantial AI tax latency imposed in managing the transfer of data between the nodes (i.e., the detection and identification stages). Without looking at the end-to-end latency, one would not realize that a large portion of time is spent waiting at brokers. This observation is *not* unique to our application; any application built on Apache Kafka (or a similarly brokered communication mechanism) will face this reality.

4.3 Overhead of Pre- and Post-Processing AI

Most AI research papers focus only on the core AI component, neglecting the other associated parts that are essential to end-to-end AI processing. However, there are pre- and post-processing steps that are unavoidable. Both play a critical role in the overall latency and both contribute to the AI tax. Pre-processing involves preparing the data for the AI kernel execution, while post-processing is loosely defined as any processing that is performed to convert the generated AI result(s) into something meaningful to the user or next stage.

To quantify the AI tax for pre- and post-processing, we refine our view of a frame's processing by looking at the time breakdown of each process using code profiling tools. Figure 8 shows, for each of the main processes (ingestion, detection, and identification), where time is spent.

Ingestion is exclusively a pre-processing stage. It shows a nearly even split between frame extraction and frame resizing (Figure 8(a)). Extraction refers to parsing the incoming video stream into individual frames. Resizing converts frames from `1920x1080` to `960x540` for the detection



(a) Ingestion. **(b) Detection.** **(c) Identification.**

Fig. 8. Process CPU time breakdowns. Though ingestion uses no AI algorithms, it consists of straightforward image processing algorithms, which should be easily accelerated [62]. Face identification is overwhelmingly AI-centric. In contrast, face detection relies heavily on supporting code to enable its AI processing.

stage. The remaining time is split between the overhead of event logging and other supporting code, including transferring frames to the co-located detection process.

During face detection (Figure 8(b)), despite being an AI-centric stage, only 42% of the time is spent executing the AI algorithm in TensorFlow. Cropping and resizing faces (to `160x160`) for identification takes 25% of the time; supporting TensorFlow and NumPy code (pre- and post-processing for each frame) take 6% and 4%, respectively; and "other" code takes a whopping 13% of the time. Code in the "other" category includes inter-process communication (from the ingest stage), additional matrix manipulation, loop management, bounding box calculation, image encoding, and so forth.

The AI-centric identification stage has a markedly different breakdown. It spends 88% of its time directly executing AI algorithms; Kafka code, though, takes 8% of the time. The remaining components contribute little to the total time.

Beyond the pre-processing of the ingestion stage, end-to-end *Face Recognition* requires substantial pre- and post-processing within the AI-centric stages. In face detection, non-AI computation constitutes 57.6% of the compute cycles. In identification, that figure drops to 12.4%, which is still far from trivial. In a complex and diverse AI-centric application such as *Face Recognition*, AI computation constitutes 55.2% of end-to-end cycles, with the remainder going to supporting code: 17.8% to resizing, 9.0% to networking, 5.2% to tensor preparation, 3.6% to Kafka processing (outside of the brokers), and the rest to other supporting tasks.

In summary, despite the massive excitement surrounding AI algorithms, AI workloads are more than just tensors and neural networks: without the supporting code, AI is impotent. We emphasize that the supporting code, far from being a minor player in a complete application, constitutes over 40% of the compute cycles, not counting the compute time in the brokers. The pre- and post-processing code is executed on the general-purpose CPU, so it motivates the need to understand the role of the CPU as AI acceleration increases.

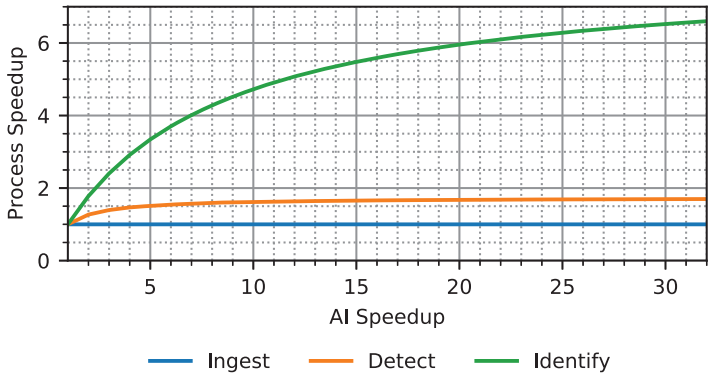


Fig. 9. Projected speedups of individual processes. Amdahl's law predicts that the realized speedups of accelerated AI diminish quickly, approaching an asymptote far short of the speedups of AI accelerators.

5 ACCELERATING AI AND ITS IMPLICATION ON AI TAX

There are numerous efforts underway to accelerate AI [9, 15, 18, 22, 29, 43, 63, 76]. But given the significance of the AI tax in end-to-end AI performance and in pre- and post-processing, it is important to understand how the tax evolves as AI is accelerated and its impact on the overall end-to-end application performance; there are performance limitations that arise beyond a certain point of AI acceleration. To build a balanced system, it is important to understand these limits.

We therefore study how varying degrees of AI speedup affect the end-to-end performance of our video analytics workload. In general, we could foresee accelerated AI coming about in various ways: CPU manufacturers may decide to integrate ML-centric hardware directly into the CPU execution pipeline; or dedicated off-chip accelerators may be utilized, including highly parallel pipelines (such as GPUs) and dedicated inference engines (such as Intel's Neural Compute Stick [22], Habana's Goya inference processor [21], or Google's Coral Edge TPU [9]). Comparisons of the efficacy of each of these solutions is the subject of other work. In this work, we look at the impact of theoretical speedups, regardless of how the speedups are achieved.

In this section, we explore the impact of accelerating AI applications up to 32×, based on the performance of existing accelerators. Habana reports that its Goya processor achieves 13.5× speedup over a two-socket Intel Xeon Platinum 8,180 [74].[1] We explore speedups approximately twice as great as this (up to 32×) in order to account for future advances.

Section 5.1 analytically estimates accelerated AI performance to show its asymptotic limits. Section 5.2 introduces our technique for emulating accelerated workloads on current hardware. In Section 5.3, we evaluate the performance of accelerated processing and discover a quickly approaching bottleneck. Section 5.4 shows that the bottleneck results from overwhelming the system's capacity to write to storage. We finish by showing in Section 5.5 how frames' waiting time in brokers grows as a fraction of end-to-end latency.

5.1 Analytical Speedups for AI Acceleration

The AI tax means that a significant portion of an AI application's compute cycles are spent on tasks other than AI and ML, and Amdahl's law dictates that the overall speedup of a system is limited by the portion of execution that is not accelerated. Application of Amdahl's law (Figure 9) shows that each of the three primary processes—ingestion, detection, and identification—is limited in how much real-world speedup it can enjoy if AI is accelerated in isolation. Ingestion, which

[1] Other than its faster clock, the 8,180 CPU is identical to the 8,176 CPUs we use.

performs no AI compute, naturally derives no benefit from acceleration. Detection, which is 42% AI, rapidly approaches its asymptotic speedup of just 1.74×, achieving 1.59× overall speedup at 8× acceleration and 1.66× overall speedup at 16× acceleration. Identification, at 88% AI, has an asymptotic speedup limit of just 8×. At 16× AI acceleration it achieves 5.6× overall speedup, and even at 32× AI acceleration it shows just 6.6× overall speedup.

The exciting speedups promised by up-and-coming inference accelerators will be severely moderated by the reality of the supporting, non-AI code—the AI tax. With the fervor surrounding acceleration of AI and ML, these results from Amdahl's law serve as an important reminder that AI applications are more than ML computation. Supporting and enabling code is a critical component of an end-to-end application and this should serve as a call to action to address the limitations imposed by that code.

5.2 Emulating AI Acceleration on Hardware

It is instructive to see how the AI tax evolves as compute is universally accelerated (i.e., overcoming the asymptotic limits of Section 5.1) on a real system. To do so, we emulate the behavior of accelerated processing. Only the most basic loop controls and Kafka code are left in their original state.

Our emulated acceleration technique relies on the observation that, from the perspective of application progress, the perspective of network traffic, and the perspective of the brokers, it is impossible to distinguish between (1) running the real application as has been described and characterized and (2) implementing artificial delays reflective of the actual compute times (Section 4) and sending meaningless data over the network of the same size as in the real application. In accelerated *Face Recognition*, rather than accelerating and executing the real algorithms, we replace the compute with calls to `sleep`, where the sleep duration is reflective of measured execution times (Section 4.1). Accordingly, rather than sending face thumbnails to brokers, we send meaningless data whose size matches the measured sizes. We can accelerate processing (both ingest/detect and identification) by an arbitrary factor by dividing the sleep times by the speedup factor. In this way, we maintain the behavior of the brokers, network, storage, and supporting code while exploring how acceleration changes the AI tax.

We emphasize that this AI acceleration emulation provides realistic performance estimation under acceleration because (1) the most basic general-purpose processing (the support code to iterate through available frames, code to coordinate communication with Kafka brokers, the brokers themselves, etc.) remains in place and is executed as usual without the benefits of acceleration; (2) from the perspective of the data center, compute time spent executing real algorithms and waiting in `sleep` are identical; and (3) the brokers are completely ignorant of and unconcerned with the execution details of both producers and consumers. Thus, our setup to accelerate AI through emulation provides a realistic and believable look at the impact of faster AI on the data center and on the workload as a whole.

5.3 Accelerated AI Impact on Total Speedup

We explore how the end-to-end frame latency will evolve as AI benefits from increasingly powerful acceleration. For this analysis, we assume that the AI algorithms will experience no latency overhead from acceleration-i.e., we assume that the latency to communicate with dedicated off-chip accelerators is factored into the emulated speeds, or, equivalently, that future CPU architectures will directly integrate accelerator hardware into their execution pipelines.

For these experiments, we maintain the same application organization as depicted in Figure 4. As we accelerate producers and consumers equally, the imbalance between the two will persist, so it is important to utilize Kafka's load balancing features at the interface between the two.

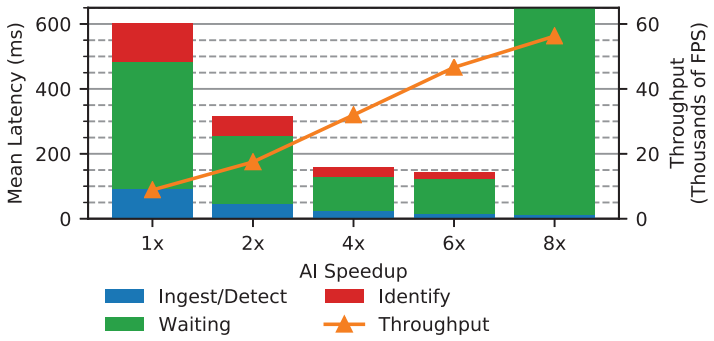


Fig. 10. Average frame latency and throughput under increasing AI acceleration. Beyond 8x speedup, the increased throughput leads to an unbalanced system. Queueing theory dictates that if elements enter the system faster than they leave, the latency increases to infinity.

For the sake of simplicity and repeatability and without loss of generality, we configure these emulation experiments so that each frame produces exactly one face. This has two impacts on performance: (1) on average, this is more faces per frame than produced by the video file used previously, which yields 0.64 faces per frame on average; and (2) because the rate of face production is constant, we do not have to provision our cluster to handle sudden spikes in traffic, allowing us to deploy fewer identification instances than for the video file. None of the conclusions we draw from these experiments is invalidated by these these two observations.

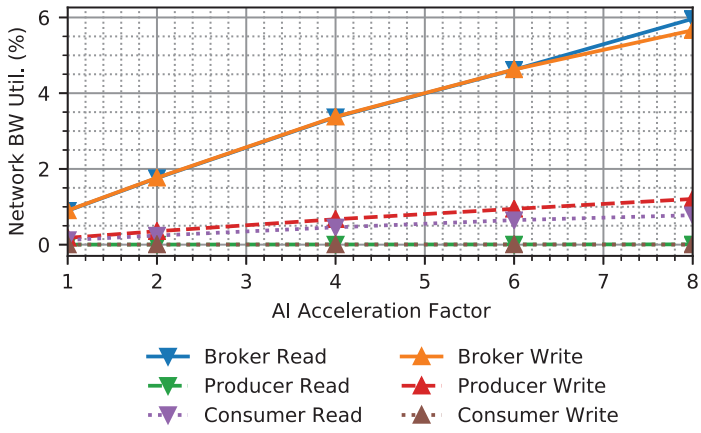
Figure 10 shows the effects of accelerating the AI components of *Face Recognition*. Note that because we assume one face per frame, which is significantly higher than the 0.64 faces per frame average produced by our default video file, the average end-to-end latency is somewhat higher at 1× speed than in Section 4.2. At higher speedups, we see a twofold benefit: first, the latency is very clearly reduced; second, the throughput is commensurately increased.

At 8× speedup, we see a new manifestation of the AI tax, with latency tending toward infinity—the longer the experiment runs, the larger the latency grows. This is an example of an unstable system in queueing theory: faces are entering the system more quickly than they are leaving. This is a major limitation that can severely hamper the prospects of AI acceleration and demands further investigation.

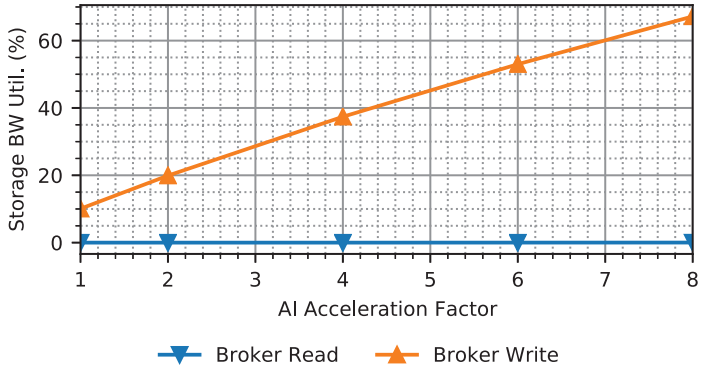
5.4 Network and Storage Bandwidth Limits

With state-of-the-art industry accelerators claiming improvements of up to 15× in inference speedup over CPUs [18], it is critical to understand why the system becomes unbalanced at 8× acceleration. Without this insight, it will be difficult to build systems to accommodate higher acceleration factors.

Intuitively, we suspect the imbalance results from the increased throughput of the system overwhelming one of two resources with limited bandwidth: either network or storage bandwidth. We measure the utilization of both bandwidths to understand the problem at increased acceleration factors (i.e., greater than 8×). In Figure 11(a), the network bandwidth utilization of all container types rises with increasing acceleration factor. Unsurprisingly, producer (ingest/detect containers) network read bandwidth is next to zero, as is consumer (identification containers) write bandwidth. Conversely, producer write and consumer read bandwidths are comparable. But the real network bandwidth hot spot is the brokers—as the point of communication between producers and consumers, they must process all network traffic generated by the producers or read by the consumers. However, even the combined network traffic flowing through the brokers constitutes a small



(a) Network bandwidth utilization. Network activity from and to producers and consumers is concentrated at the brokers; nevertheless, broker network utilization falls far short of our 100 Gbps capacity.



(b) Storage bandwidth utilization. Storage activity for producers and consumers is not shown because it is not preserved in accelerator emulation. Network activity is translated to storage activity in the brokers.

Fig. 11. Network and storage bandwidth utilization under acceleration. Whereas network utilization never exceeds 6% of network capacity, storage bandwidth utilization exceeds 67% of capacity at 8x acceleration. Storage bandwidth becomes a bottleneck much sooner than network bandwidth.

portion of the available bandwidth: at 8× accelerated AI, the read bandwidth is only 6 Gbps, a mere 6% of the available 100 Gbps.

Figure 11(b) shows the storage bandwidth requirements of the brokers. We omit the data for the producer and consumer containers, as their storage behaviors are not preserved by our emulation technique and are nevertheless expected to be near zero, as they work largely out of memory. The brokers, however, have rather high bandwidth requirements. Even at native (1×) speed, the write bandwidth is 10% of capacity (1.1 GB/s). At 8× acceleration, that rises to over 67%.

With the overhead of the operating system, managing the file system, and coordinating all the small requests to be written to storage, by 8× acceleration, 67% utilization has effectively saturated the available bandwidth. Returning to queueing theory, the inability of storage to write data to

storage (and make it available to the consumers) as fast as it is supplied leads to the imbalance and growing latency.

We note that data reads, however, use essentially none of the available bandwidth. This is easily understood: brokers are tasked with ensuring data reliability, so they must write producer data to storage, but the operating system can also cache the data in memory, allowing reads directly from memory and bypassing the storage read path.

Doubtless, fine-tuning the brokers' parameters could allow them to better utilize the storage bandwidth. An in-depth exploration of the Apache Kafka parameter space is not, however, the purpose of this article. Regardless of the ability of the brokers to utilize available bandwidth, they will hit a hard limit at the specifications of the hardware devices.

In a setup with a more conservative network bandwidth (e.g., 10 Gbps), both the storage and the network would quickly become bottlenecks when accelerating compute.

Thus, the increased throughput of a moderately accelerated end-to-end system creates a new AI tax that quickly overwhelms the communication substrate, counteracting the gains achieved through hardware acceleration of AI.

5.5 Increase in Waiting Time

Furthermore, whereas the waiting time at 1× speed constitutes 64.6% of the total latency of a frame (Figure 10), it grows to 66.4% at 2×, 68.0% at 4.0×, and 79.1% at 6×. This trend can be partially understood by Kafka's automatic batching between brokers and consumers and producers. A message from a producer can be held in the producer for a small amount of time until a larger group of messages has been accumulated to be sent as a batch. Similarly, when a consumer requests available messages from a broker, the broker can withhold messages until there exists some minimum amount of data. Thus, the broker time grows with the decrease in compute time to improve batching. Both batching behaviors are limited by timeouts to ensure that neither producer nor consumer waits excessively long. We have tuned these parameters to find settings that ensure good behavior across a variety of experiments. Nevertheless, the time spent waiting between producers and consumers approaches some lower limit beyond which no amount of tuning can help; in an application that has many more stages than *Face Recognition*, this minimum waiting time could accumulate across stages and prove prohibitively long.

6 GENERALIZABILITY OF FINDINGS

We recognize that our research is a case study of a single application. While case studies are often undervalued—despite shedding light on an application that is valuable in its own right and pioneering an evaluation approach [67]—we nevertheless acknowledge that evaluation of additional applications is beneficial. We therefore discuss some findings on a second application, *Object Detection*, that was deployed similarly using Kafka in our edge data center. While we do not study *Object Detection* in the same detail as *Face Recognition*, we discuss here some results showing that this additional application faces AI tax bottlenecks as well. In Section 6.1, we describe the purpose and design of the application. We look at the AI tax in *Object Detection* when running natively (Section 6.2) and when accelerated (Section 6.3).

6.1 Object Detection

Like *Face Recognition*, *Object Detection* analyzes video streams in real-time. Instead of recognizing faces, though, *Object Detection* uses an R-CNN [69] to identify multiple objects in each frame. Also like *Face Recognition*, *Object Detection* is split into two stages with Kafka brokers serving to transfer data between them. The two stages are termed ingestion and detection. Ingestion ingests a video

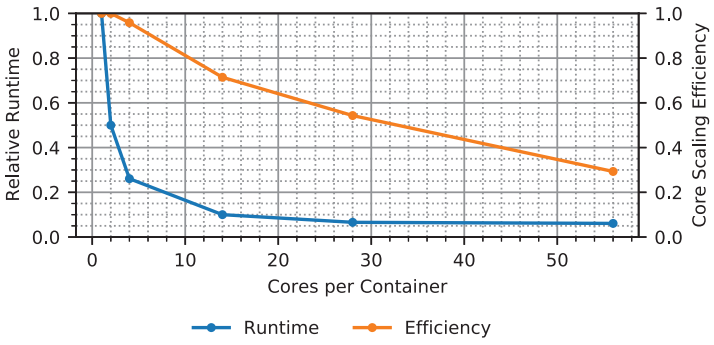


Fig. 12. Relative computational latency of *Object Detection*'s detection containers with core scaling. Core scaling shows very good efficiency in reducing runtime, particularly compared to *Face Recognition*.

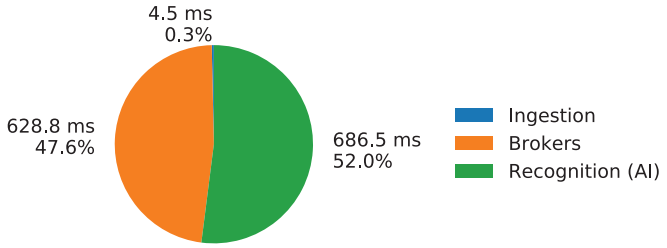


Fig. 13. Breakdown of *Object Recognition* average end-to-end frame latency. Unlike *Face Recognition*, *Object Detection* performs no AI computation in its initial stage (Ingestion); hence, it is orders of magnitude faster than the Recognition stage, where all the AI computation is performed.

stream, parsing it into separate frames, and passes those frames through Kafka to detection. AI compute is exclusively performed in this later stage in the R-CNN.

Unlike *Face Recognition*, wherein the presence or absence of faces in a frame dictates whether and how much data is sent through Kafka, in this application each frame is always sent. This decreases the variability in system load, as each detection instance always has to process precisely one frame at a time.

As shown in Figure 12, the detection stage of *Object Detection* shows near linear speedups with increasing core count. Through testing, we determined to allocate 14 cores per container; this allows us to instantiate four detection containers per server. Despite this, the ingestion stage operates orders of magnitude faster than the detection stage (see Section 6.2); we limit the ingestion rate to 30 frames per second. To balance that ingestion rate, we instantiate 96 detection containers for each ingestion container.

6.2 AI Tax

Figure 13 shows the end-to-end frame latency breakdown. The first stage, ingestion, performs no AI. As such, it completes very quickly, in only 4.5 ms. As stated, we limit the ingestion rate to 30 FPS, so for practical purposes this time is really 33.3 ms. In contrast, the final stage, detection, does all of the AI processing and weighs in at an impressive 687 ms. Waiting time in the brokers is nearly as long, averaging 629 ms.

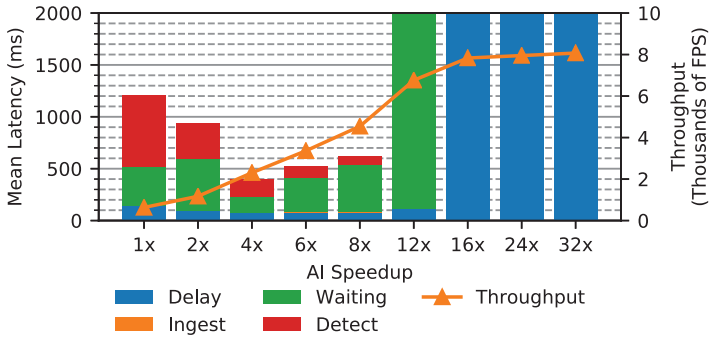


Fig. 14. *Object Detection* average frame latency and throughput under increasing AI acceleration.

6.3 Acceleration

Using our acceleration emulation methodology, we explore the implications of accelerating *Object Detection*. Given the large number of cores needed for each detection instance, and given the limited size of our cluster, we assign only a single core to each detection instance for these experiments, allowing us to deploy significantly more ingestion instances than we would be able to otherwise. This is feasible because the emulation methodology does not care about number of cores and because in an accelerated data center setup, it may be very possible to support the higher instance count.

We maintain the same ratio of producers to consumers as in our real setup, but by increasing the density of consumers we are able to scale up the experiments significantly. We use a single producer node but instantiate 21 producers on it and we use 36 consumers nodes, each with 56 consumers. We continue to use three brokers.

Since we already decided to limit the frame rate to 30 FPS, we continue this practice. With increasing acceleration, we increase the number of frames we send—effectively, the acceleration factor dictates the number of simultaneous video feeds each producer can process. Thus, at 2× acceleration, each producer sends frames at 30 FPS but it sends two frames at a time instead of one; similarly, at 8× acceleration, a producer sends eight frames at 30 FPS, and so on.

Figure 14 shows the results of acceleration. Despite the significantly lower count of producers compared to *Face Recognition*, we still see performance begin to degrade after 4× acceleration. By 12×, the average latency is not yet infinite, though it does exceed 3,000 ms. But at 16× and above, the average end-to-end latency again tends toward infinity.

While we see the broker waiting time begin to grow at 6× and particularly 12× acceleration (suggesting that the brokers are probably again facing a storage bottleneck), the real bottleneck in this application arises from a new category entirely. In Figure 14, we have added a "Delay" category for latency components; this component represents the time between when a frame (or set of frames) was supposed to start processing in ingestion and when it actually starts processing. This delay arises from a set of frames taking longer than 33.3 ms (30 FPS) and delaying the start of the subsequent set of frames.

This ingestion delay represents a new manifestation of the AI tax we had not seen previously. For every set of frames, we have opted to send each frame to the brokers separately; this ensures that they can be fully load balanced by the brokers. However, the time required to send the full set of frames rapidly grows with the larger set sizes, so that it soon exceeds the time allotted to each set: at 4.5 ms per frame, sending just eight frames should take approximately 36 ms, exceeding the allotted 33.3 ms. Kafka is well designed, however, so the producers and the brokers manage to

intelligently batch the frames before sending them; this means that at 8× and 12× speedup, Kafka manages to keep the "Delay" time manageable. But by 16× speedup, the AI tax from sending so many items so rapidly overwhelms the capacity of the producers to keep up.

We see this bottleneck reflected in the throughput as well. At 1×, the throughput is 630 FPS, as expected. That scales pretty well up to 8× speedup, but it falls short of what is expected at 12× and the system saturates by 16× speedup.

7 AI-CENTRIC DATA CENTER DESIGN

As future accelerators emerge, we seek to unlock higher speedups. In Section 5, we saw that in an accelerated AI environment, the AI tax overwhelms the communication mechanism; in particular, the storage medium is quickly saturated at relatively modest emulated compute acceleration speeds. So in Section 7.1, we explore two avenues to overcoming this bottleneck. Implementing these solutions to the tax translates to actual monetary cost, as shown in Section 7.2. We show in Section 7.3 that a purpose-built edge data center can address the tax with increased capacity to handle accelerated compute at lower TCO.

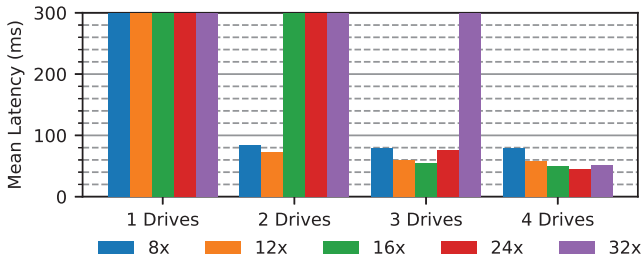
7.1 Unlocking Higher Speedups

There are three ways to deal with the limitation in the storage bandwidth: (1) utilize faster storage in the existing brokers, either through a faster storage medium (e.g., Intel Optane [23]) or through multiple drives operating in parallel; (2) create more storage bandwidth by allocating additional brokers; or (3) decrease the size of the face thumbnails, thus demanding less bandwidth. We explore all three methods (Figure 15), first increasing the installed drive count from one to four to provide greater bandwidth to each broker node, then increasing the broker count from three to eight across distinct broker nodes, and finally decreasing the face thumbnails down to one-eighth their original size.

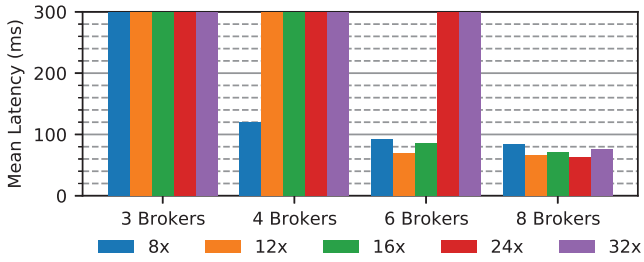
Increasing the Bandwidth. The effect of additional storage bandwidth on the existing nodes is captured in Figure 15(a). For these experiments, we instantiated additional broker instances on each broker node (one for each drive) to ensure that each drive is given the same access to compute and memory resources; in practice, only one broker should be instantiated per node to avoid replicating data on the same node. In the figure, we start with 8× speedup—the speedup that sent latency to infinity in the previous experiments—and increase the emulated speedup to 32×. With just one NVMe drive, the average end-to-end frame latency is infinite (depicted by the latency bar extending beyond the limits of the chart) at 8× and all higher speedups. These experiments rely on additional drives being installed only in the brokers—in our case, there are three of them; the remainder of the servers remain unaltered from their original configuration. In increasing the storage bandwidth by going from one drive to two drives, both 8× and 12× speedups are "unlocked"—the system gains the ability to support compute of these speeds. With three drives, the system supports up to 24× speedup, and with four drives, 32× is unlocked.

Spreading the Load. Rather than installing additional drives in each of the brokers, we can instantiate additional brokers in the data center. This spreads out the load on storage to more brokers and hence more drives. Returning each broker to its default storage configuration (one NVMe drive), we repeat our experiments with four, six, and eight brokers.

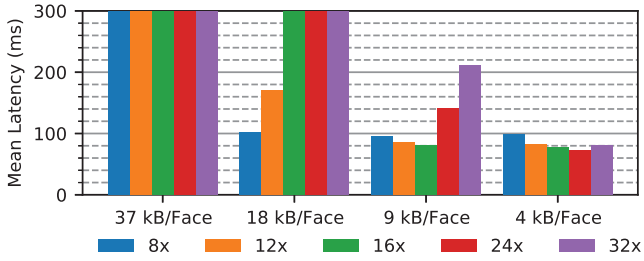
With three brokers, as we saw before, any acceleration factor at or above 8× leads to infinite latency. A small 33% increase in the broker count (going from three to four brokers), however, allows the system to handle the 8× factor, while a 2× broker increase allows for up to 16× acceleration. At eight brokers, the system can handle a 32× factor.



(a) Using additional storage devices on each node to add bandwidth.



(b) Using additional brokers to handle added bandwidth requirements.



(c) Decreasing thumbnail sizes to offset higher bandwidth demands.

Fig. 15. Average frame latency under accelerated AI. The higher bandwidth demands can be accommodated by allocating extra bandwidth or reducing face thumbnail sizes.

We find an important distinction between adding additional drives to existing brokers and adding additional brokers: the latter is more efficient. To achieve the ability for the system to support 32× accelerated AI compute, we had to increase the number of drives by a factor of 4; in contrast, we had to increase the broker count by 2.7× (going from three brokers to eight) for the same performance achievement. The significantly lower increase in storage bandwidth in the increased-brokers approach indicates that brokers may also benefit from having additional compute capacity, memory bandwidth, or network bandwidth available.

Decreasing the Demand. There exists one additional possibility for reining in the bandwidth demands on the storage: decrease the volume of data that needs to be stored. Rather than spreading the data among additional brokers, the data volume can be reduced by decreasing the average size of face thumbnails. Figure 15(c) shows the effect of face sizes at one-half, one-quarter, and one-eighth their original size. Similar to increasing the bandwidth in each broker, we see that the smaller face sizes use a smaller portion of the available bandwidth and so increase the maximum

Table 3. Homogeneous Data Center Equipment

Component	Price (US$)	Quantity
Dell PowerEdge R740xd (base server)	$28,731	1,024
Intel Xeon Platinum 8,176	Included	2
32 GB DDR4 SDRAM	Included	12
Intel SSD DC P4510 1 TB (NVMe SSD)	$399	1
Mellanox MCX415A (100 GbE adapter)	$660	1
Mellanox MSN2700-CS2F (100 GbE switch for fat-tree topology)	$17,285	160
Mellanox MCP1600 (100 GbE cable)	$100	3,072
Total	**$33,577,760**	

In a homogeneous data center similar to (but larger than) our own, there is considerable expense in ensuring that all components are equally equipped. The equipment cost of a 1,024-node data center would be around US$30.9 million.

supportable speedup, but without instantiating additional brokers or installing additional storage devices.

This solution, however, comes with serious tradeoffs. Decreasing face size using compression would require additional compute time, potentially offsetting much or all of the accelerator gains. Decreasing face size by using smaller thumbnails changes the algorithm and can detrimentally impact accuracy. Due to these severe limitations of this approach, we will focus on the previous two solutions.

7.2 The Cost of the AI Tax in the Data Center

A typical and simple approach that customers rely on to build an edge data center is to aim for homogeneity across servers (i.e., all of the server components are literally identical across the machines). But in a specialized application domain, such as edge video analytics, this ignores the unique characteristics of the applications and either significantly over-provisions some resources or severely handicaps application performance, leading to suboptimal TCO.

Table 3 shows the basic computing and networking equipment needed to build a homogeneous 1,024-node edge data center similar in compute capabilities to our own setup. This design gives each node comparable equipment to that used in our experiments: two 28-core processors, 384 GB of RAM, 100 Gbps interconnect, and a single NVMe drive. The nodes are connected in a three-level fat-tree topology using 32-port Mellanox Ethernet switches. This topology ensures full-speed non-blocking network connectivity to each node.

Using an open source TCO calculator from Coolan [36] to include power (servers, networking equipment, cooling, etc.), rack equipment, cabling costs, and so forth, and assuming a 3-year amortization life, we estimate a yearly cost of US$10.2 million for server equipment, US$1.3 million for network equipment, and US$1.4 million for power, for a total yearly cost of US$12.9 million.

While common wisdom regarding data centers suggests that the majority of the TCO is spent on power (including powering cooling equipment), simple analysis shows this is not necessarily the case. Each of the servers in our hypothetical data center is equipped with a 750 W power supply, while Mellanox reports that its routers can consume a maximum of 398 W [37]. This yields a total maximum power consumption of 921 kW. Cooling is estimated to require approximately as much power as the compute resources [10, 20], bringing the total to 1,842 kW. Assuming US$0.10 per kilowatt hour, operating the data center would cost US$184 per hour or US$1.61 million per year under maximum load.

To accommodate up to 32× accelerated compute in AI, we must either install three additional drives in each node (to maintain homogeneity) or designate a large number of the nodes as brokers.

Table 4. Video Analytics-Targeted Data Center Equipment

Component	Price (US$)	Quantity
Dell PowerEdge R740xd (compute server)	$28,731	867
Intel Xeon Platinum 8,176	Included	2
32 GB DDR4 SDRAM	Included	12
Mellanox MCX411A (10 GbE adapter)	$180	1
Dell PowerEdge R740xd (broker server)	$11,016	157
Intel Xeon Bronze 3,104	Included	2
32 GB DDR4 SDRAM	Included	12
Mellanox MCX413A (50 GbE adapter)	$395	1
Intel SSD DC P4510 1 TB (NVMe SSD)	$399	4
Mellanox MSN2700-CS2F (100 GbE switch)	$17,285	28
Mellanox MSN2700-BS2F (40 GbE switch)	$10,635	14
Mellanox MFA7A20-C010 (optical splitter 100 GbE to 2× 50 GbE)	$1,165	7
Mellanox MC2609130-003 (copper splitter 40 GbE to 4× 10 GbE)	$90	217
Mellanox MCP7H00-G002R (copper splitter 100 GbE to 2× 50 GbE)	$140	79
Mellanox MFA1A00-C030 (optical 100 GbE interconnect)	$515	192
Total	**$27,878,431**	

We use network splitter cables to supply 50 Gbps network to the brokers and, in combination with slower switches, 10 Gbps network to the compute nodes. This offers significant savings on network equipment. Further, we only install NVMe drives in the broker nodes, which are equipped with less compute power than the compute nodes.

Adding the additional NVMe drives costs US$1.23 million. Instead, we designate 157 of the nodes as brokers, 289 as producers, and 578 as consumers. This maintains the ratio of each node type as in our original *Face Recognition* experiments (15 producer and 30 consumer nodes, though with 8 brokers instead of 3) to enable support for 32× accelerated AI. Extrapolating from Figure 11(a), we estimate each producer and consumer node will consume approximately 4 Gbps of network bandwidth and each broker node 24 Gbps. The broker nodes demand less than 9 Gbps (or 1.1 GB/s) of storage write bandwidth.

7.3 AI-Specific Edge Data Center

The homogeneous data center was designed to be generic, capable of executing a variety of application classes; hence, we had to adapt the application to the data center, resulting in hugely over-provisioned network and storage. The producers and consumers constitute over 84% of the data center and use only 4% of the available network capacity and essentially none of the storage bandwidth. The brokers use a respectable 24% of the network capacity and basically all of the storage bandwidth but use very little of the compute capacity. This shows extremely inefficient allocation of limited resources in the data center. But we can do better.

Instead of forcing the application to fit into an existing data center, we propose building a data center that fits the application. We recommend a *purpose-built data center* that specifically targets the broker-specific AI tax (the demand for storage bandwidth). The AI tax, if not accounted for, can translate to non-trivial real-world costs that can drastically affect end users' needs. In contrast, by understanding the AI tax and designing to it, we demonstrate that a moderately sized edge data center can be purpose-built to better address the AI tax while yielding meaningful cost savings.

In Table 4, we see the equipment needed for this setup, designed to support up to 32× AI acceleration. In this scenario, we utilize the same highly parallel servers as in Table 3 for producers and consumers but limit the network bandwidth on these nodes to only 10 Gbps and install only basic

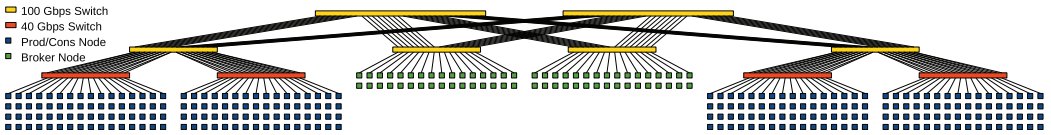


Fig. 16. Possible network configuration for an accelerated AI-centric edge data center. This network configuration is fundamentally a fat-tree built from 100 Gbps switches. However, since no node needs the full 100 Gbps bandwidth, we subdivide it. Mellanox offers splitter cables that split, for example, a 100 Gbps connection into two 50 Gbps or four 25 Gbps connections. Each pair of broker nodes shares a 100 Gbps port. Rather than providing each producer/consumer node with 50 or even 25 Gbps, we further subdivide the network connection speed using 32-port 40 Gbps switches. To provide full bandwidth to the 40 Gbps switches, each 100 Gbps port is split into two 50 Gbps connections, both of which are connected to the same 40 Gbps switch. Thus, the aggregate bandwidth provided to a 40 Gbps switch is 800 Gbps, of which the switch can use 640 Gbps. A 100 Gbps switch can connect two 40 Gbps switches. The slower switches use four-way splitter cables providing each producer/consumer node with a 10 Gbps network.

storage for operating each server. The broker nodes, in contrast, are built on far less parallel but still impressive CPUs, while enjoying 50 Gbps network connections and four NVMe SSDs.

We illustrate in Figure 16 a simple network solution that could provide the designated bandwidths to each server. At its heart, the network is still a fat-tree built from 100 Gbps Mellanox switches, but, using Mellanox splitter cables and slower 40 Gbps switches, broker nodes are provided with 50 Gbps connections while producer and consumer nodes get 10 Gbps connections. A single edge switch can connect 32 broker nodes or 128 producer/consumer nodes. We can thus build the complete data center using a two-level fat-tree of just 28 100 Gbps switches (12 edge and 16 core); seven edge switches connect to a total of fourteen 40 Gbps switches and five connect to the 157 brokers.

In designing this purpose-built data center, we wanted to avoid limiting potential advancements or upgrades during the lifetime of the data center. We designed it with double the anticipated requirements for network and storage bandwidth. The brokers were designed to accommodate the 32× speedup in two separate ways. First, we maintained the higher ratio of brokers to compute nodes, just as we did in the homogeneous design; second, we allocated four times the storage devices and bandwidth to each broker. Either one of these solutions on its own would have been adequate to accommodate the compute speedup. Furthermore, by giving 50 Gbps and 10 Gbps network connections to the broker and compute nodes, respectively, we have allowed them to grow to double their anticipated needs. In combination, we have given the data center the ability to adapt to unanticipated application speedups during its intended lifetime.

Our purpose-built data center incurs an equipment cost of US$27.9 million with a yearly power cost of US$1.4 million for a 3-year amortized yearly total cost of ownership of US$10.8 million. This is 16.6% lower than the TCO of the homogeneous data center while being better equipped to handle future accelerated compute.

8 RELATED WORK

We build on prior work that enabled and rapidly expanded AI and ML applications. Unlike most of the prior work, however, we explore the implications of accelerating AI computation and how it affects an end-to-end application flow. We present related work in five categories: (1) AI and ML benchmarking, (2) integrating AI and ML, (3) end-to-end application flow studies, (4) exploiting heterogeneity, and (5) edge data centers.

Benchmarking. MLPerf is one of the leading resources for benchmarking ML-related compute [28, 57]. It provides flexibility for benchmarking a variety of hardware across a variety of ML kernels, but it entirely ignores the issue of end-to-end application behavior and performance. In our work, we demonstrate the central importance of understanding the end-to-end application, showing that each ML kernel can constitute a relatively small portion of the pipeline and that truly optimizing ML performance requires a more holistic view of the system.

Integrated AI. In presenting the scale and deployment of ML workloads at Facebook, Hazelwood et al. acknowledged the importance of pre-processing data for training and emphasized its stress on storage, network, and CPU [54]. They acknowledged the potentially high latency of inference for top quality models but did not expose the overhead of pre- and post-processing. Nor did they discuss the resource requirements of streaming inference workloads. We emphasize both of these to show how they can pose a barrier to overall performance improvement from AI acceleration.

Microsoft recognizes the importance of latency in the data center particularly as it applies to deep neural networks [33]. Chung et al. presented Microsoft's Project Brainwave, which implements DNNs largely in FPGAs distributed throughout the data center, emphasizing the importance of accelerating increasingly complex DNNs [45]. In contrast, this work emphasizes the importance of the enabling code for AI and assumes that accelerating AI and ML will be successful, instead looking at its ultimate impact on the larger workflow.

End-to-End Application Flows. Though not specific to AI workloads, Kanev et al. offered a comprehensive look at the trends of warehouse-scale computing at Google [58]. They quantified data center "taxes"—overheads that come with applications but do not directly contribute to the end result, including compression, communication serialization, and remote procedure calls. We show that the brokers act as a tax, coordinating the activities of a distributed application.

Other work has broken down the end-to-end latency of requests, at various levels of granularity, ranging from evaluation of Internet speed and programming language patterns to operating system scheduling and memory latency [44, 61]. This more closely matches our contribution, though our analysis is restricted to latency within the data center and is focused specifically on the common communication base (Apache Kafka) of open source streaming frameworks in an effort to bring perspective to end-to-end AI application flows.

Heterogeneous Execution. Prior works sought to exploit the heterogeneity in a data center, producing benefits in speed, energy consumption, and operating costs [53, 64]. Where these papers sought to capitalize on unintentional heterogeneity (arising from workload co-location and, e.g., later upgrades), we extol the benefits of intentionally designing an on-premise data center with heterogeneous servers and network. We thereby add hardware cost savings to the existing benefits of data center heterogeneity.

Edge Data Centers. Hewlett Packard Enterprise recently demonstrated that cloud-based computing is often not the most cost-effective solution [48]. Their analysis showed for a well-utilized edge data center, TCO can be drastically lower than comparable capabilities in the cloud. Our work extends that idea, showing how the on-premise data center can be specifically tailored to the needs of AI applications.

9 CONCLUSION

It is easy to get caught up in the excitement of AI and ML; this work has brought context to those advancements, elucidating an AI tax, and serves as a call to action to address limiters of performance in realistic, edge data center deployments of AI applications. Streaming AI applications are only possible with the support of pre- and post-processing code, which is far from trivial in both latency and compute cycles and relies almost exclusively on the CPU for all of the processing. AI

applications will likely be composed of multiple inference stages, each with its own characteristics and overheads. And the enabling substrate for managing AI applications in a data center sees hot spots in both network and storage that could soon become bottlenecks if not addressed.

REFERENCES

[1] [n.d.]. AI-Benchmark. Retrieved July 25, 2019 from http://ai-benchmark.com/index.html.
[2] [n.d.]. Amazon.com: : Amazon Go. Retrieved July 25, 2019 from https://www.amazon.com/b?node=16008589011.
[3] [n.d.]. Apache Apex. Retrieved July 29, 2019 from http://apex.apache.org/.
[4] [n.d.]. Apache Flink: Stateful Computations Over Data Streams. Retrieved July 29, 2019 from https://flink.apache.org/.
[5] [n.d.]. Apache Kafka. Retrieved June 10, 2019 from https://kafka.apache.org/.
[6] [n.d.]. Apache Kafka. Retrieved June 12, 2019 from https://kafka.apache.org/documentation/streams/.
[7] [n.d.]. Apache Storm. http://storm.apache.org/.
[8] [n.d.]. Caffe | Deep Learning Framework. Retrieved July 21, 2019 from https://caffe.berkeleyvision.org/.
[9] [n.d.]. Coral. Retrieved July 24, 2019 from https://coral.withgoogle.com/.
[10] [n.d.]. Data Center Cooling Costs | Dataspan. Retrieved August 6, 2019 from https://www.dataspan.com/blog/data-center-cooling-costs/.
[11] [n.d.]. Deep Learning and Artificial Intelligence Solutions | NVIDIA. Retrieved July 16, 2019 from https://www.nvidia.com/en-us/deep-learning-ai/solutions/.
[12] [n.d.]. Docker - Build, Ship, and Run Any App, Anywhere. Retrieved June 12, 2019 from https://www.docker.com/.
[13] [n.d.]. Druid | Interactive Analytics at Scale. Retrieved July 29, 2019 from https://druid.apache.org/.
[14] [n.d.]. GitHub - baidu-research/DeepBench: Benchmarking Deep Learning operations on different hardware. Retrieved July 11, 2020 from https://github.com/baidu-research/DeepBench.
[15] [n.d.]. GitHub - basicmi/AI-Chip: A list of ICs and IPs for AI, Machine Learning and Deep Learning. Retrieved August 5, 2019 from https://github.com/basicmi/AI-Chip.
[16] [n.d.]. GitHub - davidsandberg/facenet: Face Recognition using Tensorflow. Retrieved January 9, 2019 from https://github.com/davidsandberg/facenet.
[17] [n.d.]. Google Cloud Platform Blog: Google Supercharges Machine Learning Tasks with TPU Custom Chip. Retrieved July 24, 2019 from https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html.
[18] [n.d.]. Habana Homepage - Habana. Retrieved June 17, 2019 from https://habana.ai/.
[19] [n.d.]. home. Retrieved July 11, 2020 from https://aimatrix.ai/en-us/index.html.
[20] [n.d.]. HPE Power Advisor. Retrieved July 30, 2019 from https://paonline56.itcs.hpe.com/?Page=Index#.
[21] [n.d.]. Inference - Habana. Retrieved June 17, 2019 from https://habana.ai/inference/.
[22] [n.d.]. Intel Unveils the Intel Neural Compute Stick 2 at Intel AI Devcon Beijing for Building Smarter AI Edge Devices. Retrieved June 30, 2019 from https://newsroom.intel.com/news/intel-unveils-intel-neural-compute-stick-2/.
[23] [n.d.]. Intel® Optane™ Technology. Retrieved July 31, 2019 from https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.
[24] [n.d.]. Intel® SSD DC P4510 Series (1.0TB, 2.5in PCIe 3.1 x4, 3D2, TLC) Product Specifications. Retrieved January 9, 2019 from https://ark.intel.com/content/www/us/en/ark/products/122573/intel-ssd-dc-p4510-series-1-0tb-2-5in-pcie-3-1-x4-3d2-tlc.html.
[25] [n.d.]. Intel® Xeon® Platinum 8176 Processor (38.5M Cache, 2.10GHz) Product Specifications. Retrieved January 9, 2019 from https://ark.intel.com/content/www/us/en/ark/products/120508/intel-xeon-platinum-8176-processor-38-5m-cache-2-10-ghz.html.
[26] [n.d.]. Logstash: Collect, Parse, Transform Logs | Elastic. Retrieved June 12, 2019 from https://www.elastic.co/products/logstash.
[27] [n.d.]. MLPerf. Retrieved July 28, 2019 from https://mlperf.org/.
[28] [n.d.]. MLPerf. Retrieved July 28, 2019 from https://mlperf.org/inference-overview/.
[29] [n.d.]. NVIDIA Deep Learning Accelerator. Retrieved July 19, 2019 from http://nvdla.org/.
[30] [n.d.]. On-Premise Data Centers: Coming Back or Heading Out? Retrieved July 30, 2019 from https://emconit.com/blog/on-premise-data-centers-coming-back-or-heading-out.
[31] [n.d.]. Open Source Search & Analytics · Elasticsearch | Elastic. Retrieved June 12, 2019 from https://www.elastic.co/.
[32] [n.d.]. Production-Grade Container Orchestration - Kubernetes. Retrieved June 12, 2019 from https://kubernetes.io/.
[33] [n.d.]. Project Brainwave - Microsoft Research. Retrieved July 12, 2019 from https://www.microsoft.com/en-us/research/project/project-brainwave/.
[34] [n.d.]. The Rise of Edge Data Centres - Data Economy. Retrieved July 30, 2019 from https://data-economy.com/the-rise-of-edge-data-centres/.
[35] [n.d.]. Samza. Retrieved July 29, 2019 from http://samza.apache.org/.

[36] [n.d.]. Slash Data-Ccenter Costs and Downtime by Using Coolan's TCO Model - TechRepublic. Retrieved July 30, 2019 from https://www.techrepublic.com/article/slash-data-center-costs-and-downtime-by-using-coolans-tco-model/.
[37] [n.d.]. Specifications - SN2000 Series - Mellanox Docs. Retrieved July 30, 2019 from https://docs.mellanox.com/display/sn2000pub/Specifications.
[38] [n.d.]. Stanford DAWN Deep Learning Benchmark (DAWNBench). Retrieved July 9, 2020 from https://dawn.cs.stanford.edu/benchmark/index.html.
[39] [n.d.]. TensorFlow. Retrieved June 12, 2019 from https://www.tensorflow.org/.
[40] [n.d.]. Video Analytics Market to Reach USD 25.4 Billion by 2026. Retrieved August 6, 2019 from https://www.marketwatch.com/press-release/video-analytics-market-to-reach-usd-254-billion-by-2026cisco-systems-inc-axis-communications-genetec-inc-2019-09-09.
[41] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*.
[42] Ken Birman and Thomas Joseph. 1987. *Exploiting Virtual Synchrony in Distributed Systems*. Vol. 21. ACM.
[43] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices* 49, 4 (2014), 269–284.
[44] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The mystery machine: End-to-end performance analysis of large-scale Internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. 217–231.
[45] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. 2018. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro* 38, 2 (2018), 8–20.
[46] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. *Training* 100, 101 (2017), 102.
[47] DataTorrent. [n.d.]. End-to-end "Exactly-Once" With Apache Apex. Retrieved July 30, 2019 from https://cdn.rawgit.com/dtpublic/website/b0c73294/blogs/End-to-end%20_Exactly-Once_%20_with%20Apache%20Apex%20-%20DataTorrent.htm.
[48] Hewlett Packard Enterprise. 2018. *HPE On-Prem vs. Amazon Web Services (AWS)*. Technical Report. Hewlett Packard Enterprise Company.
[49] Wanling Gao, Fei Tang, Lei Wang, Jianfeng Zhan, Chunxin Lan, Chunjie Luo, Yunyou Huang, Chen Zheng, Jiahui Dai, Zheng Cao, et al. 2019. AIBench: An industry standard internet service AI benchmark suite. *arXiv preprint arXiv:1908.08998*.
[50] Udit Gupta, Xiaodong Wang, Maxim Naumov, Carole-Jean Wu, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. 2019. The architectural implications of Facebook's DNN-based personalized recommendation. *arXiv preprint arXiv:1906.03109*.
[51] Kaylie Gyarmathy. [n.d.]. How to Reduce Latency Using Edge Computing. Retrieved July 30, 2019 from https://www.vxchnge.com/blog/how-data-center-reduces-latency.
[52] Michelle Hannula. [n.d.]. How Hybrid Cloud Simplifies Data Sovereignty Challenges | CIO. Retrieved July 30, 2019 from https://www.cio.com/article/3396631/how-hybrid-cloud-simplifies-data-sovereignty-challenges.html.
[53] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. 2017. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 625–638.
[54] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proceedings of the 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 620–629.
[55] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. 2018. AI benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European Conference on Computer Vision (ECCV'18)*. 0–0.
[56] Andrey Ignatov, Radu Timofte, Andrei Kulik, Seungsoo Yang, Ke Wang, Felix Baum, Max Wu, Lirong Xu, and Luc Van Gool. 2019. AI benchmark: All about deep learning on smartphones in 2019. *arXiv preprint arXiv:1910.06663*.
[57] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2019. MLPerf inference benchmark. Retrieved July 11, 2020 from https://edge.seas.harvard.edu/files/edge/files/mlperf_inference.pdf.

[58] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 158–169.

[59] Martin Kleppmann. [n.d.]. Apache Kafka, Samza, and the Unix Philosophy of Distributed Data. Retrieved July 30, 2019 from https://www.confluent.io/blog/apache-kafka-samza-and-the-unix-philosophy-of-distributed-data/.

[60] Charles E. Leiserson. 1985. Fat-trees: Universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers* 100, 10 (1985), 892–901.

[61] Jialin Li, Naveen Kr Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.

[62] Almudena Lindoso and Luis Entrena. 2009. Hardware architectures for image processing acceleration. In *Image Processing*. IntechOpen.

[63] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. 2016. Tabla: A unified template-based framework for accelerating statistical machine learning. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*. IEEE, 14–26.

[64] Jason Mars, Lingjia Tang, and Robert Hundt. 2011. Heterogeneity in "Homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters* 10, 2 (2011), 29–32.

[65] Robert Metzger. [n.d.]. Kafka + Flink: A Practical, How-To Guide. Retrieved July 30, 2019 from https://www.ververica.com/blog/kafka-flink-a-practical-how-to.

[66] Rajiv Onat. [n.d.]. Apache Storm and Kafka Together: A Real-time Data Refinery. Retrieved July 30, 2019 from https://hortonworks.com/blog/storm-kafka-together-real-time-data-refinery/.

[67] Keshav Pingali. 2019. A Case for Case Studies. https://www.sigarch.org/a-case-for-case-studies/.

[68] Carlo Regazzoni, Andrea Cavallaro, Ying Wu, Janusz Konrad, and Arun Hampapur. 2010. Video analytics for surveillance: Theory and practice [from the guest editors]. *IEEE Signal Processing Magazine* 27, 5 (2010), 16–17.

[69] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems*. 91–99.

[70] Daniel Richins, Dharmisha Doshi, Matthew Blackmore, Aswathy Thulaseedharan Nair, Neha Pathapati, Ankit Patel, Brainard Daguman, Daniel Dobrijalowski, Ramesh Illikkal, Kevin Long, et al. 2020. Missing the forest for the trees: End-to-end AI application performance in edge data centers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 515–528.

[71] Florian Schroff, Dmitry Kalenichenko, and James Philbin. 2015. FaceNet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 815–823.

[72] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. 2017. Inception-v4, Inception-ResNet and the impact of residual connections on learning. In *AAAI*, Vol. 4. 12.

[73] Peter Torelli and Mohit Bangale. [n.d.]. *Measuring Inference Performance of Machine-Learning Frameworks on Edge-class Devices with the MLMark™ Benchmark*. Report. EEMBC.

[74] Bob Wheeler. 2018. *Data Centers Accelerate AI Processing*. Technical Report. The Linley Group.

[75] Alex Woodie. [n.d.]. Understanding Your Options for Stream Processing Frameworks. Retrieved July 30, 2019 from https://www.datanami.com/2019/05/30/understanding-your-options-for-stream-processing-frameworks/.

[76] Xilinx. 2018. *Accelerating DNNs with Xilinx Alveo Accelerator Cards*. Technical Report. Xilinx, Inc.

[77] Fangjin Yang. [n.d.]. Building a Streaming Analytics Stack with Apache Kafka and Druid. Retrieved July 30, 2019 from https://www.confluent.io/blog/building-a-streaming-analytics-stack-with-apache-kafka-and-druid/.

[78] Kaipeng Zhang, Zhanpeng Zhang, Zhifeng Li, and Yu Qiao. 2016. Joint face detection and alignment using multi-task cascaded convolutional networks. *IEEE Signal Processing Letters* 23, 10 (2016), 1499–1503.

[79] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. 2019. AI matrix: A deep learning benchmark for Alibaba data centers. *arXiv preprint arXiv:1909.10562*.

Received July 2020; accepted November 2020