Scrooge: A Cost-Effective Deep Learning Inference System

Yitao Hu
University of Southern California
yitaoh@usc.edu

Rajrup Ghosh
University of Southern California
rajrupgh@usc.edu

Ramesh Govindan
University of Southern California
ramesh@usc.edu

Abstract

Advances in deep learning (DL) have prompted the development of cloud-hosted DL-based media applications that process video and audio streams in real-time. Such applications must satisfy throughput and latency objectives and adapt to novel types of dynamics, while incurring minimal cost. Scrooge, a system that provides media applications as a service, achieves these objectives by packing computations efficiently into GPU-equipped cloud VMs, using an optimization formulation to find the lowest cost VM allocations that meet the performance objectives, and rapidly reacting to variations in input complexity (*e.g.,* changes in participants in a video). Experiments show that Scrooge can save serving cost by 16-32% (which translate to tens of thousands of dollars per year) relative to the state-of-the-art while achieving latency objectives for over 98% under dynamic workloads.

CCS Concepts

• **General and reference** → **Performance**; • **Computing methodologies** → **Neural networks**; **Computer vision**; **Natural language processing**; • **Software and its engineering** → **Scheduling**.

Keywords

Cloud computing, deep learning inference, auto-scaling

1 Introduction

Using large labeled data-sets, powerful GPUs, and advances in algorithms, deep learning (DL) has revolutionized many fields [27]. To achieve this, DL research has developed neural network architectures suitable for tasks such as detecting or classifying objects in images (convolutional neural networks or CNNs [16, 26, 43]), recognizing speech (recurrent neural networks, RNNs [49]), or processing natural language (transformers [8, 48]).

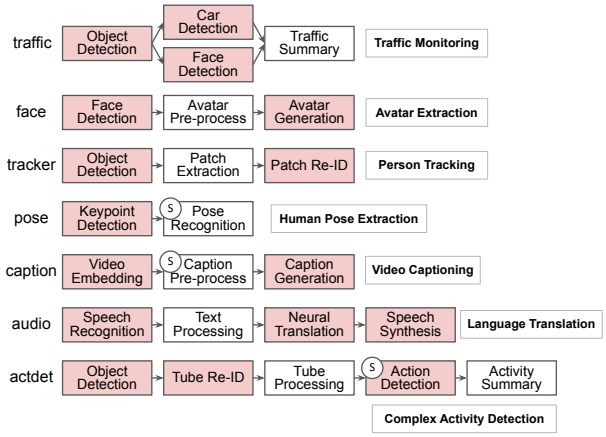


Figure 1: Examples of media applications. The pink rectangles represent GPU modules, while the white ones represent CPU modules. The circled S denotes a stateful module.

Inspired by these advances, application developers are starting to use DL models to build cloud-based applications that process video or audio streams in real-time [5, 6, 18, 40, 42, 45, 47, 54]. Figure 1 lists several examples of such *media processing applications* (henceforth just applications). Prior work has often modeled these as acyclic data-flow graphs (following [18], we call these *media DAGs* or mDAGs) in which a vertices represent either a GPU-based DL model invocation or a CPU computation, and an edge represents data transfer between the corresponding vertices. The input to an application is a stream of audio or video, and a single application can invoke one or more DL models. For example, the traffic monitoring application [55] uses an object detection model to identify pedestrians and vehicles in video streams, then uses other models to count vehicles or recognize faces. The language translation application in [18] translates an audio stream from one language to another using three DL models in sequence: speech recognition, neural translation and speech synthesis.

This paper considers the design and implementation of a system, called Scrooge, that provides mDAGs as a service hosted on the cloud. Scrooge is a generalization of a cloud-based inference serving system [5, 6, 40, 42, 45, 47, 54], which

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11.
https://doi.org/10.1145/3472883.3486993

applies DL models to client inputs (usually images and video). In Scrooge, a client sends a video or audio stream to an application mDAG hosted on the cloud; the mDAG processes the stream and returns the results to the client. Scrooge can host multiple mDAGs. It also multiplexes multiple client streams onto a single mDAG, such as when multiple clients concurrently request traffic monitoring.

Scrooge must satisfy two performance objectives of media processing. An *end-to-end latency objective* (or *latency SLO*) specifies a bound on the tail latency that the client can tolerate [5, 6, 42]. A *rate objective* ensures that the mDAG throughput matches the client's frame rate for video or audio.

Goal and Approach. The primary goal of Scrooge is to *minimize cost* while satisfying these performance objectives. Cloud providers today provide a range of GPU-enabled virtual machine (VM) configurations, at different price points (§2). To minimize cost, the mDAG service provider must choose, at each instant, the set of VM configurations that achieves this goal for its current workload. To do this, Scrooge uses three related ways to minimize cost: it finds the lowest cost VM configuration that can meet the performance objectives; it *packs* as much computation as possible into a VM by leveraging additional parallelism, especially for GPUs, whenever possible; it continuously adapts resource usage to workload changes and *input complexity* (the semantic richness in the audio or video input). This ensures just-enough resource usage, thereby also minimizing cost.

Contributions. In achieving minimum-cost mDAG serving, Scrooge makes the following contributions.

Scrooge develops more efficient methods to pack mDAGs onto VMs (specifically, into GPUs assigned to the VMs) than prior work. For DL-based workloads, prior work [5, 42] relies on batching to improve throughput and GPU utilization. Batching can adversely impact latency so the efficacy of this technique depends upon the latency objective. Recent work [33, 52] has explored spatial multiplexing, which seeks to exploit parallelism to improve throughput and GPU utilization. However, unless the degree of parallelism is chosen carefully, spatial multiplexing can actually degrade throughput and lower efficiency [18]. Scrooge uses a novel combination of batching and spatial multiplexing, by developing profiled models for each mDAG on each unique VM configuration. Each model represents the relationship between throughput and latency of that mDAG for that VM configuration, for different choices of batch sizes and the degree of parallelism.

Scrooge's second contribution is an optimization formulation that jointly, for a given workload, selects the lowest cost VM configuration while at the same time ensuring maximum packing efficiency. Unlike existing inference systems which either assume homogeneous hardware [42] or only consider one type of GPU [5, 41], Scrooge can reason about heterogeneous VM configurations. It uses the profiled models as input to a mixed-integer linear programming (MILP) formulation to derive the minimum cost configuration. To scale the optimization, Scrooge's optimization formulation leverages the elasticity of the cloud and simply selects the best fit VM configuration, decoupling this choice from placement of an mDAG module on a VM instance.

Scrooge identifies, and adapts rapidly to, new sources of dynamics that prior work has not considered. A serving system must deal with changing resource requirements resulting from client arrivals and departures. However, the processing requirements for an mDAG can also depend on *input complexity*, in two ways. First, depending on the complexity of the input video or audio stream, the data transmitted on an mDAG edge can vary over time, resulting in varying resource requirements at the receiving. For example, in the traffic monitoring application (Figure 1), the object detector extracts a bounding box of each pedestrian, and the face detector is applied to each bounding box independently. So, the processing requirements for face detection can increase linearly with the number of pedestrians. Second, input complexity can also impact the latency of processing a frame using DL models. Most prior work has considered CNN-based model architectures, whose execution time is independent of input complexity. In more recent model architectures, like RNNs and transformers, execution latency can depend upon input complexity. Scrooge develops lightweight methods to track input complexity, and minimally adapts inputs to its optimization formulation to adapt resource allocation as necessary to meet mDAG requirements while minimizing cost.

Evaluation Results. Using an implementation of Scrooge on a cluster of 16 GPUs (§4.1), we show that Scrooge is able to satisfy the end-to-end latency objective and rate objective under workload dynamics resulting from input complexity. To achieve comparable performance, Nexus [42], the state-of-the art, incurs a serving cost 1.160 to 1.319 times of Scrooge's (§4.2), even as it satisfies the latency SLO of 8% fewer frames than Scrooge. To put these numbers in perspective, Scrooge's cost advantages can result in savings of tens of thousands of dollars over a year. Scrooge's MILP formulation incurs less than 100 ms latency and produces lower-cost allocations (by up to 5% in some cases) than a greedy heuristic. Its allocations are off the optimal by about 3%, but computing the optimal using a brute-force technique is intractable for the problem sizes we consider. Spatial multiplexing reduces Scrooge's serving cost by 2% to 58%.

	VM types	Price ratio	TFlops ratio
GCP [12]	22	8.38	2.41
AWS [1]	19	7.79	2.41
Azure [30]	24	6.46	2.41

Table 1: Comparison between cloud providers.

2 Motivation

Three design dimensions distinguish Scrooge from prior work (§5): cost-awareness, packing efficiency, and adaptation to input complexity.

Scrooge must be cost-aware. Cloud providers today permit customers to choose from range of VM configurations. Table 1 shows the number of GPU-enabled VM configurations in three large cloud providers as of this writing: Google's GCP [12], Amazon's AWS [1] and Microsoft Azure [30]. Each provides 10-22 different GPU-enabled VM configurations, and the price ratio between the cheapest to the most expensive VM is 7-8. This ratio is *not* correlated with GPU resources; the ratio of highest to lowest GPU TFlops is 2.41.

The cost of a VM is also not correlated with performance. Table 2 shows the price per hour for VMs with two different GPU types. It also depicts the throughput of two models for object detection: SSD-Inception [19], and YOLO [39]. The ratio of throughputs are different for different models, and also different from the cost ratios.

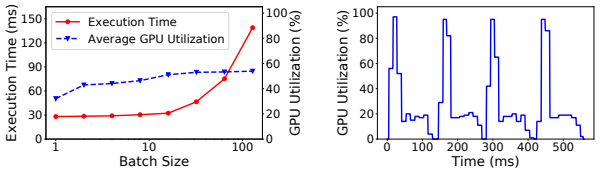
Takeaway. Scrooge must explicitly provision resources based *both* on cost and performance (throughput and latency). Some existing inference systems either assume homogeneous hardware [42] or only consider one type of GPU [5, 41].

Scrooge must pack DNN models efficiently to reduce cost. Prior work has considered two packing strategies: temporal multiplexing with batching, and spatial multiplexing.

Temporal multiplexing with batching. Temporal multiplexing uses time-slicing to share the GPU. It runs only one DL model at each instant, and switches between DL models in a round-robin fashion. To increase resource utilization, temporal multiplexing is used with batching [20], which increases throughput by combining multiple inputs into a single request. Batching can reduce I/O requests, as well as potentially invoke larger GPU kernels for DL computation to better leverage GPU's parallel compute capability [6, 11, 42]. Usually, a larger batch size leads to a higher throughput, but can also lead to higher latency per batch. For example, when increasing the batch size from 1 to 8, the throughput for SSD-Inception [19] increases from 40.98 img/sec to 67.52 img/sec, but the worst case latency also increases from 24 ms to 236 ms[1]. Therefore, existing inference systems [6, 42] pick a proper batch size to maximize inference throughput, while satisfying the user's latency objectives.

	P100	V100	Ratio
VM Pricing ($/h)	2.07	3.06	1.48
SSD-Inception [19] throughput	14.69	16.31	1.11
YOLO [39] throughput	22.17	26.72	1.21

Table 2: Pricing, hardware information and throughput for Microsoft Azure virtual machine with GPUs.



(a) **(b)**

Figure 2: (a) Execution time (blue line) and average GPU utilization (red line) for batch size from 1 to 128; (b) GPU utilization for four consecutive requests.

More generally, batching (a) exhibits diminishing returns with larger batch sizes and (b) utilizes the GPU in a bursty fashion. We are not the first to make these observations [33, 52]. To illustrate these, consider Figure 2, which shows the impact of batching for SSD-Inception on the P100 GPU (measurements obtained using NVIDIA's profiling tool [32]).

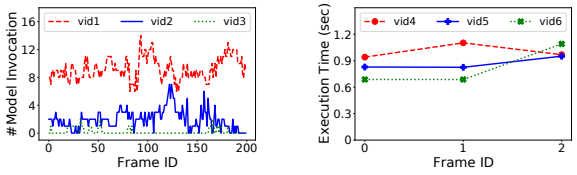
With increasing batch sizes (Figure 2a), GPU utilization levels off (blue line), but beyond a relatively small batch size of 16, latency increases almost linearly (red line). The batch size at which these diminishing returns occur depends on the DL model structure and the implementation of GPU kernels.

Figure 2b depicts GPU utilization across four consecutive DL model computations; GPU utilization is highly bursty . It reaches 100% for a short period of time, but remains below 20% for most of the time.

Spatial multiplexing. To further increase throughput, recent work [18, 33, 52] has used spatial multiplexing, employing GPU CUDA streams [13] to run multiple DL jobs on the same GPU concurrently. Concurrent execution can potentially introduce interference, in which the throughput of one DL model is impacted by another. To avoid this, prior work requires the developer to specify which jobs should be spatially multiplexed [33], uses trial-and-error to minimize interference [52], or explicitly profiles model resource usage [18].

Takeaway. One option that has not been explored in previous inference systems is a combination of batching and spatial multiplexing, which can pack models more efficiently, resulting in reduced cost. However, such an approach must

[1]The 24 ms worst case latency only includes a 24 ms execution time, while the 236 ms worst case latency includes a 118 ms execution time and another 118 ms maximum queuing delay.



(a) (b)

Figure 3: (a) Number of vehicle classifier invocation per frame; (b) Execution time per frame.

simultaneously avoid increased latency due to batching and interference due to spatial multiplexing.

Scrooge must adapt to input complexity. The semantic complexity in an audio or video stream can affect the compute latency of or the number of inputs to a node in an mDAG.

In an mDAG, a node that processes a video frame might emit a processed representation of the frame: *e.g.,* bounding boxes of objects. Depending on input complexity, the average number of elements in this processed representation per frame (the *scaling factor* [42]) can vary significantly. Figure 3a shows the scaling factor for a traffic monitoring application [17, 42] which includes three DL models: an object detector followed by a vehicle classifier or a human face detector. The scaling factor can vary significantly both within a video, and between videos. In Figure 3a, *vid*1 has an average scaling factor of 8.94 – the object detector can detect close to 9 vehicles per frame. *vid*3's average scaling factor is 0.37. Moreover, *vid*2's scaling factor ranges from 0 to 8 in a 200-frame window.

Previous inference systems [5, 42] mainly focused on scheduling CNN models, whose execution time is constant for different inputs. More recent architectures like RNNs and transformers can have a varying execution time depending on input complexity. For example, Figure 3b shows the execution time for an RNN model for video captioning, S2VT [49], for three different input video streams. The execution time for frames from different videos, as well as different frames from the same video, vary significantly, ranging from 0.686 to 1.102 seconds. We call this phenomenon *ballooning*, and represent the inflation/deflation of execution time due to changes in input complexity by a *ballooning factor* (§3.2).

Takeaway. Scrooge should explicitly track changing resource needs resulting from changes in input complexity; otherwise latency SLO may be violated significantly. For example, for *vid*2, when the scaling factor changes from 0 to 8, if the vehicle classifier or the face detector is provisioned for a low scaling factor, its execution latency will increase, potentially violating end-to-end latency. Llama [41] reports a similar observation on input complexity by scaling factor, but it does not consider ballooning (§5).

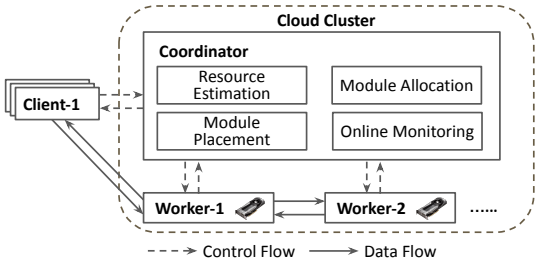


Figure 4: Scrooge overview.

3 Scrooge Design

In this section, we begin with an overview of Scrooge, followed by a detailed description of its components.

3.1 Overview

Terminology. Following prior work [18, 40, 42], Scrooge represents media processing applications with a data-flow graph called an mDAG. Each distinct media processing application has its own mDAG (Figure 1). In an mDAG, a vertex (called a *module*) represents either a CPU computation (*e.g.,* data preprocessing), or an invocation of a DL model on a GPU. An edge in the mDAG represents a data dependency. For instance, the video captioning mDAG includes 1 CPU module and 2 GPU modules: the GPU modules perform feature extraction and caption generation, and the CPU module guarantees the order of intermediate results, and maintains stateful variables[2] across frames.

Scrooge runs on a cloud cluster, and each client sends a stream of video or audio data to be processed by an mDAG. Each stream has a fixed input rate, expressed in terms of video or audio frames [46] per second. As with other cloud services, Scrooge attempts to ensure a *latency SLO* for each frame processed by a given mDAG: this represents the latency incurred by the frame across the entire mDAG, but does not include propagation and transmission latency between the client and the cloud cluster.

Scrooge aggregates all streams destined to a given mDAG in a *session*. Thus, a session includes streams from different clients, each with potentially different input rates, but all with the same latency SLO.[3] Scrooge assumes that the session input rate from a given client is fixed, and each client can join or leave the session independently at any time. Therefore, the total ingress rate for a session (the sum of all the input rates) can vary over time.

How Scrooge minimizes cost. For a given session with specific *total ingress rate* and latency SLO, the Scrooge *coordinator* schedules the session and allocates the lowest total

[2]Note that modules in Scrooge can be stateful unlike some prior work [6, 42].
[3]Today, cloud services do not expose latency SLO choices to individual end user clients, and we adopt the same model. We have left it to future work to explore a generalization where different clients can potentially express different SLOs.

cost computation resources (*i.e., worker* VMs) that can process the requests and satisfy the throughput and latency requirements (Figure 4). The coordinator directs clients to send streams directly to workers assigned to the corresponding session. Workers collectively execute various mDAG modules, while respecting data dependencies, then return the results to the clients. Each Scrooge worker is equipped with multiple CPU cores and exactly one GPU.

Scrooge achieves cost-effectiveness as follows:

► It packs as much of DL processing into a GPU as possible by combining batching and spatial multiplexing (§2). To do this, for each mDAG, it builds performance profiles that characterize the throughput-vs-latency tradeoffs for different degrees of batching and spatial multiplexing (§3.2).

► Its coordinator solves an optimization formulation that attempts to determine, for each mDAG session, at each instant, the ***type*** (Table 1) of VM worker to allocate, such that overall cost is minimized without violating SLOs for any mDAG. Scrooge's optimization scales because it decouples resource allocation (§3.3) from instance placement (§3.4): there are relatively few VM types to search over (§2), and once it has determined the type of VM, Scrooge determines which VM instance to allocate.

► Its runtime closely tracks mDAGs to determine the impact of input complexity (§3.5). Specifically, it tracks node processing latency to determine ballooning. It also tracks mDAG edges to determine scaling factor changes (§2). It then rapidly adapts resource usage by re-applying the optimization formulation. This ensures that, at each instant, Scrooge uses just-enough resources (Scrooge includes other optimizations to reduce cost, such as reusing pre-warmed up workers when possible, §3.6).

3.2 Estimating Resource Usage

Problem, Challenges and Approach. To reduce cost, Scrooge must efficiently pack mDAG modules into CPUs and GPUs. To do this, it needs to estimate each module's resource usage. Two challenges arise in doing this. First, the packing efficiency of GPU modules can be a function of both the degree of batching, and of spatial multiplexing (§2). Second, Scrooge must adapt to varying resource usage due to changes in input complexity (in addition to adapting to varying workload).

In general, there are two approaches to this problem: (a) proactively building performance models for each module through profiling, and (b) reactively adapting to variations in resource usage. A purely proactive approach is difficult, since that requires being able to accurately predict changes to input complexity. A purely reactive approach can result in unacceptable rates of latency SLO violations if the system is unable to increase a session's resource allocation quickly enough to meet demand.

Scrooge chooses a combination of these approaches. It profiles performance models for each GPU module that capture the relationship between throughput and latency for different degrees of batching and spatial multiplexing. At each instant it tracks: (a) the ingress rate for the session and (b) the scaling and ballooning factors for each module, and reactively adapts the session's resource allocations to match these.

The Throughput-Latency Profile. To enable efficient packing, Scrooge derives empirical models (by profiling offline[4]) of the relationship between throughput (work) and latency for a given mDAG module executing on a given worker (*e.g.,* a given type of VM with a fixed set of resources). For a CPU module, we simply profile the average latency of executing the module on a single input.

For a GPU module, such as SSD-Inception, the relationship between throughput and latency depends on two factors (§2): the batch size, and the degree of spatial multiplexing (we use the term *concurrency level* to denote this). To derive models of this relationship, Scrooge profiles the throughput and latency achieved by each GPU module on each worker, for different choices of batch sizes and concurrency levels. Spatial multiplexing models on a GPU can result in interference. Prior work [18] shows that, with higher spatial multiplexing, throughput increases linearly then drops at the point when interference manifests. Scrooge conservatively assumes the maximum point on this as the capacity of the GPU.

Figure 5 shows one sample profile: each curve represents a concurrency level, and each point represents a different batch size (doubling at each step from 1 to 128). In general, we can make two observations. First, at a given concurrency level, increasing the batch size increases throughput and latency up to a maximum, after which throughput drops but latency continues to increase. Thus, increasing the batch size from 1 to 8, its throughput increases from 40.98 img/sec to 67.52 img/sec. When the batch size is larger than 8, SSD-Inception invokes another type of GPU kernel, leading to a lower throughput. Second, higher concurrency levels can sustain higher maximum throughput, but at higher latency. For a batch size of 8, a concurrency level of 1 has a throughput of 67.52 img/sec, while a concurrency level of 4 (four batches of SSD-Inception with batch size of 8 run on the GPU concurrently) has a throughput of 77.51 img/sec.

Scrooge's profiler obtains each point in the profile by averaging over several hundred batches. As such, these profiles cannot capture resource need variability resulting from variations in input complexity. Scrooge reacts to changes in input complexity as described later in §3.6. We show in §4 that

[4]Most proposed inference systems use profiling [5, 6, 18, 40, 42, 54] for resource provisioning.

Figure 5: The throughput-latency profile for a GPU module. Each curve represents a different concurrency level. Each point corresponds to a different batch size, ranging from 1 to 128 (each batch is twice the previous).

the cost of profiling is very small, since these profiles can be reused across all clients of each mDAG.

Scrooge uses this profile in its allocation algorithm §3.3. Intuitively, that algorithm tries to search for a "split" of the latency SLO budget across mDAG modules such that the total cost of resources is minimized. The algorithm uses the profile as follows: if it allocates 0.2 seconds to the GPU module (the vertical line in Figure 5), then it uses the batch size and concurrency level point which has the highest throughput, but a latency lower than 0.2 sec. In Figure 5, this corresponds to a batch size of 4, and a concurrency level of 2. We now describe the allocation algorithm in more detail.

3.3 Allocation

Overview. Given an mDAG and its associated session (the aggregate client traffic for that mDAG), the primary challenge in Scrooge is to determine the minimum cost set of workers that can collectively satisfy the session's ingress rate, without violating the session's latency SLO. To tackle this challenge, Scrooge decouples *allocation* — determining the set of worker *types* needed for the mDAG from *placement* — determining the actual worker instances. By narrowing the search space, this decoupling allows Scrooge to find the minimum cost allocation using an MILP formulation, rather than resorting to heuristics as in prior work [5, 42]. §4 shows that, with this decoupling, for practical problem sizes, allocation can usually complete in about 100 ms, even though the problem is known to be NP-hard [2]. In this section, we discuss the allocation formulation; §3.4 discusses placement.

Allocation: An Example. Before we present the allocation algorithm, we use a simple example to explain the intuition behind it, and to demonstrate how it uses throughput-latency profiles. Consider an mDAG with two modules $A \rightarrow B$. For example A might be an object detector, and B might be a vehicle counting module. Table 3 and Table 4, respectively show the throughput-latency profiles for these two modules in tabular form, for two different worker types X and Y. Thus, for example, module A can sustain 60 fps on worker X with a batch size of 4 and concurrency level of 2. For the same *configuration* — we use the term configuration to denote one

Batch size	Concurrency	Latency	Throughput	Type
2	1	40	50	X
4	2	133	60	X
2	1	25	81	Y
4	2	95	84	Y

Table 3: Profile for module A on worker type X ($2 per hour) and worker type Y ($3 per hour). Latency is in milliseconds.

choice of batch size and concurrency level — module A can achieve 84 fps at a lower latency on Y.

Now, suppose Scrooge wishes to allocate resources to the mDAG to satisfy an end-to-end latency SLO of 300 ms. Suppose also that, at the current instant, the ingress rate to the mDAG is 80, and the scaling factor is 4.0. Thus, the *input rate* (the number of data items input per second to a module) for module A is 80, while the input rate for module B is 320. These, together with the two profiles, constitute the input to the allocation formulation.

The key idea behind the MILP solver is to try to split the 300 ms end-to-end latency between A and B, while minimizing cost and satisfying the throughput requirements. Consider, for example, an allocation in which module A uses the $< 4, 2 >$ configuration (batch size of 4, concurrency level of 2) on Y. This satisfies the input rate of 80 and incurs a latency of 145 ms (95 ms of processing and 50 ms delay to accumulate a batch of 4 frames). Then, B can use $2 < 2, 1 >$ configured instances of Y to satisfy its input rate, resulting in a total latency of 0.171 s and a total cost of $9 (per hour). However, this allocation does not have the lowest cost!

Instead, the cost-optimal allocation incurs a cost of $7.55 an hour at a total latency of 0.273 s (still within the SLO), as follows. For A, it assigns two workers. First, it assigns an X worker with a $< 4, 2 >$ configuration to satisfy an input rate of 60. This is the maximum sustainable rate on the X worker, and, since no other module can be scheduled on this worker, we call it a *full allocation*. Then, it assigns a Y worker with a $< 2, 1 >$ configuration to handle the residual input rate of 20. Since this only partially utilizes the capacity of the worker (it can sustain a rate of 81), we call this a *partial allocation*. Finally, for B it allocates two Y workers using a $< 4, 2 >$ configuration; one of these is a full allocation, and the other is a partial allocation.

This example highlights a few insights. The solver trades off latency to lower cost, while still ensuring the latency

Batch size	Concurrency	Latency	Throughput	Type
2	1	20	100	X
4	2	67	120	X
2	1	13	160	Y
4	2	40	200	Y

Table 4: Profile for module B on worker type X ($2 per hour) and worker type Y ($3 per hour). Latency is in milliseconds.

SLO. The optimal solution doesn't always select the cheaper worker type for every module; X is cheaper than Y, but is not selected for B. Finally, partial allocations can share a worker instance. For instance, both A's and B's partial allocations use worker type Y and can be placed on the same worker instance, avoiding resource fragmentation and thereby reducing cost. For this, Scrooge needs to assess worker capacity. Prior work [18] has demonstrated *input rate proportionality*: if a module has an input rate of f, but the worker can sustain a maximum rate of F, then the module uses f/F of the worker's capacity. Using this, we can see that A's partial instance of Y uses $20/81$ of the capacity, while B's uses $120/200$; so, those two can be co-located in one instance (§3.4).

MILP Formulation. We now formally describe the allocation algorithm.

Inputs. The inputs to the allocation algorithm, summarized in Table 5, include (a) an mDAG g, such that $g(m,n)$ denotes an edge between modules m and n, (b) the input rate r_i for the i-th module, (c) a set of k throughput-latency profile entries $p^k_{i,j}$ for the i-th module on the j-th worker type, where the k-th entry maps a configuration $< b^k_{ij}, e^k_{ij}, d^k_{ij} >$ to a profiled throughput and latency value (b is the batch size, e the concurrency level, and d is the batch execution time)[5], (d) the cost of the j-th worker type c_j (*e.g.*, as in Table 2), and (e) the latency SLO budget l_{SLO} for the entire mDAG.

Outputs. The algorithm has two outputs (Table 6). First, which worker types a module should use — the indicator variable x^k_{ij}, if set, denotes that module i should use configuration k on type j. Second, how many workers it should use — the indicator variable u^k_{ij}, if set, denotes how many instances of configuration k on type j i should use. For example, if u^k_{ij} is 4.6 for module i on worker type j, then i has 4 full allocations and one partial allocation (at 0.6 capacity). Thus, u^k_{ij} is a measure of the cluster capacity allocated to i (Table 6). These outputs are fed into the placement algorithm (§3.4).

The cost model. The allocation algorithm minimizes total cost for the entire mDAG, including both the CPU modules and the GPU modules as shown in Figure 1. In doing so, it assumes that the cost charged to a module is proportional to the capacity of the worker that it uses. In our example above, an instance of module i is only charged 0.6 of the cost of worker type j (c_j) for partial allocation, while total cost of fully allocated instances is $4c_j$.

Decision Variables. In addition to the output variables, the formulation uses three other decision variables (Table 6). Throughput rate t^k_{ij} is non-zero only for decision $x^k_{ij} = 1$ and captures the part of the total input rate of module i assigned to workers of type j using configuration k. It is related to $u^k_{i,j}$ as follows:

$$u^k_{ij} = \begin{cases} \frac{t^k_{ij}}{\frac{b^k_{ij} \times e^k_{ij}}{d^k_{ij}}}, & \text{if } x^k_{ij} = 1 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

The fraction in the denominator $\frac{b^k_{ij} \times e^k_{ij}}{d^k_{ij}}$ represents the rate at which a single worker can process inputs (d^k_{ij} is the profiled execution time for a batch size b^k_{ij}), so the complete term determines the number of workers allocated to i of type j at configuration k.

To meet the latency SLO, the allocation algorithm attempts to track the critical path latency on the mDAG. It does this using two decision variables. l_i for module i is the latency along the longest path in the mDAG from source module to module i. l_{max} is the critical latency for the entire mDAG.

Optimization formulation. An mDAG can be considered as partial order relations among the vertices represented by their edges [50]. If an edge goes from vertex m to n, an input can be processed by n after m. The allocation decision needs to find a worker type along with its configuration for each module, such that the longest latency for an input in an mDAG satisfy the SLO. We can formulate this scheduling problem as a mixed-integer linear programming (MILP) optimization satisfying linear constraints:

$$\min \quad \sum_i \sum_j c_j \times \sum_{k=0}^{|p_{ij}|} u^k_{ij} \quad (2a)$$

$$\text{s.t.} \quad \sum_j \sum_{k=0}^{|p_{ij}|} x^k_{ij} \times t^k_{ij} = r_i, \quad \forall i, \quad (2b)$$

$$l_m + x^k_{mj} \times W(l^k_{mj}) \le l_n, \quad \forall (k, j, g(m,n) = 1), \quad (2c)$$

$$l_{max} \le l_{SLO} \quad (2d)$$

In this formulation $W(l^k_{ij})$ is the worst-case latency, determined by the sum of the time to form a batch, and the profiled batch execution time d^k_{ij}. The batch creation time for a module i is dependent on selected batch size and the maximum rate that a worker type j can support. For example, the maximum rate supported by j with configuration <4, 2, 200> (batch size = 4, concurrency level = 2 and batch execution

Symbol	Description
g	mDAG, $g(m,n) = 1$ denotes an edge between m and n.
p^k_{ij}	k^{th} profile $<b^k_{ij}, e^k_{ij}, d^k_{ij}>$ for module i on worker type j.
c_j	Cost of worker type j.
r_i	Input rate for module i.
l_{SLO}	Latency SLO budget for the mDAG.

Table 5: Input parameters to MILP.

[5]Each module can be either a CPU module or a GPU module as shown in Figure 1. For CPU modules, batch size is always 1.

Variables	Description
$x_{ij}^k \in \{0,1\}$	Module i use worker type j.
$t_{ij}^k \in \mathbb{R}_{\geq 0}$	Throughput rate for module i on worker type j.
$u_{ij}^k \in \mathbb{R}_{\geq 0}$	Capacity allocated to module i on worker type j.
$l_i \in \mathbb{Z}_{\geq 0}$	Critical latency for module i in milliseconds.
$l_{max} \in \mathbb{Z}_{\geq 0}$	Critical latency for mDAG in milliseconds.

Table 6: Decision Variables in MILP formulation.

time = 200 ms) for i is $4 \times 2/0.2 = 40$ img/sec. It takes $4/40 = 100$ ms to create a batch of size 4. Thus, the worst-case latency for module i on this worker j is $100 + 200 = 400$ ms.

The optimization is subject to three constraints: **(2b)** the sum of the input rates for all module instances should equal total input rate for the module; **(2c)** if module m immediately precedes module n in an mDAG, the critical latency for module n from the source should be greater than sum of critical latency of module m from source and m's worst case latency; **(2d)** the critical latency l_{max} of the entire mDAG, the sum of critical latency of the last node and it's worst case latency, must be within the latency SLO.

Dealing with partial allocations. When profiling a module with a configuration k, the profile determines the highest throughput achievable for that configuration, and the corresponding latency. This latency includes the processing latency, and the queueing delay required to form a batch.

In a partial allocation, because the assigned input rate is lower, the queueing delay to wait for a batch to form is higher. Thus, the MILP output, if it has any partial allocations, when actually run on the cluster, may violate SLO. To avoid this, the allocation algorithm refines the *configuration* of the partial allocation in a second step. To do this, it replaces $W(l_{ij}^k)$ with

$$W(l_{ij}^k)^* = d_{ij}^k + \frac{b_{ij}^k}{t_{ij}^k} \tag{3}$$

in **2c** and re-runs the MILP for only this worker on this machine type, searching for a smaller batch size configuration that will preserve the latency SLO.

However, this makes **2c** a non-convex constraint because we use an inverse of a decision variable. The Gurobi [29] solver internally translates the constraint to linear approximations. For some inputs, the solver doesn't converge (but returns quickly when it determines it is infeasible); in these cases, Scrooge reverts to a greedy search for the best configuration for the partial allocation. Partial allocations result is slightly sub-optimal allocations (§4).

3.4 Placement

For the i-th module in an mDAG, the allocation algorithm determines that it should use k_i worker instances. For each instance, the algorithm determines (a) the type of worker and (b) the configuration to use. In general, $k_i - 1$ of these instances will be *full* — the module requires the entire capacity of the worker, and no other computation can run on that instance — and one other worker may be *partially* allocated.

Finding a complete instance. Scrooge is able to decouple placement from allocation because of elasticity of cloud services; it assumes that it can (almost always) find an unused instance of a given type of worker to allocate to the module. When it cannot find a worker of type j, it simply re-runs the allocation algorithm *without* that worker type.

Finding a partial instance. An mDAG service will serve a number of different mDAGs, and each mDAG's modules will likely occupy a worker partially. The *residual capacity* of a partial worker is determined by input rate proportionality (§3.1). To accommodate a partial instance, Scrooge searches for the best-fit residual capacity. If it cannot find such a worker, it requests a free instance from the cloud service, and allocates that worker to the module; the remaining capacity can be allocated to another module from the same mDAG, or another mDAG.

When are allocation and placement invoked? The Scrooge coordinator invokes allocation and placement when (a) a client leaves or joins and (b) when input complexity changes significantly. We discuss the latter below.

Re-routing client streams. If, as a result of these changes, the allocation changes and instances have to be allocated/de-allocated, the coordinator must re-route client streams to different worker instances (and possibly migrate state). We use mechanisms for these inspired by the literature on network function virtualization (NFV [25, 36, 51]) and omit the details of these mechanisms for brevity.

Dealing with worker failure. The coordinator uses the same mechanisms for client re-routing and state migration when a worker instance fails (this assumes that modules use resilient underlying cloud storage).

3.5 Reacting to Input Complexity Variation

Input complexity can either (a) increase the input rate to a module, or (b) inflate the execution time of a module.

Monitoring. Each Scrooge worker continuously monitors (a) the average execution time of the module (to determine ballooning §2), and (b) the average input rate to the module (to determine the scaling factor). When either of these changes significantly, the worker notifies the coordinator, which invokes the allocation and (possibly) placement algorithms.

Invoking the allocation algorithm. When the input rate to the i-th module changes, the coordinator simply invokes

the allocation algorithm for the mDAG, with the new observed input rate r_i (Table 5). When the execution time of the module increases, the coordinator re-runs the allocation algorithm, but with the observed execution latency, instead of the profiled execution latency.[6][7]

3.6 Other Cost Optimizations

Worker reuse and turn-off delay. Popular serving frameworks [45, 47] use just-in-time (JIT) compilation [14] to optimize DNN execution, so the first few frames can incur a significantly longer latency. To minimize JIT's impact on the latency SLO, Scrooge's placement always re-uses a worker that has already loaded the module, when possible. To maximize worker reuse, Scrooge places an unused worker in a *warmup* state for a configurable interval (called the *turn-off* delay) before returning the worker to the cloud service. In §4, we show that this optimization can reduce the rate of latency SLO violations.

Module pipelining. A Scrooge's GPU module requires some CPU operations for serializing input and output, resizing the inputs *etc.* Scrooge automatically pipelines CPU operations with GPU computation to maximize throughput, and to minimize end-to-end latency.

Early dropping. When the workload increases, inputs wait in a module's input queue until the coordinator increases resource allocation for the module. Before executing each queued input, the Scrooge worker uses the profiled (or recently observed) execution latency to determine if this input would violate the latency SLO given that it may have incurred queueing delay. If so, the worker drops the input without invoking the module on the input.

4 Evaluation

We compare Scrooge against the state-of-the-art inference system, Nexus [42], then quantify the importance of various design decisions.

4.1 Methodology

Implementation. We have implemented all the features of Scrooge described in §3. Each Scrooge worker runs in a separate container, and uses TF-Serving [45] to serve DL models on the worker's dedicated GPU. DL models are either written in Tensorflow [44] or converted[8] from other languages like PyTorch [37] or MXNet [4] using tools like ONNX [35]. Scrooge has 14,930 lines of Python code, which includes the client, coordinator and worker implementations. The mDAG library is an additional 8,617 lines of Python code.

Testbed. We deployed Scrooge on a cluster containing 16 VMs with a total of 16 GPUs (8 P100 GPUs and 8 V100 GPUs, each with 6 vCPUs, 112 GB RAM and a 128 GB SSD.) from Microsoft Azure [30]. We deliberately designed the testbed to demonstrate Scrooge's capability to minimize the total serving cost across heterogeneous workers.

Comparison Alternative.[9] Nexus [42] is a DL serving system that schedules DL execution in a GPU cluster, while satisfying client's latency SLO. Nexus profiles DL latency under different batch sizes, and derives the maximum batch size that can satisfy the latency SLO to maximize throughput. It monitors incoming workload, and adjusts scheduling and placement decisions accordingly.

Workloads. We use all seven mDAGs in Figure 1 in our experiments. Traffic monitoring (*traffic*) [55] uses SSD-Inception [19] to identify pedestrians and vehicles in video streams, then uses other models to count vehicles or recognize faces. Avatar extraction (*face*) uses a face detector and PRNet [10] to extract facial keypoints. Person tracking (*tracker*) detects pedestrians in video streams and then uses Triplet [15] for person re-identification across frames. Human pose extraction (*pose*) uses OpenPose [3] to detect human keypoints and then uses an action recognition model [38] to recognize human poses. Video captioning (*caption*) [49] uses S2VT to generate text description of video streams by analyzing the output of the fully-connected layer from AlexNet [26]. Language translation (*audio*) [18] translates audio script from English to German. Complex activity detection (*actdet*) [28] detects human activity across cameras.

The video and audio streams we use in our evaluations comes from public datasets or the application developer [9, 28, 38, 42, 49]. Our comparisons (§4.2) only use three of these mDAGs (*traffic*, *face* and *tracker*), because the other four (*pose*, *caption*, *audio* and *actdet*) either require stateful modules or take audio streams as input, neither of which Nexus supports. We use all seven mDAGs in other evaluations (§4.4).

Metrics. We focus on two metrics: (1) The *cost* to serve the media application charged by the cloud provider. For example, for the pricing shown in Table 2, if two P100 machines

[6] In theory, Scrooge can overreact if input changes significantly every frame. In practice, we expect input complexity to vary at human activity timescales of a few seconds (Figure 3a), not on every frame. Scrooge tracks execution time and input rate changes over sub-second windows, so it can avoid overreacting.

[7] More precisely, let's say that, for the selected batch size and concurrency level, the profiled execution time was l_p and the observed execution time is l_o. Then, Scrooge inflates all profiled execution latencies by the *ballooning factor* l_o/l_p and then re-runs the allocation algorithm.

[8] Future work can extend Scrooge to support pytorch models without conversion using torchserve [47].

[9] Table 7 tabulates related work. Of these, Nexus has been shown to be better than Clipper. Others do not support DAGs (*e.g.,* Triton), are focused on edge rather than cloud (Rim), build upon FaaS unlike Scrooge (*e.g.,* Llama), or have not made their code available (Inferline).

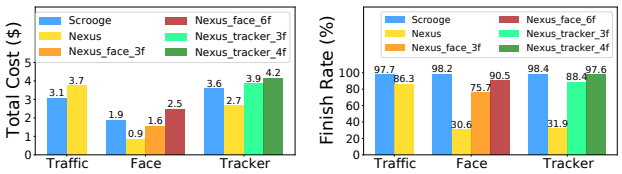


Figure 6: (a) The total cost for *traffic*, *face* and *tracker*; (b) The finish rate for *traffic*, *face* and *tracker*. For *face* and *tracker*, Nexus's total cost is lower than Scrooge's; however, its finish rate is less than 1/3 of Scrooge's. When finish rates are comparable, Nexus's total cost is 16-32% higher than that of Scrooge.

and one V100 machine are used, the total cost is $7.20/hr. (2) The *finish rate* for each session in the cluster, defined as the ratio between the number of frames that satisfy the latency SLO and the number of frames sent during a one second window. For example, for a time window of $[t, t+1)$, if Scrooge receives 20 frames, but processes only 19 of them within the latency SLO, the finish rate for $[t, t+1)$ is 95%. The former measures resource efficiency, while the latter measures the ability of the system to adapt to dynamics. A good DL serving system should have a high finish rate with a low total cost.

4.2 Comparison Results

Experiment details. To demonstrate Scrooge's capability to minimize the cost and to achieve high finish rate under dynamic workload, and to compare it with Nexus, we generate three workloads for *traffic*, *face* and *tracker*. Each workload lasts for 1000 seconds and has 12 clients, with clients joining and leaving at random times. Each client stream for *traffic* has a fixed frame rate between 2 and 5, for *face* between 12 and 30, and for *tracker* between 15 and 40. The highest ingress rate for each mDAG saturates our 16 node cluster, hence the choice of frame rates. All three workloads have a latency SLO of 400 ms to match Nexus evaluations.

Results. Figure 6 summarizes the results of the comparison; we discuss results for each mDAG separately.

Traffic. Scrooge outperforms Nexus for this mDAG. Nexus's end-to-end cost is 1.218 times of Scrooge's and its finish rate is 86.3% while Scrooge's finish rate is 97.7%. Figure 7 shows the time evolution of cost and finish rate during the 1000-sec window. The top panel shows the input rate for *traffic*'s modules in the second layer, which varies significantly with time, illustrating variability due to changes in input complexity and client dynamics. (Figure 3(a) more directly quantifies variability in input complexity; we use those videos in our experiments). The middle panel shows the real-time cost for Nexus and Scrooge; Nexus's cost is higher than Scrooge for most of the experiment. The bottom panel shows the real-time finish rate; Nexus exhibits significant SLO violations. Scrooge outperforms Nexus in both total cost and finish rate for three major reasons.

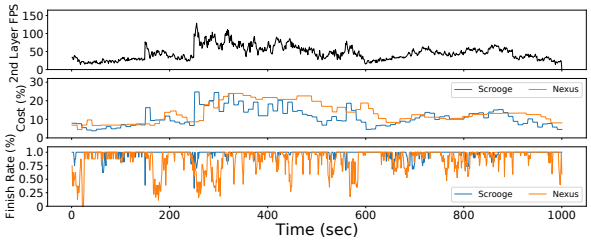


Figure 7: The cost and finish rate on workload of *traffic* for Scrooge and Nexus.

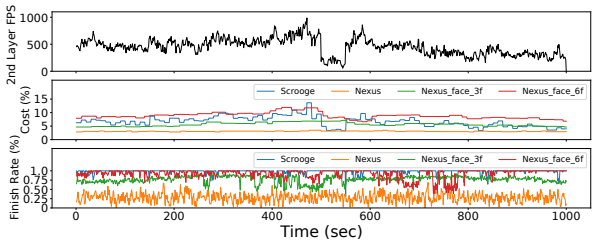


Figure 8: The cost and finish rate on *face* for Scrooge and Nexus.

First, on GPU modules, Nexus uses batching only, while Scrooge uses spatial multiplexing and batching to further increase the maximum throughput each machine can support, leading to a lower total cost. In theory, Scrooge should, at every instant of time, have a higher cost than that of Nexus. In practice, due to experimental variability, the ingress rates at each instant for each system are not identical; Scrooge's ingress rates are slightly higher sometimes, resulting in occasionally higher cost Figure 7. §4.4 explores the impact of Scrooge's spatial multiplexing in greater detail.

Second, Scrooge decouples placement from allocation, so can derive a near-optimal solution using the Gurobi solver, while Nexus achieves scheduling and placement using a greedy heuristic, leading to potentially sub-optimal solutions. In §4.4, we quantify the optimality and execution time of Scrooge's decoupling strategy.

Third, our experimental testbed is deliberately heterogeneous (§4.1). Scrooge's allocation algorithm takes workers heterogeneity into account, while Nexus's scheduling algorithm implicitly assumes homogeneous hardware.

Face. As shown in Figure 6, for *face*, Nexus's end-to-end cost is only 0.456 times of Scrooge's. However, Nexus's finish rate was 30.6% — nearly 70% of the requests to Nexus either violated the latency SLO or were never received by the client for the final output. Meanwhile, Scrooge's 98.2% finish rate is much higher than Nexus's.

In addition to the reasons we mentioned above, Scrooge outperforms Nexus for *face* because Scrooge profiles and

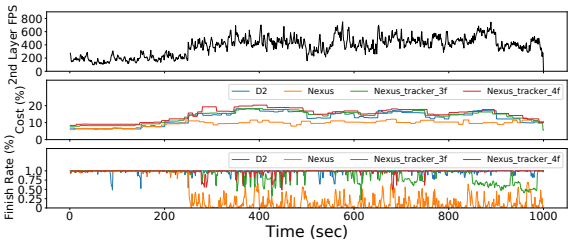


Figure 9: The cost and finish rate on workload of *tracker* for Scrooge and Nexus.

provisions resources for both CPU and GPU modules, while Nexus focuses on GPU modules and executes CPU-based computation[10] in its front-end [42]. However, for *face*, the CPU module can be compute-intensive. In our evaluation, Nexus allocated only one front-end for *face*'s CPU module, which resulted in long queues at front-end leading to increased latency SLO violations.

To verify this hypothesis, we manually increased the number of front-ends for Nexus from 1 to 3 then 6, denoted as $Nexus\_face\_3f$ and $Nexus\_face\_6f$. Figure 6 and Figure 8 shows the overall total cost and finish rate, as well as the real-time cost and finish rate for $Nexus\_face\_3f$ and $Nexus\_face\_6f$. By increasing the number of front-ends from 1 to 3, Nexus's finish rate increased from 30.6% to 75.7%, suggesting that 3 front-ends aren't enough for this workload. Nexus requires a minimum of 6 frontends to avoid queuing for the CPU module in *face*. Thus, $Nexus\_face\_6f$ always uses 6 workers [11] during the 1000-sec window. With 6 frontends, Nexus's finish rate is 90.5%, and its total cost is 1.319 times of Scrooge's. Thus, for *face*, Nexus requires significant trial-and-error to determine the optimal number of front-ends, and even then has lower finish rate and higher cost than Scrooge.

Tracker. Like *face*, *tracker* also has a CPU module to extract image patches from the object detection results. With a single front-end, Nexus's cost is only 0.740 times of Scrooge's, but its finish rate is 31.9%. Therefore, we manually increased Nexus's number of frontends for *tracker*'s CPU module from 1 to 3 then 4, denoted as $Nexus\_tracker\_3f$ and $Nexus\_tracker\_4f$, as shown in Figure 6 and Figure 9. Nexus requires at least 4 frontends to avoid queuing at the front-ends. With these, Nexus's finish rate is 97.6%, and its total cost is 1.160 times Scrooge's.

Summary. Nexus' cost is 16-32% higher than that of Scrooge. While this may seem modest, this translates to significant cost savings for a long-lived inference system. Consider the *traffic* mDAG in Figure 6. During our approximately 15 minute run with just 12 clients, the cost

[10]In the Nexus architecture, front-ends receive requests from clients and perform any necessary CPU processing, while backends serve DL models.
[11]For CPU modules, Scrooge and Nexus is charged at the rate of a non-GPU enabled worker, $0.83/h.

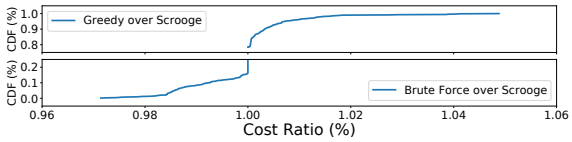


Figure 10: The cost ratio of Scrooge over optimal cost derived from brute force.

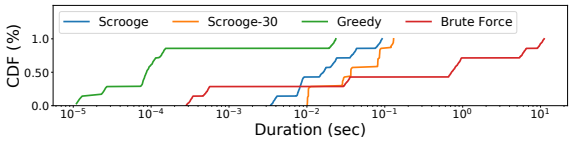


Figure 11: Latency distribution for allocation decisions

savings from Scrooge was $0.6. Over a year, even with this modest workload, the total cost savings for a cluster of 16 VMs running the *traffic* mDAG is over $21K! At larger scales (~1000 VMs), Scrooge can theoretically save over $1M over a year. Thus, we argue that careful, cost-effective resource allocation is especially important for DL-based media processing workloads.

4.3 The Importance of Cost-Aware Allocation

MILP-based allocation is essential. We now explore whether, instead of using MILP: (a) we could have searched brute-force for the optimal allocation and (b) we could have used a fast greedy heuristic.

Figure 10 plots the distribution of the cost of the allocations generated by Scrooge, relative to these two other approaches. This distribution is over all the allocation decisions we make for the experiments in §4.2. This figure, whose y-axis is truncated for clarity, shows that Scrooge generates the same cost configurations as the brute-force optimal about 85% of the time. However, it deviates from the optimal by less than 3% in the rest of the cases. This is due to partial allocations. In a partial allocation, our algorithm re-runs the MILP formulation with a non-convex constraint, relying on the underlying solver to use approximations to converge, and falling back to a greedy heuristic when the solver cannot converge (§3.3). This is the sole cause for the sub-optimality (we have verified this by inspection). However, brute force is clearly infeasible: brute-force can take over 10 s to compute the optimal solution, while MILP always converges in less than 100 ms, and often faster (Figure 11).

Our experiments only use 2 VM types mainly due to budget limits; however, even with 30 VM types — more than the number of VM types in Table 1 — our MILP completes within 130 ms across all our chains (line labeled Scrooge-30 in Figure 11). At this scale, brute force is unable to determine an allocation on our server.

Figure 10 also depicts the performance of a greedy heuristic, which selects the module (and associated worker type and configuration) whose ratio of throughput gain to the product of cost and latency increase is highest. This runs faster than MILP of course (Figure 11), and produces solutions with identical cost 80% of the time. However, as Figure 10 shows, it incurs sub-optimal costs (up to 5% higher in some cases) for the rest of the decisions.

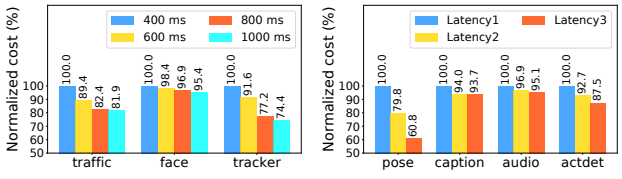
For this reason, Scrooge uses MILP-based allocation; it is tractable in our setting, and provides lower cost allocations than our greedy heuristic.

Less stringent SLOs lead to lower cost allocations. To evaluate the impact of latency SLO on total cost, we run Scrooge under 4 different latency SLOs. Figure 12 shows the normalized cost of Scrooge under different latency SLOs. In Figure 12a, when the latency SLO increases from 400 ms to 1000 ms, the total cost reduced up to 18.14% for *traffic* mDAG, 4.63% for *face* and 25.59% for *tracker*. In Figure 12b, for the other four mDAGs, a larger latency SLO can reduce the total cost up to 39.20%, 6.29%, 4.87% and 12.54% respectively.

A larger latency SLO increases the latency budget of each module in the mDAG, and Scrooge can more aggressively batch and/or spatially multiplex to increase throughput when latency budget is larger (Figure 5). Specifically, for *traffic* mDAG, when increasing latency SLO from 400 ms to 1000 ms, Inception's average batch size increased from 1.91 to 7.78, its average concurrency level decreased from 2.77 to 2.06 [12], and its average throughput increased from 24.9 to 33.6.

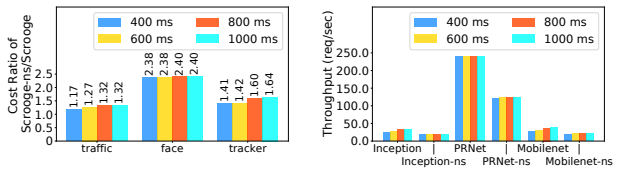
Not all mDAGs benefit equally from relaxing the latency SLO; the gains depend on the design of DL models. For example, by increasing latency SLO from 400 ms to 1000 ms, the *tracker* mDAG can reduce 25.59% of cost, while the *face* mDAG only benefits by 4.63% of cost. Figure 13b explains why — the average throughput of Mobilenet from the *tracker* mDAG increased from 27.9 to 38.2, while the average throughput of PRNet from the *face* mDAG only increased from 239.9 to 240.8.

[12] A larger latency budget might decrease module's batch size or concurrency level, but its throughput will always increase.



(a) (b)

Figure 12: (a) The normalized cost of Scrooge under different latency SLOs for *traffic*, *face* and *tracker*; (b) The normalized cost of Scrooge under different latency SLOs for *pose*, *caption*, *audio* and *actdet*.



(a) (b)

Figure 13: (a) The cost ratio of Scrooge-ns over Scrooge under various latency SLO for *traffic*, *face* and *tracker*; (b) The average throughput for Inception, PRNet and Mobilenet under various latency SLO.

4.4 Justifying Design Decisions

Spatial multiplexing is crucial. We now compare Scrooge against Scrooge-no-spatial (Scrooge-ns), which uses a fixed concurrency level of 1.

Figure 13a shows the cost ratio between Scrooge-ns and Scrooge under various latency SLO for three mDAGs used in §4.2. For *traffic*, the cost ratio ranges from 1.17 to 1.32 (14.48%-24.36% savings). For *face* and *tracker*, cost savings ranges are 57.90%-58.28% and 29.16%-39.10% respectively. Figure 13b explains the impact of relaxing the SLO budget. For *traffic*'s Inception and *tracker*'s Mobilenet, throughput gains are evident in Scrooge, but less so in Scrooge-ns. This suggests that spatial multiplexing is *essential* for exploiting relaxed SLOs.

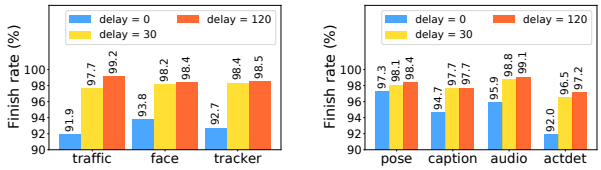
The other four chains (*pose*, *caption*, *audio*, and *actdet*) confirm these observations (omitted for brevity). Of these, *caption* shows the least cost reductions from relaxed SLOs; its RNN-based S2VT model only accepts a batch size of 1.

Profiling costs are negligible. Scrooge explicitly chose to profile mDAGs offline. Profiling enables it to get good initial estimates of mDAG resource needs. For the mDAGs we used in this paper, Scrooge's profiling cost was $7.67. These profiles can be used in perpetuity, unless the mDAGs change. To put this number in content, the cost of running the *traffic* mDAG for just *one* day is $267.

Turnoff delay can increase finish rate. In this experiment, we quantify turnoff delay's influence on finish rate for each mDAG (§4.3). Figure 14a shows the *finish rate* for *traffic*, *face* and *tracker* under three turnoff delay values: 0, 30 and 120 sec. When there is no turnoff delay (blue bars in Figure 14a), Scrooge's finish rate ranges from 91.9% to 93.8%. This is due to its input complexity; when the scaling factor increases sharply, the finish rate drops until Scrooge is able to provision additional resources. It can take up to 5 seconds [14, 18] to load the model weight into GPU memory and finish just-in-time compilation. Turn-off delay avoids this overhead thereby reducing impact on finish rate: a turnoff delay of 30 seconds can increase the finish rate to over 97%, while a turnoff delay of 120 seconds can further increase to over 98%.

System	VM/FaaS	Heterogeneity	Media Type	Cost Optimization	Input Complexity	Multiplexing	CPU Module
Scrooge	VM	yes	video and audio	optimization	scaling and ballooning	spatial + batching	yes
Nexus [42]	VM	no	video	heuristic	scaling	batching	no
Clipper [6]	VM	no	video and audio	no	no	any	no
Rim [18]	VM/Edge	yes	video and audio	no	no	spatial	yes
Triton [34]	VM	no	video and audio	no	no	any	yes
InferLine [5]	VM	no	video	heuristic	scaling	batching	yes
INFaaS [40]	VM	no	video	heuristic	scaling	batching	no
OoO [23]	VM	no	video	no	no	spatial	no
MArk [54]	VM and FaaS	no	video and audio	heuristic	scaling	batching	no
Llama [41]	FaaS	no	video	heuristic	scaling	batching	yes

Table 7: Comparison of Inference Systems



(a) (b)

Figure 14: (a) The finish rate of Scrooge under different turnoff delay for *traffic*, *face* and *tracker*; (b) The finish rate of Scrooge under different turnoff delay for *pose*, *caption*, *audio* and *actdet*.

Figure 14b shows the *finish rate* for the other four mDAGs from Figure 1 under three turnoff delay values. For *pose*, Scrooge achieves high finish rate even with a turnoff delay of 0 sec. This is because *pose*'s resource requirements are not significantly affected by input complexity. For *caption*, *audio* and *actdet*, a turnoff delay of 30 seconds can increase the finish rate to over 96%, while a turnoff delay of 120 seconds can further increase to over 97%.

5 Related Work

DNN Inference Systems. DL model serving has been explored a fair bit in recent years, but Scrooge occupies a unique point in the design space. It differs from Nexus [42] in supporting CPU modules, heterogeneous hardware, using spatial multiplexing and dealing with ballooning exhibited by newer model architectures. Inferline [5] and INFaaS [40] only supports a single GPU type and uses batching. MArk [54] focuses on provisioning GPU modules only and uses heuristic methods to make resource allocation decisions. Moreover, InFaaS and MArk do not support module DAGs. Clipper [6] does not adapt resource allocation to input complexity changes, and does not use spatial multiplexing. Rim [18] also does not adapt to input complexity changes, and performs heuristic allocation (but uses spatial multiplexing and supports heterogeneous GPUs). Triton [33], OoO [23] and Space-Time [22] combine batching and spatial multiplexing, but do not adapt to input complexity changes and do not explicitly seek to optimize cost. Llama [41] supports inference on FaaS platforms, which can have high latency. It supports scaling factor adaptation, but not ballooning factor adaptation.

DL Training Systems. Scrooge is also inspired by DL training systems. Gandiva [52] identified the importance of spatial multiplexing for training, Gandiva$_{fair}$ the importance of supporting heterogeneous GPU hardware for training and DS2 [24] for reactively scaling stream processing in response to changes in processing time.

Dataflow Systems. Prior work has developed dataflow languages and run-time support [7, 21, 31, 53] for cloud-based applications. As in Scrooge, these approaches represent applications as DAGs, and automatically manage resources and schedule computations. However, these systems focus on throughput for general dataflow workloads, while Scrooge is specialized for DL inference and minimizes the total serving cost while ensuring latency and finish rate constraints.

6 Conclusions

In the paper, we presented a cost-efficient deep learning inference system named Scrooge, which packs DL workload efficiently to maximize inference throughput while satisfying DL application's latency and rate objectives. Scrooge decouples placement from allocation to derive the most cost-efficient scheduling for most cases, and continuously adapts resource allocation to workload changes, which ensures system performance under dynamic workload. Experiments show that Scrooge can save 16 – 32% of serving cost over the state-of-art while achieving latency objectives for over 98%.

Acknowledgments

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Amazon Web Services 2020. https://aws.amazon.com/.
[2] D. Bernstein, M. Rodeh, and I. Gertner. 1989. On the complexity of scheduling problems for parallel/pipelined machines. *IEEE Trans. Comput.* 38, 9 (1989), 1308–1313. https://doi.org/10.1109/12.29469

[3] Zhe Cao, Gines Hidalgo, Tomas Simon, Shih-En Wei, and Yaser Sheikh. 2019. OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. *IEEE transactions on pattern analysis and machine intelligence* 43, 1 (2019), 172–186.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[5] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 477–491.

[6] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[7] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[9] EarthCam 2021. https://www.earthcam.com/.

[10] Yao Feng, Fan Wu, Xiaohu Shao, Yanfeng Wang, and Xi Zhou. 2018. Joint 3d face reconstruction and dense alignment with position map regression network. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 534–551.

[11] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. 2018. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*. 1–15.

[12] Google Cloud Platform 2020. https://cloud.google.com/.

[13] GPU Pro Tip: CUDA 7 Streams Simplify Concurrency 2021. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

[14] Mark Harris. 2013. CUDA Pro Tip: Understand Fat Binaries and JIT Caching. https://devblogs.nvidia.com/cuda-pro-tip-understand-fat-binaries-jit-caching/.

[15] Alexander Hermans, Lucas Beyer, and Bastian Leibe. 2017. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737* (2017).

[16] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861

[17] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Shivaram Venkataraman, Paramvir Bahl, Matthai Philipose, Phillip B. Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 269–286. https://www.usenix.org/conference/osdi18/presentation/hsieh

[18] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. 2021. Rim: Offloading Inference to the Edge.. In *Proceedings of the 6th ACM/IEEE Conference on Internet of Things Design and Implementation*.

[19] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. 2017. Speed/accuracy trade-offs for modern convolutional object detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 7310–7311.

[20] Inference: The Next Step in GPU-Accelerated Deep Learning 2015. https://developer.nvidia.com/blog/inference-next-step-gpu-accelerated-deep-learning/.

[21] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. 2007. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. 59–72.

[22] Paras Jain, Xiangxi Mo, Ajay Jain, Harikaran Subbaraj, Rehan Sohail Durrani, Alexey Tumanov, Joseph Gonzalez, and Ion Stoica. 2018. Dynamic Space-Time Scheduling for GPU Inference. *arXiv preprint arXiv:1901.00041* (2018).

[23] Paras Jain, Xiangxi Mo, Ajay Jain, Alexey Tumanov, Joseph E Gonzalez, and Ion Stoica. 2019. The OoO VLIW JIT Compiler for GPU Inference. *arXiv preprint arXiv:1901.10008* (2019).

[24] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. 2018. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 783–798.

[25] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/conference/nsdi18/presentation/katsikas

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*. 1097–1105.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[28] Xiaochen Liu, Pradipta Ghosh, Oytun Ulutan, BS Manjunath, Kevin Chan, and Ramesh Govindan. 2019. Caesar: cross-camera complex activity recognition. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*. 232–244.

[29] Gurobi Optimization LLC. 2020. Gurobi Optimizer Reference Manual. http://www.gurobi.com.

[30] Microsoft Azure 2020. https://azure.microsoft.com/en-us/.

[31] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[32] NVIDIA System Management Interface 2012. https://developer.nvidia.com/nvidia-system-management-interface.

[33] NVIDIA Triton Inference Server 2021. https://developer.nvidia.com/nvidia-triton-inference-server.

[34] NVIDIA's TensorRT 2019. https://developer.nvidia.com/tensorrt.

[35] ONNX 2021. https://onnx.ai/.

[36] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[37] PyTorch 2021. https://pytorch.org/.

[38] Realtime action recognition 2019. https://github.com/felixchenfy/Realtime-Action-Recognition.

[39] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: an Incremental Improvement. *arXiv* (2018).

[40] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. 2019. INFaaS: A model-less inference serving system. *arXiv preprint arXiv:1905.13348* (2019).

[41] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. 2021. Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines. *arXiv preprint arXiv:2102.01887* (2021).

[42] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[43] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, Inception-Resnet and the Impact of Residual Connections on Learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[44] TensorFlow 2021. https://www.tensorflow.org/.

[45] TensorFlow Serving 2021. https://github.com/tensorflow/serving.

[46] The Private Life of MP3 Frames 2021. http://id3lib.sourceforge.net/id3/mp3frame.html.

[47] TorchServe 2021. https://pytorch.org/serve/.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[49] Subhashini Venugopalan, Marcus Rohrbach, Jeffrey Donahue, Raymond Mooney, Trevor Darrell, and Kate Saenko. 2015. Sequence to sequence-video to text. In *Proceedings of the IEEE international conference on computer vision*. 4534–4542.

[50] Kent Wilken, Jack Liu, and Mark Heffernan. 2000. Optimal instruction scheduling using integer programming. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2000*. 121–133. https://doi.org/10.1145/349299.349318

[51] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 299–312. https://www.usenix.org/conference/nsdi18/presentation/woo

[52] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. 2018. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 595–610. https://www.usenix.org/conference/osdi18/presentation/xiao

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[54] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 1049–1062.

[55] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. 2017. Live video analytics at scale with approximation and delay-tolerance. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 377–392.