Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics

Anonymous Author(s)

ABSTRACT

Deep neural networks (DNNs) power modern visual analytics systems due to their high accuracy. However, DNNs can be expensive to execute, often taking billions of floating point operations. To improve DNN inference, researchers have proposed systems and optimizations under the assumption that executing DNNs is the bottleneck. However, through a novel measurement study, we show that the *preprocessing of data* (e.g., decoding, resizing) can be the bottleneck in many visual analytics systems on modern hardware. Furthermore, this bottleneck varies by resolution and file format. Based on these observations, we develop SysX, a preprocessing-aware runtime engine that automatically optimizes DNN inference over visual databases. SysX introduces two novel contributions, leveraging two key observations: 1) preprocessing and DNN execution can be pipelined and 2) the behavior of a DNN is tightly coupled with its input format. SysX's first contribution is a preprocessing-aware cost model for selecting DNNs for visual analytics. SysX's second contribution is to *jointly optimize* preprocessing and DNN execution by 1) optimizing common vision preprocessing operations, 2) leveraging natively present, low-resolution visual data, and 3) partially decoding visual data. However, naively using low-resolution data degrades accuracy, so we provide a DNN training procedure that recovers high accuracy on low-resolution data. We evaluate SysX on six visual datasets and show that its optimizations can achieve up to 4.6× *end-to-end* throughput improvements at a fixed accuracy.

ACM Reference Format:
Anonymous Author(s). 2020. Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics. In *Proceedings of SIGMOD '20: ACM SIGMOD/PODS International Conference on Management of Data (SIGMOD '20).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Deep neural networks (NNs) now power a range of visual analytics tasks and systems [8, 33, 37] due to their high accuracy. When deploying these DNNs, application developers face different resource constraints. As a result, these systems often provide methods for trading off accuracy and throughput [8, 33, 37]. This trade off is inherent in DNNs: higher accuracy DNNs generally require more computation.

Even executing smaller DNNs can be computationally intensive, requiring up to many billions of floating point operations. As a result, researchers have built systems to optimize DNN execution for visual analytics [8, 33, 37, 51]. These systems improve the Pareto frontier of accuracy and throughput by leveraging a range of techniques. These techniques range from model compression [27, 32] to the use of *specialized NNs* [37]. Specialized NNs are inexpensive NNs that are used to approximate more expensive target DNNs [12, 33, 42]. They can be used to filter inputs so the target DNN will be executed fewer times, improving throughput. This technique is used by systems such as NoScope [37] and Tahoma [8]. These optimizations have led to orders of magnitude in throughput improvements in executing DNNs. For example, the specialized NNs used in NoScope and Tahoma execute at tens of thousands of images per second (im/s).

While these speedups are impressive, performance measurements in this existing work largely ignore a key bottleneck in *end-to-end* DNN inference: preprocessing, or the process of reading, decoding, and transferring image data to DNN accelerators. In the first measurement study of its kind, in this paper, we show that preprocessing costs often *dominate end-to-end DNN inference* when carefully using advances in hardware accelerators and compilers. For example, simply decoding JPEG compressed images achieves a throughput of only 686 images/s on an inference optimized `g4dn.xlarge` Amazon Web Services (AWS) instance, which has the latest NVIDIA line of inference optimized GPUs, the T4. On the `g4dn.xlarge`, the throughput of simply decoding images is 3× slower than the throughput of ResNet-50, a DNN that is widely used for benchmarking and has historically been considered expensive [1, 20, 21]. These trends additionally hold for energy usage (Section A). Furthermore, this gap widens for specialized NNs and will continue: next generation accelerators reportedly outperform the T4 by over 7× [62].

As accelerators and systems that use these accelerators become more efficient [3, 24, 62], we argue that it will becoming increasingly critical to account for the *end-to-end* cost of DNN inference, in particular including the preprocessing time.

In light of these observations, we examine opportunities for more principled joint optimization between preprocessing and DNN execution. We leverage two insights: 1) DNN execution and preprocessing can be pipelined and 2) the accuracy and throughput of a DNN is closely coupled with its input format. Thus, rather than treating the input format as fixed, we consider methods of using transformed inputs as a key step in DNN architecture search and training. This yields two novel opportunities for accelerating inference: 1) preprocessing-aware, cost-based methods of selecting DNNs for higher accuracy or improved throughput and 2) input-aware training/specialization. We implement these in SysX, a runtime engine for end-to-end DNN inference that can be integrated into existing visual analytics systems.

SysX's first contribution is to construct preprocessing-aware cost models for generating both regular and specialized NNs. Existing cost models for selecting DNNs ignore the cost of preprocessing [37, 42] or ignore the fact that compute between preprocessing and DNN execution can be pipelined [8]. These assumptions are largely correct when DNN execution is the overwhelming bottleneck, but do not hold on modern hardware for a wide range of DNNs. As DNN-based visual analytics contains a fundamental trade-off between accuracy and computation, we show that ignoring preprocessing costs can select DNNs that are cheap to execute, but are less accurate. In contrast, by correctly incorporating preprocessing costs, SysX can select the most accurate DNN that matches the speed of preprocessing.

SysX's second contribution is to further balance preprocessing and DNN execution by *jointly* optimizing these steps by: 1) optimizing common vision preprocessing operations, 2) exploiting native low-resolution and partially decoded visual data, and 3) presenting a training procedure that achieves high accuracy on this data. In contrast to prior work that treats DNN inputs as fixed, we present two ways to reduce the cost of preprocessing. First, we leverage low-resolution visual data present in many systems. For example, YouTube stores multiple resolutions of video and Instagram stores thumbnails of images. Second, when low-resolution visual data is not present, we optimize preprocessing and DNN execution by partially decoding visual data. We show that naively using low-resolution data and partial decoding can degrade accuracy. To rectify this, we augment DNN training to be aware of low-resolution data by explicitly using data augmentation for the target resolution. We show that this procedure can achieve high accuracy, comparable to or exceeding the accuracy of full-resolution data.

We integrate SysX into NoScope and Tahoma and evaluate SysX on six visual datasets, including video and image datasets. We verify our choice of cost modeling through benchmarks on public cloud hardware and show that SysX can achieve up to 4.6× improved throughput compared to recent work in visual analytics.

In summary, we make the following contributions:

(1) We show that preprocessing costs can dominate end-to-end DNN-based analytics when carefully using modern hardware.
(2) We propose a method to modify cost models so that they are preprocessing aware and demonstrate that prior cost models are inaccurate.
(3) We show how to use low-resolution and partially decoded visual data to further balance preprocessing and DNN execution, and provide a training procedure that achieves high accuracy on this data. We show that these optimizations can achieve up to a 4.6× speedup over recent work in visual analytics.

2 BACKGROUND

We describe how DNNs are deployed in modern visual analytics systems, the full end-to-end inference of DNNs, and advances in hardware/compilers for DNNs. In this work, we focus on high throughput DNN inference.

Visual analytics systems. The majority of modern visual analytics systems we are aware of are configured with (possibly user-defined) target DNN(s) [8, 33, 37, 42]. Many of these modern visual analytics systems train smaller proxy models, which we refer to as specialized NNs, for improved performance [28, 37]. These specialized NNs are typically used to filter irrelevant data or classify easy examples before executing the larger, target DNN. The specialized NNs are typically trained on a portion of the data and used in a cascade at inference time.

NoScope and Tahoma. As a case study, we consider NoScope [37] and Tahoma [8]. These systems aim to reduce the cost of DNN inference by training specialized NNs to filter inputs before executing an expensive *target DNN*. NoScope ignores the cost of decoding video and selects cheap specialized NNs. Tahoma considers preprocessing costs by estimating the cost of a model by the sum of time to preprocess the input and execute the DNN *but ignores that preprocessing and DNN execution can be pipelined*. We describe how these choices can lead to inaccurate throughput estimation for DNN selection in Section 5. Furthermore, both of these systems were implemented on older hardware (the NVIDIA P100 and K80 GPUs respectively) where inference is significantly slower. We show the effect of preprocessing and modern inference hardware in these systems in Section 8.

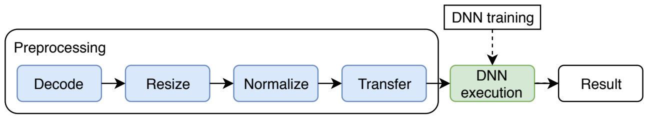


Figure 1: A block diagram of end-to-end DNN inference for visual analytics tasks, including preprocessing and DNN execution. Much research has focused on optimizing DNN execution, but preprocessing can dominate execution times on modern hardware.

DNN end-to-end inference. In this work, we primarily focus on the *end-to-end* inference of the DNNs (as opposed to training). End-to-end DNN inference consists of four steps: 1) ingesting data, 2) preprocessing the data, 3) transferring the data to the accelerator, and 4) executing the DNN computation graph on the accelerator. We show a diagram of this process in Figure 1. A large body of research focuses on the DNN execution, but preprocessing has been relatively neglected.

The first step of end-to-end DNN inference is ingesting the data. In visual analytics systems, this step involves reading the compressed visual data, e.g., JPEG compressed image, into memory from disk or over the network. This step typically involves high levels of branching so it is inefficient for massively parallel arithmetic-intensive accelerators.

The second step involves decoding the compressed data, e.g., to raw pixels, and typically applying a set of transformations over the decompressed data. For example, the image will be resized to 224x224, converted to floating point, and per-channel whitened for the standard ResNet-50.

The third step transfers the transformed data to the accelerator.

The fourth step executes the DNN computation graph on the accelerator and returns the result. This step has been optimized, both in the form of more efficient accelerators [24, 26, 34] and in the form of better DNN compilers [3, 39, 52].

Advances in hardware and compilers for DNNs. Since deep learning has become widely adapted, accelerators for executing DNN computation graphs have become dramatically more efficient. These new accelerators largely rely on the fact that DNNs can be approximated with little to no loss in accuracy [27] and that they often contain repeated computational patterns [31].

As a result, contemporary DNN accelerators use reduced precision [27] and hardware units specific to DNNs, such as Tensor Cores [43]. For example, these are present in the NVIDIA T4 GPU. These were not leveraged heavily in older accelerators, e.g., the NVIDIA K80 and P100 GPUs, and can result in substantial throughput improvement in DNN execution.

Furthermore, next generation accelerators reportedly achieve significantly higher DNN execution throughput by further reducing precision and continuing to specialize hardware. Next generation hardware reports up to a 2× improvement in throughput [62]. As a result, the preprocessing bottleneck will only grow.

These new hardware substrates are used in conjunction with new DNN compilers that take computation graphs and emit optimized execution plans [3, 39, 52]. These compilers can efficiently use the hardware for DNN execution. As we show, this can result in up to a 10× improvement in throughput.

3 MEASUREMENT STUDY OF END-TO-END DNN INFERENCE

In this section, we benchmark DNNs and visual data preprocessing on public cloud hardware. We show that more efficient accelerators and compilers have dramatically reduced the cost of DNN execution.

In light of these reduced costs, we further show that *preprocessing costs* can now dominate when using these accelerators efficiently, even for DNNs that have been considered expensive.

Experimental setup. We benchmarked the popular ResNet-50 model for image classification [31], which has widely been used in benchmarking [1, 21] and has been considered expensive. Specialized NNs are typically much cheaper than ResNet-50.

We benchmarked the time for only DNN execution and the time for preprocessing separately to isolate bottlenecks.

We benchmarked on the publicly available NVIDIA T4 GPU. We used the `g4dn.xlarge` AWS instance which has 4 vCPU cores (hyperthreads). This instance type is optimized for DNN inference; similar instances are available on other cloud providers. We used the TensorRT compiler [3] unless specified otherwise. While we benchmarked on the T4, other contemporary, non-public accelerators report similar or improved results [24, 34].

Effect of software on inference throughput. We benchmarked ResNet-50 throughput on the inference optimized T4 GPU using three software systems for DNNs to show the effect of more efficient software on throughput. We benchmark using Keras [18], PyTorch [48], and TensorRT [3]. We note that Keras was was used by Tahoma and TensorRT is an optimized DNN computational graph compiler.

As shown in Table 1, efficient use of accelerators via efficient compilers (TensorRT in this case) can result in up to a 10× improvement in throughput. Without the efficient use of accelerators, preprocessing is *not* the bottleneck.

Execution environment	Throughput (im/s)
Keras	243
PyTorch	424
TensorRT	2,172

Table 1: Throughput of ResNet-50 on the T4 with three different execution environments. Notably, Keras was used in [8]. As shown, the efficient use of hardware can result in over a 8.9× improvement in throughput. We used the optimal batch size for each framework.

Breakdown of end-to-end DNN inference. DNN inference consists of steps beyond the execution of the DNN computation graph. For the standard ResNet-50 configuration, the preprocessing steps are:

1. Decode the compressed image, which is typically stored in JPEG.
2. Resize the image with an aspect-preserving resize such that the short edge of the image is 256 pixels. Centrally crop the image to 224x224.
3. Convert the image to float32. Divide the pixel values by 255, subtract a per-channel value, and divide by a per-channel value (these values are derived from the training set). We refer to this step as "normalizing" the image.
4. Rearrange the pixel values to channels-first (this step depends on the DNN configuration).

To see the breakdown of preprocessing the costs, we implemented these steps in hand-optimized C++, ensuring best practices for high performance C++, including reuse of memory to avoid allocations. We used `libturbo-jpeg`, a highly optimized library for JPEG decompression, for decoding the JPEG images. We used OpenCV's optimized image processing libraries for the resize and normalization. We run all benchmarks on a standard `g4dn.xlarge` AWS instance and use multithreading to utilize all the cores.

As shown in Figure 2, simply decoding the JPEG files achieves lower throughput than the throughput of ResNet-50 execution. All together, the preprocessing of the data achieves 3.5× lower throughput than ResNet-50 execution. These overheads increase to up to 9.6× for ResNet-18.

4 SYSX OVERVIEW

SysX is an optimized runtime engine for end-to-end DNN inference that can be integrated into existing visual analytics systems. We integrate SysX into NoScope [37] and Tahoma [8] and note that it could additionally be integrated into other contemporary visual analytics systems that do not account for preprocessing costs correctly, including probabilistic predicates [42] and BlazeIt [36].

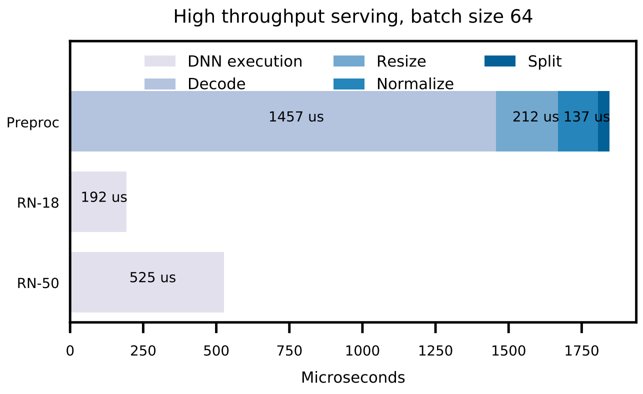


Figure 2: Breakdown of end-to-end inference of ResNet-50 for a batch size of 64 on the inference-optimized AWS `g4dn.xlarge` instance type (one NVIDIA T4 GPU and 4 vCPU cores). The DNN was executed on the T4 and the preprocessing was parallelized across all CPU cores. As shown, the execution of the computation graph is bottlenecked by over 3.5× compared to preprocessing data. This overhead increases to 9.6× for ResNet-18.

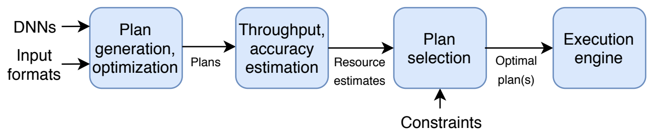


Figure 3: System diagram of SysX. As input, SysX takes a set of DNNs, visual input formats, and optional constraints. As output, SysX returns an optimal set of plans or plan, depending on the constraints. SysX will generate plans, estimate the resources for each plan, and select the Pareto optimal set of plans.

We describe SysX's architecture and show a schematic in Figure 3. At a high level, SysX aims to generate efficient, end-to-end execution plans for DNN inference given a set of DNNs and visual input formats.

Deployment setting. In this work, we focus on high throughput batch settings, as prior work does [8]. SysX's goal is to achieve the highest throughput on the available hardware resources. For example, a visual analytics engine might ingest images or videos daily and run a batch analytics job each night. Nonetheless, several of our techniques, particularly in jointly optimizing preprocessing and inference, also apply to the low-latency or latency-constrained throughput settings.

We note that in high throughput batch settings, visual data is almost always stored in compressed formats which require preprocessing. Uncompressed visual data is large: a single hour of 720p video is almost 900GB of data whereas compressed video can be as small as 0.5GB per hour.

Inputs and outputs. As input, SysX takes a set of DNNs (e.g., ResNets) and a set of allowed visual data formats (e.g., full resolution JPEG images, thumbnail JPEGs). We denote the set of DNNs as $\mathcal{D}$ and the set of visual data formats as $\mathcal{F}$. SysX further takes a set of calibration images (i.e., a validation set) to estimate the accuracy.

SysX optionally takes a throughput or accuracy constraint. If a constraint is specified, SysX will return an optimized execution plan that respects these constraints. Otherwise, SysX will return the Pareto optimal set of execution plans (in accuracy and throughput).

Components. SysX contains three major components: 1) a plan optimizer, 2) a cost estimator, and 3) an execution engine. We show these three components in Figure 3.

SysX will first generate query plans from the provided $\mathcal{D}$ and $\mathcal{F}$ by taking the cross product of $\mathcal{D}$ and $\mathcal{F}$, namely $\mathcal{D} \times \mathcal{F}$. Then, SysX will optimize these plans using its optimizer. Given a set of optimized plans, SysX will estimate the accuracy and throughput of these plans using its cost model. Finally, SysX will return the best query plan if a constraint is specified or the Pareto optimal set of query plans if not.

Example. We describe NoScope in this framework as an example. NoScope considers a fixed input format, namely the provided input format, which is H.264 video. NoScope considers 12 models, each of which are cascaded with a target DNN. Thus, $|\mathcal{F}| = 1|$ and $|\mathcal{D}| = 12$. NoScope aims to return the configuration with the highest throughput for a given accuracy. NoScope estimates the throughput of $D_i \in \mathcal{D}$ by ignoring preprocessing costs.

We describe SysX's cost model and optimizations below.

5 COST MODELING FOR VISUAL ANALYTICS

Overview. When deploying DNN-based visual analytics systems, application developers have different resource constraints. As such, these systems often expose a way of trading off between accuracy and throughput. Higher accuracy DNNs typically require more computation; we show this phenomenon on ImageNet in Figure 4. Prior work has designed high throughput specialized DNNs for filtering [8, 37]; we do not focus on the design of DNNs in this work and assume that the set of possible DNNs is provided to SysX.

To select system configurations, system designers have developed algorithms to select DNNs given system constraints. One of the most popular methods for selecting DNNs is to use a cost model [8, 37]. We describe cost modeling for DNNs and critically how prior work estimated the throughput of

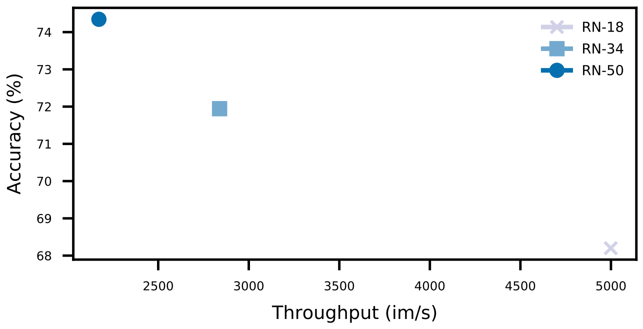


Figure 4: Throughput vs accuracy (top one) for ResNets of different depths. As shown, there is a trade-off between accuracy and throughput (computation).

DNNs. Importantly, we show that prior methods of throughput estimation can lead to inaccurate results (Table 2). We then describe how to make cost models preprocessing-aware.

Cost models. Given a set of resource constraints and metrics to optimize, a system must choose which DNNs to deploy to maximize these metrics while respecting resource constraints. For example, one popular constraint is a minimum throughput and one popular metric is accuracy. In this exposition, we focus on throughput-constrained accuracy and accuracy-constrained throughput, but other constraints could be used.

Specifically, denote the possible set of system configurations as $C_1, ..., C_n$. Denote the resource consumption estimate of each configuration as $R(C)$ and the resource constraint as $R_{\max}$. Denote the metric to optimize as $M(C)$.

In its full generality, the optimization problem is

$$\begin{aligned} &\max_i M(C_i) \\ &\text{s.t. } R(C_i) \leq R_{\max}. \end{aligned} \tag{1}$$

In this framework, both accuracy and throughput can either be constraints or metrics. For example, for throughput-constrained accuracy, $R(C_i)$ would be an estimate of the throughput of C_i and $M(C_i)$ would be an estimate of the accuracy of C_i. Similarly, for accuracy-constrained throughput, $R(C_i)$ would be an estimate of the accuracy and $M(C_i)$ would be an estimate of the throughput.

As an example, Tahoma generates $C_i = [D_{i,1}, ..., D_{i,k}]$ to be a sequence of k models, $D_{i,j}$, that are executed in sequence. The resource $R(C_i) = A(C_i)$ is the accuracy of configuration C_i and the metric $M(C_i) = T(C_i)$ is the throughput of configuration C_i.

Prior work has focused on expanding the set of C_i or evaluating $R(C_i)$ and $M(C_i)$ efficiently [8, 12, 37, 42]. A common technique is to use a smaller model (e.g., a specialized NN) to filter data before executing a larger, target DNN in a *cascade*. For example, when detecting cars in a video, NoScope will

Configuration	Preprocessing throughput (im/s)	DNN execution throughput (im/s)	Pipelined throughput (im/s)	SysX estimate (im/s, % error)	NoScope estimate (im/s, % error)	Tahoma estimate (im/s, % error)
Balanced	4001	4999	4056	**4001, 1.4%**	4999, 23.2%	2239, 44.8%
Preproc-bound	534	4999	557	**534, 4.1%**	4999, 797.5%	505, 9.3%
DNN exec-bound	5876	1844	1720	**1844, 7.2%**	**1844, 7.2%**	1330, 22.7%

Table 2: We show measurements of preprocessing, DNN execution, and pipelined end-to-end DNN inference for three configurations of DNNs and input formats: balanced, preprocessing-bound, and DNN-execution bound. We measure the throughput in images per second of preprocessing, DNN execution, and end-to-end DNN inference on the left. We show throughput estimations and their errors from three cost models on the right. We bold the most accurate estimate. As shown, SysX matches or ties the most accurate estimate for all conditions.[1]

train an efficient model to filter out frames without cars [37]. Cascades can significantly expand the feasible set of configurations.

For cost models to be effective, the accuracy and throughput measurements must be accurate. We discuss throughput estimation below. Accuracy can be estimated using best practices from statistics and machine learning. A popular method is to use a held-out validation set to estimate the accuracy [10]. Under the assumption that the test set is from the same distribution as the validation set, this procedure will give an estimate of the accuracy on the test set.

Throughput estimation. A critical component of cost model for DNNs is the throughput estimation of a given system configuration C_i; recall that each configuration C_i is represented as a sequence of one or more DNNs, $D_{i,j}$. Given a specific DNN $D_{i,j}$, estimating its throughput simply corresponds to executing the computation graph on the accelerator and measuring its throughput. As DNN computation graphs are typically fixed, this process is efficient and accurate.

Prior work (e.g., NoScope) [37, 42] has used the throughput of $D_{i,j}$ to estimate the throughput of end-to-end DNN inference. Specifically, denoting the throughput estimate as $T(C_i)$, we have that

$$T(C_i) \approx \sum_{j=1}^{k} \alpha_j T_{\text{execution}}(D_{i,j}) \tag{2}$$

where α_j is the pass-through rate of DNN $D_{i,j}$ and $T_{\text{execution}}(D_{i,j})$ is the throughput of executing $D_{i,j}$. This approximation holds when the cost of preprocessing is small compared to the cost of executing the DNNs. However, we show that this approximation is inaccurate for other conditions, namely when preprocessing costs dominate DNN execution costs or when preprocessing costs are approximately balanced with DNN execution costs (Table 2).

[1]Preprocessing having lower throughput than both in the preprocessing-bound and balanced conditions are due to the experimental harness being optimized for pipelined execution. The experimental harness does not significantly affect throughput when compared to without the harness.

Other systems (e.g., Tahoma) [8] have incorporated preprocessing costs in the form of

$$T(C_i) \approx T_{\text{preprocess}}(C_i) + T_{\text{execution}}(C_i). \tag{3}$$

This approximation holds when either preprocessing or DNN execution is the overwhelming bottleneck, but is inaccurate for other conditions, namely when preprocessing costs are approximately balanced with DNN execution costs (Table 2).

However, these approximations of throughput (ignoring preprocessing costs and summing preprocessing and DNN execution costs) ignore two critical factors: first, that input preprocessing can dominate inference times and second, that input preprocessing can be pipelined with DNN execution on accelerators. We describe a more accurate throughput estimation scheme below.

Corrected throughput estimation. For high throughput DNN inference on accelerators, the DNN execution and preprocessing of data can be pipelined. As a result, a more accurate throughput estimate for a given configuration is:

$$T(C_i) \approx \min\left(T_{\text{preprocess}}(C_i), \sum_{j=1}^{k} \alpha_j T_{\text{execution}}(D_{i,j})\right) \tag{4}$$

Importantly, as we have shown in Section 3, preprocessing can dominate end-to-end DNN inference.

If preprocessing costs are fixed, then it becomes optimal to maximize the accuracy of the DNN subject to the preprocessing throughput. Namely, the goal is to pipeline the computation as effectively as possible. We give two examples of how this can change which configuration is chosen below. Furthermore, in Section 6, we show how SysX jointly optimizes preprocessing and DNN execution for improved throughput or accuracy.

First, when correctly accounting for preprocessing costs in a throughput-constrained accuracy deployment, it is not useful to select a throughput constraint higher than the throughput of preprocessing. Second, for an accuracy-constrained throughput deployment, the most accurate DNN subject to the preprocessing throughput should be selected.

6 OPTIMIZING ACROSS PREPROCESSING AND DNN INFERENCE

To increase the throughput of end-to-end DNN inference, we must jointly optimize preprocessing and DNN execution. In this section, we describe two ways to jointly optimize preprocessing and DNN execution by leveraging reduced fidelity visual data (i.e., low-resolution or partial decoding) or reduced preprocessing computation.

In Section 8.2, we show that naively using reduced fidelity visual data can degrade accuracy. As a result, we propose a method to train DNNs to be aware of reduced fidelity visual data. We show that this method can recover accuracy or even improve accuracy in certain cases.

We first describe how to optimize common preprocessing optimizations (Section 6.1) and give background on visual compression formats (Section 6.2). We then describe how to leverage visual compression formats for improved preprocessing speeds by describing how to use low-resolution data (Section 6.3), describing how to use partial decoding (Section 6.4), and describing our low-resolution training procedure (Section 6.5).

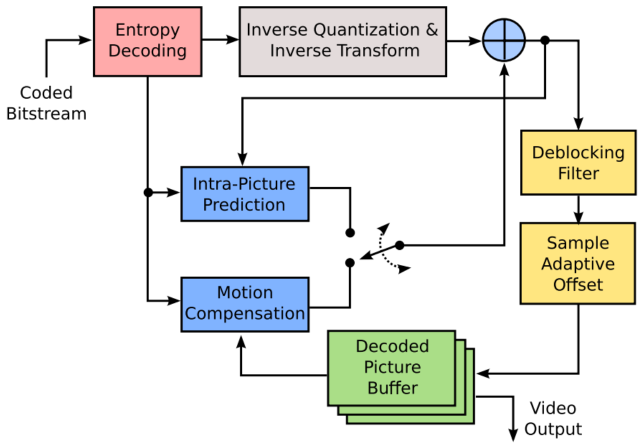
6.1 Optimizing Common Preprocessing Operations

A large class of common visual DNN preprocessing operations fall under the steps described in Section 3. Briefly, they include resizing, cropping, pixel-level normalization, data type conversion, and channel reordering. We can optimize these operations at inference time by fusing, reordering, and pre-computing operations.

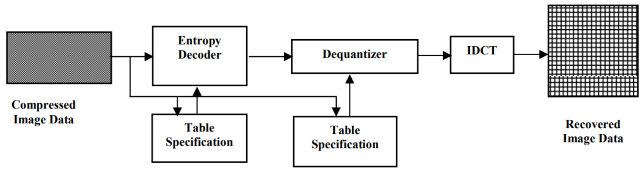
To optimize these steps, SysX will accept the preprocessing steps as a computation graph and performs a combination of rule-based and cost-based optimization of these steps. SysX contains rules of allowed operation reordering:

(1) Normalization and data type conversion can be placed at any point in the computation graph.
(2) Normalization, data type conversion, and channel reordering can be fused.
(3) Resizing and cropping can be swapped.

Given these rules, SysX will apply cost-based optimization to select the preprocessing graph with lowest cost. When considering a computational graph, SysX will also fuse and pre-compute all possible operations as we have found that fusion and pre-computation improve performance. For example, normalization and data type conversion can be precomputed with a lookup table that fits in L2 cache. Finally, as resizing is cheaper on smaller data formats and smaller sub-images, we place resizing after cropping and before data type conversion where possible.



(a) HEVC video decoder block diagram. Taken from [29].



(b) JPEG still image decoder block diagram. Taken from [38].

Figure 5: Block diagrams for a video and still image decoder. Entropy decoding requires heavy branching, so it is inefficient for arithmetic-intensive accelerators. Several steps can be omitted for faster decoding at the cost of visual fidelity, e.g., the deblocking filter.

6.2 Overview of Visual Compression Formats

We briefly describe the salient properties of the majority of popular visual compression formats, including the popular JPEG, HEVC/HEIC, and H.264 compression formats. We describe the decoding of the data and defer a description of encoding to other texts [50, 56, 61]. We show block diagrams of the JPEG and HEVC formats in Figure 5; many other formats are similar, including H.264, VP9, and JPEG2000.

Importantly, the entropy decoders in both JPEG and HEVC (Huffman decoding and arithmetic decoding respectively) are not efficient on accelerators for DNNs as it requires substantial branching. Furthermore, Certain parts of decoding can be omitted, e.g., the deblocking filter, for reduced fidelity but faster decoding times.

First, the data bitstream is decoded to transformed data via an arithmetic decoder [56, 61] or a Huffman decoder [50]. These decoders generally involves sequential decisions and a high degree of branching. As a result, these decoders are not well suited for highly parallel accelerators.

Second, the transformed coefficients are reverse-transformed and dequantized. Typically, a DCT-based transformation is used, although JPEG2000 uses a wavelet transform. The transformation is generally lossless and the compression comes from quantization. In some cases, intra-picture prediction is also applied.

Third, for video decoding, inter-picture motion prediction can be applied.

Fourth, several compression formats, including H.264 and HEVC/HEIC, contain an optional deblocking filtering. Applying the deblocking filter can improve visual fidelity, but can also be disabled for reduced computation.

6.3 Low-resolution Visual Data

We have found that many services that serve images store low-resolution versions of the full-resolution data, either for previewing purposes or for low-bandwidth situations. For example, Instagram stores 161x161 previews of images at the time of writing [9]. Similarly, YouTube stores several resolutions of the same video for different bandwidth requirements, e.g., 240p up to 4K video.

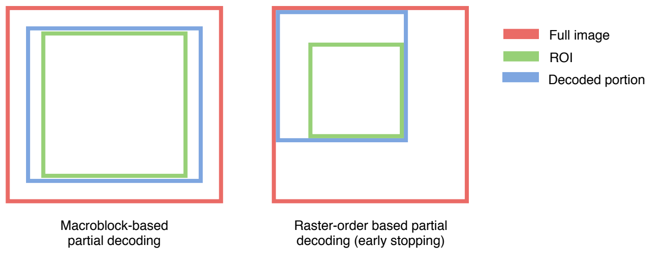
Decoding low-resolution visual data is more efficient than decoding full resolution data. To leverage this low-resolution data for improved preprocessing throughput, SysX could decode and then upscale the low-resolution visual data. However, we show that naively upscaling gives low accuracy results. Instead, SysX will train DNNs to be aware of low-resolution data, as described in below (Section 6.5).

For single DNN systems, SysX will train a low-resolution aware DNN and use this DNN if the accuracy threshold is met on the validation set. Otherwise, SysX will default to the standard-resolution DNN. For systems with specialized DNNs or cascades, SysX will choose the configuration with the highest throughput/accuracy given the accuracy/throughput constraint.

6.4 Partial and Low-Fidelity Decoding

When low-resolution visual data is not available, SysX can optimize preprocessing by partially decoding visual data. Many DNNs only require a portion of the image for inference, or *regions of interest (ROI)*. For example, many image classification networks centrally crop images, so the ROI is the central crop. Computing face embeddings crops faces from the images, the ROIs are the face crops. Furthermore, these networks often take standard image sizes, e.g., 224×224 for the standard ResNet-50. We show two examples in Figure 6.

Many image compression formats allow for partial decoding explicitly in the compression standard and all compression formats we are aware of allow for early stopping of decoding. We give three instantiations of partial decoding



Full image
ROI
Decoded portion
Macroblock-based partial decoding
Raster-order based partial decoding (early stopping)

Figure 6: Examples of partial decoding for images. On the left, the ROI is the central crop of the image. For JPEG images, SysX can decode only the macroblocks that intersect the ROI. For image formats that do not allow for independently decoding macroblocks, SysX can partially decode based on raster order (right). Thus, only the blue portion need be decoded. As decoding is generally more expensive than other parts of the preprocessing pipeline, partial decoding can significantly improve throughput.

in popular visual compression formats and provide a list of popular visual data compression formats and which features they contain in Table 3. We then describe how to use these decoding features for optimized preprocessing.

First, for the JPEG image compression standard, each 8x8 block, or *macroblock*, in the image can be decoded independently (partial decoding) [60]. Second, the H.264 and HEVC video compression standards contain deblocking filters, which can be turned off at the decoding stage for reduced computational complexity at the cost of visual fidelity (reduced fidelity decoding) [56, 61]. Third, the JPEG2000 image compression format contains "progressive" images, i.e., downsampled versions of the same image, that can be partially decoded to a specific resolution (multi-resolution decoding) [57].

Partial decoding. We present two methods of partially decoding visual data. We show examples of each in Figure 6.

ROI decoding. When only a portion of the image is needed, e.g., for central cropping or when selecting a region of interest (ROI), only the specified portion of the image need be decoded. To decode this portion of the image, SysX will first find the smallest rectangle that aligns with the 8x8 macroblock border and contains the region. Then, SysX will decode the rectangle and return the crop. This procedure is formalized in Algorithm 1.

Early stopping. For compression formats that do not explicitly allow for partial decoding, SysX can terminate decoding on parts of the image that are not necessary. For example, if only the top $N \times N$ pixels are required for inference, SysX will terminate decoding after decoding the top $N \times N$ pixels.

Algorithm 1 Partial JPEG decoding for a fixed DNN input resolution of 224 × 224

```
Ensure: $img \in (3, h, w)$
  $w', h' \leftarrow RatioPreservingResize(img)$
  $l = \frac{(w'-224)}{2}$
  $r = l + 224$
  $t = \frac{h'-224}{2}$
  $b = t + 224$
  $scale \leftarrow \frac{\min(h, w}{224}$
  $l', r', t', b' = scale \cdot [l, r, t, b]$
  img.SetCropline($l'$, $r'$, $t'$)
  while rowIdx $\leq b'$ do
    $img$[rowIdx] = ReadScanlines(rowIdx)
    rowIdx + = 1
  end while
```

Format	Type	Low-fidelity features
JPEG	Image	Partial decoding
JPEG2000	Image	Multi-res. decoding
PNG, WebP	Image	Early stopping
HEIC/HEVC	Image/Video	Reduced fidelity decoding
H.264	Video	Reduced fidelity decoding
VP8	Video	Reduced fidelity decoding
VP9	Video	Reduced fidelity decoding

Table 3: A list of popular visual data formats and their low-fidelity features. Many popular formats contain methods of decoding parts of the visual data, including the popular JPEG, H.264, and HEVC formats.

Reduced-fidelity decoding. Several visual compression formats contain options for reduced fidelity decoding. While there several ways to reduce the fidelity of decoding for decreased preprocessing costs, we focus on methods that are easily specified with existing decoding APIs. Specifically, we explore reduced fidelity in the form of disabling the deblocking filter. SysX will profile the accuracy of the specialized and target NNs with and without the deblocking filter and choose the option that maximizes throughput.

Progressive decoding. Recall that several image formats support progressive decoding, or decoding lower resolutions of the same image. For visual compression formats that support progressive decoding, SysX will decode the smallest resolution image that is larger than the specified resolution for target DNNs. For specialized NNs, SysX will profile the accuracy of stopping decoding at earlier stages and upsampling the resulting image.

6.5 Training DNNs for Low-resolution Visual Data

As described above, SysX can use low-resolution visual data to decrease preprocessing costs. However, naively using low-resolution can decrease accuracy, especially for target DNNs. For example, using a standard ResNet-50 with native 161x161 images results in a *10.8% absolute drop in accuracy*: this drop in accuracy is larger than switching from a ResNet-50 to a ResNet-18 or nearly reducing the depth by a third. To alleviate the drop in accuracy, SysX can train DNNs to be aware of low-resolution. This procedure can recover, or even exceed, the accuracy of standard DNNs.

SysX trains DNNs to be aware of low-resolution by augmenting the input data at training time. At training time, SysX will downsample the full-resolution inputs to the desired resolution and then upsample them to the DNN input resolution. SysX will do this augmentation in addition to standard data augmentation. By purposefully introducing downsampling artifacts, these DNNs can be trained to recover high accuracy on low-resolution data.

We show that this training procedure can recover the accuracy of full resolution DNNs when using lossless low-resolution data, e.g., PNG compression. However, when using lossy low-resolution data, e.g., JPEG compression, low-resolution DNNs can suffer a drop in accuracy. Nonetheless, we show that using lossy low-resolution data can be more efficient than using smaller, full-resolution DNNs.

7 IMPLEMENTATION

Overview. We implement a SysX prototype with the above optimizations. The training phase of SysX is implemented in Python and PyTorch. We reimplemented Tahoma and NoScope in PyTorch. The execution engine is written in C++.

Components. SysX consists of three major components: 1) data ingestion, 2) preprocessing, and 3) DNN execution components. We describe each component in turn.

SysX provides an API to accept visual data (i.e., images or video). SysX provides a default method for ingesting visual data from disk. Currently, SysX supports JPEG compressed images or H.264 compressed video due to their popularity. SysX could be easily extended to a wide range of formats and similar techniques would apply. For example, JPEG2000 has progressively sized images [19], which could easily be integrated with low-resolution decoding.

SysX provides standard methods for decoding and preprocessing visual data. SysX currently supports colorspace conversion, resizing, cropping, and color normalization. These methods cover a large range of visual analytics tasks, including classification [31], object detection [30, 41], and face detection [64]. A user of SysX can also provide user-defined preprocessing methods.

SysX currently accepts DNNs in the form of Open Neural Network Exchange (ONNX) computation graphs [4]. We choose ONNX as it is widely supported, e.g., PyTorch, TensorFlow, and, Keras can both export to ONNX, and TensorRT accepts ONNX computation graphs. SysX could easily be extended to support other backends as new standards and hardware emerge.

Low-level details. To achieve high throughput, SysX leverages three core techniques: 1) the careful use of memory, 2) lightweight pipelining, and 3) the use of optimized libraries.

Memory optimizations. To optimize memory usage, SysX allocates buffers once and reuses them when possible. For example, the input to the DNN computational graph generally has the same shape so SysX will allocate a buffer for a batch of data and reuse it between successive batches of data. SysX will also pin memory that will be subsequently transferred to the GPU. Pinning memory substantially improves copy performance.

Pipelining. For lightweight pipelining, SysX uses a thread pool for preprocessing workers and CUDA streams for parallel execution on the GPU. To communicate between the preprocessing workers and the CUDA execution streams, SysX uses a multi-producer, multi-consumer queue. Additionally, SysX only passes pointers between workers, avoiding excessive memory copies.

Optimized libraries. Finally, SysX uses highly optimized libraries where possible. For its multi-producer, multi-consumer queue, SysX uses `folly`'s `MPMCQueue` [2]. For its DNN computation graph compiler, SysX currently uses TensorRT [3]. We have found TensorRT to be effective on GPUs, but other compilers could be used for different hardware substrates. Finally, SysX uses `FFmpeg`, `libspng`, and `libturbo-jpeg` for loading video, PNG images, and JPEG images respectively.

8 EVALUATION

We evaluated SysX on six visual datasets and show that SysX can outperform baselines by up to 4.6× for image datasets and 10× for video datasets at a fixed accuracy level.

8.1 Experimental Setup

Overview. We evaluate our optimizations on four image datasets and two video datasets. The task for the image datasets is image classification. The task for the video datasets is binary object detection, or the presence or absence of a target object class. For the image datasets, we use accuracy and throughput as our primary evaluation metrics. For the video datasets, we use the F1 score and throughput as our primary metrics. We choose the F1 score as these datasets are class imbalanced.

Dataset	# of classes	# of train im.	# of test im.
`bike-bird`	2	23k	1k
`animals-10`	10	25.4k	2.8k
`birds-200`	200	6k	5.8k
`imagenet`	1,000	1.2M	50K

Table 4: Summary of dataset statistics for the still image datasets we used in our evaluation. The datasets range in difficulty and number of classes. `bike-bird` is the easiest dataset to classify and `imagenet` is the hardest to classify.

Dataset	Target class	Occupancy
`taipei`	Bus	11.9%
`amsterdam`	Car	44.7%

Table 5: Summary of dataset statistics for the video datasets we used in our evaluation. Occupancy is the percent of frames with the target object.

Datasets. The image datasets we use are `bike-bird` [11], `animals-10` [7], `birds-200` [59], and `imagenet` [22]. These datasets vary in difficulty and number of classes (2 to 1,000). In contrast, several prior systems study only binary filtering [8, 12, 42]. We summarize dataset statistics in Table 4. For the image datasets, we further considered a standard short size of 161 encoded in PNG, JPEG ($q = 75$), and JPEG ($q = 95$).

For the video datasets, we used `taipei` and `amsterdam` as used in [37]. We provide dataset statistics in Table 5. As the original videos were not available, we scraped additional video from the same stream. We used the original resolutions, which were 720p. We additionally encoded the videos to 480p for the low-resolution versions.

Model configuration and baselines. For SysX, we use the standard configurations of ResNets, specifically 18, 34, and 50. We find that these models span a range of accuracy and speed while only requiring training three models. We note that if further computational resources are available at training time, further models could be explored.

Image datasets. For the image datasets, we use the following two baselines. First, we use standard ResNets and vary their depths, specifically choosing 18, 34, and 50 as these are the standard configurations [31]. We refer to this configuration as the naive baseline; the naive baseline does not have access to other image formats. Second, we use Tahoma as our other baseline, specifically a representative set of 8 models from Tahoma cascaded with ResNet-50, our most accurate model.

We choose 8 models due to the computational cost of training these models, which can take up to thousands of GPU hours for the full set of models.

Video datasets. For the video datasets, we use six representative models from NoScope and cascade them with the state-of-the-art Mask R-CNN [30]. Mask R-CNN is significantly more accurate than the YOLOv2 used in [37]. We did not deploy the frame skipping used in NoScope to isolate the effect of the specialized NNs. We used a separate day of video for training and testing. We evaluated over 150,000 frames of video.

Hardware environment. Throughout, we use the AWS `g4dn.xlarge` instance type with a single NVIDIA T4 GPU attached. The `g4dn.xlarge` has 4 vCPU cores with 15 GB of RAM. A vCPU is generally a hyperthread, so 4 vCPUs consists of 2 physical cores. As such, compute intensive workloads, such as image decoding, will achieve sublinear scaling compared to a single hyperthread.

The `g4dn.xlarge` instance type is optimized for inference. Namely, the T4 GPU is significantly more power efficient than GPUs designed for training, e.g., the V100. However, they achieve lower throughput as a result; our results are more pronounced when using the V100 (e.g., using the `p3.2xlarge` instance). We further describe our choice of hardware environment in Section A.

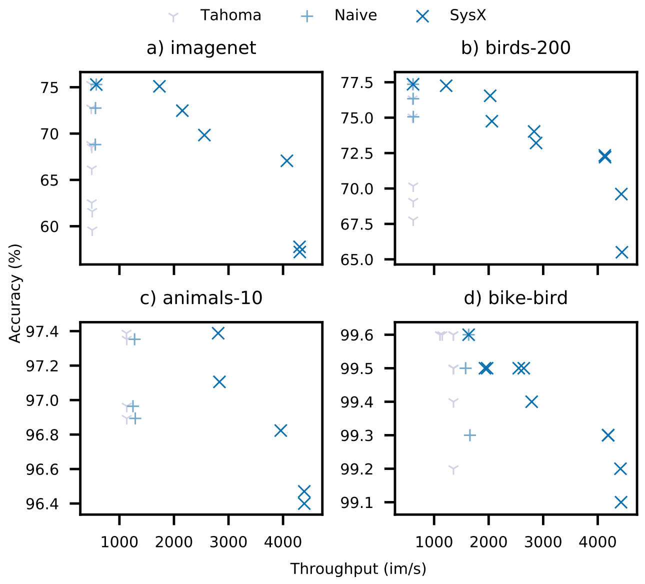


Figure 7: Throughput vs accuracy for the naive baseline, Tahoma, and SysX on the four image datasets. SysX can improve throughput by up to 4.6× with no loss in accuracy. Furthermore, SysX can improve the Pareto frontier compared to both baselines.

8.2 SysX can Significantly Improve Image Analytics Throughput

End-to-end speedups. We evaluated SysX on the four image datasets shown in Table 4. We further evaluated the baselines as described above.

We first investigated whether SysX outperforms baselines on these datasets when all optimizations were enabled. As shown in Figure 7, SysX can improve throughput by up to 4.6× with no loss in accuracy relative to ResNet-18 and up to 2.2× with no loss in accuracy relative to ResNet-50. Furthermore, SysX can improve the Pareto frontier of all baselines. We note that Tahoma's specialized models performs poorly on complex tasks and are also bottlenecked on image preprocessing.

Importantly, we see that the naive baselines (i.e., all ResNet depths) *are bottlenecked by preprocessing* for all datasets. Any further optimizations to the DNN execution alone, including model compression, will *not improve end-to-end throughputs.* The differences in baseline throughputs are due to the native resolution and encoding of the original datasets: `birds-200` contains the largest average size of images. The throughput variation between ResNets depths is due to noise; the variation is within margin of error.

The speedups that SysX can achieve are data dependent. We describe two observations of the source of speedups. First, for the easier datasets, SysX can achieve the same or even higher accuracy by simply using low-resolution data. Second, for the most difficult dataset (`imagenet`), a fixed model will result in slightly lower accuracy (<1%) when using lossless image compression. However, when using a *larger* model, SysX can recover accuracy.

Comparison against Tahoma. Tahoma underperforms the naive solution of using a single, accurate DNN for preprocessing bound workloads. This is primarily due to overheads in cascades, namely coalescing and further preprocessing operations. Specifically, Tahoma cascades a small DNN into a larger DNN. These smaller DNNs are significantly less accurate than the larger DNNs and thus require many images to be passed through the cascade for higher accuracy, especially on the more complex tasks. The images that are passed through must be copied again and further resized if the input resolutions are different.

Ablation and lesion studies. We further investigated the source of speedups of SysX's optimizations by performing ablation and lesion studies.

We performed a lesion study by individually removing the 1) preprocessing optimizations and 2) low-resolution data

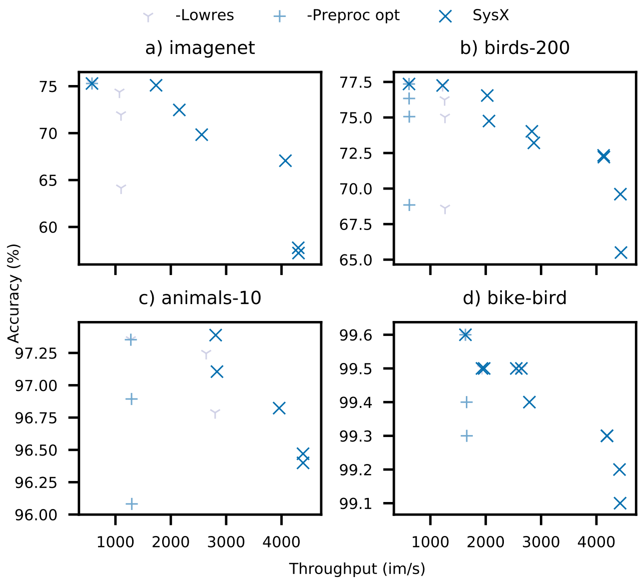


Figure 8: Lesion study for image datasets in which we individually removed the preprocessing optimizations and low-resolution data. As shown, both optimizations improve the Pareto frontier for all datasets.

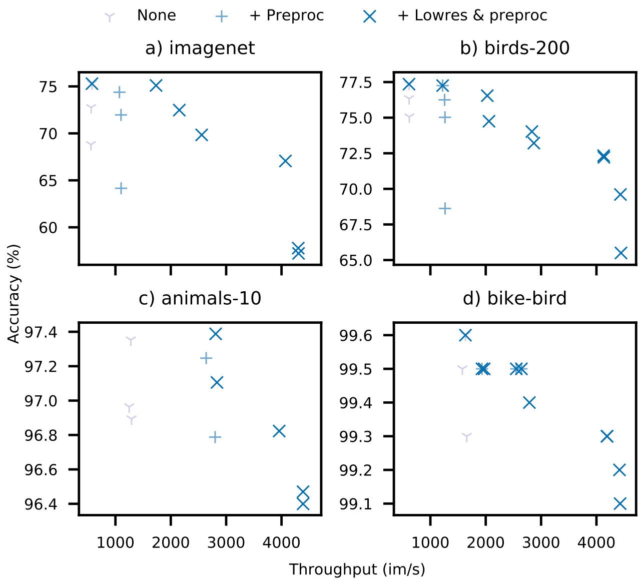


Figure 9: Ablation study for image datasets in which we successfully add the preprocessing optimizations and then the low-resolution data. As shown, both optimizations improve the Pareto frontier for all datasets.

from SysX. As shown in Figure 8, removing either optimization shifts the Pareto frontier.

We performed an ablation study by successively adding the preprocessing optimizations and the low-resolution data to SysX. As shown in Figure 9, both optimizations improve throughput. We further see that the optimizations are task-dependent: the easiest task (`bike-bird`) can achieve high throughput at fixed accuracies with only the preprocessing optimizations. However, many real-world tasks are significantly more complicated than binary classification of birds and bikes.

Effect of training procedure. We investigated the effect of the training procedure in the context of low-resolution input formats. We trained ResNet-50 on four input formats: 1) full resolution, 2) 161 short-side PNG, 3) 161 short-side JPEG ($q = 95$), and 4) 161 short-side JPEG ($q = 75$).

We show the accuracy of these conditions in Table 6 for `imagenet`, our hardest dataset. As shown, low-resolution aware training can nearly recover the accuracy of full resolution data even on this difficult dataset. Low-resolution training can fully recovery accuracy on `bike-bird` and `animals-10`.

Benchmarking SysX. We further investigated the efficiency of pipelining in SysX and our choice of using min in cost modeling (Section 5). To study these, we measured the throughput of SysX when only preprocessing, only executing the DNN computational graph, and when pipeline both stages. We used the low-resolution images encoded in JPEG $q = 75$ to ensure the system was under full load.

As shown in Table 7, SysX can successfully pipeline the decode and DNN computation graph execution with only a 6.7% overhead. Importantly, these results justify our choice of using min in cost models as our cost estimate is within 6.7% of the true throughput even at high loads. The 6.7% overhead likely comes from the cost of transferring memory to the accelerator, which increases cache load.

8.3 SysX can Significantly Improve Video Analytics Throughput

We evaluated SysX on the two video datasets, `taipei` and `amsterdam`; we provide dataset statistics in Table 5. We evaluated NoScope as the baseline.

As shown in Figure 10, SysX can improve throughput by up to 10× at a fixed accuracy level. Furthermore, SysX can improve the Pareto frontier compared to NoScope.

SysX's primary speedups for both datasets come from more accurate, but more expensive specialized NNs. Despite being more expensive, the more expensive specialized NNs can still be pipelined with preprocessing.

SysX's primary speedups for `taipei` come from SysX foregoing the target DNN and primarily using the specialized NN. SysX's primary speedups for `amsterdam` come from the target DNN executing fewer times.

Format	Acc (reg train, 50)	Acc (low-resol train, 50)	Acc (reg train, 34)	Acc (low-resol train, 34)
Full resol	75.29%	55.12%	72.75%	64.86%
161, PNG	70.77%	75.10%	68.24%	72.47%
161, JPEG ($q = 95$)	68.74%	72.25%	66.64%	69.84%
161, JPEG ($q = 75$)	63.78%	63.40%	62.01%	62.38%

Table 6: Effect of training procedure and input format on accuracy for ResNet-50 and ResNet-34 on `imagenet`, the most difficult dataset. SysX can achieve an accuracy throughput trade-off by simply changing the input format, e.g., low-resolution ResNet-50 (low-resol train, 50) on 161, JPEG ($q = 95$) achieves approximately the same accuracy as ResNet-34 (reg train, 34) on full resolution data (full resol), namely 72.25% accuracy compared to 72.75% accuracy. SysX can also achieve no loss in accuracy for easier datasets (e.g., `bike-bird`).

Condition	Throughput (im/s)
Preproc only	5,876
DNN-exec only	1,844
Both	1,720

Table 7: Throughput of preprocessing only, DNN execution only, and the full end-to-end DNN inference on the `imagenet` dataset with low-resolution images encoded at $q = 75$. As shown, SysX only incurs a 6.7% overhead, justifying our choice of min **in our cost model.**

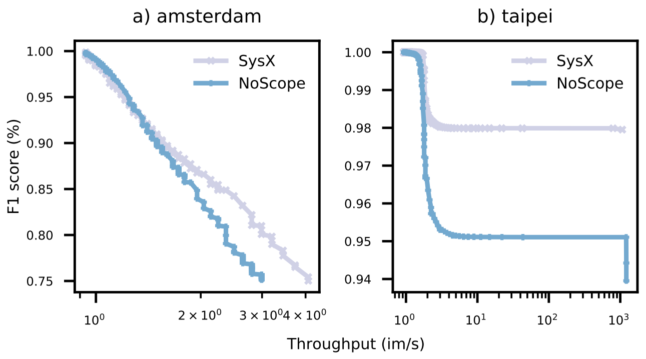


Figure 10: Throughput vs F1 score for NoScope and SysX on the two video datasets we evaluated. SysX can improve throughput by up to 10× at a fixed accuracy level. Furthermore, SysX can improve the Pareto frontier compared to NoScope.

In both cases, SysX does not achieve substantial speedups at high accuracy levels, which are in line with results in [37]. These results are because all models must pass through nearly all inputs to the target DNN for high accuracy.

9 RELATED WORK

Visual analytics systems. Contemporary visual analytics systems leverage DNNs for high accuracy annotations and largely focus on optimizing the cost of executing these DNNs [8, 12, 36, 37, 42]. These systems typically use smaller proxy models, such as specialized NNs to accelerate analytics. However, as we have showed, modern hardware and compilers can create bottlenecks elsewhere in the end-to-end execution of DNNs. SysX can be integrated into these systems to alleviate preprocessing costs.

Other video analytics systems, such as Scanner [51] or VideoStorm [63] optimize queries as a black box. In particular, Scanner [51] uses all available processing resources for decoding video, but cannot jointly optimize preprocessing and DNN execution.

Systems for optimized DNN execution and serving. Researchers have proposed compilers for optimizing DNN computation graphs, including XLA [39], PyTorch JIT [52], TVM [14], and TensorRT [3]. We leverage TensorRT in this work. These compilers generally cannot jointly optimize preprocessing and DNN execution. Furthermore, as they generate more efficient code, the bottleneck of preprocessing will only increase.

Further optimizations for DNN execution. Researchers have proposed machine learning techniques from model distillation [32] to model compression [27] to reduce the cost of DNN execution. These techniques generally take a given DNN architecture and improve its accuracy or speed. Notably, we are unaware of work in the machine learning literature for preprocessing-aware optimizations. These optimizations further improve DNN throughput, but will only increase the gap between preprocessing and DNN execution.

Optimizing DNN preprocessing. To the best of our knowledge, the only system that focuses on optimizing DNN preprocessing is NVIDIA DALI [45]. However, DALI optimizes preprocessing for DNN training and focuses on data augmentation. DNN training is significantly slower than inference and, while DALI is sufficient for training workloads, it does not match the throughput for inference.

Accelerators for DNNs. Due to the computational cost of DNN computational graphs, researchers and companies have created accelerators for DNN execution [24, 34]. In surveying hardware-based DNN acceleration, we were unable to find a single paper that measured *end-to-end* execution time [5, 6, 13, 15–17, 23–26, 34, 35, 40, 44, 46, 47, 49, 53–55, 58]. Until very recently, executing the DNN computational graph was the overwhelming bottleneck in DNN execution, but we show evidence that trend has reversed (Section 3). Thus, we believe it is critical to reason about *end-to-end* performance.

10 CONCLUSION

In this work, we show that preprocessing can be the bottleneck in end-to-end DNN inference. We show that the preprocessing costs are accounted for incorrectly in cost models for selecting models in visual analytics applications. To address these issues, we build SysX, an optimizing runtime engine for end-to-end DNN inference. SysX contains two novel optimization for end-to-end DNN inference: 1) an improved cost model for estimating DNN throughput and 2) joint optimizations for preprocessing and DNN execution that leverage low-resolution data. We evaluate SysX and these optimizations on six visual datasets and show that SysX can achieve up to 4.6× improvement in throughput. These results demonstrate the promise of jointly optimizing preprocessing and DNN execution.

A HARDWARE ENVIRONMENT AND POWER DISCUSSION

Overview. Throughout, we use the `g4dn.xlarge` instance as our testing environment. The `g4dn.xlarge` instance has a single NVIDIA T4 GPU, 4 vCPU cores (which are hyperthreads), and 15GB of RAM. The CPU type is the Intel Xeon Platinum 8259CL CPU, which is a proprietary CPU developed specifically for this instance type. As the specifications are not publicly available, we estimate the power draw by the similar Xeon Platinum 8268 CPU, which has a power draw of 205 watts, or 4.27 watts per vCPU core.

Discussion. We note that there are other `g4dn` instances which contain a single T4 GPU and 8, 16, 32, and 64 vCPU cores. These other instances types could be used to improve the preprocessing throughput by using more cores.

Nonetheless, our speedup numbers can be converted to cost savings when considering other instance types and our conclusions remain unchanging with respect to cost. For example, a 3× improvement in throughput of preprocessing can be translated to using 3× fewer cores. Furthermore, the cost of additional cores dominates: around 3.4 vCPU cores is the same price as the T4 when estimating the cost of vCPU cores and the T4 (see below).

Using these price and power estimates, we can estimate the relative price and power of preprocessing and DNN execution. Using the configuration in Figure 2, we see that preprocessing is significantly more expensive than DNN execution for the ResNet-50 ($0.218 vs $0.894 per hour) and has approximately the same power draw (60W vs 70W). For ResNet-18, these differences are more prominent: $0.218 vs $2.453 and 164W vs 70W for price and power respectively.

Core price estimation. We estimate the price per vCPU core using a linear interpolation, assuming the T4 is a fixed price and the remaining price is split equally among the cores. Using this method, we find that the hourly cost of the T4 accelerator is approximately $0.218 and the cost of a single vCPU core is approximately $0.0639. The R^2 value of this fit is 0.999. Thus, approximately 3.4 vCPU cores is the same hourly price of a T4.

GPU	Release date	Throughput (im/s)
K80	2014	159
P100	2016	1,955
T4	2019	2,172
V100	2017	7,151
Habana	N/A	15,000 (reported)

Table 8: Throughput of ResNet-50 on three GPU accelerators. Throughput has improved by over 44× in three years and will continue to improve. The T4 is an inference optimized accelerator that is significantly more power efficient than the V100, but contains similar hardware units.

B TRENDS IN HARDWARE ACCELERATION FOR DNNS

We benchmarked ResNet-50 throughput on the K80, P100, T4, and V100 GPUs to show the effect of improved accelerators on throughput; we further show the reported throughput of an unreleased accelerator [62]. We used a batch size of 64 for experiments on GPUs. As shown in Table 8, throughput has improved by 44× in three years. Furthermore, accelerators will become more efficient.

While we do not study low latency serving in this work, we note that similar trends hold for latency. The T4 GPU can achieve a latency of 0.93 ms for ResNet-50 at a batch size of 1, which is far lower than the latency of decoding an image.

REFERENCES

[1] 2018. MLPerf. https://mlperf.org/. (2018).
[2] 2019. folly. https://github.com/facebook/folly. (2019).
[3] 2019. NVIDIA TensorRT. (2019). https://developer.nvidia.com/tensorrt
[4] 2019. ONNX. (2019). https://onnx.ai/
[5] Jorge Albericio, Alberto Delmás, Patrick Judd, Sayeh Sharify, Gerard O'Leary, Roman Genov, and Andreas Moshovos. 2017. Bit-pragmatic deep neural network computing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 382–394.

[6] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 1–13.

[7] Corrado Alessio. 2019. Animals-10. (2019). https://www.kaggle.com/alessiocorrado99/animals10

[8] Michael R Anderson, Michael Cafarella, Thomas F Wenisch, and German Ros. 2019. Predicate Optimization for a Visual Analytics Database. *ICDE* (2019).

[9] Elizabeth Arens. 2019. Always Up-to-Date Guide to Social Media Image Sizes. (2019). https://sproutsocial.com/insights/social-media-image-sizes-guide/

[10] Christopher M Bishop. 2006. *Pattern recognition and machine learning.* springer.

[11] Tom B Brown, Nicholas Carlini, Chiyuan Zhang, Catherine Olsson, Paul Christiano, and Ian Goodfellow. 2018. Unrestricted adversarial examples. *arXiv preprint arXiv:1809.08352* (2018).

[12] Christopher Canel, Thomas Kim, Giulio Zhou, Conglong Li, Hyeontaek Lim, David Andersen, Michael Kaminsky, and Subramanya Dulloor. 2019. Scaling Video Analytics on Constrained Edge Nodes. *SysML* (2019).

[13] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. *ACM SIGARCH Computer Architecture News* 38, 3 (2010), 247–257.

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.

[15] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2016. DianNao family: energy-efficient hardware accelerators for machine learning. *Commun. ACM* 59, 11 (2016), 105–112.

[16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 367–379.

[17] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 27–39.

[18] François Chollet et al. 2015. Keras. (2015).

[19] Charilaos Christopoulos, Athanassios Skodras, and Touradj Ebrahimi. 2000. The JPEG2000 still image coding system: an overview. *IEEE transactions on consumer electronics* 46, 4 (2000), 1103–1127.

[20] Cody Coleman, Daniel Kang, Deepak Narayanan, Luigi Nardi, Tian Zhao, Jian Zhang, Peter Bailis, Kunle Olukotun, Chris Re, and Matei Zaharia. 2018. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *arXiv preprint arXiv:1806.01427* (2018).

[21] 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.

[22] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition.* Ieee, 248–255.

[23] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on.* IEEE, 109–116.

[24] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture.* IEEE Press, 1–14.

[25] Vinayak Gokhale, Jonghoon Jin, Aysegul Dundar, Berin Martini, and Eugenio Culurciello. 2014. A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops.* 682–687.

[26] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on.* IEEE, 243–254.

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

[28] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. 2016. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *MobiSys.* ACM, 123–136.

[29] Ilkka Hautala, Jani Boutellier, Jari Hannuksela, and Olli Silvén. 2014. Programmable low-power multicore coprocessor architecture for HEVC/H. 265 in-loop filtering. *IEEE Transactions on Circuits and Systems for Video Technology* 25, 7 (2014), 1217–1230.

[30] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. 2017. Mask r-cnn. In *ICCV.* IEEE, 2980–2988.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR.* 770–778.

[32] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[33] Kevin Hsieh, Ganesh Ananthanarayanan, Peter Bodik, Paramvir Bahl, Matthai Philipose, Phillip B Gibbons, and Onur Mutlu. 2018. Focus: Querying Large Video Datasets with Low Latency and Low Cost. *OSDI* (2018).

[34] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on.* IEEE, 1–12.

[35] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. 2016. Stripes: Bit-serial deep neural network computing. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on.* IEEE, 1–12.

[36] Daniel Kang, Peter Bailis, and Matei Zaharia. 2019. Challenges and Opportunities in DNN-Based Video Analytics: A Demonstration of the BlazeIt Video Query Engine. CIDR.

[37] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: optimizing neural network queries over video at scale. *PVLDB* 10, 11 (2017), 1586–1597.

[38] Bheshaj Kumar, Kavita Thakur, and GR Sinha. 2012. A new hybrid jpeg image compression scheme using symbol reduction technique. *arXiv preprint arXiv:1202.4943* (2012).

[39] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. *TensorFlow Dev Summit* (2017).

[40] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. Drisa: A dram-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, 288–301.

[41] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. 2016. Ssd: Single shot

multibox detector. In *European conference on computer vision*. Springer, 21–37.

[42] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating Machine Learning Inference with Probabilistic Predicates. In *SIGMOD*. ACM, 1493–1508.

[43] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 522–531.

[44] Bert Moons and Marian Verhelst. 2016. A 0.3–2.6 TOPS/W precision-scalable processor for real-time large-scale ConvNets. In *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*. IEEE, 1–2.

[45] NVIDIA. 2019. NVIDIA DALI. (2019). https://docs.nvidia.com/deeplearning/sdk/dali-developer-guide/docs/index.html

[46] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 27–40.

[47] Seong-Wook Park, Junyoung Park, Kyeongryeol Bong, Dongjoo Shin, Jinmook Lee, Sungpill Choi, and Hoi-Jun Yoo. 2015. An energy-efficient and scalable deep learning/inference processor with tetra-parallel MIMD architecture for big data applications. *IEEE transactions on biomedical circuits and systems* 9, 6 (2015), 838–848.

[48] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. (2017).

[49] Maurice Peemen, Arnaud AA Setio, Bart Mesman, Henk Corporaal, et al. 2013. Memory-centric accelerator design for Convolutional Neural Networks.. In *ICCD*, Vol. 2013. 13–19.

[50] William B Pennebaker and Joan L Mitchell. 1992. *JPEG: Still image data compression standard*. Springer Science & Business Media.

[51] Alex Poms, William Crichton, Pat Hanrahan, and Kayvon Fatahalian. 2018. Scanner: Efficient Video Analysis at Scale (To Appear). (2018).

[52] PyTorch Team. 2018. The road to 1.0: production ready PyTorch. (2018). https://pytorch.org/blog/the-road-to-1_0/

[53] Atul Rahman, Jongeun Lee, and Kiyoung Choi. 2016. Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*. IEEE, 1393–1398.

[54] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 267–278.

[55] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 535–547.

[56] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, and Thomas Wiegand. 2012. Overview of the high efficiency video coding (HEVC) standard. *IEEE Transactions on circuits and systems for video technology* 22, 12 (2012), 1649–1668.

[57] David Taubman and Michael Marcellin. 2012. *JPEG2000 image compression fundamentals, standards and practice: image compression fundamentals, standards and practice*. Vol. 642. Springer Science & Business Media.

[58] Swagath Venkataramani, Ashish Ranjan, Subarno Banerjee, Dipankar Das, Sasikanth Avancha, Ashok Jagannathan, Ajaya Durg, Dheemanth Nagaraj, Bharat Kaul, Pradeep Dubey, et al. 2017. Scaledeep: A scalable compute architecture for learning and evaluating deep networks. In *ACM SIGARCH Computer Architecture News*, Vol. 45. ACM, 13–26.

[59] Catherine Wah, Steve Branson, Peter Welinder, Pietro Perona, and Serge Belongie. 2011. The caltech-ucsd birds-200-2011 dataset. (2011).

[60] Gregory K Wallace. 1992. The JPEG still picture compression standard. *IEEE transactions on consumer electronics* 38, 1 (1992), xviii–xxxiv.

[61] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. 2003. Overview of the H. 264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology* 13, 7 (2003), 560–576.

[62] William Wong. 2018. Habana Enters Machine-Learning Derby with Goya Platform. (2018). https://www.electronicdesign.com/industrial-automation/habana-enters-machine-learning-derby-goya-platform

[63] 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 *NSDI*, Vol. 9. 1.

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