CBinfer: Exploiting Frame-to-Frame Locality for Faster Convolutional Network Inference on Video Streams

CBinfer: Exploiting Frame-to-Frame Locality for Faster Convolutional Network Inference on Video Streams

Lukas Cavigelli, Luca Benini

***Abstract*— The last few years have brought advances in computer vision at an amazing pace, grounded on new findings in deep neural network construction and training as well as the availability of large labeled datasets. Applying these networks to images demands a high computational effort and pushes the use of state-of-the-art networks on real-time video data out of reach of embedded platforms.**

Many recent works focus on reducing network complexity for real-time inference on embedded computing platforms. We adopt an orthogonal viewpoint and propose a novel algorithm exploiting the spatio-temporal sparsity of pixel changes. This optimized inference procedure resulted in an average speed-up of $9.1\times$ over cuDNN on the Tegra X2 platform at a negligible accuracy loss of $< 0.1\%$ and no retraining of the network for a semantic segmentation application. Similarly, an average speed-up of $7.0\times$ has been achieved for a pose detection DNN on static camera video surveillance data. These throughput gains combined with a lower power consumption result in an energy efficiency of 511 GOp/s/W compared to 70 GOp/s/W for the baseline.

I. Introduction

Computer vision (CV) technology has become a key ingredient for automatized data analysis over a broad range of real-world applications: smart cameras for video surveillance, robotics, industrial quality assurance, medical diagnostics, and advanced driver assistance systems have recently become popular due the rising accuracy and robustness of CV algorithms. This industry interest has fostered the procedure of a wealth of research projects yielding a fierce competition on many benchmarks datasets such as the ImageNet/ILSVRC [1], MS COCO [2], and Cityscapes [3] benchmarks, on which scientists from academia and big industry players evaluate their latest algorithms.

In recent years, the most competitive approaches to address many CV challenges have relied on machine learning with complex, multi-layered, trained feature extractors commonly referred to as *deep learning* [4]. The most frequently used flavor of deep learning techniques for CV are convolutional neural networks (ConvNets, CNNs). Since their landslide success at the 2012 ILSVRC competition over hand-crafted features, their accuracy has further improved year-over-year even exceeding human performance on this complex dataset [1], [5]. CNNs keep on expanding to more areas of computer vision and data analytics in general [6]–[9] and are moving towards analyzing video data for action recognition [10], tracking [11], and improved object detection [12], [13].

The authors would like to thank *armasuisse Science & Technology* for funding this research. This project was supported in part by the EU's H2020 programme under grant no. 732631 (OPRECOMP).

L. Cavigelli and L. Benini are with ETH Zürich, Zürich, Switzerland (e-mail: {cavigelli,benini}@iis.ee.ethz.ch).

Unfortunately, the high accuracy of CNNs comes with a high computational cost, requiring powerful GPU servers to train these networks for several weeks using hundreds of gigabytes of labeled data. While this is a miassive effort, it is a one-time endeavour and can be done offline for many applications. However, the inference of state-of-the-art CNNs also requires several billions of multiplications and additions to classify even low resolution images by today's standards [14]. While in some cases offloading to centralized compute centers with powerful GPU servers is also possible for inference after deployment, it is extremely costly in terms of compute infrastructure and energy. Furthermore, collecting large amounts of data at a central site raises privacy concerns and the required high-bandwidth communication channel causes additional reliability problems and potentially prohibitive cost of deployment and during operation [15]. For many applications the introduced latency is prohibitive.

The alternative, on-site near sensor embedded processing, largely solves the aforementioned issues by transmitting only the less sensitive, condensed information—potentially only security alerts in case of a smart surveillance camera—but imposes restrictions on available computation resources and power. These push the evaluation of such networks for real-time semantic segmentation or object detection out of reach of even the most powerful embedded platforms available today for high-resolution video data [14]. However, exactly such systems are required for a wide range of applications limited in cost (CCTV/urban surveillance, perimeter surveillance, consumer behavior and highway monitoring) and latency (aerospace and UAV monitoring and defence, visual authentication) [15], [16].

Large efforts have thus already been taken to develop optimized software implementations for heterogeneous platforms [14], [17]–[20], to design specialized hardware architectures [9], [21]–[25], and to adapt the networks to avoid expensive arithmetic operations by reducing arithmetic precision, exploiting sparsity, and developing more compact DNNs [8], [26]. However, they either do not provide a strong enough performance boost, are already at the theoretical limit of what can be achieved on a given platform, are inflexible and not commercially available, or incur a considerable accuracy loss. It is thus essential to extend the available options to efficiently perform inference on CNNs.

In this paper, we propose a novel method performing **c**hange-**b**ased **infer**ence (hence named CBinfer) for convolutional neural networks on video data from a static camera with limited frame-to-frame changes. We extend our preliminary work in [27]:

1) Enhancements to the algorithm for improved compute

time and ensuring a consistent input-output relation for each convolution layer.
2) In-depth analysis how changes propagate through the CBinfer DNN.
3) Analysis of accuracy, compute time and energy efficiency for long frame sequences.
4) Additional evaluations for a pose detection application with a much deeper network and a dataset without annotations.
5) Optimizations and evaluations on the more recent Nvidia Tegra X2 platform[1].
6) Discussion and evaluation of the processing steps and how the chosen configuration provides the highest performance gain.

Overall the proposed method provides an average speed-up by a factor of 9.1–7.0 over an implementation relying on cuDNN and introducing only negligible accuracy loss. It thus significantly outperforms previous approaches exploiting frame-to-frame locality which all have measured performance gains in the range of a few ten percents while introducing accuracy losses of several percent (cf. Section II-C). Our method can be combined with most single-frame optimizations such as exploiting weight sparsity or the development of more compact DNN models. The code is available online at https://github.com/lukasc-ch/CBinfer.

II. Related Work

In this section, we first describe existing optimized implementations for CNN inference and existing approximations trading accuracy for throughput. We then specifically survey related approaches exploiting the limited changes in video data to reduce the computational effort required to perform CNN inference. Finally, we discuss available datasets and CNNs with which we can evaluate our proposed algorithm.

Most per-frame optimization techniques can be combined with the method we propose herein. Existing approaches targeting video data have very limited gains and have not been specifically optimized for static camera frame sequences.

A. Optimized Embedded System Implementations

The latest wave of interest in neural networks can be attributed to their sudden success driven by the availability of large datasets and the increasingly powerful computing platforms. One of the most economical and practicable solutions for training medium-sized CNNs is to use a workstation with GPUs. The available software frameworks to implement and train CNNs provide strong support for this kind of platform.

The massive amounts of compute time spent training CNNs has spurred the development of highly optimized GPU implementations. First, most widely used frameworks relied on their own custom implementations which have all converged to methods relying on matrix-multiplications, leveraging the availability of highly optimized code in BLAS libraries and the fact that GPUs are capable of achieving a throughput within a few percent of their peak performance with this type of workload. Specialized libraries such as Nvidia's cuDNN and Nervana Systems' Neon provide some additional performance gains through assembly-level implementations [19] and additional algorithmic improvements such as Winograd and FFT-based convolution [20]. A specific implementation for non-batched inference on an embedded platform building on a matrix multiplication is documented in [14], also showing that more than 90% of time is spent computing convolutions.

B. Approximations Trading Accuracy for Throughput

DNNs commonly require a high computation effort in the order of 20 GOp/frame for classification of a 224×224 pixel image (1 multiply-add is counted as 2 operations) [28]. Extracting features when working with high resolution images (e.g. for object detection or semantic segmentation) scales up the effort proportional to the number of pixels, quickly reaching few 100 GOp/frame.

Admitting limited accuracy losses in order to gain a higher throughput by approximating existing networks, inference algorithms, and arithmetic operations can help overcome the computational obstacles preventing widespread adoption of CNN-based algorithms on embedded and mobile platforms. Several such approaches are surveyed and compared in [29], [30]. In this section, we will provide an overview of different options that can be exploited.

One such option is the reduction of the required arithmetic precision to evaluation NNs. Various methods from normal fixed-point analysis to retraining networks to adapt for quantized weights and activations exist. On some off-the-shelf software programmable platforms, 16-bit or 8-bit arithmetic operations can be vectorized to obtain a performance boost [31]. Extreme methods go as far as to enforce binary weights [32], [33], and in some cases also binary activations [26]. This means that multiplications can be dropped entirely, and in case of binary activations even collapse some of the add/subtract operations into XNOR and bit count operations. Many networks can be quantized with 8 bit without an increase in error rate, before there is a trade-off between precision and accuracy [21], [34]. Some methods try reducing the computational effort by pruning many very small weights to zero, making it possible to skip some operations [35], or even dynamically skip operations when the activations are zero [36]. More sophisticated quantization schemes such as vector quantization exist and can further compress a trained CNN model, but they require specialized hardware to bring an improvement in energy efficiency [36], [37].

Further research has focused on optimizing semantic segmentation and object detection algorithms to better reuse already computed features by eliminating any non-convolutional elements from the network [38], [39] or introducing structured sparsity [40]. Simplifying the operations in a network, such as low-rank approximations of 2D convolutions or by simply designing smaller networks with state-of-the-art methods have been evaluated in [28], [41].

The method we propose in this paper does not supersede these methods, but can be combined with the aforementioned approximation methods to further improve throughput.

[1]The Nvidia Tegra X2 is a system-on-chip available on an embedded board with an affordable power budget (< 15 W) for a stationary camera.

C. Video-based Computation Reduction

Obtaining per-frame features naturally seems like an easier task when these frames belong to a video sequence rather than a random collection of images. Limited movement of objects in a frame can be exploited in object tracking by working with a limited search window within the frame [42], not only reducing the problem size, but also simplifying the regression task—up until the tracked target is occluded by a large object.

Clockwork CNNs [43] specifically target CNNs for semantic segmentation with a structure similar to [39]. They have extended this work on fully convolutional networks, which presents a CNN with skip connections and deconvolution layers to refine the lower-resolution feature maps obtained deep within the network using the features extracted early in the network. They exploit that lower-resolution feature maps within the network are more stable over time than the full-resolution input. They thus propose to reevaluate the first few layers and the last layers affected through the skip connections more frequently than the more coarse grained feature maps. This is a strong limitation on the set of CNNs this method can be applied to. They present evaluations based on a static as well as a dynamic, content-adaptive reevaluation schedule, showing that they can reduce the number of full-frame convolutions by about 40% before the accuracy starts to drop on the Youtube-Objects dataset. However, this approach is limited to updating entire frames, whereas we exploit that often only small parts of the scene change and need to be reevaluated, which leads to larger savings.

CNNCache [44] describes a general approach pursuing a similar direction of work. They describe their method as a caching mechanism, where blocks of the image are matched to blocks in the previous frame, thereby fetching results of similar block from the cache instead of recomputing the results. Similarly to our work, this requires the selection of a threshold, and on top of that a block size and a cache depth in the form of an expiration time. The block matching allows to handle video data where the camera is not fully static, but it does not allow perspective changes. They have shown that their method achieves an average speed-up in the order of 20% at a top-1 accuracy loss of 3.5% performing image classification relative to the *ncnn* framework's default implementation. The capability to recall convolution results even when the specific image tile has moved introduces a significant overhead comparing image tiles, thereby limiting the potential speed-up significantly. Further, this methods requires a relatively high tolerance when comparing image tiles to be able to find matches, thereby introducing significant accuracy losses.

DeepMon [45] proposes another method combining convolution layer decomposition, half-precision computation, and convolutional layer caching. Similarly to CNNCache, they divide the input to each convolutional layer into blocks and reuse the result when a block matches to the one in the previous frame. To reduce overhead, they do not directly compare the blocks, but instead extract histogram-based features. They apply their technique only to the first few layers, because in later layers the caching overhead exceeds the compute latency savings. They show a speed-up attributable to caching of 18% for object detection and 36% for image classification at an accuracy loss in the order of 3.8% to 6.2%. While their histogram-based comparison method for the image tiles reduces overhead, it still remains significant and the introduced accuracy loss increases further.

Sigma-Delta Quantized Networks [46] is the most similar method to ours. They combine quantizing the network and decomposing the input to each convolution layer with the difference of the current frame's values to the previous frame's values and accumulate the result over time. They show a $4-10\times$ reduction in the number of operations in total, of which $2-3\times$ can be attributed to the temporal differences aspect of their method at an accuracy drop. However, whether this reduction in number of multiply-add operations can be put into performance gains after all the introduced overhead remains an open question.

D. Suitable Datasets and Neural Networks

We show the applicability of the concept to various applications, namely by evaluating the proposed method for semantic segmentation and pose detection. These are both often applied to high-resolution images and video streams with high frame rates above 10 frame/s for meaningful applications.

We are specifically interested in video sequences obtained from a static camera. While some such datasets exist (e.g. person or vehicle detection or re-identification), most of them are limited to extremely few (1-3) classes and rarely target semantic segmentation. However for first application scenario—semantic segmentation—the dataset[2] used in [47] provides ground truth labels for 10-class semantic segmentation from an urban street surveillance perspective, and while they work with individual images, several surrounding unlabeled frames and a trained convolutional network are available. An example image labeled with the provided CNN is shown in Figure 1, and a sample sequence of 3 images is visualized in Figure 2.

For the second application—pose detection—several datasets to detect joints and limbs exist in the form of annotated images or a moving camera frame sequences, but none with a static camera. To overcome this and to show the feasibility of applying CBinfer without annotated data, we use

[2]Available online at https://doi.org/10.3929/ethz-b-000276417

TABLE I
SEMANTIC SEGMENTATION CNN USED FOR EVALUATIONS.

	Type	Outp. Res.	Feat. Maps	CT [ms]	rel. CT
L1	conv 7×7	541×871	$3\to16$	72.9	13.6%
L2	act., pool 2×2	271×436	$16\to16$	10.7	2.0%
L3	conv 7×7	271×436	$16\to64$	116.2	21.7%
L4	act., pool 2×2	136×218	$64\to64$	10.2	1.9%
L5	conv, act. 7×7	136×218	$64\to256$	309.4	57.8%
L6	conv, act. 1×1	136×218	$256\to64$	14.4	2.7%
L7	conv 1×1	136×218	$64\to8$	1.6	0.3%
	Total			535.4	

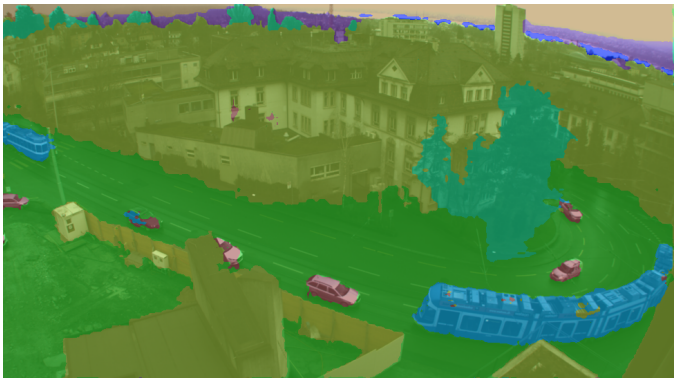

Fig. 1. Example output of the scene labeling network of [47] on which we evaluate our algorithm.

Fig. 2. A sample video sequence from the dataset of [47] showing the frame-by-frame changes by overlaying a sequence of length 3. Moving objects are only a small part of the overall scene and affect only a small share of the pixels.

unlabeled frame sequences from the CAVIAR dataset[3] and take the pretrained network to generate the reference output. The dataset contains scenes recorded using surveillance cameras with wide-angle lenses and captures the interaction of few people. It has a resolution of 384×288 pixel and a frame rate of 25 frame/s. A few sample frames are shown in Figure 3.

III. Methodology

The most straight-forward pixel-level approach is to detect changing pixels on the input frame based on a threshold on the difference to the previous frame, and then update all the pixels affected by them. This increases the number of pixels to be updated layer-after-layer due to the convolution operations. Thus for e.g. a 7×7 convolution, a one-pixel change triggers an update of 49 pixels in the next layer and 169 pixels after another 7×7 convolution. Strided operations (often used with pooling layers) reduce this effect, but do not prevent it. This issue might seem prohibitive for multi-layer CNNs, particularly when considering that individual pixels might keep exceeding the threshold due to noise.

[3]Available at http://homepages.inf.ed.ac.uk/rbf/CAVIAR/, collected through the EC Funded CAVIAR project/IST 2001 37540.

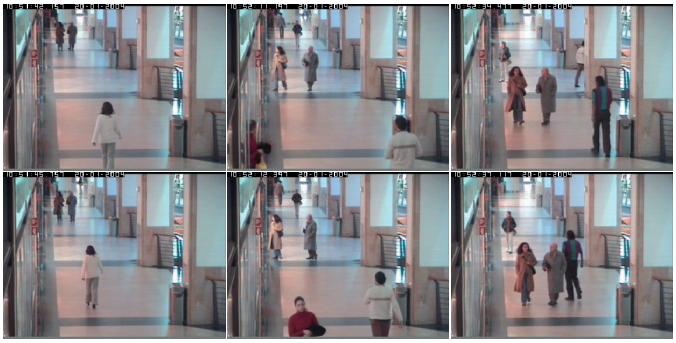

Fig. 3. Sample frames from sequences of the CAVIAR dataset on which we perform evaluations for detection.

However, the change is not only spatially local at the input, but also at the output. Furthermore, noise-like changes will likely not have strong impacts on feature maps deeper within the network. We thus propose to perform the change-detection not only at the input, but before each convolution layer—relative to its previous input—and to compute an updated value only for the affected output pixels. This can be done without modifications to the training of the CNN, can be applied to existing pre-trained networks, and is not specific to the CNN on which we evaluate the proposed algorithm.

We propose to replace all spatial convolution layers (conv layers) with *change-based* spatial convolution layers (CBconv layers). This means adapting the widely used, simple and well-performing matrix-generation and matrix-multiplication sequence of operations [14], [48]. The convolution layer computes

$$y_o(j,i) = b_o + \sum_{c \in \mathcal{C}_{in}} \sum_{(\Delta j, \Delta i) \in \mathcal{S}_k} k_{o,c}(\Delta j, \Delta i) x_c(j - \Delta j, i - \Delta i),$$

where o indexes the output channels $\mathcal{C}_{out}$ and c indexes the input channels $\mathcal{C}_{in}$. The pixel is identified by the tuple (j,i) and $\mathcal{S}_k$ denotes the support of the filters kernels k. This can be computed by performing a matrix multiplication

$$\mathbf{Y} = \mathbf{K}\mathbf{X}, \quad \mathbf{Y} \in \mathbb{R}^{|\mathcal{C}_O| \times h_o \cdot w_o}, \tag{1}$$

$$\mathbf{K} \in \mathbb{R}^{|\mathcal{C}_O| \times |\mathcal{C}_I| \cdot h_k \cdot w_k}, \quad \mathbf{X} \in \mathbb{R}^{|\mathcal{C}_I| \cdot h_k \cdot w_k \times h_o \cdot w_o}. \tag{2}$$

The image matrix $\mathbf{X}$ is constructed as $X((kh_k + j)w_k + i, y_o w_o + x_o) = x(k, j + y_o, i + x_o)$ with $k = 1, \ldots, |\mathcal{C}_{in}|$, $j = 1, \ldots, h_k$, $i = 1, \ldots, w_k$ and $y_o = 1, \ldots, h_o$, $x_o = 1, \ldots, w_o$. The filter matrix $\mathbf{K}$ is given by $K(o, (ch_k + j)w_k + i) = k(o, c, j, i)$ for $o = 1, \ldots, |\mathcal{C}_{out}|$, $c = 1, \ldots, |\mathcal{C}_{in}|$, $j = 1, \ldots, h_k$ and $i = 1, \ldots, w_k$. The result matrix is stored as $Y(o, y_o w_o + x_o) = y(o, y_o, x_o)$. Zero-padding can be applied during the construction of the $\mathbf{X}$ matrix and an efficient strided convolution can be computed by dropping the unused rows.

We replace this matrix multiplication by the following sequence of processing steps, thereby drastically reducing the size of the matrix used in the main computation step.

A. Processing Steps

We modify the standard approach and use a sequence of processing steps (cf. Figure 4, top/*feed-forward*): change detection, change indexes extraction, matrix generation, matrix

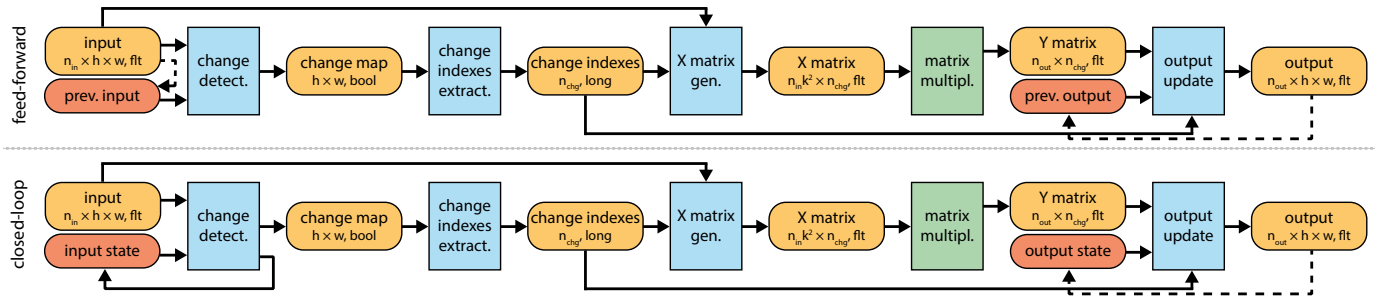

Fig. 4. Processing flow and intermediate data tensors of CBinfer. Color code: custom processing kernels, cuBLAS kernel, variables sharable among layers, and variables to be stored per-layer. Size and data type of intermediate results are indicated below the variable name.

multiplication, and output update. In the following, we will explain the individual steps.

a) Change Detection: In this step, changed pixels are detected. We define a changed pixel as one where the absolute difference of the current to the previous input of any feature map/channel exceeds some threshold τ, i.e.

$$m(j,i) = \bigvee_{c \in \mathcal{C}_I} \left(|x^{(t)}(c,j,i) - x^{(t-1)}(c,j,i)| > \tau \right). \quad (3)$$

The computation effort of this step is crucial, since it is executed independently of whether any pixel changed. Each of these changes affects a region equal to the filter size, and these output pixels are marked for updating:

$$\widetilde{m}(j,i) = \bigvee_{(\Delta j, \Delta i) \in \mathcal{S}_k} m(j + \Delta j, i + \Delta i), \quad (4)$$

where $\mathcal{S}_k$ is the filter kernel support, e.g. $\mathcal{S}_k = \{-3, \ldots, 3\}^2$ for a 7×7 filter. All of this is implemented on GPU by clearing the change map to all-zero and having one thread per pixel, which—if a change is detected—sets the pixels of the filter support neighborhood in the resulting *change map*.

b) Change Indexes Extraction: In this step, we condense the change map $\widetilde{m}$ to 1) a list of pixel indexes where changes occurred and 2) count the number of changed pixels. This has been implemented by relying on the Thrust[4] copy_if function. The computed index list is later on needed to access only the needed pixels to assemble the matrix for the convolution.

c) Matrix Generation & Matrix Multiplication: Matrix multiplications are used in many applications, and highly optimized implementations such as the GEMM (general matrix multiplication) function provided by the Nvidia cuBLAS library come within a few percent of the peak FLOPS a GPU is capable to provide. Matrix multiplication-based implementations of the convolution layer relying on GEMM are widely available and are highly efficient [14], [49] as described above. The $\mathbf{X}$ matrix in (1) is not generated full-sized, but instead only those columns corresponding to the relevant output pixels are assembled, resulting in a reduced width equal to the number of output pixels affected by the changes in the input image. The $\mathbf{K}$ matrix is made up of the filters trained using normal convolution layers and keeps the same dimensions, so the computation effort in this step is proportional to the number of changed pixels and the matrix multiplication is in the worst case only as time consuming as the full-frame convolution.

d) Output Updating: We use the previously stored results and the newly computed output values along with the change indexes list to provide the updated output feature maps. To maximize throughput, we also include the ReLU activation of the affected pixels in this step, reducing the compute time by 1) not writing the value to memory and immediately reading them again—an independent ReLU layer is strongly memory bandwidth limited, and 2) only applying the ReLU operation to changed pixels.

B. Memory Requirements

The memory requirements of DNN frameworks are known to be very high, up to the point where it becomes a limiting factor for increasing the mini-batch size during learning and thus reducing the throughput when parallelizing across multiple GPUs. These requirements are very different when looking at embedded inference-only systems:

1) Inference is typically done on single frames and creating mini-batches would introduce often unacceptable latency and the benefit of doing so is limited to a few percent of additional performance [14].
2) During training, the input of each layer has to be stored in order to be able to compute the gradients. This is not required during inference.
3) Batch normalization layers, Dropout layer, etc. (if present) are considered independent layers during training. They can be absorbed into the convolution layer for inference.

To obtain a baseline memory requirement, we compute the required memory of common DNN frameworks performing convolutions using matrix multiplication with a batch size of 1. We assume an optimized network minimizing the number of layers, e.g. by absorbing batch normalization layers into the convolution layers or using in-place activation layers. This way 30M values need to be stored for the intermediate results, 264M values for the $\mathbf{X}$ matrix, and 873k values for the parameters. This can further be optimized by sharing $\mathbf{X}$ among all convolution layers and by keeping only memory allocated to storing only the output of two layers and switching back-and-forth between them, layer-by-layer. This reduces the

[4] https://thrust.github.io

memory footprint to 9M, 93M, and 872k values, and a total of 103M values for our baseline.

Applying our algorithm requires a little more memory, because we need to store additional intermediate results (cf. Figure 4) such as the change matrix, the changed indexes list, and the $\mathbf{Y}$ matrix, which can all again be shared between the layers. We also need to store the previous output to use it as a basis for the updated output and to use it as the previous input of the subsequent layer. For our sample network, this required another $\sim$60M values to a total of 163M values (+58%, total size $\sim$650 MB)—an acceptable increase and not a limitation, considering that modern graphics cards typically come with around 12 GB memory and even GPU-accelerated embedded platforms such as the Nvidia Jetson TX2 module provide 8 GB of memory.

C. Closed-Loop Formulation

In Figure 4a and Section III-A we describe the processing steps for a *feed-forward* implementation of CBinfer. However, note that this structure allows gradually changing inputs (e.g. two images are morphed over several frames with increments below the change detection threshold) to never trigger any update within the network and thus keep a stale result. In an outdoors surveillance setting, the effects could be even worse: consider a static scenery with a sunset and thus gradually changing brightness without triggering any update operation. Now a moving object passes, leaving a dark trace behind which has been updated under the changed lighting conditions.

To overcome such issues, we are proposing a closed-loop version of CBinfer as shown in Figure 4, bottom/*closed-loop*. Rather than storing the previous input, we now have an *input state*, which is updated only for those pixels which have triggered a change. This can be done directly in the change detection phase. This way, the previous output is consistently the convolution result of the input state and ensured not to drift far away from ideal result.

Since the previous input had to be stored before as well, this does not introduce any memory overhead. Moreover, in many cases it can even decrease compute time since only the few values where changes occurred have to be copied over from the *input* to the *input state*. For the feed-forward CBinfer the entire input tensor would have to be copied[5].

D. Fine-Grained Change-based Inference

In the proposed scheme, every output value affected by any change at the input is recomputed. As the convolution operation is linear, updates based on the difference to the previous frame can be computed to reduce the number of multiplications and additions in two ways:

1) Fine-grained across feature maps (FG-FM): Only some of the input feature maps affecting a given output value might have changed. An incremental update of the affected feature maps based on the difference of the change in input values relative to the previous frame would be sufficient.
2) Spatially fine-grained (FG-SP): Just because an output pixel is affected by an input pixel does not require that it is completely recomputed. With a 3×3 filter, a single pixel marked as change would trigger the re-computation of 9 pixels. Also here an incremental update based on differences is possible.

However, there are some drawbacks and limitations:

- For both approaches the structure of the core computation effort is less regular and can not be written as a dense matrix multiplication.
- The compute effort of the *change indexes extraction* scales linearly with the number of values that have to be checked for changes. In case of (1), the effort in this step is thus scale up by a factor of the number of input feature maps.
- The potential gains in case of (2) are limited. Changing pixels are typically clustered together and all that is being saved is a small halo on the change map around the changes. This can in most cases be expected to be in the range of a few percent.

E. Propagating Changes & Pooling

Change detection and change indexes extraction can contribute up to half of the compute time (cf. Section IV-F). In some cases, it is thus worth considered to skip these steps and

1) reuse the previous layer's change map and performing a simple update on it assuming worst-case propagation, or
2) when no propagation is happening (e.g. in case of 1×1 kernels), reuse directly the previous layer's change indexes.

Avoiding change detection also implies saving the memory to store the previous input for that layer. Besides the aforementioned advantages, there are some potential drawbacks.

In case of (1), only the change detection step can be avoided and replaced with a change propagation step, and the change indexes have to be extracted again. Also the changes spread out at every layer this is done.

For (2), there is no propagation of changes and both, change detection and change indexes extraction, can be skipped. So the only drawback is that a few changes might be updated although they would be discarded if the input would be checked against the current layer's threshold.

Besides for accelerating convolution layers, the above is also interesting for pooling layers which can also be implemented using a change-based approach. Since they typically follow a convolution layer, case (2) can be applied and the change-based update introduces no significant overhead but saves

[5]Note that one such tensor always has to be copied when applying CBinfer. Consider two CBinfer layers after each other. During the *update output* step of the first CBinfer layer, we copy the newly computed values into the *previous output* tensor and feed it to the next CBinfer layer as the *input*. If we would not copy the data from input to *previous input* here and instead just keep the memory address of the previous frame's input, it will be at the same location where the output of the first CBinfer layer's result will be stored when processing the next frame, thus directly modifying the *previous input* variable and thereby introducing incorrect behavior (i.e. there are never any changes, since ultimately the *input* and *previous input* would point to the same memory location)

compute time—mostly by reducing memory bandwidth as pooling layers are memory-bound operations.

F. Threshold Selection

The proposed algorithm adds one parameter to each convolution layer, the detection threshold. It is fixed offline after the training based on sample video sequences. A threshold of zero yields identical results to the non-change-based implementation, which has been used for functional verification.

For our evaluations, we perform an automated threshold selection process. First, all convolution layers are converted to change-based convolutions and batch normalization and ReLU layers are absorbed into the CBinfer layers wherever possible. We define and choose

1) A performance metric such as pixel-wise classification accuracy, intersect-over-union (IoU), mean average precision (mAP)—possibly, the loss function of the network,
2) a set of frame sequences to evaluate the network, where the last frame is ideally annotated. An obvious alternative in case of a lack of frame sequences with annotated last frame is the generation the comparison of the change-based network model's output to the output of the original model using an appropriate metric, and
3) an initial threshold, a factor determining the rate with which we adjust the threshold, and a maximum acceptable increment in quality loss per layer.

We then set all thresholds to zero and start to iteratively step from the first to the last layer of the network. For each layer, we set an initial threshold value and evaluate the model with the aforementioned metric and dataset. We increment the threshold by a fixed factor (e.g.1.1), re-evaluate, and repeat until the quality loss introduced by the current layer (with respect to a zero threshold) exceeds the maximum acceptable limit and then take the previous threshold value.

In case of a DNN with (re-)convergent paths, we perform the threshold selection on these paths independently while setting the thresholds for the other paths to zero.

The maximum acceptable quality loss can be set equally for all layers of the network. We focus on low accuracy loss configurations, and thus we are trying to select the threshold values such that they are right at the point where implementation losses are starting to occur. Nevertheless, we have observed best results by splitting the overall acceptable loss unevenly, allowing the first layer to introduce most of the loss.

IV. Results & Discussion

In this section, we will first present the evaluation environment and analyze the baseline compute time breakdown. We then analyze the threshold selection, the effect on accuracy and achievable throughput. We then perform a more in-depth analysis of the throughput to verify the quality of the GPU implementation and investigate how the changes propagate in the network. We then establish why more fine-grained change detection does not pay off and how implementation loss and performance gains behave on longer sequences.

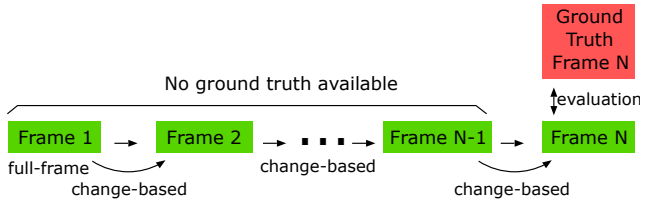

Fig. 5. Scheme of the image sequence used for the evaluations.

A. Evaluation Environment

We evaluate our method for two application scenarios: semantic segmentation and pose detection. For the first, we perform our evaluations on the urban surveillance dataset described in Section II-D and [47] and using the corresponding scene labeling CNN, not using the multispectral imaging data. The dataset provides 51 training images and 6 validation images with 776×1040 pixel with the corresponding ground-truth scene labeling, classifying each pixel into one of the following 8 classes: building, road, tree, sky, tram, car/truck, water, distant background. For the validation set, the labeled images are part of short video sequences with 5 additional frames available before the frame for which the ground truth labeling is available. A trained network on this data is described in [47] and its parameters are reused unaltered for our evaluations. The procedure with which we perform our evaluation is visualized in Figure 5.

For the pose detection application, we use frame sequences from the CAVIAR dataset without ground truth annotations and the trained body estimation network of OpenPose [50] with $T = 2$ stages. The frames are re-sampled to 368×490 pixel as in the original OpenPose implementation to enable a meaningful comparison. The frame sequences are subsampled in time by a factor of 6 to arrive at a frame rate of around 4 frame/s. In this setting, we measure the accuracy loss in terms of mean-squared error (MSE) relative to the output of the non-change-based network. We have found a MSE of $2 \cdot 10^{-4}$ on the network's output to be sufficient for the pose detection to work reliably. With this dataset we run change-based inference for 9 frames before the accuracy and throughput measurements are performed on Frame 10 to avoid any start-up transients. As we will show later in Figure 15, these transients are very short and the error does not accumulate over time.

We have implemented the proposed algorithm in the PyTorch framework using custom CUDA kernels, including functions to aid in converting DNNs to CBinfer (automatic conversion and threshold selection). We have evaluated the performance on a Jetson TX2 board. Our performance baseline is the PyTorch implementation using Nvidia's cuDNN backend. It includes optimizations such as the Winograd algorithm and FFT-based convolutions mentioned in Section II-A. Our evaluations were conducted using half-precision floating point numbers which have no negative impact on accuracy for both DNNs.

B. Baseline Throughput and Computation Breakdown

Before we discuss the performance of the proposed algorithm, we analyze the baseline throughput and compute time

TABLE II
PERFORMANCE BASELINE COMPUTE TIME BREAKDOWN

Layer	Conv.	Activ.	Pooling	total	share
1	72.9 ms	7.4 ms	3.3 ms	83.6 ms	15.6%
2	116.2 ms	6.9 ms	3.3 ms	126.4 ms	23.6%
3	302.8 ms	6.6 ms	—	309.4 ms	57.8%
4	12.7 ms	1.7 ms	—	14.4 ms	2.7%
5	1.6 ms	—	—	1.6 ms	0.3%

breakdown of the segmentation DNN in Table II. Clearly, the convolution operations are dominant, taking up 94.5% (506.2 ms) of the overall computation time (535 ms). This reaffirms the focus on the convolution layers and will later on show that after accelerating the convolution operation significantly, optimizations for activation and pooling become relevant.

C. Threshold Selection

Our algorithm introduces a threshold parameter for each layer, for which we outline the selection process in Section III-F. In Figure 6 we visualize the relation between accuracy and each layer's change detection threshold. We proceed similarly to our selection process, allowing an accuracy drop of 0.04% per layer for the semantic segmentation network. Starting from all-zero thresholds ($\tau_i = 0, i = 1, \ldots, 3$), we sweep and select the optimal threshold parameter for each layer iteratively. The main purpose is to align the tipping points of the threshold-accuracy curve, such that not a single layer's threshold is limiting the overall accuracy.

After the selection of the thresholds, we can scale them jointly to analyze the trade-off against the classification accuracy more concisely as can be observed in Figure 7 (left). The accuracy of the individual test sequences (different traces) clearly show a similar behavior with a plateau up to a clear point where there is a steep increase in error rate. We repeated this analysis for the much deeper pose detection network (cf. Figure 8), showing similar behavior for the MSE with respect to the baseline DNN.

D. Throughput Evaluations

The motivation for the proposed algorithm was to increase throughput by focusing only on the frame-to-frame changes. We show the performance gain in Figure 7 (right) with the indicated baseline analyzing the entire frame with the same network using cuDNN. In the extreme case of setting all thresholds to zero, the entire frame is updated, which results in a clear performance loss because of the change detection overhead as well as fewer optimization options such as less cache-friendly access patterns when generating the $\mathbf{X}$ matrix. Nevertheless, few operations are skipped where the pixels did not change at all.

When increasing the threshold factor, the average throughput increases rapidly to about 20 frame/s, where it starts saturating because the change detection step as well as other non-varying components like the pooling and pixel classification layers are becoming dominant and the number of detected changed pixels does not further decrease. We almost reach this plateau already for a threshold factor of 1, where we have by construction almost no accuracy loss. The average frame rate over the different sequences is near 18 frame/s at this point—an improvement of 9.1× over the cuDNN baseline of 1.96 frame/s.

One sequence (Figure 7, ) has—while still being close to 5.1× faster than the baseline—a significantly lower throughput than the other sequences. While most of them show typical scenarios such as shown in Figure 2, this sequences shows a very busy situation where the entire road is full of vehicles and all of them are moving. The effective number of operations (add or multiply operations) to compute the convolution updates is visualized in Figure 7 (center). For most frame sequences the savings are above 10× while the aforementioned exceptional cases has a significantly higher share with savings of around 5×.

Running the same analysis for the pose detection network yields very similar results. For the cuDNN baseline we get a frame rate of 0.72 frame/s and CBinfer achieves a rate of 3–8 frame/s for a threshold factor of 1 or a speed-up of 4.2× to 11.1×. A noticable difference are performance gains for the zero threshold configuration. Here the overhead of CBinfer is outweighed by the savings due to many pixels at the input not changing at all and therefore not triggering an update even for a zero threshold, yielding a performance gain even in a completely loss-less configuration.

We have repeated the performance measurements with fp32 precision on a workstation with a Nvidia GTX 1080 Ti GPU to compare them to the Tegra X2 platform, obtaining an almost identical throughput-threshold trade-off and compute time breakdown up to a scaling factor of 13.9×—as can be expected for a largely very well parallelizable workload and a 14.1× more powerful device with a similar architecture[6].

E. Accuracy-Throughput Trade-Off

While for some scenarios any drop in accuracy is unacceptable, many applications allow for some trade-off between accuracy and throughput—after all choosing a specific CNN already implies selecting a network with an associated accuracy and computational cost.

We analyze the trade-off directly in Figure 9. The most extreme case is updating the entire frame every time resulting in the lowest throughput at the same accuracy as full-frame inference. Increasing the threshold factor in steps of 0.25 immediately results in a significant throughput gain and for most sequences the trade-off only starts at frame rates close to saturation above 20 frame/s. The same frame sequence that already deviated from the norm before behaves differently here as well. However, an adaptive selection of the threshold factor with a control loop getting feedback about the number of changed pixels could allow for a guaranteed throughput by reducing the accuracy in such cases and is left to be explored in future work.

[6]Tegra X2: 437-750 GFLOPS (fp32), 874-1500 GFLOPS (fp16), and 58.4 GB/s DRAM bandwidth.
GTX 1080 Ti: 10609 GFLOPS (fp32) and 484 GB/s.

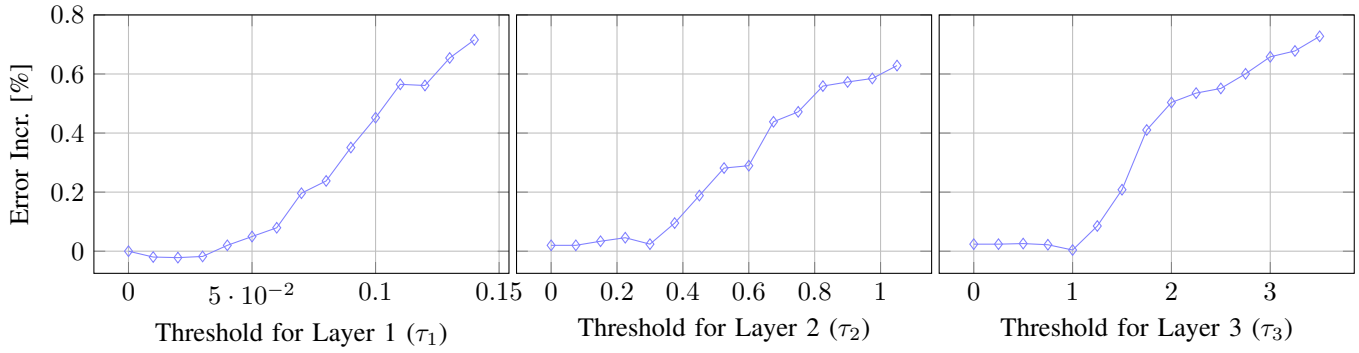

Fig. 6. Analysis of the increase in pixel classification error rate by selecting a certain change detect threshold. This analysis is conducted layer-by-layer, where the error increase of any layer includes the error introduced by the previous layers' threshold choice ($\tau_1 = 0.04, \tau_2 = 0.3, \tau_3 = 1.0$).

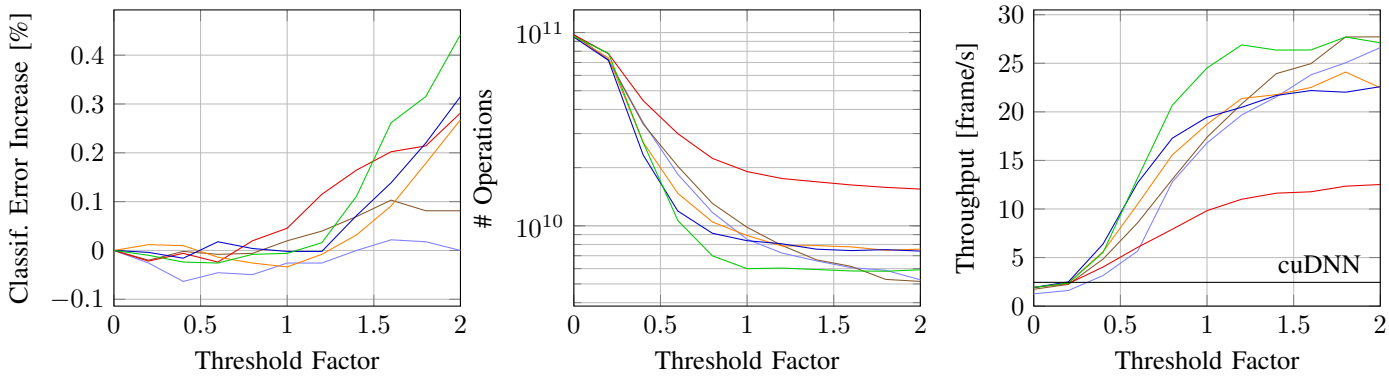

Fig. 7. Evaluation of the impact of jointly scaling the change detection thresholds on the classification error, the number of detected changed pixels (sum over all 3 layers), and the throughput.

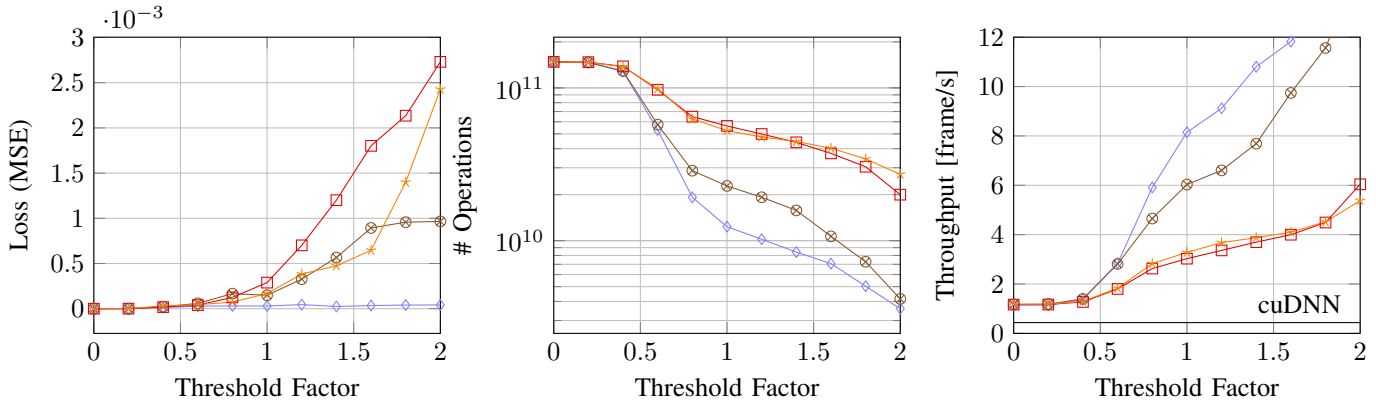

Fig. 8. Evaluation of introduced loss (left), effective number of compute operations (center) and measured throughput (right) for several frame sequences running the pose detection network and varying the change detection threshold.

F. Compute Time Breakdown

In Section IV-B and specifically in Table II, we already discussed the compute time breakdown of the entire network when using frame-by-frame analysis. To gain more in-depth understanding of the limiting factors of our proposed algorithm, we show a detailed compute time breakdown of only the change-based convolution layers in Figure 10. The time spent on change detection is similar across all 3 convolution layers, which aligns well with our expectations since the feature map volume at the input of $n_{ch} \cdot h \cdot w$ values is identical for L2 and L3, and 25% smaller for L1. That this step already makes up for more than 23.4% of the overall time underlines the importance of a very simple change detection function: any increase in compute time for change detection has to be offset by time savings in the other steps by reducing the number of changes significantly. The change indexes extraction effort is linear to the number of pixels $h \cdot w$ and the clear drop from L1 to L2 is as expected. However, since it is not well parallelizable, there is not much additional gain when comparing L2 to L3. The effort to generate the $\mathbf{X}$ matrix

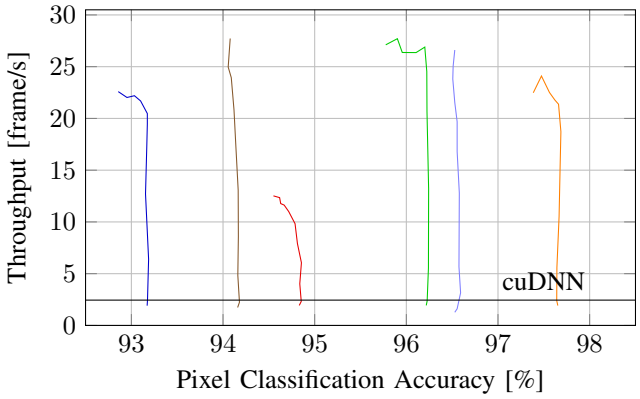

Fig. 9. Evaluation of the throughput-accuracy trade-off for all 6 video sequences.

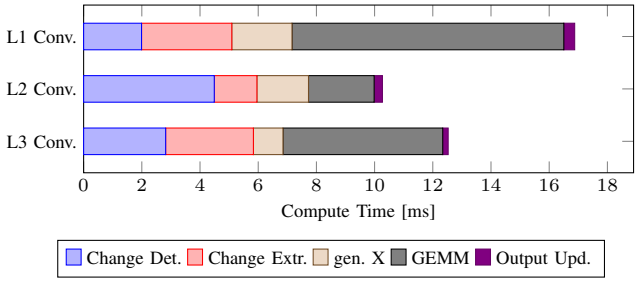

Fig. 10. Compute time for the individual processing steps per layer running on the GPU for a typical frame sequence.

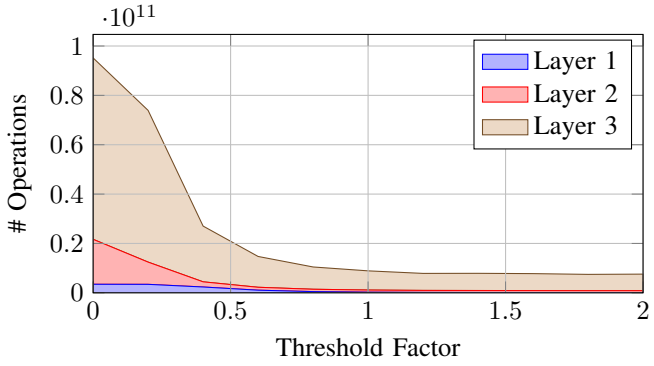

Fig. 11. Cumulative number of multiply and add operations for the scene labeling network.

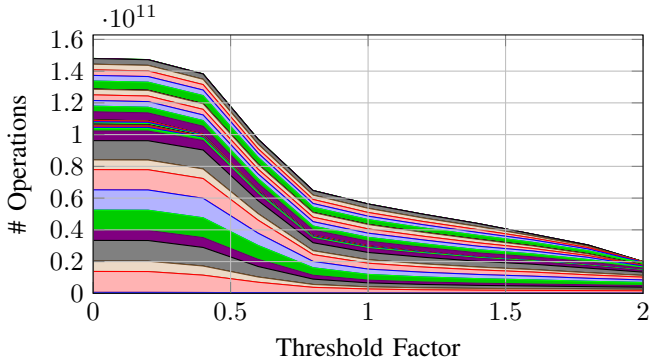

Fig. 12. Number of multiply and add operations for the pose detection network stacked by layer with first layer on bottom.

is very dependent on the number of changed pixels, the number of feature maps, and the filter size. It is, however, most important that the time spent on shuffling data around to generate $\mathbf{X}$ is significantly smaller than the actual matrix multiplication, which clearly makes up the largest share. The subsequent update of the output values including activation only uses a negligible part of the overall processing time.

An important aspect is not directly visible: The overall compute time for the dominant convolution layers, has shrunk tremendously by more than $12.9\times$ from $512.8\,\mathrm{ms}$ to about $39.7\,\mathrm{ms}$. This makes the pooling layer a non-negligible contributor to the overall compute time (total $6.6\,\mathrm{ms}$). As outlined in Section III-E, we can perform the pooling also with a change-based approach and skip the change detection and indexes extraction by relying on the preceeding convolution layer's change indexes. This provides an additional speed-up by an average of $5.8\times$ and $4.5\times$ for the first and second pooling layer, respectively.

G. Change Propagation

During the construction of the algorithm we argued that change detection should be performed for every convolution layer not only for modularity, but also justifying that the worst-case change propagation would result in a rapid growth of the share of changed pixels as we proceed deeper into the network. However, skipping change detection and instead assuming worst-case propagation for some intermediate layers might improve performance. Our experiments have shown that for neither of the networks this pays off. An experiment has shown that for Layer 2, the number of changes is reduced by $6.8\times$ from 7.57% to 1.11% and for Layer 3 from 2.58% to 1.94% by $1.33\times$. Not repeating change detection for some layer affects the compute time:

1) reducing the compute time by substituting the change detection step with a more light-weight change propagation step,
2) scaling up the compute effort from the matrix generation through the output update proportionally to the increase in the number of pixels marked as change, and
3) leaving the execution time of the change indexes extraction unaffected.

Combining this with the results in Figure 10, skipping change detection for Layer 2 would result in an increase in execution time by a factor of $2.9\times$. For Layer 3 it would result in approximately no effect on performance.

An immediate concern evaluating a CNN based on changing pixels is the spreading of the affected regions though the convolutions. We have thus analyzed the effective number of compute operationsin Figure 11 and Figure 12 for the semantic segmentation and the pose detection networks, respectively. For each layer the number of compute operations is shown in dependence of the joint threshold scaling factor. Layers in parallel branches of the network are shown sequentially. The changes are neither spreading out nor vanishing as we proceed deeper into the DNN.

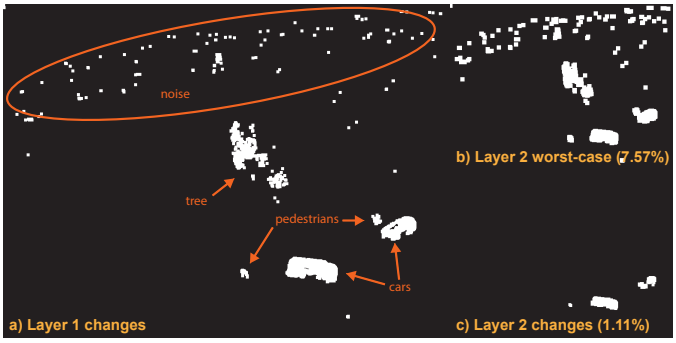

Fig. 13. Analysis of the change propagation. (a) shows the changes detected in Layer 1 using the thresholds determined in Section IV-C, in the upper part of the image there are several single-pixel changes due to noise. We show the changed pixels for Layer 2 based on worst-case propagation as assumed when dropping the Layer 2 change detection step (b) and those when applying change detection instead (c).

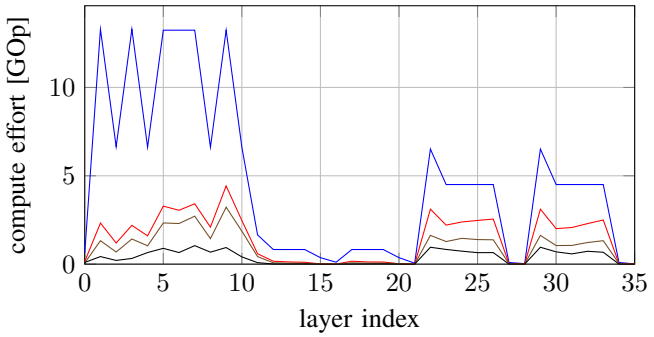

Fig. 14. Analysis of the compute effort by layer for the pose detection network. We compare not using CBinfer ( ) to different granularities of CBinfer: normal (coarse-grained) ( ), spatially fine-grained ( ), and feature map fine-grained ( ).

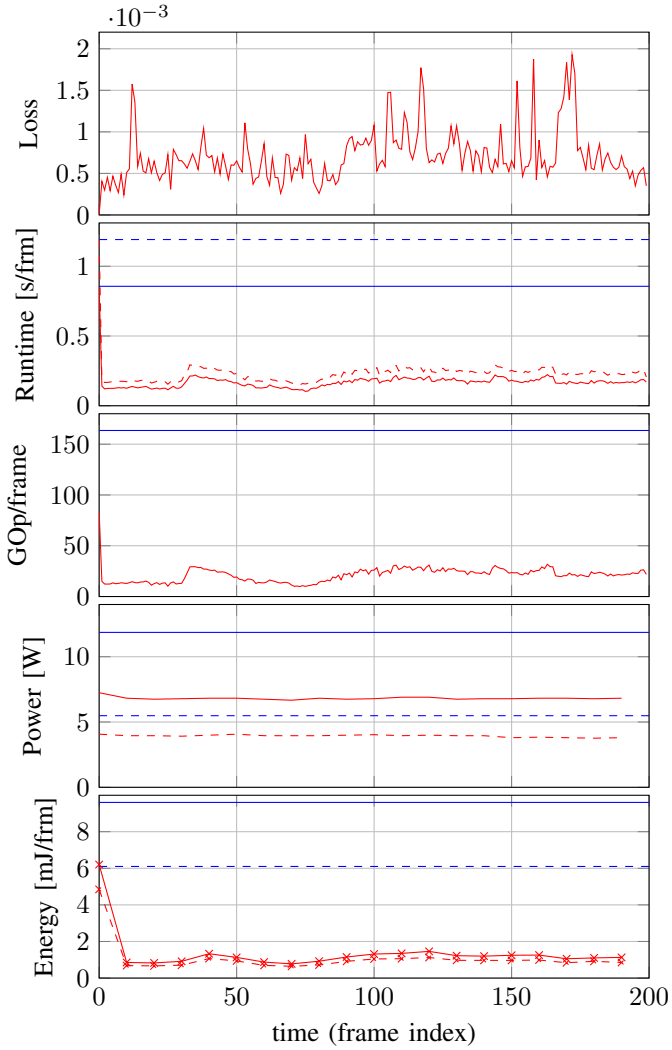

Fig. 15. Evaluation of accuracy effect, runtime, number of compute operations, power and energy when performing inference for the pose detection network on a 200-frame video sequence. **Legend:** CBinfer, cuDNN; continuous: max-N (max. performance) power mode, dashed: max-Q (max. efficiency) power mode.

Besides the effect on performance, the visualization of these changes in Figure 13 provides insight into the inner workings of the DNN. As expected, single-pixel artifacts such as noise disappear due to the smoothing effect of the convolution. Changes originating from moving objects such as pedestrians and cars are propagated to the next layer as desired. A particularly interesting observation is the effect on the region marked as tree: In the input frame sequence the leaves of the tree move in the wind, but already after the first convolution layer the resulting changes completely vanish. We construe these pixels in this reagion to already be represented more abstractly as *leaves*.

H. Fine-grained CBinfer

In Section III-D we have introduced two types of fine-grained CBinfer to further reduce the number of multiply-add operations: spatially and across feature maps. We analyze this effect by running change-based inference and comparing the compute effort to the number of detected changes and number of operations to perform per change in Figure 14. The drawbacks discussed in Section III-D are confirmed:

- Spatially fine-grained (SP-FG) CBinfer reduces the number of operations only by around 20% while exploiting this lets the operations become much less regular and the larger matrix multiplication decays into many smaller ones followed by an aggregation step, where both introduce a massive memory bandwidth overhead.
- The results for fine-grained evaluation by feature map show much more potential based on a reduction of multiply and add operations by around 65%. However, such an implementation also requires a change map per feature map and thus the change extraction step has to be performed on a 3D tensor rather than a 2D tensor. The effort is scaled up by the number of feature maps at the input of the convolution layer (often 16, 64, 256, or more), thereby pushing this computation overhead (for normal CBinfer from 10-20% of compute time, cf. Figure 10) to several times (160-5120%) the overall compute effort of normal CBinfer.

I. Energy Efficiency

We have measured the power consumption of the Tegra X2 module using the on-board sensors for two of its power modes: maximum performance (max-N) and maximum efficiency (max-Q). When idling, the power consumption is 1.80 W and 1.77 W for max-N and max-Q, respectively. The measurements under load have been conducted while running pose detection on a 200-frame sequence and are visualized in Figure 15. Generally, we can see a clear correlation between the number of operations that have to be computed and the runtime, where the later has a clear offset due to the overhead of change detection and change indexes extraction. We can also observe that there is no long-term rise in the introduced loss. The power is very constant for the cuDNN baseline in max-N (——, 12 W) and max-Q mode (- - -, 5.3 W) as well as for the CBinfer implementation (—×—, 6.8 W) and (- × -, 4.8 W), respectively. Note that we were processing the frames without duty-cycling. The resulting energy efficiency is shown in the trace at the bottom. The baseline uses around 9.6 J/frame in max-N mode and 6.1 J/frame in max-Q mode, whereas the CBinfer implementation uses an average of 1.1 J/frame and 0.8 J/frame, respectively. This corresponds to energy savings of $8.7\times$ and $7.6\times$ and an equivalent average energy efficiency of 148 GOp/s/W and 204 GOp/s/W for the max-N and max-Q power modes, respectively.

For the scene labeling network and the max-N power mode we have measured a power consumption of 6.8 W with CBinfer and 10.5 W with cuDNN and thus 411 and 3003 mJ/frame, respectively. With a frame requiring 210 GOp, this results in an energy efficiency of 511 and 70 GOp/s/W—an improvement by $5.9\times$.

V. Conclusion

We have proposed and evaluated a novel algorithm for change-based evaluation of CNNs for video recorded with a static camera setting, exploiting the spatio-temporal sparsity of pixel changes. The method introduces a set of parameters to trade-off accuracy and throughput. Even when choosing the parameters conservatively to introduce no significant accuracy loss, we have observed an average speed-up by $9.1\times$ for a semantic segmentation DNN and $7.0\times$ for a pose detection DNN relative to cuDNN using our GPU implementation. The resulting boost in energy efficiency over per-frame evaluation is an average of $8.7\times$ and $5.9\times$ for the two applications, respectively. This corresponds to an equivalent energy efficiency of 511 GOp/s/W on the Tegra X2 platform for the pose detection DNN. We have analyzed various flavors of the proposed algorithm and how the changes propagate through the DNNs to further underline the optimality of the structure of the proposed algorithm.

Further gains might be possible by training the network on videos using change-based inference for the forward propagation or by introducing noise to simulate the slight deviations for the ideal feature maps. The proposed method should also not be limited to video data, but work on any data where changes in at least one dimension are sparse (e.g. spectrograms of audio data). Finally, reducing the granularity of the algorithm by 2×2 or 4×4 would allow an implementation using Winograd's convolution algorithm for additional speed-up, like it is being done in the cuDNN baseline.

References

[1] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[2] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Proc. ECCV*, 2014.

[3] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The Cityscapes Dataset for Semantic Urban Scene Understanding," in *Proc. IEEE CVPR*, 2016, pp. 3213–3223.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proc. IEEE ICCV*, dec 2015, pp. 1026–1034.

[6] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan, "YouTube-8M: A Large-Scale Video Classification Benchmark," *arXiv:1609.08675*, 2016.

[7] P. Fischer, A. Dosovitskiy, E. Ilg, P. Haeusser, C. Hazirbas, V. Glokov, P. Van der Smagt, D. Cremers, and T. Brox, "FlowNet: Learning Optical Flow with Convolutional Networks," in *arXiv:15047.06852*, 2015.

[8] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *Proc. ACM ICCAD*, New York, NY, USA, 2016.

[9] W.-s. Park and M. Kim, "CNN-based in-loop filtering for coding efficiency improvement," in *Proc. IEEE IVMSP*, 2016.

[10] P. Wang, Y. Cao, C. Shen, L. Liu, and H. T. Shen, "Temporal Pyramid Pooling Based Convolutional Neural Networks for Action Recognition," vol. 27, no. 12, pp. 2613–2622, 2015. [Online]. Available: http://arxiv.org/abs/1503.01224

[11] K. Chen and W. Tao, "Once for All: a Two-flow Convolutional Neural Network for Visual Tracking," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8215, no. c, pp. 1–10, 2017.

[12] K. Kang, H. Li, J. Yan, X. Zeng, B. Yang, T. Xiao, C. Zhang, Z. Wang, R. Wang, X. Wang, and W. Ouyang, "T-CNN: Tubelets with Convolutional Neural Networks for Object Detection from Videos," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8215, no. c, pp. 1–11, 2017.

[13] Z. Jie, W. F. Lu, S. Sakhavi, Y. Wei, E. H. F. Tay, and S. Yan, "Object Proposal Generation With Fully Convolutional Networks," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 28, no. 1, pp. 62–75, jan 2018.

[14] L. Cavigelli, M. Magno, and L. Benini, "Accelerating Real-Time Embedded Scene Labeling with Convolutional Networks," in *Proc. ACM/IEEE DAC*, 2015.

[15] G. Ananthanarayanan, P. Bahl, P. Bodik, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, "Real-Time Video Analytics: The Killer App for Edge Computing," *Computer (Long. Beach. Calif).*, vol. 50, no. 10, pp. 58–67, 2017.

[16] L. T. Nguyen-Meidine, E. Granger, M. Kiran, and L.-A. Blais-Morin, "A comparison of CNN-based face and head detectors for real-time video surveillance applications," in *Proc. IEEE IPTA*, nov 2017.

[17] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," *arXiv:1802.04730*, 2018.

[18] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," in *arXiv:1410.0759*, 2014.

[19] A. Lavin, "maxDNN: An Efficient Convolution Kernel for Deep Learning with Maxwell GPUs," in *arXiv:1501.06633v3*, 2015.

[20] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," in *Proc. IEEE CVPR*, 2016, pp. 4013–4021.

[21] L. Cavigelli and L. Benini, "Origami: A 803-GOp/s/W Convolutional Network Accelerator," *IEEE TCSVT*, vol. 27, no. 11, pp. 2461–2475, nov 2017.

[22] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An Ultra-Low Power Convolutional Neural Network Accelerator Based on Binary Weights," in *Proc. IEEE ISVLSI*, 2016, pp. 236–241.

[23] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A Convolutional Network Accelerator," in *Proc. ACM GLSVLSI*. ACM Press, 2015, pp. 199–204.

[24] Y.-H. Chen, T. Krishna, J. Emer, and V. Sze, "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks," in *Proc. IEEE ISSCC*, 2016, pp. 262–263.

[25] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun, "NeuFlow: A Runtime Reconfigurable Dataflow Processor for Vision," in *Proc. IEEE CVPRW*, 2011, pp. 109–116.

[26] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks," in *Proc. ECCV*, 2016, pp. 525–542.

[27] L. Cavigelli, P. Degen, and L. Benini, "CBinfer: Change-Based Inference for Convolutional Neural Networks on Video Data," in *Proc. ACM ICDSC*, 2017.

[28] A. Canziani, E. Culurciello, and A. Paszke, "Evaluation of neural network architectures for embedded systems," in *Proc. IEEE ISCAS*, may 2017.

[29] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *arXiv:1801.04381*, 2018.

[30] F. Iandola and K. Keutzer, "Small neural nets are beautiful," in *Proc. IEEE/ACM/IFIP CODES*. ACM Press, 2017.

[31] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented Approximation of Convolutional Neural Networks," in *ICLR Work.*, 2016.

[32] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights," in *Proc. ICLR*, 2017.

[33] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "Hyperdrive: A Systolically Scalable Binary-Weight CNN Inference Engine for mW IoT End-Nodes," in *Proc. IEEE ISVLSI*, 2018.

[34] S. Hashemi, N. Anthony, H. Tann, R. I. Bahar, and S. Reda, "Understanding the Impact of Precision Quantization on the Accuracy and Energy of Neural Networks," in *arXiv:1612.03940*, 2016.

[35] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning Filters for Efficient ConvNets," *arXiv:1608.08710*, 2016.

[36] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, R. Tapiador-Morales, I.-A. Lungu, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, and T. Delbruck, "NullHop: A Flexible Convolutional Neural Network Accelerator Based on Sparse Representations of Feature Maps," *arXiv:1706.01406*, jun 2017.

[37] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," in *Proc. ACM/IEEE ISCA*, 2016, pp. 243–254.

[38] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," in *Proc. IEEE CVPR*, jun 2016, pp. 779–788.

[39] J. Long, E. Shelhamer, and T. Darrell, "Fully Convolutional Networks for Semantic Segmentation," in *Proc. IEEE CVPR*, 2015.

[40] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices," in *arXiv:1707.01083*, 2017.

[41] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," in *arXiv:1602.07360*, N. Navab, J. Hornegger, W. M. Wells, and A. Frangi, Eds., vol. 9349, Cham, feb 2016.

[42] D. Held, S. Thrun, and S. Savarese, "Learning to Track at 100 FPS with Deep Regression Networks," in *LNCS*, ser. Lecture Notes in Computer Science, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, vol. 9905, pp. 749–765.

[43] E. Shelhamer, K. Rakelly, J. Hoffman, and T. Darrell, "Clockwork Convnets for Video Semantic Segmentation," *arXiv:1608.03609*, 2016.

[44] M. Xu, X. Liu, Y. Liu, and F. X. Lin, "Accelerating Convolutional Neural Networks for Continuous Mobile Vision via Cache Reuse," *arXiv:1712.01670*, 2017.

[45] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications," in *Proc. ACM MobiSys*, 2017, pp. 82–95.

[46] P. O. Connor and M. Welling, "Sigma-Delta Quantized Networks," *ICLR*, 2017.

[47] L. Cavigelli, D. Bernath, M. Magno, and L. Benini, "Computationally efficient target classification in multispectral image data with Deep Neural Networks," in *Proc. SPIE Secur. + Def.*, vol. 9997, 2016.

[48] Y. Jia, "Caffe: An Open Source Convolutional Architecture for Fast Feature Embedding," 2013. [Online]. Available: http://caffe.berkeleyvision.org

[49] J. Jin, V. Gokhale, A. Dundar, B. Krishnamurthy, B. Martini, and E. Culurciello, "An efficient implementation of deep convolutional neural networks on a mobile coprocessor," in *Proc. IEEE MWSCAS'14*, 2014, pp. 133–136.

[50] Z. Cao, T. Simon, S.-E. Wei, and Y. Sheikh, "Realtime Multi-person 2D Pose Estimation Using Part Affinity Fields," in *Proc. IEEE CVPR*. IEEE, jul 2017, pp. 1302–1310.