Accelerate Inference of CNNs for Video Analysis While Preserving Exactness Exploiting Activation Sparsity

Toshiaki Wakatsuki [1] **Sekitoshi Kanai** [1] **Yasuhiro Fujiwara** [1]

Abstract

This paper proposes a *range-bound-aware convolution layer* that accelerates the inference of rectified linear unit (ReLU)-based convolutional neural networks (CNNs) for analyzing video streams. Since video analysis systems require to process each video frame in real-time, the computational cost of inference of CNNs must be reduced. Several techniques heuristically skip the computation for the current frame and reuse the results of the previous frame when the current and previous frames are sufficiently similar. However, for critical applications such as surveillance systems, their accuracy can be unsatisfactory because they sacrifice accuracy for efficiency. In contrast, our method reduces the computational cost of convolution layers accompanied by ReLU while producing exactly the same inference results as an original model. We utilize both temporal similarity of video frames and activation sparsity in ReLU-based CNNs to guarantee to skip truly redundant computations. We experimentally confirm that our method can accelerate widely used pre-trained CNNs with both CPU and GPU implementations.

1 Introduction

Since video cameras have become pervasive, analyzing video streams is one of the key components of real-world applications (Jiao et al., 2019), such as monitoring security and transportation surveillance. Video analysis systems (Zhang et al., 2017; Kang et al., 2017) for these applications inevitably rely on convolutional neural networks (CNNs) because they have significantly improved accuracy of image classification and object detection. To analyze video streams, CNNs should process each video frame in real-time to detect events with low latency (Canel et al., 2019). In addition to real-time processing, the accuracy of the model is also important to accurately track objects (Bewley et al., 2016). Even though CNNs can achieve high accuracy, the computational cost of CNNs is too high to process video streams in real-time because each inference of single video frame requires billions of floating-point operations (FLOPs) (Bianco et al., 2018). Most of the FLOPs on CNNs are dominated by the computation of convolution layers, which is performed by general matrix-matrix multiplication (GEMM) of a weight matrix and an input matrix. While the utilization of relatively high-performance GPUs for GEMM is being supported by deep learning compilers (Wang et al., 2019), the inference of CNNs is often run on CPUs in real applications because CPUs are more widespread than GPUs (Wu et al., 2019).

To reduce the computational cost of convolution layers for video analysis, there are roughly two types of approaches: utilizing temporal redundancy of video streams and sparsification of matrices in CNNs. Prior works often utilized temporal redundancy of video streams, i.e., they utilized the similarity between consecutive video frames by detecting differences, caching results, and screening pixel-wise (Kang et al., 2017; Loc et al., 2017; Cavigelli & Benini, 2020). These methods heuristically skip the computation for the current frame and reuse the results of the previous frame when the current and previous frames are sufficiently similar. However, these methods lack theoretical guarantees to preserve accuracy of an original model since the model may fail to catch non-trivial changes that affect the inference results; it leads that event detection systems can miss events, or object tracking systems can fail to recognize the objects.

On the other hand, sparsification of matrices in CNNs can reduce FLOPs of matrix multiplication in convolution layers by skipping multiplications with zero via sparse matrix-dense matrix multiplication (SpMM). There have been many attempts to exploit sparsity in the weight matrix and the input matrix (Zhu & Gupta, 2017; Ji et al., 2018; Gale et al., 2019; Shomron & Weiser, 2019; Blalock et al., 2020). To obtain sparse weight matrices, Han et al. (2015) pruned nearly zero weights and fine-tuned the model. Furthermore, several studies improved the efficiency of SpMM by inducing structured sparsity (Wen et al., 2016), e.g., block-wise (Gray et al., 2017) and filter-wise (Luo et al., 2017). However, sparsification of the weight matrix and levels of structured

[1]Nippon Telegraph and Telephone Corporation, Tokyo, Japan. Correspondence to: Toshiaki Wakatsuki <toshiaki.wakatsuki.xg@hco.ntt.co.jp>.

Proceedings of the 4th MLSys Conference, San Jose, CA, USA, 2021. Copyright 2021 by the author(s).

sparsity sacrifice accuracy for efficiency, which can be unsatisfactory for critical applications that require high accuracy, such as surveillance systems.

The input matrices in intermediate convolution layers are naturally sparse since the model uses a rectified linear unit (ReLU) (Glorot et al., 2011) as an activation defined by

$$\mathrm{ReLU}(x) = \max(x, 0). \tag{1}$$

Since ReLU converts negative output values of a convolution layer to zero, the input matrix for the next convolution layer contains exact zero elements. Treating the input matrix as a sparse matrix yields a theoretical reduction of the FLOPs via SpMM and produces exactly the same results as an original model. However, it is difficult to gain practical speed-up because sparsity is not high enough for generic SpMM kernels to outperform optimized GEMM kernels in terms of convolution. Structured sparsity is difficult to induce in the input matrix because its sparsity depends on processed input data. Instead of directly utilizing sparsity of the input matrix, several researchers have attempted to predict output elements of convolution layers that become zeros after ReLU (Shomron & Weiser, 2019; Ren et al., 2018). They use the coarse-grained prediction to keep the prediction cost low and efficiently reduce the computational cost by structured sparsity. However, the coarse-grained prediction can wrongly treat non-zero output elements as zero regardless of the accurate values and deteriorate the accuracy of the inference.

This work aims to accelerate the inference on video streams while the accelerated model produces exactly the same inference results as an original model. We propose a *range-bound-aware convolution layer*, which can reduce the FLOPs of convolution layers in CNNs accompanied by ReLU activation. This is a drop-in replacement for the standard convolution layer because it produces exactly the same activation maps. Unlike the prior methods, our method utilizes both the temporal redundancy of video streams and activation sparsity so that it guarantees exactness. Our range-bound-aware convolution layer has two phases. First, it computes *range-bound*, which is our proposed upper bound of each output element. We define the upper bound derived from the difference between adjacent video frames utilizing similarity. Second, our method skips the computation of output elements that surely result in zero after ReLU using this upper bound. This enables removing the whole dot-product for one output element, which is more efficient than removing each scalar multiplication like SpMM.

In the experiments, we show that our method can effectively reduce FLOPs on widely used models such as VGG (Simonyan & Zisserman, 2015), ResNet (He et al., 2016), and SSD (Liu et al., 2015) on video streams from the YUP++ dataset (Feichtenhofer et al., 2017) without any changes in inference results for an original model. We found widely used models such as variants of VGG achieve 1.1–2.1× speed-up for each convolution layer using the straightforward serial implementations of our method run on CPUs compared with SpMM-based implementations. We also conducted experiments to demonstrate speed-up on GPUs, and VGG achieves 4% and up to 8% end-to-end speed-up with the careful implementation compared with a TVM deep learning compiler (Chen et al., 2018) with a vendor-optimized library (cuDNN[1]).

2 Notation

A 2D convolution layer takes two inputs: an input tensor $\mathcal{X} \in \mathbb{R}^{CHW}$ and a filter tensor $\mathcal{W} \in \mathbb{R}^{KCRS}$, where C and K are the number of input and output channels, respectively. $H \times W$ and $R \times S$ are the size of a 2D input and a filter kernel, respectively. An output element of convolution layer $Y_{k,i,j}$, which is in the k-th output channel at the spatial location (i, j), is given by

$$\begin{aligned} Y_{k,i,j} &= \textstyle\sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} \mathcal{X}_{c,i+r,j+s} \mathcal{W}_{k,c,r,s} & (2) \\ &= \boldsymbol{x}_{(i,j)} \cdot \boldsymbol{w}_{(k)}, & (3) \end{aligned}$$

where $\boldsymbol{x}_{(i,j)}$ and $\boldsymbol{w}_{(k)}$ are vectors of length CRS obtained by unrolling the summation of $\mathcal{X}$ and $\mathcal{W}$, respectively. They are given by

$$\boldsymbol{x}_{(i,j)} = [\mathcal{X}_{0,i,j}, \mathcal{X}_{0,i,j+1}, \ldots, \mathcal{X}_{C-1,i+R-1,j+S-1}]^T \tag{4}$$

$$\boldsymbol{w}_{(k)} = [\mathcal{W}_{k,0,0,0}, \mathcal{W}_{k,0,0,1}, \ldots, \mathcal{W}_{k,C-1,R-1,S-1}]^T \tag{5}$$

We omit the other convolution hyperparameters, such as stride and padding, for simplicity here. Note that they only affect the access pattern to $\mathcal{X}$ for constructing $\boldsymbol{x}_{(i,j)}$. Since we use $\boldsymbol{x}_{(i,j)}$ instead of $\mathcal{X}$ in the rest of paper, our method can be used with any stride and padding hyperparameters. Let $Z_{k,i,j}$ be an element of activation maps. A typical convolution layer accompanied by ReLU is given by

$$Z_{k,i,j} = \mathrm{ReLU}(Y_{k,i,j} + b_k), \tag{6}$$

where b_k is a bias term of a k-th filter.

In this paper, we cover a video analysis system that executes the model inference for each frame in a video stream. We use the frame number $t \in \{1, 2, \ldots, T\}$ to distinguish variables for each frame, e.g., $\boldsymbol{x}^{(t)}_{(i,j)}$ and $\boldsymbol{x}^{(t-1)}_{(i,j)}$ are variables for the current frame and the previous frame, respectively.

3 Proposed Method

In this section, we describe a method that reduces FLOPs without any changes in inference results for an original model. We propose a *range-bound-aware convolution layer*,

[1] https://developer.nvidia.com/cudnn

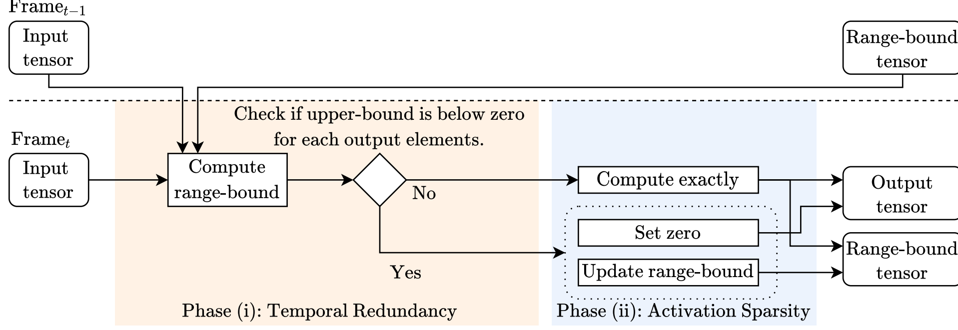

Figure 1. An overview of the *range-bound-aware convolution layer*.

which is a drop-in replacement for the standard convolution layer with ReLU. The range-bound-aware convolution layer not only produces exactly the same activation maps as the standard convolution layer after applying ReLU but also reduces FLOPs utilizing the temporal redundancy of video streams and the activation sparsity of ReLU-based CNNs. Figure 1 shows an overview of our method that consists of two phases. (i) We introduce *range-bound*, which is the upper bound of each output element derived from the previous frame's inference results. (ii) By using range-bound, we can identify and skip the computation of output elements that become zeros after ReLU. First, we describe the key idea using the typical convolution layer shown in Eq. (6). Then, we extend our method to support a wide variety of ReLU-based CNNs as discussed later in Section 3.3.

3.1 Key Idea

For the first phase, we define *range-bound* $\overline{Y}^{(t)}_{k,i,j}$, which is the upper bound of an output element $Y^{(t)}_{k,i,j}$ computed incrementally from the previous output element $Y^{(t-1)}_{k,i,j}$ or range-bound $\overline{Y}^{(t-1)}_{k,i,j}$, as follows:

Definition 1. *We define range-bound $\overline{Y}^{(t)}_{k,i,j}$ as follows:*

$$\overline{Y}^{(t)}_{k,i,j} = \|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|\|\boldsymbol{w}_{(k)}\| + Y^{(t-1)}_{k,i,j}. \quad (7)$$

If the previous inference succeeded to skip the computation of $Y^{(t-1)}_{k,i,j}$, we use the following definition given by

$$\overline{Y}^{(t)}_{k,i,j} = \|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|\|\boldsymbol{w}_{(k)}\| + \overline{Y}^{(t-1)}_{k,i,j}. \quad (8)$$

We use this definition because of the balance between the amount of computation and the tightness of the upper bound. Since video streams have temporal redundancy (Richardson, 2004), the pixels in consecutive video frames have similar values. Therefore, $\|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|$ becomes small, and the upper bound becomes tight. Since we can reuse $\|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|$ across K output channels, we only need to compute $\|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|$ once. $\|\boldsymbol{w}_{(k)}\|$ is constant for inference. When we have $\|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|$ and $\|\boldsymbol{w}_{(k)}\|$, the computation of $\overline{Y}^{(t)}_{k,i,j}$ costs $\mathcal{O}(1)$ time.

We introduce the following lemma to verify range-bound is always the upper bound of $Y^{(t)}_{k,i,j}$.

Lemma 1. *If we set $\overline{Y}^{(1)}_{k,i,j} = Y^{(1)}_{k,i,j}$, we have $Y^{(t)}_{k,i,j} \le \overline{Y}^{(t)}_{k,i,j}$ for $t = 1, \ldots, T$.*

Proof. We give a proof by mathematical induction to t.

Base case: $Y^{(1)}_{k,i,j} = \overline{Y}^{(1)}_{k,i,j}$ holds from the assumption.

Induction hypothesis: $Y^{(t-1)}_{k,i,j} \le \overline{Y}^{(t-1)}_{k,i,j}$ holds.

Inductive step: From Eq. (3), we have

$$Y^{(t)}_{k,i,j} = \boldsymbol{x}^{(t)}_{(i,j)} \cdot \boldsymbol{w}_{(k)} - \boldsymbol{x}^{(t-1)}_{(i,j)} \cdot \boldsymbol{w}_{(k)} + Y^{(t-1)}_{k,i,j} \quad (9)$$

$$= (\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}) \cdot \boldsymbol{w}_{(k)} + Y^{(t-1)}_{k,i,j}. \quad (10)$$

From the induction hypothesis $Y^{(t-1)}_{k,i,j} \le \overline{Y}^{(t-1)}_{k,i,j}$ and the Cauchy-Schwarz inequality, $Y^{(t)}_{k,i,j} \le \|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|\|\boldsymbol{w}_{(k)}\| + \overline{Y}^{(t-1)}_{k,i,j}$ holds. Since the right hand is equal to Definition 1, we have $Y^{(t)}_{k,i,j} \le \overline{Y}^{(t)}_{k,i,j}$, which completes the inductive step. Since the base case and the inductive step hold, we complete the proof. □

Lemma 1 shows our proposed range-bound is the valid upper bound for all frames. For the second phase, we skip the computation of output elements by using range-bound. We can guarantee that the output element results in zero after ReLU by the following lemma:

Lemma 2. *If a computation of a convolution layer and ReLU can be represented as $Z^{(t)}_{k,i,j} = \mathrm{ReLU}(Y^{(t)}_{k,i,j} + b_k)$, we have $Z^{(t)}_{k,i,j} = 0$ when the inequality*

$$\overline{Y}^{(t)}_{k,i,j} + b_k \le 0 \quad (11)$$

is satisfied.

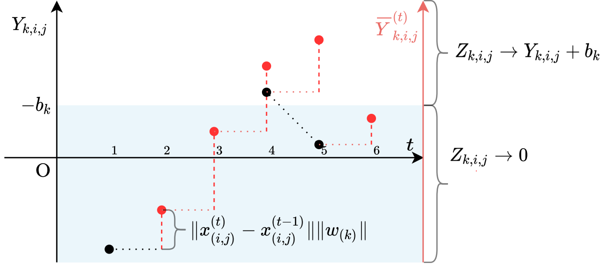

Figure 2. The time evolution of range-bound $\overline{Y}^{(t)}_{k,i,j}$ (red circle) for each frame t.

Proof. From the condition and Lemma 1, we have $Y^{(t)}_{k,i,j} + b_k \leq \overline{Y}^{(t)}_{k,i,j} + b_k \leq 0$ when Eq. (11) is satisfied. Since $Y^{(t)}_{k,i,j} + b_k$ is surely a non-positive value, we have $Z^{(t)}_{k,i,j} = \mathrm{ReLU}(Y^{(t)}_{k,i,j} + b_k) = 0$ from the definition of ReLU in Eq. (1), which completes the proof. □

Lemma 2 indicates that when $\overline{Y}^{(t)}_{k,i,j} + b_k$ is lower than zero, we can safely set $Z^{(t)}_{k,i,j}$ to zero instead of the exact computation of convolution for $Y^{(t)}_{k,i,j}$. Therefore, our method skips the computation of output elements of the convolution layer accompanied by ReLU using Lemma 2 while maintaining the valid range-bound. Note that although we skip the exact computation, Lemma 2 will still valid for the next frame thanks to Eq. (8) in Definition 1 as described below.

For maintaining range-bound, $\overline{Y}^{(t)}_{k,i,j}$ increases monotonically as long as Eq. (11) is satisfied from Definition 1. When Eq. (11) is not satisfied, we need to compute exact $Y^{(t)}_{k,i,j}$ and update $\overline{Y}^{(t)}_{k,i,j} = Y^{(t)}_{k,i,j}$. For example, Figure 2 shows the time evolution of range-bound. The red and black circles represent range-bound $\overline{Y}^{(t)}_{k,i,j}$ and the exactly computed $Y^{(t)}_{k,i,j}$, respectively. First, we perform exact computation at $t = 1$. In the next frame, we first compute range-bound and check that Lemma 2 holds, and thus, we can skip the exact computation. For $t = 3$, it is the same procedure as $t = 2$. For $t = 4$, range-bound exceeds the $-b_k$, and thus, we compute exact $Y^{(4)}_{k,i,j}$. Since $Y^{(4)}_{k,i,j} > -b_k$, Lemma 2 does not hold for $t = 5$, and thus, we compute exactly and update range-bound $\overline{Y}^{(5)}_{k,i,j} = Y^{(5)}_{k,i,j}$. We can skip computation for $t = 5$ because Lemma 2 holds again. Summarizing the above, we need exact computations only for $t = 1, 4$, and 5 while we can skip computations for $t = 2, 3$, and 6.

Algorithm 1 is the formal procedure of the range-bound-aware convolution layer. In addition to $\boldsymbol{x}^{(t)}_{(i,j)}$ and $\boldsymbol{w}_{(k)}$, we require the previous frame's inference results $\boldsymbol{x}^{(t-1)}_{(i,j)}$ and range-bound $\overline{Y}^{(t-1)}_{k,i,j}$. We compute $\|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|$ once across K output channels (line 4). From Lemma 2, we check whether $Z^{(t)}_{k,i,j}$ is zero (line 6). If Eq. (11) is satisfied, we skip the exact computation and assign zero to $Z^{(t)}_{k,i,j}$ (line 8). Otherwise, we exactly computes $Z^{(t)}_{k,i,j}$ (lines 10 and 11). We update range-bound for the next frame (lines 7 and 10).

Algorithm 1 The range-bound-aware convolution layer.

```
1: Input: x^(t)_(i,j), x^(t-1)_(i,j), Ȳ^(t-1)_k,i,j, w_(k), ||w_(k)||, b_k
2: for i = 0 → H − 1 do
3:   for j = 0 → W − 1 do
4:     Compute ||x^(t)_(i,j) − x^(t-1)_(i,j)||
5:     for k = 0 → K − 1 do
6:       if Eq. (11) holds then
7:         Ȳ^(t)_k,i,j = Ȳ_k,i,j + ||x^(t)_(i,j) − x^(t-1)_(i,j)|| ||w_(k)||
8:         Z^(t)_k,i,j = 0
9:       else
10:        Ȳ^(t)_k,i,j = x^(t)_(i,j) · w_(k)
11:        Z^(t)_k,i,j = ReLU(Ȳ^(t)_k,i,j + b_k)
12:      end if
13:    end for
14:  end for
15: end for
```

3.2 Theoretical Analysis

We introduce the following theorems to prove the exactness and the computational cost of our method.

Theorem 1. *The range-bound-aware convolution layer produces exactly the same activation maps as the standard convolution layer accompanied by ReLU.*

Proof. First, Algorithm 1 updates range-bound $\overline{Y}^{(t)}_{k,i,j}$ as Definition 1 (line 7 or 10). Thus, Lemma 1 holds. Since the possible value of $Z^{(t)}_{k,i,j}$ is positive or zero, we prove both cases. If $Z^{(t)}_{k,i,j} > 0$, we have $0 < Y^{(t)}_{k,i,j} + b_k < \overline{Y}^{(t)}_{k,i,j} + b_k$ from Lemma 1. Since Eq. (11) does not hold in Lemma 2, Algorithm 1 updates $\overline{Y}^{(t)}_{k,i,j}$ and outputs exact $Z^{(t)}_{k,i,j}$ (lines 10–11). If $Z^{(t)}_{k,i,j} = 0$, Algorithm 1 outputs zero by skipping computation or computing exactly (line 8 or 11). Thus, Algorithm 1 produces exactly the same activation maps as the standard convolution layer accompanied by ReLU. □

Let α be the ratio of $Z^{(t)}_{k,i,j}$ satisfying Eq. (11). The time complexity of our algorithm is given by

Theorem 2. *The range-bound-aware convolution layer requires $\mathcal{O}((K(1-\alpha)+1)CRSHW)$ time.*

Proof. We first consider the time complexity inside the loop of H and W (lines 4–13). The computation of $\|\boldsymbol{x}^{(t)}_{(i,j)} -$

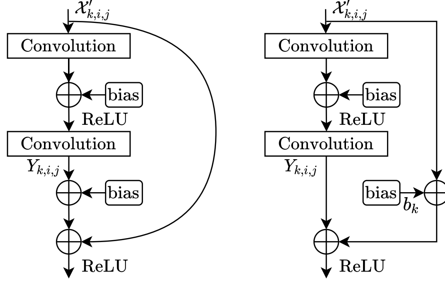

Figure 3. A model architecture with shortcut connection (left). The order of additions is exchanged to apply Lemma 2 (right).

$\boldsymbol{x}^{(t-1)}_{(i,j)}\|$ requires $\mathcal{O}(CRS)$ time since the length of $\boldsymbol{x}_{(i,j)}$ is CRS (line 4). Inside the loop of K, if Eq. (11) holds, it requires $\mathcal{O}(1)$ time (lines 7–8). Otherwise, it requires $\mathcal{O}(CRS)$ time for dot-product (lines 10–11). In total, the loop of K requires $\mathcal{O}(K(1-\alpha)CRS + CRS)$ time. Thus, Algorithm 1 requires $\mathcal{O}((K(1-\alpha)+1)CRSHW)$ time. □

Our method reduces time complexity roughly proportional to $1-\alpha$ since the standard convolution layer requires $\mathcal{O}(KCRSHW)$ time. In Section 5, we investigate α because it indicates the theoretical efficacy of our method.

3.3 Extensions

In this section, we discuss the extensions of our method to non-trivial model architectures and the utilization of activation sparsity of the input matrix.

3.3.1 Use for Complicated Model Architectures

For simple CNNs, the relation between the convolution layer and ReLU is often represented as Eq. (6). However, modern CNNs use more sophisticated model architecture. Thus, we need to take these into account and modify Eq. (11) for Lemma 2. For example, Figure 3 shows a model architecture with shortcut connection (He et al., 2016) given by

$$Z_{k,i,j} = \mathrm{ReLU}((Y_{k,i,j} + b_k) + \mathcal{X}'_{k,i,j}), \tag{12}$$

where the input of the second ReLU is the sum of the output of the second convolution layer, bias, and the input of the first convolution layer $\mathcal{X}'_{k,i,j}$. In this case, we can modify Eq. (11) as follows:

$$\overline{Y}^{(t)}_{k,i,j} + (b_k + \mathcal{X}'_{k,i,j}) \leq 0. \tag{13}$$

Another example is batch normalization (Ioffe & Szegedy, 2015), which performs normalization between convolution and ReLU, given by

$$\mathrm{BN}(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\gamma + \beta, \tag{14}$$

where μ, σ, ϵ, γ, and β are constants for inference. The batch normalization can be optimized out using the following simplification and constant folding (Jiang et al., 2018):

$$\mathrm{BN}(Y_{k,i,j} + b_k) = \frac{\boldsymbol{x}_{(i,j)} \cdot \boldsymbol{w}_{(k)} + b_k - \mu}{\sqrt{\sigma^2+\epsilon}}\gamma + \beta \tag{15}$$

$$= \boldsymbol{x}_{(i,j)} \cdot \boldsymbol{w}'_{(k)} + b'_k, \tag{16}$$

where $\boldsymbol{w}'_{(k)} = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}\boldsymbol{w}_{(k)}$ and $b'_k = \frac{b_k-\mu}{\sqrt{\sigma^2+\epsilon}}\gamma + \beta$. Thus, batch normalization can also be represented as Eq. (6). Therefore, our method can be applied to these models.

3.3.2 Utilize Input Sparsity

We can combine the utilization of sparsity of the input matrix with Algorithm 1. We write $\boldsymbol{x}_{(i,j)}$ and $\boldsymbol{w}_{(k)}$ as $\boldsymbol{x}_{(i,j)} = [x_0, x_1, \ldots, x_{CRS-1}]^T$ and $\boldsymbol{w}_{(k)} = [w_0, w_1, \ldots, w_{CRS-1}]^T$. Then, we construct a set P that contains indices of non-zero elements in $\boldsymbol{x}_{(i,j)}$ given by $P = \{q \in \{0, 1, \ldots, CRS-1\} | x_q \neq 0\}$. Let P_i be the i-th smallest index in P and n be the number of elements in P. We can construct two compressed vectors $\hat{\boldsymbol{x}}_{(i,j)} = [x_{P_0}, x_{P_1}, \ldots, x_{P_{n-1}}]^T$ and $\hat{\boldsymbol{w}}_{(k)} = [w_{P_0}, w_{P_1}, \ldots, w_{P_{n-1}}]^T$. The construction of $\hat{\boldsymbol{x}}_{(i,j)}$ can take place at the same time as line 4. Thus, $Y_{k,i,j} = \hat{\boldsymbol{x}}_{(i,j)} \cdot \hat{\boldsymbol{w}}_{(k)}$ requires $\mathcal{O}(n)$ time instead of $\mathcal{O}(CRS)$ time (line 10), where $n \leq CRS$.

4 Implementation on GPUs

In this section, we describe the implementation details of the range-bound-aware convolution for GPUs. Algorithm 1 performs suboptimally on GPUs because selective computation of dot-products (line 10) is inefficient compared with the GEMM kernel for convolution. Our algorithm is similar to sampled dense-dense matrix multiplication (SDDMM) given by $O = D_1 D_2 \odot S$, where D_1 and D_2 are dense matrices, and S is a boolean sparse matrix. In our case, S corresponds to the condition of Eq. (11) of each output element, and thus, sparsity corresponds to α, which is the ratio of output elements satisfying Eq. (11). Although there are efficient SDDMM implementations on GPUs (Hong et al., 2019; Jiang et al., 2020), it is difficult to gain practical speed-up directly. This is because these implementations of SDDMM can achieve speed-up only if S is highly sparse while average activation sparsity is typically below 60% in our experiments. They also assume the sparse matrix is static to preprocess the data format while positions of non-zero elements in activation maps changes in every frame.

Therefore, we develop a more adaptive implementation for CNNs. Our strategy is based on the observation that activation sparsity is not uniformly distributed over channels. For example, Figure 4 shows histograms of the number of channels versus the activation sparsity ratio, which is the

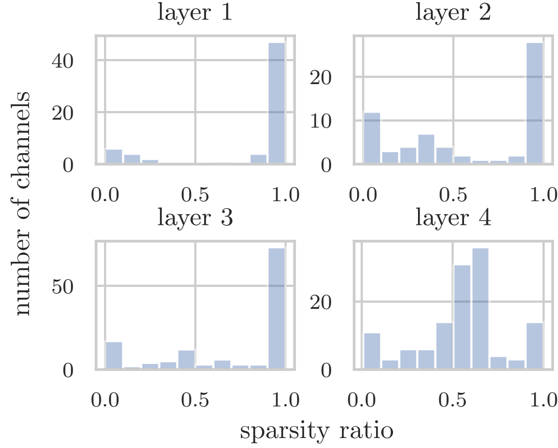

Figure 4. The histograms of the number of channels versus the ratio of zero elements for each channel's activation map in VGG19 with batch normalization.

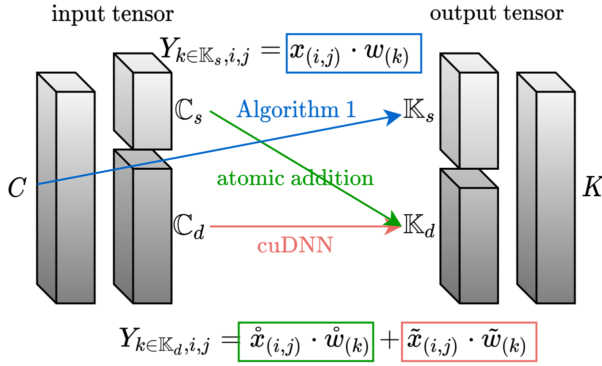

Figure 5. The overview of the implementation on GPUs.

ratio of the number of zero in an activation map, on VGG19 with batch normalization. There are channels in which the activation sparsity ratio is over 90%. We separate channels into these sparse channels and dense channels, and only apply the proposed algorithm to sparse channels.

4.1 Procedure

Figure 5 illustrates the overview of our implementation on GPUs. Let $\mathbb{K}_s$ and $\mathbb{K}_d$ be the set of sparse output channels and the set of dense output channels in the output tensor, respectively. Sparse channels mean that their sparsity ratio is higher than a certain threshold, and dense channels mean that their sparsity ratio is lower than the threshold. Similarly, $\mathbb{C}_s$ and $\mathbb{C}_d$ are the set of sparse input channels and the set of dense input channels in the input tensor, respectively.

Separate channels: We use the first four video frames to separate channels. Practically, we reorder channels and separate channels by a separation point K'. Therefore, we have $\mathbb{K}_s = \{0, 1, \dots, K'-1\}$ and $\mathbb{K}_d = \{K', K'+1, \dots, K-1\}$. In the first frame, we compute all $Y_{k,i,j}$ exactly by the standard convolution layer. In the second frame, we compute all $\overline{Y}_{k,i,j}$ and calculate the ratio α, which is the ratio of output elements satisfying Eq. (11), for each channel. We sort output channels by α and reorder corresponding parameters such as the filter tensor. In the third frame, we execute the inference with reordered parameters and compute all $Y_{k,i,j}$ exactly. In the fourth frame, we compute all $\overline{Y}_{k,i,j}$ and calculate the ratio α again, and then, we set the separation point K' for a channel in which α exceeds the threshold. We only consider the separation of the output tensor because the input tensor is already separated in the previous layer except for the first layer. Therefore, $\mathbb{C}_s$ and $\mathbb{C}_d$ correspond to $\mathbb{K}_s$ and $\mathbb{K}_d$ of the previous layer, respectively.

Compute sparse output channels $k \in \mathbb{K}_s$: For sparse output channels $k \in \mathbb{K}_s$, we use Algorithm 1 because we expect to skip computations of most output elements by using Lemma 2. First, we compute $\|\mathcal{X}^{(t)}_{i,j} - \mathcal{X}^{(t-1)}_{i,j}\|^2$ for each spatial point (i, j) in the input tensor in parallel given by $\sum_{c=0}^{C-1}(\mathcal{X}^{(t)}_{c,i,j} - \mathcal{X}^{(t-1)}_{c,i,j})^2$. Second, we compute $\|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|^2$ for each spatial point (i, j) in the output tensor in parallel given by $\sum_{r=0}^{R-1}\sum_{s=0}^{S-1}\|\mathcal{X}^{(t)}_{i+r,j+s} - \mathcal{X}^{(t-1)}_{i+r,j+s}\|^2$, where the square root of this is used for Lemma 2 and updating range-bound. Then, the list of output elements that do not satisfy Lemma 2's inequality is extracted using an efficiently parallelized stream compaction (Baxter, 2016). Finally, we compute exact $Y_{k,i,j} = \boldsymbol{x}_{(i,j)} \cdot \boldsymbol{w}_k$ by dot-product for each output element in this list in parallel.

Compute dense output channels $k \in \mathbb{K}_d$: Let $\mathring{\boldsymbol{x}}_{(i,j)}$ and $\tilde{\boldsymbol{x}}_{(i,j)}$ be the vectors from sparse input channels $\mathbb{C}_s$ and dense input channels $\mathbb{C}_d$, respectively. Let $\mathring{\boldsymbol{w}}_{(k)}$ and $\tilde{\boldsymbol{w}}_{(k)}$ be the corresponding filter vectors. With this notation, the computation of the output element is given by $Y_{k,i,j} = \mathring{\boldsymbol{x}}_{(i,j)} \cdot \mathring{\boldsymbol{w}}_{(k)} + \tilde{\boldsymbol{x}}_{(i,j)} \cdot \tilde{\boldsymbol{w}}_{(k)}$. For dense output channels $k \in \mathbb{K}_d$, we compute $\tilde{\boldsymbol{x}}_{(i,j)} \cdot \tilde{\boldsymbol{w}}_{(k)}$ using the optimized GEMM kernel for convolution from the cuDNN library. Then, we compute $\mathring{\boldsymbol{x}}_{(i,j)} \cdot \mathring{\boldsymbol{w}}_{(k)}$ using sparse operations, which will require low computational cost because $\mathring{\boldsymbol{x}}_{(i,j)}$ has few non-zero elements. Specifically, the list of non-zero elements in sparse input channels $c \in \mathbb{C}_s$ is extracted using the parallelized stream compaction. After that, each non-zero element is multiplied by the corresponding filter weight and added to the corresponding output elements using atomic addition to avoid race conditions.

5 Evaluation

We describe experimental setups and results to evaluate the range-bound-aware convolution layer. First, we evaluate α, which is the ratio of reducible output elements. Second, we summarize the reduction of FLOPs by Algorithm 1. Third, to show algorithmic benefits of each component of our method by evaluating the reduction of wall-clock time,

Table 1. The number of convolution layers that can be applied with the range-bound-aware convolution layer for each model.

VGG19_bn	ResNet50	WRN-101-2	SSD_VGG16
16/16	49/53	100/104	25/39

Table 2. Scenes that have the highest α and the lowest α in the second convolution layer. α is shown in mean (standard deviation).

Scene	VGG19_bn	ResNet50	WRN-101-2
FallingTrees	0.51 (0.06)	0.37 (0.10)	0.50 (0.08)
Ocean	0.40 (0.05)	0.16 (0.05)	0.33 (0.04)

we compare four straightforward serial implementations run on CPUs: (a) standard dense convolution, (b) utilizing the input sparsity with compressing input, (c) the range-bound-aware convolution (Section 3.1), and (d) the range-bound-aware convolution with compressing input (Section 3.3.2). The former two are baselines and the latter two are proposed methods. Also, to confirm practical performance considering hardware-dependent optimizations on GPUs, we compare the GPU implementation of our method (Section 4) with the optimized baseline (TVM with cuDNN).

5.1 Setup

Models: For image classification models, we use VGG19 with batch normalization (VGG19_bn), ResNet50, and Wide ResNet-101-2 (WRN-101-2) from torchvision,[2] which are pre-trained on the ImageNet dataset with an image size of 224×224. For object detection models, we use SSD_VGG16 from gluoncv,[3] which is pre-trained on the COCO dataset with an image size of 512×512. Table 1 shows the number of convolution layers that can be applied with the range-bound-aware convolution layer. We can replace all convolution layers in VGG19_bn, which uses batch normalization. Although ResNet50 and Wide ResNet (WRN-101-2) have shortcut connections, we can replace the majority of convolution layers. For object detection models, the majority of convolution layers in SSD_VGG16 are replaceable but not convolution layers whose outputs are used to produce detection results without applying ReLU.

Video Streams: We use video streams of various scenes from the YUP++ dataset (Feichtenhofer et al., 2017), which contains 1,200 video streams. There are 20 scene categories, and each category has 60 video streams. Half of them were taken with a static camera, and the other half with a moving camera. The frame rate is in the range of 24 to 30 frames per second. We preprocess video frames by resize and center crop and feed video frames sequentially to CNNs that classify images or detect objects.

[2]https://pytorch.org/docs/stable/torchvision/models.html

[3]https://gluon-cv.mxnet.io/model_zoo/index.html

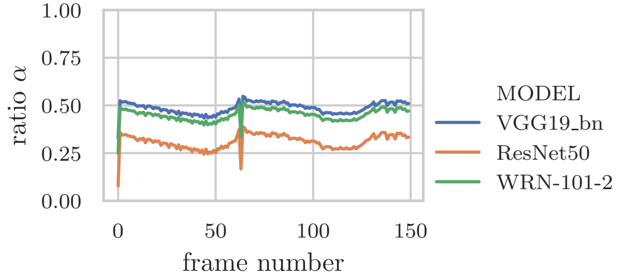

Figure 6. The time evolution of α in the second convolution layer on one video stream (Street_static_cam_13).

Environment: The straightforward serial implementations of Algorithm 1 are written in C++ and run on Core i7-8700 with 64GB memory. The GPU implementations described in Section 4 are written in CUDA (v10.0) and run on Jetson Nano with 128 CUDA cores and 4GB memory. We develop the model inference runtime with our method using TVM (v0.6.0) since TVM outperforms deep learning frameworks by various optimizations for inference (Wang et al., 2019). We use cuDNN (v7.6.3) and fix the convolution algorithm to `IMPLICIT_PRECOMP_GEMM` for consistency of results.

5.2 Results

5.2.1 *Evaluating the ratio α*

We show the ratio α, which is the ratio of output elements satisfying Eq. (11), to investigate the efficacy of the range-bound-aware convolution layer. Higher α means higher reduction of FLOPs by using Lemma 2. To understand the dependence on video streams, Table 2 shows the mean and standard deviation of α in the second convolution layer for selected two scene categories that respectively achieve the highest and lowest α using video streams taken with a static camera. `Falling Trees` scene has the highest α because video frames are almost static except for the small duration in which trees are falling. Therefore, $\|\boldsymbol{x}^{(t)}_{(i,j)} - \boldsymbol{x}^{(t-1)}_{(i,j)}\|$ remains nearly zero. `Ocean` scene has the lowest α due to the glitter of the water surface that makes the difference larger. The range of α is around $\pm 10\%$ over 20 scenes, and α depends more on models. Figure 6 shows an example of the time evolution of α in the second convolution layer on one video stream in `Street` scene. As we can see, the curves of α are almost the same regardless of the model and relatively smooth over time.

Figure 7 shows the box plots of average ratio α of all 1,200 video streams for each range-bound-aware convolution layer. Our range-bound guarantees that around 50% of output elements are zero in about three layers in VGG19_bn and WRN-

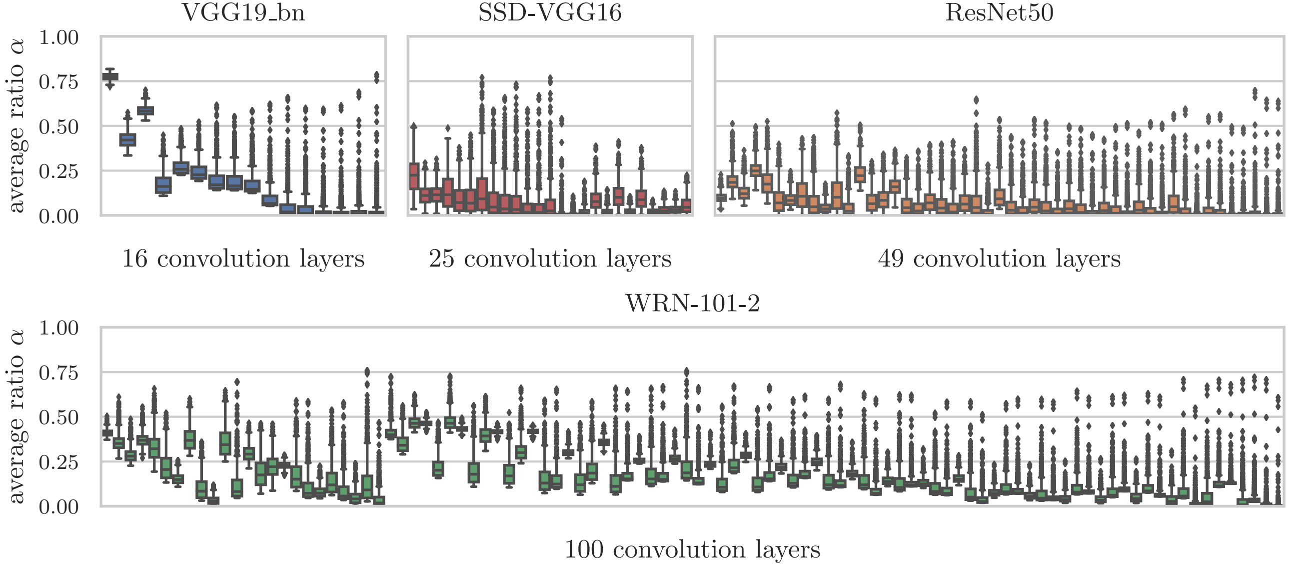

Figure 7. The box plots of ratio α, which is the ratio of reducible output elements, of 1,200 video streams from YUP++ for each range-bound-aware convolution layer. α is averaged over all frames (5 seconds, 120–150 frames) in a video stream.

Table 3. The total number of output elements in the range-bound-aware convolution layers, the average number of exactly computed output elements in our method, and non-zero output elements over 1,200 video streams for each model. The number of activations per inference of our method and standard convolution correspond to Computed and Total columns, respectively.

	Computed	Non-zero	Total
VGG19_bn	8,892,686	4,967,704	14,852,096
ResNet50	8,613,847	5,466,429	9,608,704
WRN-101-2	15,607,509	9,126,446	19,719,168
SSD_VGG16	63,587,213	39,273,634	73,334,528

101-2. All models tend to decrease α as layers progress. This is because, in later layers, $\|\boldsymbol{x}_{(i,j)}^{(t)} - \boldsymbol{x}_{(i,j)}^{(t-1)}\|$ suffers from the effect of difference in larger regions compared with earlier layers due to preceding convolution layers. Thus, it is difficult to guarantee that the output element results in zero by range-bound. Table 3 shows the relationship between the total number of output elements in the range-bound-aware convolution layers, the average number of exactly computed output elements in our method, and non-zero output elements over 1,200 video streams. Our method only needs to compute 1.6–1.8$\times$ of output elements compared with the number of non-zero elements to guarantee exactness.

5.2.2 *Reduction of FLOPs*

To evaluate the reduction of FLOPs by Algorithm 1 compared with the total number of FLOPs in the original model inference, we calculate the number of reduced FLOPs and show the average FLOP reduction in Table 4. We

Table 4. The average FLOPs reduction in terms of the theoretical operation per inference with breakdown of video streams taken with a static camera and a moving camera.

	All	Static	Moving
VGG19_bn	18.3%	19.9%	16.7%
ResNet50	4.92%	6.38%	3.46%
WRN-101-2	16.5%	17.8%	15.2%
SSD_VGG16	7.68%	9.98%	5.24%

count the FLOPs in the number of multiply-add operations as in (Bianco et al., 2018; Shomron & Weiser, 2019). VGG19_bn and WRN-101-2 reduce FLOPs by 18.3%, and 16.5%, respectively. The difference of reduction of FLOPs between video streams taken with a static camera and a moving camera is around $\pm 2\%$ since our method can absorb the changes between adjacent video frames by the proposed upper bound even for video streams taken with a moving camera. Note that this does not include the utilization of input sparsity described in Section 3.3.2. From the results of the wall-clock time reduction shown in the next paragraph, we can expect that the utilization of the input sparsity further reduces FLOPs.

5.2.3 *Reduction of Wall-clock time:*

Figure 8 depicts the experimental results of four straightforward serial implementations on CPUs to show reduction of wall-clock time. For each model, we show the first 10 convolution layers. We can observe the pure efficacy of our method by comparing between (a) standard convolution and (c) the range-bound-aware convolution layer. Specifically,

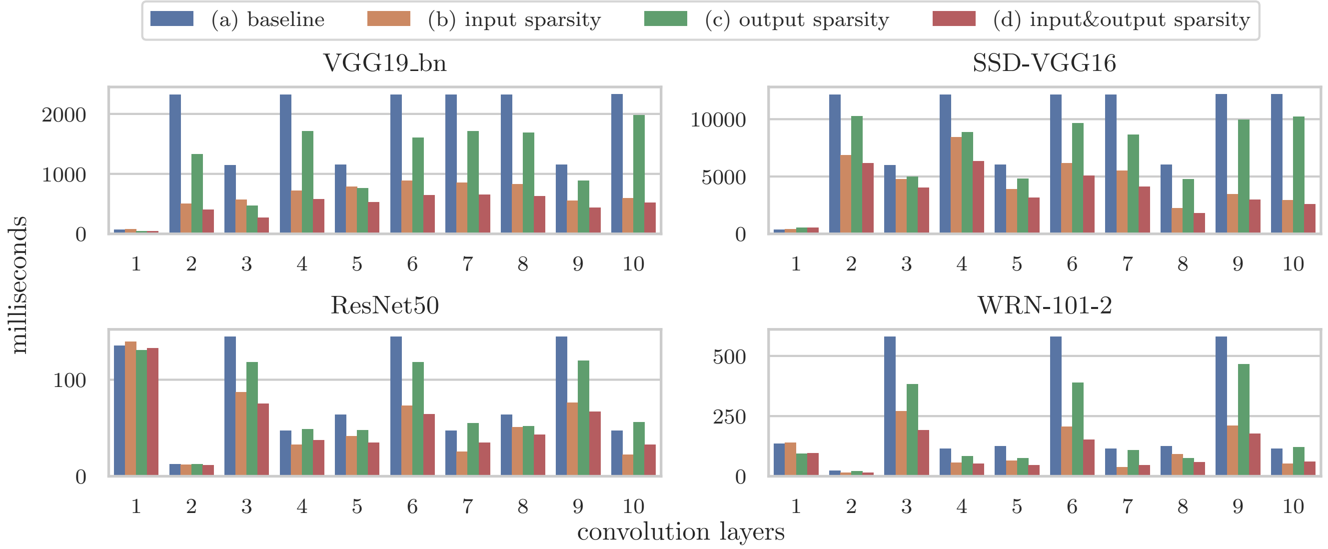

Figure 8. Wall-clock time (ms) of the straightforward serial implementations on one video stream (Street_static_cam_13) run on CPUs.

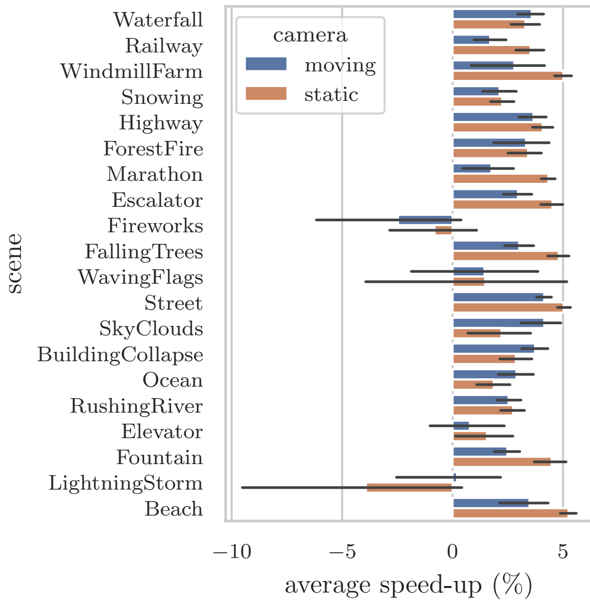

Figure 9. Relative speed-up of our method for GPUs compared with the baseline on VGG19_bn.

VGG19_bn achieves 1.2–2.4× speed-up for each convolution layer. We can observe performance improvement of our method against SpMM-based implementations that utilize input sparsity by comparing between (b) utilizing only the input sparsity and (d) the proposed method that combines the utilization of input sparsity. In this case, VGG19_bn achieves 1.1–2.1× speed-up for each convolution layer. The range-bound-aware convolution layers hardly improve the performance of several layers such as the seventh and the ninth in ResNet50 that has low α because of the overhead of computing range-bound. We can avoid performance degradation by falling back to the standard convolution layer on the basis of α for the inference of the next frame.

Table 5. The mean squared error (MSE) of the output vector of the last layer compared between our implementation and the baseline implementation.

	mean	min	max
MSE	2.73e-12	6.63e-14	7.89e-11

For the implementation on GPUs described in Section 4, we measure the wall-clock time from feeding a video frame to obtaining an inference result. We use the model compiled using TVM with the same configuration as ours for the baseline implementation. We plot the relative speed-up to the baseline on VGG19_bn for each scene category in Figure 9. The plotted speed-up is averaged over the period from the fifth frame to the last frame because we use the first–fourth frames for separating channels. Our method achieves 4% speed-up in median and up to 8% speed-up while producing exactly the same inference results as an original model. Table 5 compares the mean squared error of the output vector of the last layer between our implementation and the baseline implementation. There are negligible errors from the limited precision of floating-point arithmetic, which verifies that our method guarantees to produce exactly the same inference results as an original model. The amount of speed-up depends on the scene and the video stream itself. For example, video streams in the `Street` scene speed up by 2%–7%. In this experiment, the threshold to determine the separation point is set to 99% because α decreases as frames progress, and we use the standard convolution when the number of sparse channels is less than

20%. Note that several video streams fail to speed up because the frames used in the separation of channels do not represent the rest of the video frames such as video frames are black at beginning of video streams.

6 Related Work

Since the computational cost of CNNs must be reduced for video analysis systems, numerous methods (Kang et al., 2017; Canel et al., 2019) and hardware architectures (Buckler et al., 2018; Zhu et al., 2018; Song et al., 2020) have been proposed to accelerate inference utilizing the temporal redundancy of video streams. Loc et al. (2017) explored the reduction of the computational cost of CNNs by reusing the results of the previous video frames when the current and previous video frames are sufficiently similar. Specifically, they cache convolution outputs for each divided region of the input video frame. Their similarity measure is based on the distance derived from color histograms. DeepCache (Xu et al., 2018) enhances cache hit rates by comparing not only the same region but also multiple regions in the video frame using the search algorithm. Although this increases the processing overhead of the searching algorithm, this enables more effective cache-reuse when the camera is moving in a certain direction. Another method to utilize the temporal redundancy is pixel-wise screening. Cavigelli & Benini (2020) proposed change-based spatial convolution layers for video streams taken with a static camera. First, they compute the absolute difference between the current and previous frames for each pixel. Then, they only update output pixels affected by changed input pixels whose difference exceeds a certain threshold. However, the above methods sacrifice accuracy for efficiency. To the best of our knowledge, our method is the first work that accelerates inference utilizing the temporal redundancy of video streams while producing exactly the same inference results as an original model. Loc et al. (2017) provided a detailed breakdown for each of the optimization methods that they used to accelerate inference. The speed-up achieved by their convolutional layer caching, which utilizes the temporal redundancy of video streams from the UCF101 dataset (Soomro et al., 2012), is $1.36\times$ with a decrease in accuracy of 2.83% on VGG16. Prior works often specialized the model combining multiple methods such as model distillation (Kang et al., 2017) and the use of half-precision floating points (Loc et al., 2017). However, our method can also obtain the benefits of these methods as it can accelerate these models as an original model.

Shomron & Weiser (2019) proposed a value-prediction-based method that predicts negative elements in the output tensor of the convolution layer for ReLU-based CNNs. Since ReLU converts negative output elements to zero, we can skip exact computations of them. Their prediction method is based on a hypothesis that there is a spatial correlation in positions of zero elements in activation maps. They compute output elements sampled diagonally. If a computed output element is below zero, they skip computations of nearby elements. They reported that their method reduces FLOPs by 30.8% while decreasing top-5 accuracy by 2.0% on VGG16 without fine-tuning. In addition, they reported that the fine-tuning can compensate for the decrease in accuracy by 0.4%. Our method reduces FLOPs by 18.3% on VGG19_bn without fine-tuning and produces exactly the same results as the original pre-trained model.

6.1 Enhancing the Activation Sparsity

Since our method depends on activation sparsity, the amount of activation sparsity in the model is important for acceleration. Moreover, we utilize the non-uniformity of activation sparsity for each channel in the implementation on GPUs. Although the relationship between accuracy and activation sparsity is not well understood, Georgiadis (2019) reported that L_1 regularization on the input of the convolution layer increases activation sparsity with negligible accuracy drop. In addition, Mehta et al. (2019) empirically observed that the choice of hyperparameters for training leads to emerging filter level sparsity in CNNs with batch normalization and ReLU. Therefore, if we use these methods, our method can further accelerate ReLU-based CNNs.

7 Conclusion

For accelerating the model inference of CNNs on video streams, we proposed a *range-bound-aware convolution layer*. Our method guarantees exactly the same inference results as an original model. The proposed convolution layer can replace convolution layers accompanied by ReLU, and reduce the computational cost of matrix multiplication. Our method utilizes two properties: the temporal redundancy on video streams and activation sparsity of the model. We verify the exactness of our method by using these properties and conduct experiments to show the efficacy across video streams of various scenes.

We show that our method reduces FLOPs by 18.3%, and 16.5% on average for VGG19_bn and WRN-101-2, respectively. For the reduction of wall-clock time, VGG19_bn achieves 1.1–$2.1\times$ speed-up for each convolution layer using the straightforward serial implementations of our method run on CPUs compared with SpMM-based implementations. Although it is difficult to gain practical speed-up due to constraints on GPUs, we achieved 4% and up to 8% speed-up in the evaluation of end-to-end wall-clock time on GPUs. Finally, we expect further research for understanding activation sparsity and efficient sparse operations on GPUs to lead to more efficient models and implementations, and expect our method to contribute to further accelerate video analysis systems.

REFERENCES

Baxter, S. moderngpu 2.0, 2016. URL https://github.com/moderngpu/moderngpu/.

Bewley, A., Ge, Z., Ott, L., Ramos, F., and Upcroft, B. Simple online and realtime tracking. In *ICIP*, pp. 3464–3468, 2016.

Bianco, S., Cadene, R., Celona, L., and Napoletano, P. Benchmark analysis of representative deep neural network architectures. *IEEE Access*, 6:64270–64277, 2018.

Blalock, D., Gonzalez Ortiz, J. J., Frankle, J., and Guttag, J. What is the state of neural network pruning? In *MLSys*, volume 2, pp. 129–146. 2020.

Buckler, M., Bedoukian, P., Jayasuriya, S., and Sampson, A. EVA^2: Exploiting temporal redundancy in live computer vision. In *ISCA*, pp. 533–546, 2018.

Canel, C., Kim, T., Zhou, G., Li, C., Lim, H., Andersen, D. G., Kaminsky, M., and Dulloor, S. Scaling video analytics on constrained edge nodes. In *MLSys*, pp. 406–417. 2019.

Cavigelli, L. and Benini, L. CBinfer: exploiting frame-to-frame locality for faster convolutional network inference on video streams. *IEEE Transactions on Circuits and Systems for Video Technology*, 30(5):1451–1465, 2020.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E. Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: an automated end-to-end optimizing compiler for deep learning. In *OSDI*, pp. 578–594, 2018.

Feichtenhofer, C., Pinz, A., and Wildes, R. P. Temporal residual networks for dynamic scene recognition. In *CVPR*, pp. 7435–7444, 2017.

Gale, T., Elsen, E., and Hooker, S. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

Georgiadis, G. Accelerating convolutional neural networks via activation map compression. In *CVPR*, pp. 7085–7095, 2019.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *AISTATS*, pp. 315–323, 2011.

Gray, S., Radford, A., and Kingma, D. P. GPU kernels for block-sparse weights, 2017. URL https://openai.com/blog/block-sparse-gpu-kernels/.

Han, S., Pool, J., Tran, J., and Dally, W. J. Learning both weights and connections for efficient neural networks. In *NIPS*, pp. 1135–1143, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, pp. 770–778, 2016.

Hong, C., Sukumaran-Rajam, A., Nisa, I., Singh, K., and Sadayappan, P. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP*, pp. 300–314, 2019.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, pp. 448–456, 2015.

Ji, Y., Liang, L., Deng, L., Zhang, Y., Zhang, Y., and Xie, Y. TETRIS: TilE-matching the TRemendous Irregular Sparsity. In *NIPS*, pp. 4115–4125. 2018.

Jiang, P., Hong, C., and Agrawal, G. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *PPoPP*, pp. 376–388, 2020.

Jiang, Z., Chen, T., and Li, M. Efficient deep learning inference on edge devices. In *SysML*, 2018.

Jiao, L., Zhang, F., Liu, F., Yang, S., Li, L., Feng, Z., and Qu, R. A survey of deep learning-based object detection. *IEEE Access*, 7:128837–128868, 2019.

Kang, D., Emmons, J., Abuzaid, F., Bailis, P., and Zaharia, M. NoScope: optimizing neural network queries over video at scale. *Proc. VLDB Endow.*, 10(11):1586–1597, August 2017.

Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S. E., Fu, C., and Berg, A. C. SSD: single shot multibox detector. *arXiv preprint arXiv:1512.02325*, 2015.

Loc, H. N., Lee, Y., and Balan, R. K. DeepMon: Mobile GPU-based deep learning framework for continuous vision applications. In *MobiSys*, pp. 82–95, 2017.

Luo, J., Wu, J., and Lin, W. ThiNet: a filter level pruning method for deep neural network compression. In *ICCV*, pp. 5068–5076, 2017.

Mehta, D., Kim, K. I., and Theobalt, C. On implicit filter level sparsity in convolutional neural networks. In *CVPR*, pp. 520–528, 2019.

Ren, M., Pokrovsky, A., Yang, B., and Urtasun, R. SBNet: Sparse blocks network for fast inference. In *CVPR*, pp. 8711–8720, 2018.

Richardson, I. E. *H. 264 and MPEG-4 video compression: video coding for next-generation multimedia.* John Wiley & Sons, 2004.

Shomron, G. and Weiser, U. Spatial correlation and value prediction in convolutional neural networks. *IEEE Computer Architecture Letters*, 18(1):10–13, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

Song, Z., Wu, F., Liu, X., Ke, J., Jing, N., and Liang, X. VR-DANN: Real-time video recognition via decoder-assisted neural network acceleration. In *MICRO*, pp. 698–710, 2020.

Soomro, K., Zamir, A. R., and Shah, M. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

Su, S., Delbracio, M., Wang, J., Sapiro, G., Heidrich, W., and Wang, O. Deep video deblurring for hand-held cameras. In *CVPR*, pp. 237–246, 2017.

Wang, L., Chen, Z., Liu, Y., Wang, Y., Zheng, L., Li, M., and Wang, Y. A unified optimization approach for cnn model inference on integrated gpus. In *ICPP*, 2019.

Wen, W., Wu, C., Wang, Y., Chen, Y., and Li, H. Learning structured sparsity in deep neural networks. In *NIPS*, pp. 2074–2082. 2016.

Wu, C., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., Leyvand, T., Lu, H., Lu, Y., Qiao, L., Reagen, B., Spisak, J., Sun, F., Tulloch, A., Vajda, P., Wang, X., Wang, Y., Wasti, B., Wu, Y., Xian, R., Yoo, S., and Zhang, P. Machine learning at facebook: Understanding inference at the edge. In *HPCA*, pp. 331–344, 2019.

Xu, M., Zhu, M., Liu, Y., Lin, F. X., and Liu, X. DeepCache: principled cache for mobile deep vision. In *MobiCom*, pp. 129–144, 2018.

Zhang, H., Ananthanarayanan, G., Bodik, P., Philipose, M., Bahl, P., and Freedman, M. J. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, pp. 377–392, 2017.

Zhu, M. and Gupta, S. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

Zhu, Y., Samajdar, A., Mattina, M., and Whatmough, P. Euphrates: Algorithm-soc co-design for low-power mobile continuous vision. In *ISCA*, pp. 547–560, 2018.

A Profiling Results

We provide profiling results of our method sampled on VGG19_bn at processing the 100-th frame of Street_static_cam_13. Figure 10 shows the breakdown of the run time of our CPU implementation (Figure 8 (c)). On CPU, the run time for range-bounds is relatively short for the overall run time. Figure 11 shows the breakdown of the run time of our GPU implementation described in Section 4. On GPU, while the run time for computing dense channels by cuDNN is reduced, atomic additions sometimes become time consuming. The first, eighth and later convolution layers are processed by the standard convolution because the number of sparse channels is not enough for speed-up.

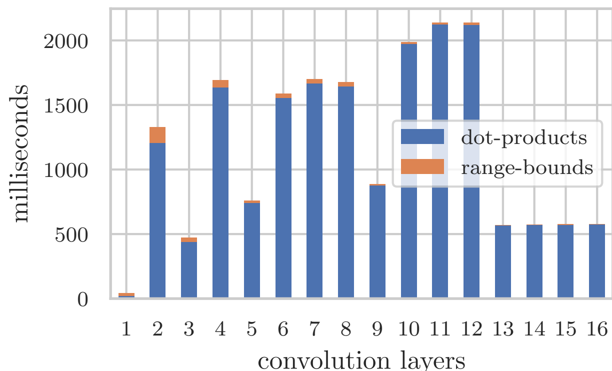

Figure 10. Run time for range-bounds and dot-products of our CPU implementation corresponding to (c) in Figure 8.

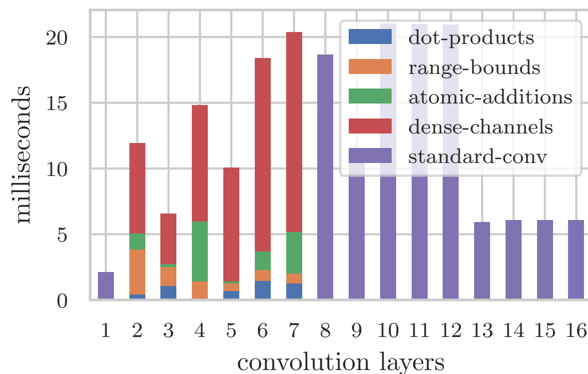

Figure 11. Run time for computing range-bounds, dot-products, atomic additions, and dense channels of our GPU implementation described in Section 4. Each operation is corresponding to Figure 5.

B Videos with Higher Frame Rate

We prepare videos with 240, 120, 60, and 30 fps by subsampling frames of 720p_240fps_[1..6].mov from adobe240fps dataset (Su et al., 2017). Figure 12 shows the averaged run time of our CPU implementation (Figure 8 (c)) for each frame rate. While our method reduces run time with 30 fps as shown in Section 5, the run time further slightly decreases as frame rate increases to 240 because of increase in the temporal redundancy.

Table 6. (error rate, reduction rate) of VGG19_bn changing threshold of CBinfer for each convolution layer and our method.

Threshold Conv layer	0.01	0.05	0.1	0.5	our Algorithm 1
1	0.0%, 17.9%	11.5%, 34.6%	25.9%, 51.7%	63.2%, 84.2%	0.0%, 76.9%
2	0.2%, 15.5%	9.1%, 28.8%	23.4%, 47.3%	59.3%, 80.6%	0.0%, 40.7%
3	0.1%, 11.4%	5.1%, 21.8%	16.2%, 37.4%	53.6%, 73.4%	0.0%, 57.6%
4	0.1%, 9.6%	1.4%, 15.2%	8.6%, 27.9%	53.1%, 72.9%	0.0%, 15.7%
5	0.0%, 5.9%	0.2%, 8.4%	2.7%, 14.5%	37.7%, 57.0%	0.0%, 25.8%
6	0.0%, 5.1%	0.6%, 9.3%	4.0%, 17.0%	54.2%, 70.4%	0.0%, 23.0%
7	0.0%, 4.3%	0.4%, 8.2%	2.5%, 14.1%	55.5%, 71.4%	0.0%, 17.8%
8	0.0%, 3.9%	0.4%, 7.5%	1.9%, 12.3%	50.3%, 66.4%	0.0%, 17.3%
9	0.0%, 2.1%	0.1%, 3.9%	0.6%, 6.3%	29.3%, 42.6%	0.0%, 15.5%
10	0.0%, 1.8%	0.2%, 4.3%	1.5%, 8.2%	48.3%, 60.5%	0.0%, 7.6%
11	0.0%, 1.5%	0.4%, 4.4%	2.2%, 9.3%	62.1%, 73.9%	0.0%, 2.9%
12	0.0%, 1.3%	0.6%, 4.7%	3.5%, 10.8%	73.1%, 82.8%	0.0%, 1.9%
13	0.0%, 0.6%	0.3%, 2.4%	1.8%, 6.2%	66.3%, 73.6%	0.0%, 0.9%
14	0.0%, 0.4%	0.4%, 2.2%	2.3%, 6.5%	70.5%, 78.7%	0.0%, 0.7%
15	0.0%, 0.3%	0.5%, 2.2%	2.9%, 7.3%	77.4%, 85.0%	0.0%, 0.9%
16	0.0%, 0.3%	0.7%, 3.1%	5.2%, 11.5%	89.2%, 94.1%	0.0%, 0.8%

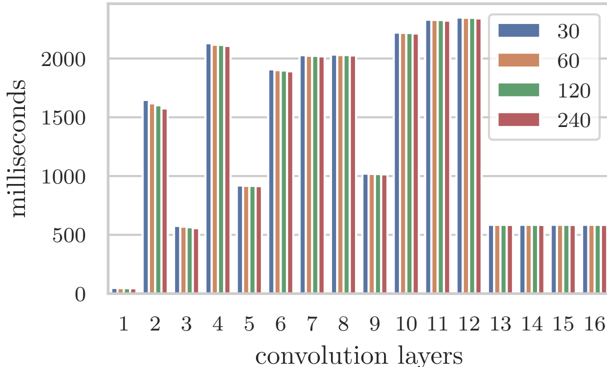

Figure 12. Run time of our CPU implementation corresponding to (c) in Figure 8 for videos with 240, 120, 60, and 30 frame rate subsampled from adobe240fps dataset.

C COMPARISON WITH CBINFER

Since CBinfer (Cavigelli & Benini, 2020) is the closest to our method, we evaluate the trade-off between accuracy and reduction of computations of CBinfer. From YUP++ dataset, we collect pairs of two consecutive frames that the original model (VGG19_bn) classifies them into different classes. We apply CBinfer to convolution layers one by one to avoid tuning a combination of thresholds for multiple convolution layers and test error rate of CBinfer using the second frames in the sets. Table 6 shows the error rate and the reduction rate of CBinfer for each threshold and our method. The reduction rate is the ratio of reducible output elements. For example, CBinfer of the 3rd convolution layer with threshold 0.1 introduced 16.2% error and 37.4% reduction while ours achieved no error and 57.6% reduction. Note that the error rate of CBinfer will increase when applying it to multiple layers. It may be effective to combine our method and CBinfer by applying it to later layers where our method yields low reduction rate.