DeltaCNN: Efficient processing of CNN inference for continuous mobile vision

Seungjoo Lee
KAIST
juicelee@kaist.ac.kr

Jaemin Shin
KAIST
jaemin.shin@kaist.ac.kr

Abstract

Convolutional Neural Network (CNN) models for *continuous mobile vision* are recently being widely deployed on various mobile applications. While it is desired to run CNN models on mobile devices to avoid exposure of privacy-sensitive mobile data, processing complex CNN models on a resource-constrained mobile device became a bottleneck in assuring model accuracy and QoE. Several studies were conducted to accelerate the processing of CNN model on a mobile device, but the state-of-the-art technology only achieves 600ms to 3500ms latency for processing a single image. To address this issue, we propose *DeltaCNN*, which is a system that efficiently reduces the latency of CNN inference by leveraging the sparsity of the intermediate image frames that hardware-based encoder and decoder processes. From the experimental result, *DeltaCNN* shows the reduced latency as the sparsity of the matrix grows, while the latency of the *DeltaCNN* remains higher than the legacy convolution processing. We expect the latency of the *DeltaCNN* to outperform the legacy processing if it is fully implemented on the hardware-based encoder layer.

1 Introduction

With the rapid development in deep learning and computer vision, *continuous mobile vision* is currently serving various applications in end-user mobile devices, including facial recognition[4], street navigation[7], Augmented Reality (AR)[1, 2], and etc. While traditional continuous mobile vision tasks were dominated by hand-crafted features with linear classifier such as Support Vector Machine (SVM), state-of-the-art algorithms for these tasks mostly leverage Convolutional Neural Networks (CNN) with significantly higher accuracy. As such CNN model inference requires excessive computing power, it is usually offloaded on external cloud servers rather than running on a resource-constrained mobile device. However, transmitting input video stream from an end-user mobile device to the cloud server for offloading exhibits serious privacy concerns. Moreover, offloading CNN inference at the cloud server often degrades the Quality of Experience (QoE) on user due to unreliable wireless network conditions at end-user mobile device.

To mitigate this issue, there have been several following approaches that aims to ease the computational complexity of CNN inference to run it on resource-constrained mobile devices. SparseSep [5] reduced runtime internal memory usage of CNN model layers for processing on the resource-constrained environment. Euphrates [19] proposed an algorithm-architecture co-designed system that exploits pixel motion data generated at Image Signal Processor (ISP) for low-power CNN inference. DeepMon [11] introduced a new caching scheme for CNN model inference on continuous mobile vision, which reduces processing latency of convolutional layers. As convolutional layer processing takes significant portion of entire CNN processing (> 85%), DeepMon caches the intermediate processing results from a frame and reuses it at the similar region of the next frame. DeepCache [18] further improved such caching scheme by efficiently searching similar image blocks between two frames that are not in the same position of the image.

Despite all these efforts to accelerate the processing of CNN model for continuous mobile vision, the state-of-the-art technology [18] achieves 600ms to 3500ms latency for processing CNN with a single low-resolution image (below 500 pixels per side), which is still long to process a multiple input frames in real-time. As recent smartphone cameras are taking high-resolution images up to 4k, such latency would be more substantial in practice. To mitigate this issue, this paper focuses on further improving the CNN inference latency and its computational cost.

To this end, we propose *DeltaCNN*, which is a system that accelerates the CNN inference on mobile devices by leveraging the hardware-based encoder and decoder on a mobile device. While the previous works [11, 18] calculates the similarity of the contiguous raw frames, *DeltaCNN* calculates the similarity of contiguous frames with the video codec encoder, which outputs a compressed frame that contains information about the difference between two frames. Our intuition of using video codec encoder comes from the fact that the encoding process could be done in negligible latency with the hardware-based encoder on a mobile device. Such design of *DeltaCNN* is expected to greatly reduce the substantial overhead of similarity calculation of previous works and enable real-time CNN processing on multiple frames on a mobile device.

To test the feasibility of our design of *DeltaCNN*, we implemented the matrix sparsification and sparse matrix multiplication using several APIs (`PyTorch` and SciPy) and compared the processing latency of convolutional layers on different size of the image input. As a result, we found that the latency of *DeltaCNN* gets reduced as the sparsity of the image

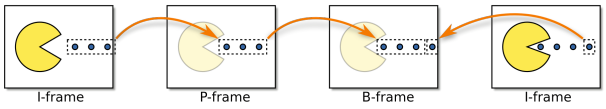

Figure 1. A sequence of video frames, consisting of two keyframes (I), one forward-predicted frame (P) and one bi-directionally predicted frame (B).

grows. The processing latency of *DeltaCNN* is higher than the legacy processing method in the evaluation, but we believe that the latency of *DeltaCNN* would be greatly reduced when it is implemented on a hardware encoder layer, directly leveraging the P-frame for CNN processing.

2 Background

2.1 Video compression

There are two approaches to achieve video compression, which are intra-frame and inter-frame compression. Intra-frame compression uses the current video frame for compression, which is fundamentally same as image compression. Inter-frame compression uses one or more preceding and/or succeeding frames to compress the contents of the current frame. Among them, *H.264*, which is inter-frame compression, is the most commonly used format, used by 91% of video industry developers as of September 2019[6].

Inter-frame compression consists of I-frame (Intra-coded picture), P-frame (Predicted picture), and B-frame (Bidirectional predicted picture). The I-frame contains complete information of an image, like a JPEG or BMP. It can be decoded as a whole frame without any dependencies. The P-frame holds only the changes in the frame from the previous one. Since it only encodes the changes, size of the video can be reduced significantly. The B-frame saves even more space by encodes differences between the current frame and both the preceding and following frames to save the current frame. Fig 1 shows how inter-frame compression works.

2.2 Related work

To enable NN inferencing on resource-constrained mobile environment, many techniques are proposed including cloud offloading and convolutional layer caching.

Cloud offloading Rather than processing NN models directly on mobile device, it is common to conduct inference using cloud services[14]. However, this approach imposes serious privacy concern as it can leak sensitive user data through the network, including images, voices, and textual contents. To keep the user's privacy, several approaches are proposed, such as using the cryptography method[9, 10], and intel SGX[13]. They achieves a high level of user privacy, but their performance is not enough to be used in practice, and they often limit the size of NN model. Moreover, cloud-based network processing can degrade the Quality of Experience

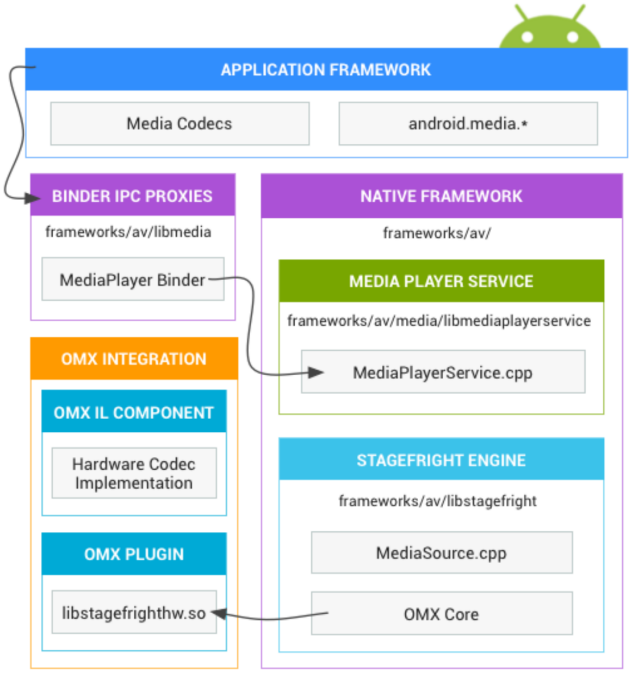

Figure 2. The media architecture of Android. There are hardware components that accelerate the encoding and decoding of a video frame.

(QoE) of the user due to unreliable wireless network conditions.

Convolutional layer caching Some works focused on continuous vision applications on mobile environment, where CNN models are usually used. They cache the result of convolution computation and exploit redundancy between frames in continuous video stream. DeepMon[11] divides a frame into square grids and compares corresponding blocks between current frame and previous frame using the histogram of colour distribution. If there are reusable regions (cache hit), it fetches previous convolution computation results for those regions from the cache. DeepCache[18] also caches the calculation result of previous frame and compares it to current frame. Nevertheless, unlike DeepMon, it uses diamond search[16] that enables comparison between grids in different location, thus tolerates the scene variation.

3 System Overview

The goal of the *DeltaCNN* is to reduce the computational complexity and processing latency of CNN inference at general continuous mobile vision tasks. Prior to the *DeltaCNN*, other approaches [11, 18] calculated the similarity between contiguous raw frames of video, which results in substantial overhead in computation. *DeltaCNN* aims to minimize this overhead by encoding the input video frame with a video codec, which outputs a compressed version of video frame that describes the difference between the contiguous frames.

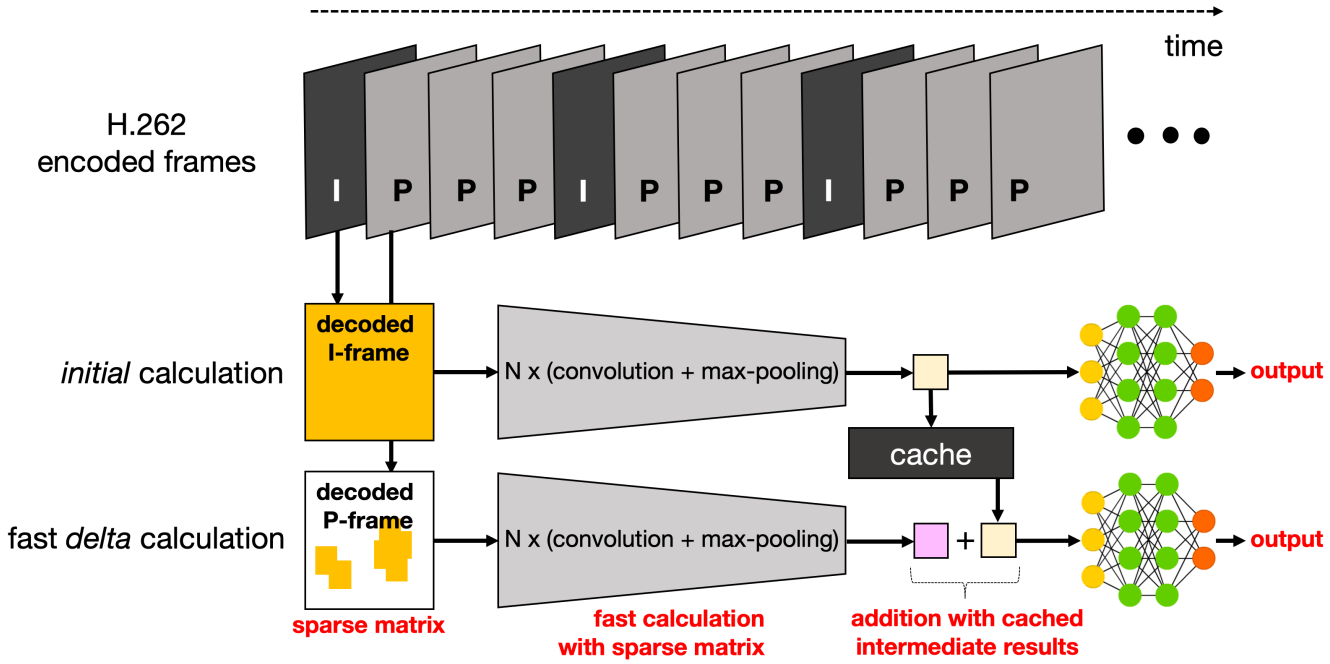

Figure 3. System overview of DeltaCNN.

The main benefit of encoding the input video frame comes from the existence of hardware-level encoder and decoder at the mobile devices, which performs encoding and decoding of video with negligible overhead. As an example, Figure 2 shows the media architecture of Android which includes the hardware components for encoding and decoding. Thus, *DeltaCNN* requires continuous mobile vision task to encode its input video frames with certain video codec. In this paper, we focus on demonstration of *DeltaCNN* with H.262 (MPEG-2) codec [3] and further discuss how it could be extended at other types of video codecs.

Figure 3 shows the overall functional architecture of our system, *DeltaCNN*. Each H.262 encoded frames of the input stream are processed through CNN model, which includes multiple convolutional layers with max-pooling and fully-connected layers. As the stream of encoded frames consists of multiple set of one I-frame followed by multiple P-frames, *DeltaCNN* processes the CNN model with 2-stage processing on each set: (1) initial I-frame calculation and (2) fast P-frame (*delta*) calculation.

Initial I-frame calculation When a I-frame comes as a new frame to be processed, *DeltaCNN* decodes the frame with hardware-based decoder. Then, *DeltaCNN* processes the decoded I-frame through convolutional layers and pooling layers, then cache the intermediate result afterwards. Next, the intermediate result gets processed through fully-connected layers and *DeltaCNN* outputs result for the I-frame.

Fast P-frame (*delta*) calculation After the I-frame being processed, a P-frame comes as a new frame to be processed. This process holds same for multiple P-frame that appears after a I-frame. The *DeltaCNN* decodes a P-frame then processes the decoded frame through convolutional layers and pooling layers. Since the P-frame only contains the motion vector and the residual difference between the frames, the decoded P-frame is sparse, containing much less information compared to decoded I-frame. Processing a sparse matrix through convolutional layers and pooling layers could be done much faster compared to the same calculation with decoded I-frame. After the processing, *DeltaCNN* adds the intermediate result with the cached intermediate result from the initial I-frame calculation. Then, *DeltaCNN* processes the added value through fully-connected layers and outputs result for the P-frame. The design rationale behind this processing design is explained in detail in Section 4.

4 DeltaCNN Design Rationale

4.1 Interpreting the P-frame

When the P-frame is encoded, the current frame is divided into 16 pixel × 16 pixel macroblocks and compared with the previous frame (reference frame). When there are matched blocks between two frames, the encoder calculates the offset between two blocks and encodes it as an "motion vector". Motion vector is calculated at the granularity of half-pixel, and frequently zero due to redundancy of video stream. It also encodes residual to compensate the error of matching. If there is no suitable match, for example, when the new

object shows up in the scene, macroblocks for that region are treated like an I-frame block.

In result, we can get a motion vector, which is frequently zero, and a residual from the P-frame. With cached previous frame, we can get a difference between two frames.

4.2 Addition of CNN

Sine the convolution is a linear operator, it has a distributive property. Thus, following equation holds for convolution, where $x(n)$ is a weight of convolutional layer, and $h_1(n)$ and $h_2(n)$ are input.

$$x(n) * \{h_1(n) + h_2(n)\} = x(n) * h_1(n) + x(n) * h_2(n)$$

This means if we have the convolution result of previous frame, we can compute the convolution of current frame only by apply convolution to *delta* like the equation below.

$$Img_{cur} = Img_{prev} + \Delta$$
$$x(n) * \{Img_{cur}\} = x(n) * Img_{prev} + x(n) * \Delta$$

However, there are layers in CNN that is not a linear operator like max-pooling and activation. If the *delta* passes those layers and just added to the calculation result of previous calculation, it would produce wrong results. We will explore and report the effect of those errors on the accuracy of NN models on evaluation section.

4.3 Sparse Matrix Multiplication

As shown in Fig 4, convolution operation with multiple channels and multiple filters is represented as matrix-matrix multiplication. As *delta* is a sparse matrix due to the redundancy of the video stream, we can compute the matrix-matrix multiplication much faster.

5 Implementation & Evaluation

In this section, we discuss the implementation and evaluation of our system in several platforms including `Pytorch` and `SciPy`. Here, we do not directly use the P-frame from the hardware encoder, rather subtract RGB value of one frame from that of the previous frame to imitate the P-frame input and show the feasibility of our design.

5.1 PyTorch implementation

Our system needs the convolutional layer to allow sparse matrix representation. However, as there is no deep learning framework that allows this behavior, we make our own convolutional layer on `PyTorch`. We mimic the behavior of original implementation of the convolutional layer that changes the convolutional calculation into one matrix-matrix multiplication[8], as shown in Fig 4.

Fig 5 shows the structure of implemented convolution layer. It saves last input to calculate the difference between current input and last input, and changes calculated difference into sparse matrix representation such as COO or CSR format. Using the sparse matrix, it calculates the convolution by matrix multiplication using `torch.sparse` module, adds the result with saved last output, and finally produces output. In this design, there is no accuracy drop even there are non-linear activations between convolutional layers since it caches the last input and output for each convolutional layer. However, there exists additional overhead as every convolutional layer have to change the representation of the matrix.

We evaluate our implementation with UCF101 dataset[17] which contains 13,421 short videos including 101 types of human activities. We randomly select 10 types of activities and evaluate *DeltaCNN* over them. We use AlexNet[12] CNN model to verify *DeltaCNN*. AlexNet is pre-trained on ILSVRC 2012 dataset[15], and transferred learned on UCF101 dataset. We modified the last fully-connected layer of the model to classify activity shown on video frame, and all convolution layers to our own layer in Fig 5. Using the transfer learned model, we feed the video frames of the validation dataset sequentially to the RTX TITAN GPU, mimicking the continuous vision application in real world.

Fig 7 shows the evaluation result of *DeltaCNN* in terms of average processing latency and accuracy. `dense` calculates the original convolution operation without changing to sparse matrix representation, `sparse` means convolution with sparse matrix representation, and `sparse_heuristic` applies heuristic to make the input more sparse, which zeros out the input values below the mean of each channel. As expected, there is no accuracy drop using the `sparse`, the accuracy of it is same with `dense`. `sparse_heuristic` drops the accuracy significantly about 50% since it changes the input value and error accumulates more and more by caching the previous input and output. However, the processing latency of `dense` is faster than the `sparse` and `sparse_heuristic`. When the heuristic is applied, it improves the processing latency than the `sparse`, but it is still slower than the `dense`.

We suspected that the `PyTorch` framework is not efficient at sparse matrix processing, because it is currently in experimental stage and not stabilized. The processing time is slower than the `dense` even if we input the zero tensor to the all the convolutional layers.

5.2 SciPy implementation

We test our implementation with `SciPy`, using stabilized sparse matrix multiplication API on CPU. We use the same CNN model and own convolutional layer implementation using `PyTorch` in section 5.1. However, when the convolutional layer calculates the matrix multiplication, we copy the matrices into CPU cache and calculates the result using `SciPy` and copy back the result into GPU.

Here, we evaluate with first 1,000 frames of the validation dataset to show the general trend. Fig 6 shows the average processing latency of each convolution layer on CPU. `mm_dense` means matrix multiplication without changing to sparse matrix, `to_sparse` means changing from dense matrix representation to sparse matrix representation, and

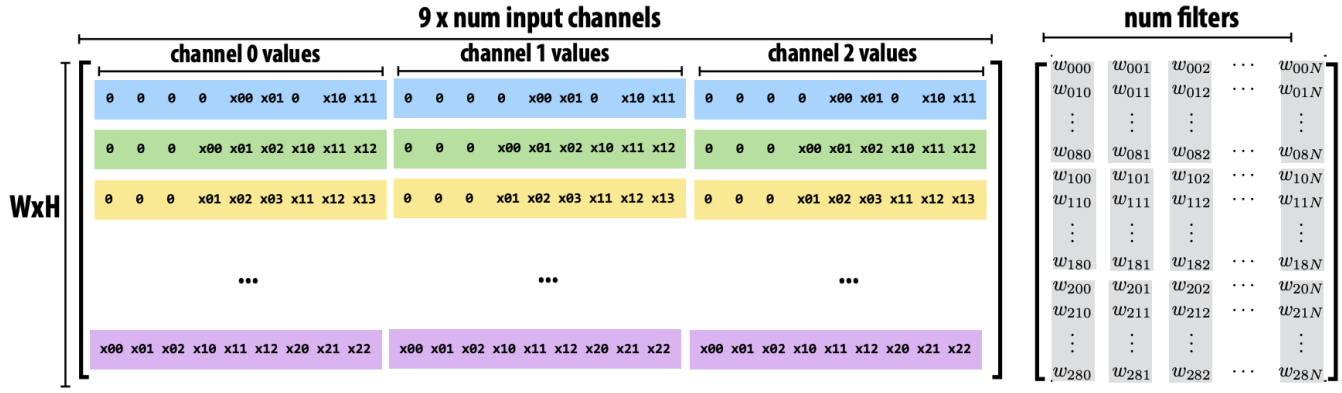

Figure 4. Convolution operation with multiple channels and multiple filters

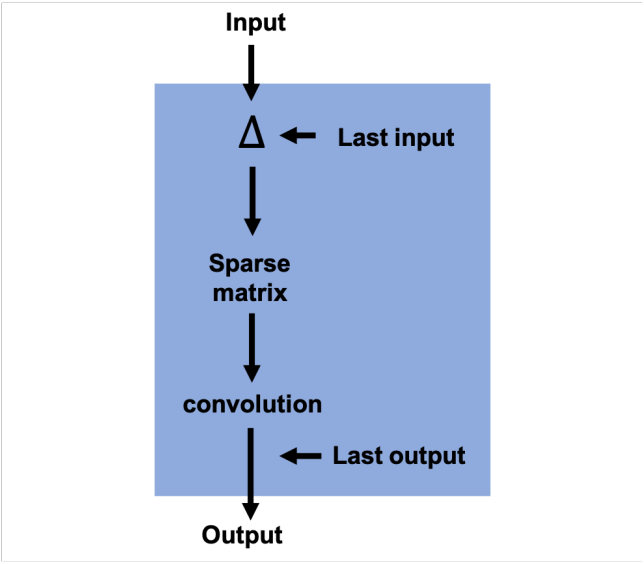

Figure 5. Convolution layer structure of our implementation

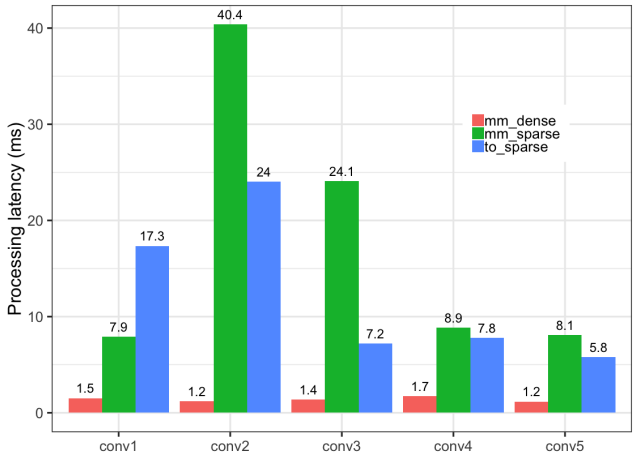

Figure 6. Processing latency of each convolution layer on CPU

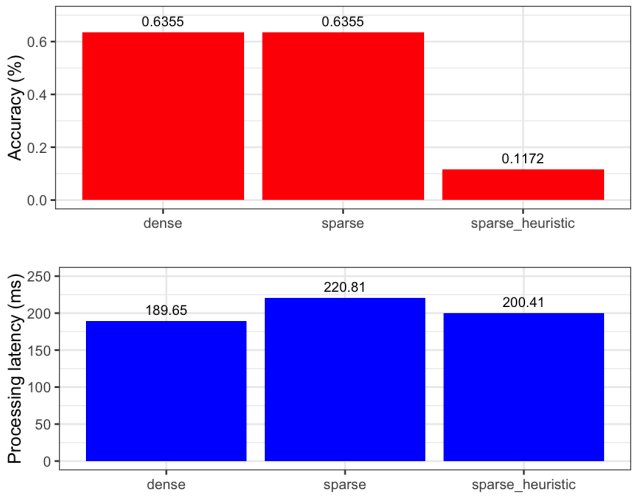

Figure 7. Accuracy and processing latency of pytorch implementation

`mm_sparse` means matrix multiplication in sparse matrix representation. Thus, the latency for *DeltaCNN* is `to_sparse` + `mm_sparse`, and the latency for normal convolution is `mm_dense`.

We can observe that the latency for `mm_dense` is much shorter than the `mm_sparse` which even does not account sparsification overhead, which is identical result with GPU evaluation (section 5.1). However, the video size of UCF101 dataset is 320×240, which is too small to get the benefit from sparse matrix multiplication.

5.3 Efficiency of sparse matrix multiplication

To test the efficiency of sparse matrix multiplication in extreme environment, we feed the randomly-generated 4K resolution image (3840×2160) to the one layer of our convolutional layer with 64 $3 \times 11 \times 11$ kernel with different

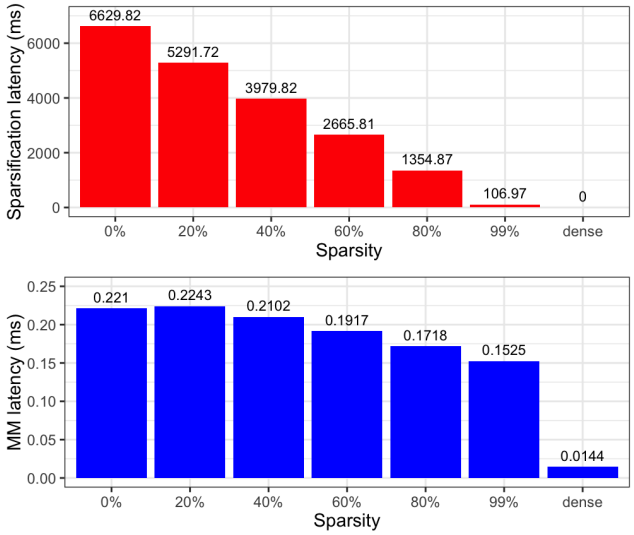

Figure 8. Processing latency of mm and sparsification of matrices with different sparsity

sparsity. We report the average latency with 20 trials for each setting, executing on RTX TITAN GPU.

Fig 8 shows the result in terms of sparsification latency and matrix multiplication latency. `dense` means matrix multiplication without sparse matrix representation, so it does not have sparsification overhead. When the sparsity increases, the sparsification latency and matrix multiplication latency drops. However, sparse matrix multiplication is much slower than `dense` even with 99% sparse matrix.

To get the benefit of sparse matrix multiplication, the input matrix should be much larger so that the number of required computation is much smaller than dense multiplication. Unfortunately, the input matrix for continuous vision multiplication is not large enough to get the advantage from sparsification, even with 4K resolution.

6 Discussion

In the previous section, we have evaluated the latency of *DeltaCNN* in various configurations varying the matrix multiplication API and the input image size. Our experimental results show that the processing latency of *DeltaCNN* which includes `to_sparse` (matrix sparsification) and `mm_sparse` (sparse matrix multiplication) is much greater than the processing latency of the legacy dense matrix multiplication in the convolution layer. However, we believe that the evaluation results do not mean that *DeltaCNN* is not effective in reducing the CNN inference latency, since we used implemented third-party APIs for the evaluation and have not fully investigated on how each functionality is implemented. We suspect that the latency of dense matrix multiplication is so small due to various optimization techniques applied over a long period of frequent usage, while the latency of sparse matrix API is relatively big since it is a newly published experimental feature and its implementation is not fully optimized.

Moreover, when *DeltaCNN* could be directly applied on the layer of hardware encoder where I, P, B frames are generated, *DeltaCNN* would have more chance of being beneficial over the legacy convolution process in terms of processing latency. Since matrix sparsification is done on hardware encoder on both *DeltaCNN* and legacy process, additional overhead from the matrix sparsification becomes zero unlike the experimental results in Section 5. However, input image size and the average sparsity among images that is required to have less latency with *DeltaCNN* is still unveiled, as the sparse matrix multiplication still takes unnelgigible amount of overhead. We leave the implementation of *DeltaCNN* on the hardware encoder layer and the latency evaluation on different input size and sparsity as a future work of this research.

7 Conclusion

The convolutional neural network (CNN) models for *continuous mobile vision* have recently been widely implemented in various mobile applications. While it is desired to run CNN models on mobile devices to avoid exposing privacy-sensitive mobile data, the processing of complex CNN models on a mobile device with limited resources became a bottleneck to ensure the accuracy of the model and QoE. Several studies were conducted to speed up the processing of the CNN model on a mobile device, but cutting-edge technology only reaches a latency of 600ms to 3500ms to process a single image. To address this problem, we propose *DeltaCNN*, which is a system that efficiently reduces the latency of CNN inference by taking advantage of the sparsity of intermediate image frames that is processed at hardware-based encoders and decoders. From the experimental result, *DeltaCNN* shows reduced latency as the matrix sparsity increases, while the latency of *DeltaCNN* remains higher than the legacy convolutional layer processing. We expect *DeltaCNN* latency to be less than the legacy processing when it is fully implemented in the hardware-based encoder layer.

References

[1] 2016. *Google glass.* Retrieved December 8, 2016 from https://developers.google.com/glass/

[2] 2016. *Hololens.* Retrieved December 8, 2016 from https://www.microsoft.com/microsoft-hololens/en-us

[3] 2019. *MPEG-2.* Retrieved October 28, 2019 from https://en.wikipedia.org/wiki/MPEG-2

[4] 2019. *Windows hello.* Retrieved November 27, 2019 from https://www.microsoft.com/en-us/windows/windows-hello

[5] Sourav Bhattacharya and Nicholas D. Lane. 2016. Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys '16).* ACM, New York, 176–189. https://doi.org/10.1145/2994551.2994564

[6] Bitmovin 2019. *2019 VIDEO DEVELOPER REPORT*. Retrieved September, 2019 from https://go.bitmovin.com/video-developer-report-2019

[7] Tiffany Chen, Hari Balakrishnan, Lenin Ravindranath, and Paramvir Bahl. 2016. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. *GetMobile: Mobile Computing and Communications* 20 (07 2016), 26–29. https://doi.org/10.1145/2972413.2972423

[8] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[9] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2016. *CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy*. Technical Report MSR-TR-2016-3. https://www.microsoft.com/en-us/research/publication/cryptonets-applying-neural-networks-to-encrypted-data-with-high-throughput-and-accuracy/

[10] Ehsan Hesamifard, Hassan Takabi, and Mehdi Ghasemi. 2017. CryptoDL: Deep Neural Networks over Encrypted Data. *CoRR* abs/1711.05189 (2017). arXiv:1711.05189 http://arxiv.org/abs/1711.05189

[11] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*. ACM, New York, 82–95. https://doi.org/10.1145/3081333.3081360

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[13] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. 2019. Occlumency: Privacy-preserving Remote Deep-learning Inference Using SGX. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19)*. ACM, New York, NY, USA, Article 46, 17 pages. https://doi.org/10.1145/3300061.3345447

[14] Microsoft Azure Cognitive Services 2019. *Microsoft Azure Cognitive Services*. Retrieved July 18, 2019 from https://azure.microsoft.com/en-us/services/cognitive-services/

[15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. 2014. ImageNet Large Scale Visual Recognition Challenge. *CoRR* abs/1409.0575 (2014). arXiv:1409.0575 http://arxiv.org/abs/1409.0575

[16] Shan Zhu and Kai-Kuang Ma. 1997. A new diamond search algorithm for fast block matching motion estimation. In *Proceedings of ICICS, 1997 International Conference on Information, Communications and Signal Processing. Theme: Trends in Information Systems Engineering and Wireless Multimedia Communications (Cat.*, Vol. 1. 292–296 vol.1. https://doi.org/10.1109/ICICS.1997.647106

[17] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. 2012. UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild. *CoRR* abs/1212.0402 (2012). arXiv:1212.0402 http://arxiv.org/abs/1212.0402

[18] Mengwei Xu, Mengze Zhu, Yunxin Liu, Felix Xiaozhu Lin, and Xuanzhe Liu. 2018. DeepCache: Principled Cache for Mobile Deep Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18)*. ACM, New York, 129–144. https://doi.org/10.1145/3241539.3241563

[19] Yuhao Zhu, Anand Samajdar, Matthew Mattina, and Paul Whatmough. 2018. Euphrates: algorithm-SoC co-design for low-power mobile continuous vision. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. ACM, New York, NY, USA, 547–560. https://doi.org/10.1109/ISCA.2018.00052