Zoper - High Performance Direct Convolution Without Runtime Packing and Reordering On CPUs*

Fuheng Wu[†] *Pingbo Zhang*[‡] *Ivan Davchev*[§] *Jun Qian*[¶]

Abstract

Convolution, a fundamental operation in deep-learning neural networks, particularly in Convolutional Neural Networks (CNNs), is highly demanding in terms of computation. Consequently, it has been the subject of extensive investigation by both academia and industry for many years. Previous studies have proposed two primary methods for computing convolutions on CPUs, including the "Im2Col + BLAS" technique, which incurs significant memory overhead, and the "NCHWc direct convolution" approach, which necessitates input and output data reordering, thereby increasing the peak memory usage and introducing extra complexities into the neural network. This paper introduces novel direct convolution techniques that avoid the need for runtime packing and reordering, resulting in a substantial reduction in memory usage and latency. Our benchmarking demonstrates that the proposed methods can achieve a memory reduction of 10-50% and a latency reduction of up to 70% compared to the current state-of-the-art approaches in production use cases.

1 Introduction

Convolutional Neural Networks (CNNs) are a class of deep neural networks that are widely used in computer vision applications such as image and video recognition, object detection, and segmentation. At a high level, the numerical algorithm used in CNN computation involves performing a series of matrix multiplications or convolutions between the input data and learnable weights (aka filters or kernels) of the network. The output of each layer is then passed through an activation function and fed into the next layer. In all the layers, convolution layer is the most computationally intensive operations, so optimizing convolution is critical to improve inference performance. A significant restriction of CNN models on mobile and low-power devices is the considerable computational burden associated with consecutive convolutional layers where high-end expensive GPU is unavailable. Even for most AI services providers, CPU serving is still a pragmatic solution due to the cost control pressure.

There are many ways to do convolution, but GEMM and direct convolution are the most widely studied and adopted in the industry and academia. GEMM has a major drawback of big memory overhead. The process involves packing overlapping image blocks, whose sizes match those of the kernel, into the columns of a sizeable temporary matrix. This extra packing operation and memory overhead has been a big problem in many use cases, so that direct convolution is proposed as a more efficient algorithm. The existing direct convolution algorithm doesn't need data packing but it need runtime input and output data reordering. Therefore, it is not real zero-memory overhead because the temporary memory used in the reordering process is the same as input data size, so it is not negligible. The reordering also adds extra complexity to the neural networks because the output reordering, as the next layer's input, need to be considered. Furthermore, the existing direct convolution involves several hyperparameters such as input block size, kernel thunk size, output block size, making the implementation even more complicated and error-prone.

2 Background

All the previous researches try to provide a generic solution, but ignores the use cases when input data is small. Take OCR as an example, it requires detection and recognition of many small text bounding boxes. Figure 1 is a typical input image of an OCR service.

We use Figure 1 as the input and profile the OCR service with HiQ, an observability and optimization tool open-sourced by Oracle(Fuheng et al., 2023). There are many small size inputs as shown in the profiling report Figure 2.

The diminutive size of the bounding boxes, such as those with dimensions of 32x21 and 32x22, belies their significant cumulative impact on processing

*This work is done at Oracle Cloud AI Services. Corresponding author: fuheng.wu@oracle.com. Mailing address: OCI Cloud AI Services, 100 Oracle Pkwy, Redwood City, CA 94065, USA.

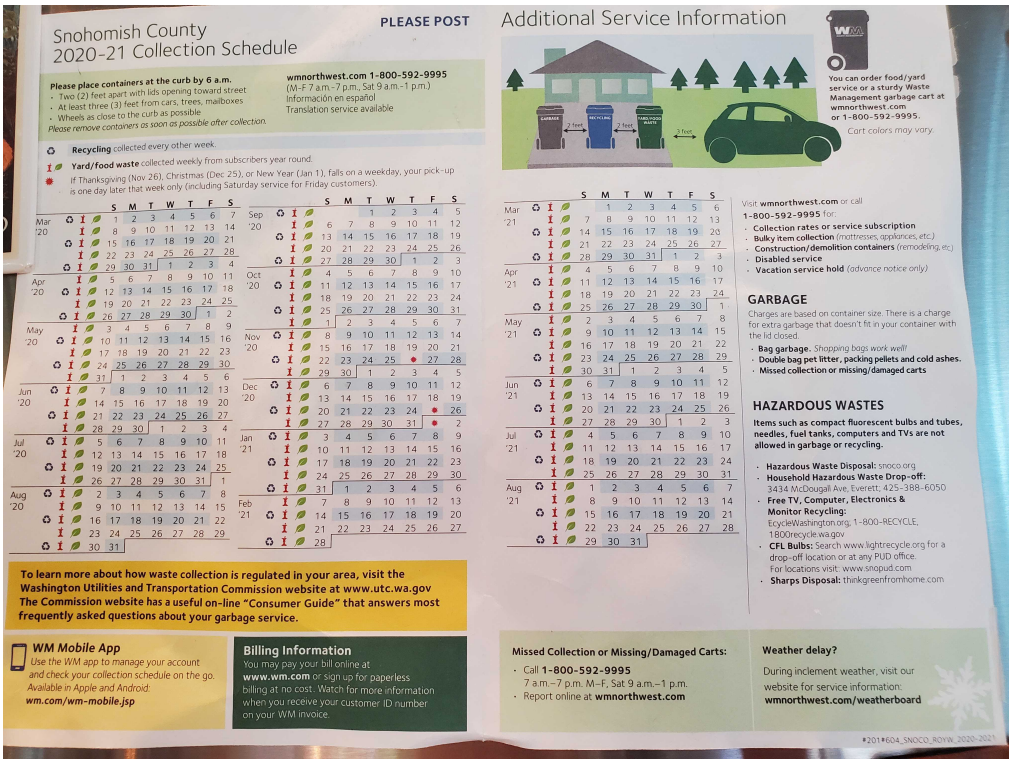

Fig. 1: Sample Input Image

```
|___rec(1.7044) ({'args': 'TextRecognizer(),list(952,ndarry(41,542,3),ndarry(33,60,3),ndarry(46,89,3
|   |___ort_run(0.0207) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,21)'])"})
|   |___ort_run(0.0201) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,21)'])"})
|   |___ort_run(0.0198) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,22)'])"})
|   |___ort_run(0.0210) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,23)'])"})
|   |___ort_run(0.0214) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,23)'])"})
|   |___ort_run(0.0213) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,24)'])"})
|   |___ort_run(0.0233) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,25)'])"})
|   |___ort_run(0.0256) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,31)'])"})
|   |___ort_run(0.0275) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,34)'])"})
|   |___ort_run(0.0294) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,35)'])"})
|   |___ort_run(0.0319) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,37)'])"})
|   |___ort_run(0.0317) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,39)'])"})
|   |___ort_run(0.0312) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,39)'])"})
|   |___ort_run(0.0339) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,40)'])"})
|   |___ort_run(0.0357) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,40)'])"})
|   |___ort_run(0.0310) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,40)'])"})
|   |___ort_run(0.0334) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,41)'])"})
|   |___ort_run(0.0344) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,41)'])"})
|   |___ort_run(0.0364) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,42)'])"})
|   |___ort_run(0.0334) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,42)'])"})
|   |___ort_run(0.0348) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,43)'])"})
|   |___ort_run(0.0379) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,43)'])"})
|   |___ort_run(0.0367) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,44)'])"})
|   |___ort_run(0.0362) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,47)'])"})
|   |___ort_run(0.0400) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,52)'])"})
|   |___ort_run(0.0481) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,60)'])"})
|   |___ort_run(0.0537) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,72)'])"})
|   |___ort_run(0.0721) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,95)'])"})
|   |___ort_run(0.0812) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,110)'])"})
|   |___ort_run(0.0993) ({'args': "InferenceSession(),None,dict(k:['x'],v:['ndarry(30,3,32,135)'])"})
```

Fig. 2: OCR Model Profiling(Partial)

Tab. 1: Notations in Convolution

variable	description
N	batch of input, also known as group
C	input or kernel channel number
H	input height
W	input width
K	batch number of kernel
R	kernel height
L	kernel width
K	number of output channel
OH	output height
OW	output width
U	kernel batch-wise block size
B	input channel-wise block size
D	output channel-wise block size

time. While each individual bounding box requires less than 100 milliseconds to process, the overall time required to complete processing of all such boxes can exceed one second, as illustrated in the accompanying graph. For small input data, the extant algorithms are beset by inefficiencies due to the algorithmic overhead incurred by the use of multi-threading strategies and runtime data padding and reordering. This paper presents an efficient algorithm that significantly outperforms the state-of-the-art generic algorithm, offering superior processing speed with a reduced memory footprint.

2.1 Notations and Conventions

Figure 3 is a graphic view of three components in convolution: `input`, `kernel` and `output`. Sometimes, `kernel` is also called `filter` since its main purpose is to filter out features from input data.

Notations used in this paper are listed in table 1. Please be noted that `N` is input batch, and also the output batch. We will assume N equals to 1 in this page, but the conclusion can be extended to cases when N is greater than 1. `K` is kernel batch number and also equal to output channel number. Input channel number and kernel channel number must be equal according to the definition of convolution, which are represented by `C`. There are other parameters like stride, padding, dilutions which are not covered by this paper.

2.2 Data Format

During the convolution operation, the input data is usually represented as a tensor. There are two common data formats for representing tensors in convolution: `NCHW` and `NHWC`.

`NCHW` stands for "batch size, number of channels, height, and width." In this format, the input tensor is a 4D array where the first dimension represents the batch size, the second dimension represents the number of input channels, and the third and fourth dimensions represent the height and width of the input image. This format is commonly used in deep learning frameworks such as PyTorch, Caffe and ONNX.

`NHWC` stands for "batch size, height, width, and number of channels." In this format, the input tensor is a 4D array where the first dimension represents the batch size, the second and third dimensions represent the height and width of the input image, and the fourth dimension represents the number of input channels. This format is commonly used in deep learning frameworks such as TensorFlow and Keras.

The choice of data format can have an impact on the performance of the convolution operation. For example, the `NHWC` format can provide better performance on GPU's generic core due to the way they handle memory accesses. However, the `NCHW` format can be more efficient for certain cases, such as depth-wise convolution, GPU tensor cores and CPUs.

In this paper, we discuss convolution in CPU and compare with the state-of-arts method which also use `NCHW` format, so we assume the input data format is `NCHW` in this paper.

2.3 Kernel Size

The use of 3x3 kernels as a standard in Convolutional Neural Networks (CNNs) has become widely adopted, as evidenced by its inclusion in both traditional models such as VGG(Simonyan and Zisserman, 2014) and contemporary cutting-edge models such as Stable Diffusion(Chen et al., 2023). This convention has emerged as a result of a combination of factors, including the computational efficiency of 3x3 kernels, their hierarchical structure and their efficacy in promoting translation invariance. (Camgözlü and Kutlu, 2021)

The computational efficiency of 3x3 kernels is noteworthy, requiring fewer parameters to train and fewer operations to execute than larger kernel sizes like 5x5 or 7x7. This attribute renders them particularly well-suited to the demands of large-scale deep learning models that necessitate many convolutional layers. CNNs typically possess a hierarchical structure that applies smaller kernels at the outset of the network and larger kernels later on. This design affords the network the capacity to learn elementary features like

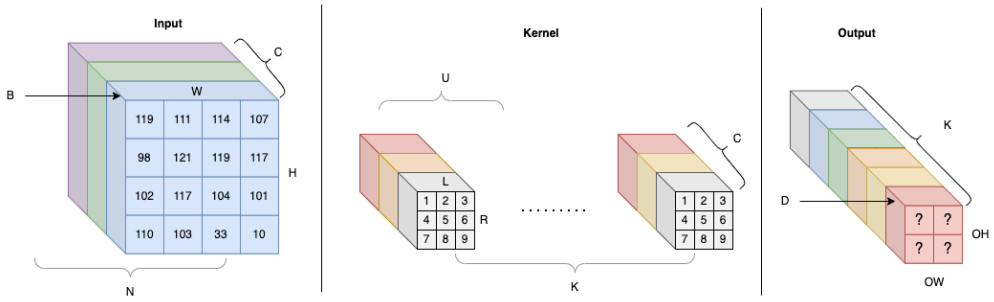

Fig. 3: Notations in Convolution

edges and corners in the initial layers and more intricate features like patterns and objects in later layers. The compact dimensions of 3x3 kernels make them an intuitive choice for the initial layers of the network, given their ability to capture small-scale details while also facilitating a receptive field that encompasses a reasonable area of the input image. Finally, CNNs are designed to be translation-invariant, allowing for pattern recognition independent of an object's placement within the input image. In this regard, 3x3 kernels are particularly effective in capturing local features, such as edges and corners, that are integral to preserving translation invariance.

2.4 Naive Convolution Algorithm

The naive convolution algorithm is generally applicable to all the use cases as follows:

```
REP(k,0,K){
 REP(h,0,OH){
  REP(w,0,OW){
   int i=output_index(k,h,w);
   REP(c,0,C){
    REP(r,0,R){
     REP(l,0,L){
      output[i] +=
       input[input_index(c,h+r,w+l)] *
       kernel[kernel_index(k,c,r,l)];
     }
    }
   }
  }
 }
}
```

This is a straightforward algorithm because it is how convolution is defined in DNN. The drawback of this algorithm is its inefficiency in CPU cache and register usage.

2.5 Related Work

Many memory efficient algorithms have been proposed to reduce the convolution computation overhead. MEC(Cho and Brand, 2017) improves im2col memory efficiency but is based on GEMM + BLAS so it is often suboptimal in computation. Additionally, it still causes considerable memory overhead.

A high performance direct convolution algorithm(Zhang et al., 2018) is proposed for CPUs supporting vector registers and FMA instruction. This algorithm has been widely used by industry's mainstream inference engines including Amazon NeoCPU(Liu et al., 2019), Microsoft ONNX Runtime. One drawback is it requires runtime input data reordering, which incurs `O(C*H*W)` extra memory usage. This algorithm introduces several hyperparameters including input block size(B), kernel chunk size(U), output block size(D), which makes the implmentation very complex. An optimization scheme search algorithm(Liu et al., 2019) is proposed to tune and find the best hyperparamter for a model. Because the computation nodes are chained, the input data layout change often impacts the entire neural network structure so that layout transformation elimination has to be considered. The implementation difficlty is exponentially increased with all the things considered. Microsoft ONNX Runtime has the best implementation with all assembly code written by hands and meet the requirements in most of use cases.

A super simple algorithm SMM-Conv(Ofir and Ben-Artzi, 2022) is proposed as a direct convolution algorithm using sliding windows similar to EMC but without GEMM. However, the algorithm requires extraction of sub-matrices which still requires extra `O(H*W)` memory. Another drawback is the latency increases dramatically due to both the sub-matrix extraction and inefficient usage of CPU registers.

3 Zero-Overhead Direct Convolution

AVX/AVX2 is widely availabele in many server or even consumer-grade CPUs. This paper presents a real zero-oeverhead algorithm for performing direct convolution on CPUs without requiring packing or reordering.

We describe the algorithm without CPU vector extension support, followed by the optimal register allocation algorithm that employs CPU Fused-Multiply-Add (FMA) instructions. Additionally, we present experimental results that demonstrate on-par or superior performance with less memory consumption.

3.1 Observations

With the conventional `NCHW` data format, direct convolution encounters performance issues due to the unfavourable memory access pattern dictated by the CNN convolution definition. The process involves loading data from the kernel, performing a dot-product with corresponding data across multiple rows in the input data, and summing them to obtain a single numeric value in the output. This results in frequent reloading of the same data, leading to inefficiency. Our algorithm overcomes this issue by reducing the frequency of register data reloading while maintaining a low CPU cache miss rate, resulting in greater efficiency.

3.2 Zoper Direct Convolution with FMA

In the AVX architecture, the fused multiply-add (FMA) instructions can be applied to the 16 256-bit YMM register files, namely Y0 - Y15.

3.2.1 Algorithm One

For the case of a 3 by 3 kernel, one good register allocation strategy involves using one YMM register, Y0, to store input data, another register, Y1, to load kernel data, and 12 registers, equivalent to 4 x 3, to store the FMA results. Similarly, for a 2x2 kernel, the best strategy is to allocate Y0 for input data, Y1 for kernel, and 14 registers, equivalent to 7x2, to store the FMA results by kernel's batch. If the total number of registers is denoted as G, then we can process (G-2)//R batches of kernels during each run.

For the purpose of improving cache locality with `NCHW` format, assuming that N==1, we split the outer dimension channel without rearranging the data. Specifically, the channel is partitioned into C/B subsets, where each subset has B channels, as illustrated in the notation section. Since input channel is a reduction dimension according to convolution definition and it is outside of H and W, we try the best to avoid the previous channel evicted from the cache when accessing a new channel. We split the input data in channel dimension to make each channel data remain in the CPU cache line as much as possible. In that case, the data format that we use is actually NCBHW. When B is small, cache miss rate is low but register reloading are more frequent. To strike a balance between them, in our experiments, we set B=16 because it shows better performance than 8 or 32, and the channel number is a multiplier of 16 in all of our models.

To illustrate the algorithm, we take the 3x3 kernel as an example. The procedure is as follows:

```
REP(k, 0, C/B){
  REP2(w, 0, OW, 8){
    REP(h, 0, H){
      REP(c, 0, B){
        y0=input[c,h,w]
        y1=input[c,h,w+1]
        y2=input[c,h,w+2]
        REP(ki,0,4){
          y3=kernel at row 0
          y6,9,12,15=fma(y0, y3, y6,9,12,15)
          y3=kernel at row 1
          y5,8,11,14=fma(y1, y3, y5,8,11,14)
          y3=kernel at row 2
          y4,7,10,13=fma(y2, y3, y4,7,10,13)
        }
      }
      write y4,7,10,13 to output
      y4,7,10,13<-y5,8,11,14<-y6,9,12,15
    }
  }
}
```

The efficiency of this algorithm comes from both the `reusage` and `satuation` of YMM registers. All the 16 registers are fully used. Input data in Y0, Y1, Y2 are used for computation without storing and loading from cache or RAM repeatedly. Intermediate results are stored in Y4 to Y15 without storing and loading from cache or RAM.

3.2.2 Algorithm Two

We reorder kernel to format: `C'K'cRLk` where `cRLk` are small blocks cut off in channel and batch dimension. For AVX2, the `k` dimension has max value of 8 = 256/32. The `c` dimension could be any value, but we normally set it as 8.

4 Experiments

The experiments in this study were conducted on an Oracle Compute E3 VM equipped with an AMD EPYC 7742 2.2GHz CPU featuring 64KB L1, 512KB L2 cache, and a 1.6MB L3 cache shared by 48 cores. To simulate the small bounding box input observed in production, experiments were performed using data of similar dimensions.

Our benchmarking shows that for the same convolution algorithm Assembly code can be 10%-30% faster than C++ code. C++ compiler optimization is enabled at `O3` level but plain C++ code cannot arrange registers in the best allocation strategy. Our code is written in C++ Intel intrinic nemocis while the SOTA algorithm code is in pure assembly, but our code is faster than SOTA in most cases even if we are at disadvantage in terms of code execution speed. `vmovaps` or `vmovdqa` can be used to copy data between registers and `vfmadd231ps` can be used for FMA operation, `vxorps` for reset register.

4.1 Zoper Vs. SOTA(NCHWc Direct Convolution)

Our findings indicate that while the state-of-the-art (SOTA) method outperforms Zoper by a small margin for input data and kernel sizes greater than 780KB, Zoper surpasses SOTA for data below this threshold. Notably, in the case of 64 input channels and a kernel batch size of 4 or 8, Zoper demonstrated a remarkable 30-fold speedup over SOTA. Moreover, in terms of memory efficiency, Zoper consistently outperforms SOTA across all input sizes.

4.2 Zoper Vs. SMM-Conv

We tested SMM-Conv algorithm in various platforms including Raspberry Pi 4, AMD and Intel. When FMA instruction is unavailable or not used, SMM-Conv performs better than other algorithms due to its relatively high efficient CPU cache usage. However, when FMA instruction set is available, SMM-Conv's drawback becomes obvious because it fails to fully leverage CPU register files thus incurs high data reloading. Also it doesn't split in channel dimension, so when input channel increase, the performance drops due to high cache miss rate. When output channel increase, due to frequent duplicate data reloading, the performance drops so dramatically that it performs even worse than EMC algorithm. It is a neat and useful algorithm for mobile and embedded devices where computing resources are limited. 7

4.3 Input Pattern Aware Inference

The proposed algorithm doesn't work with larger kernels due to the limited register numbers in CPU. While 3x3 filters have become the standard in CNNs, there are cases where larger filters may be more appropriate such as image segmentation, where larger filters can be used to capture more context and improve segmentation accuracy. The NCHWc algorithm can be used for large kernels and Zoper algorithm can be used for 3x3 kernels.

5 Conclusion

This paper presents a novel direct convolution algorithm that is optimized for memory efficiency and is specifically designed for executing convolutions on a CPU. The proposed algorithm is demonstrated to achieve superior performance in terms of both memory usage and latency when processing small input data, outperforming state-of-the-art approaches. The algorithm's ability to handle large volumes of small input data makes it well-suited for a range of use cases, including small object detection and recognition, OCR, and satellite image detection.

Furthermore, this research motivates the development of future deep neural network model inference engines that are capable of effectively identifying input data patterns. While the proposed algorithm presents significant improvements, it is important to note that there may be alternative solutions that offer similar or superior performance. The author has considered various ideas for potential future work but has not yet had the opportunity to implement or test them due to limited resources, thus inviting the reader to further explore this area of research.

References

Yunus Camgözlü and Yakup Kutlu. Analysis of filter size effect in deep learning. *CoRR*, abs/2101.01115, 2021. URL https://arxiv.org/abs/2101.01115.

Yu-Hui Chen, Raman Sarokin, Juhyun Lee, Jiuqiang Tang, Chuo-Ling Chang, Andrei Kulik, and Matthias Grundmann. Speed is all you need: On-device acceleration of large diffusion models via gpu-aware optimizations, 2023.

Minsik Cho and Daniel Brand. MEC: memory-efficient convolution for deep neural network. *CoRR*, abs/1706.06873, 2017. URL http://arxiv.org/abs/1706.06873.

Wu Fuheng, Davchev Ivan, and Qian Jun. Hiq: Ob-

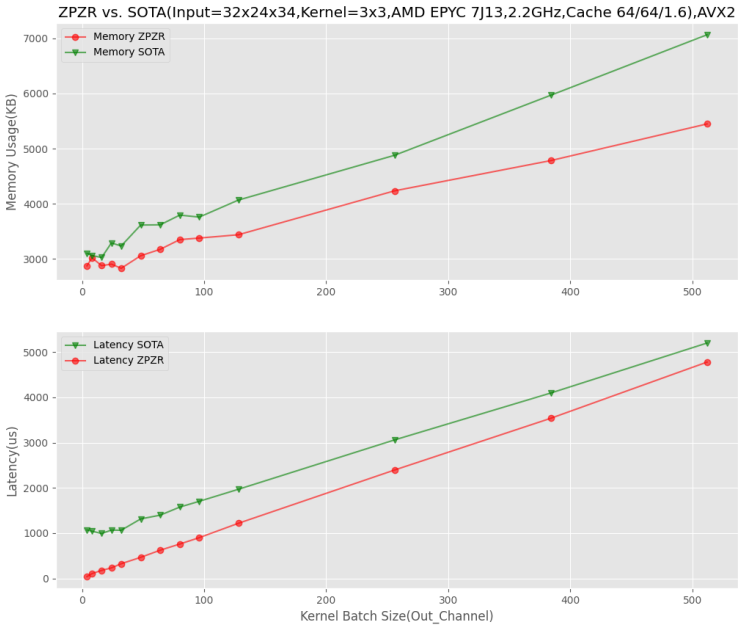

Fig. 4: Zoper Vs SOTA @ Input Channel 32

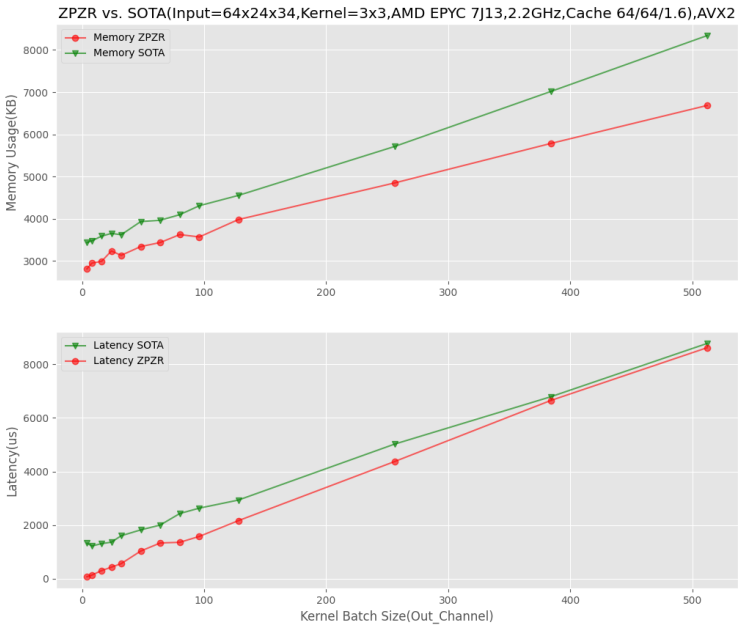

Fig. 5: Zoper Vs SOTA @ Input Channel 64

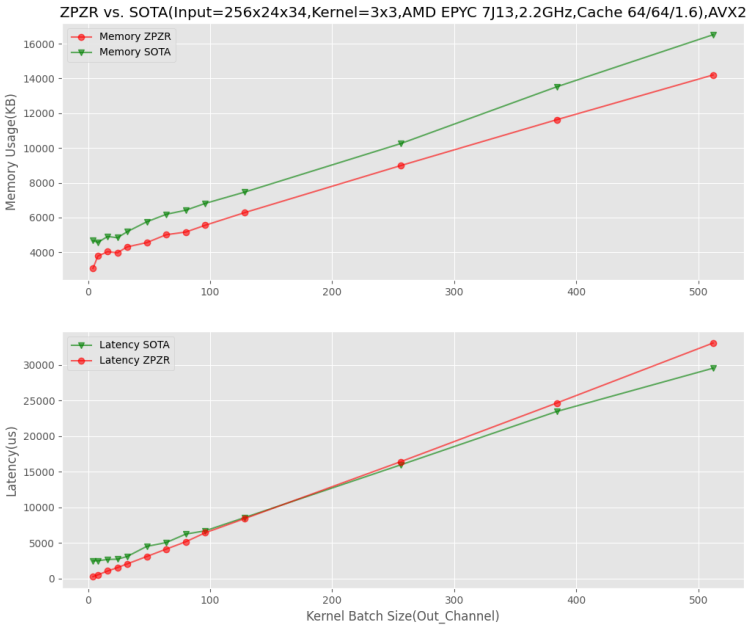

Fig. 6: Zoper Vs SOTA @ Input Channel 256

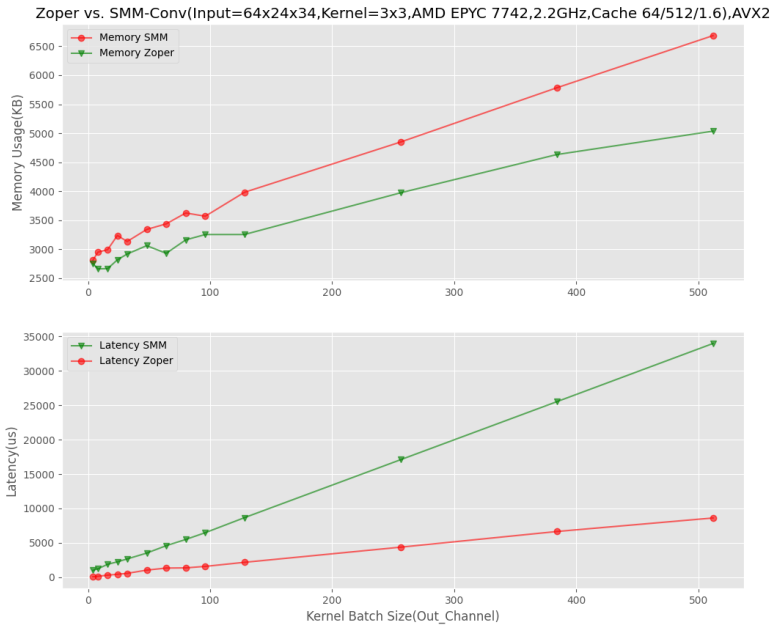

Fig. 7: Zoper Vs SMM-Conv @ Input Channel 64

servability and optimization in modern ai era, 2023. URL https://github.com/oracle/hiq.

Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/liu-yizhi.

Amir Ofir and Gil Ben-Artzi. Smm-conv: Scalar matrix multiplication with zero packing for accelerated convolution. In *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 3066–3074, 2022. doi: 10.1109/CVPRW56347.2022.00346.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

Jiyuan Zhang, Franz Franchetti, and Tze Meng Low. High performance zero-memory overhead direct convolutions. *CoRR*, abs/1809.10170, 2018. URL http://arxiv.org/abs/1809.10170.