Fast and Scalable 2D Convolutions and Cross-correlations for Processing Image Databases and Videos on CPUs

Cesar Carranza*, Daniel Llamocca† and Marios Pattichis‡
*Sección Electricidad y Electrónica, Departamento de Ingeniería
Pontificia Universidad Católica del Perú, Lima-32, Perú. Email: acarran@pucp.edu.pe
†Electrical and Computer Engineering Department
Oakland University, Rochester, MI, USA. Email: llamocca@oakland.edu
‡Department of Electrical and Computer Engineering, and
Center for Collaborative Research and Community Engagement, College of Education
The University of New Mexico, Albuquerque, NM, USA. Email: pattichi@unm.edu

***Abstract*—The dominant use of Convolutional Neural Networks (CNNs) in several image and video analysis tasks necessitates a careful re-evaluation of the underlying software libraries for computing them for large-scale image and video databases. We focus our attention on developing methods that can be applied to large image databases or videos of large image sizes.**

We develop a method that maximizes throughput through the use of vector-based memory I/O and optimized 2D FFT libraries that run on all available physical cores. We also show how to decompose arbitrarily large images into smaller, optimal blocks that can be effectively processed through the use of overlap-and-add. Our approach outperforms Tensorflow for 5×5 kernels and significantly outperforms Tensorflow for 11×11 kernels.

***Keywords*-Convolution, Parallel Processing, CPU, SIMD, MIMD, Image Processing.**

I. Introduction

Convolutions and cross-correlations dominate image processing operations. In implementing convolutional neural networks (CNNs), it is estimated that the computations associated with the convolutional layers account for about 90% of the total computations [1]. For neural-network training on large image databases or digital videos, there is clearly a need to compute 2D convolutions and cross-correlations efficiently. Clearly though, several image and video processing operations can benefit from fast convolutions and cross-correlations [2], [3].

Over the years, there has been a large body of research on how to compute 2D convolutions and cross-correlations effectively. We provide a short summary and then motivate the approach by the current paper. For small kernels, we have the computation of fast convolution using a sliding window [4], the use of separable convolutions [5], and the separable approximation of non-separable kernels [6]. Naturally though, the use of the transform-domain methods is needed for larger kernels. Over sufficiently large kernels, we expect the 2D FFT-based methods to dominate (e.g., [7]). Yet, the use of the 2D FFT requires complex-valued floating point arithmetic. Alternatively, 2D convolutions and cross-correlations can be computed very fast in hardware using the 2D Discrete Periodic Radon Transform (DPRT) [6]. In [6], it was shown that 2D convolutions of images and kernels of size $(N/2) \times (N/2)$ can be computed in-parallel as fast as $6N + 5n + 17$ clock cycles ($n = \text{ceiling}(\log_2(N))$) based on the fast DPRT architecture developed in [8].

Without access to custom hardware implementations, there is still a great need for computing fast convolutions and cross-correlations on modern CPUs. The obvious advantage of CPU methods comes from the fact that they provide access to relatively low-cost large-scale memory for training on large image formats. In other words, while CPUs can readily access 32GB or 64GB of the main memory, large amounts of memory remain prohibitively expensive for GPUs and FPGAs. Furthermore, it is important to recognize that we have advanced 2D FFT libraries that take advantage of multiple physical cores, SIMD instructions, as well as the conjugate-symmetry for real-valued images and videos (e.g., [9], [10]). On the other hand, for small convolution kernels, we also have the development of the Tensorflow libraries that are widely used for training deep learning systems [11].

We seek to develop a fast and scalable system that can be used to compute convolutions and cross-correlations for large image or video databases of arbitrarily large image sizes. More specifically, we develop a system that can provide sustainable high-throughput performance for reasonably large convolution kernels and large images. Naturally, we need to deal with moving large images effectively through the memory hierarchy through the use of vector-based operations. To deal with the need to process arbitrary sizes without causing cache misses, we propose the use of an overlap-and-add approach that fits optimal image blocks into the local memory. Furthermore, for fast I/O, we use high-performance vector copy operations that move image blocks that move images through local memory. We also integrate our approach with existing high-performance 2D FFT libaries to deliver fast throughputs. Our method is general and avoids low-level hardware dependencies by abstracting the necessary vector operations that should be available on all candidate CPUs.

We compare our approach against the spatial-convolution/cross-correlation methods available through Tensorflow 2.0. Our results clearly demonstrate that we can reach very high performance for arbitrary kernels and images, significantly outperforming Tensorflow 2.0 for kernels that are as small as 5×5.

The rest of the paper is organized as follows. We provide a general framework of our method in section II. We present details on our implementation and comparative results in section III. Conclusions and future work are given in section IV.

II. Methodology

We summarize our approach into three sections. First, we provide a description of optimal convolution using 2D FFTs in section II-A. Second, we summarize overlap-and-add in section II-B. Performance optimization is described in section II-C.

A. Basic Convolution Block

We present a general framework for computing convolutions in Fig. 1. The basic algorithm can be used for computing convolutions of entire images or use overlap and add to compute convolutions over large images with small kernels. The algorithm computes $f = g * h$ using 2D FFTs. The approach expands the standard use of 2D FFTs for computing convolutions of image sequences of arbitrary sizes on modern architectures. To support wider applications, we focus our description on describing the basic architectural elements needed to support higher performance.

We begin with memory allocation. Initially, as given in lines 2-7, we need to pre-allocate memory for f, g, h, H. The reserved memory size $N \times N$ needs to satisfy $N \geq P + Q - 1$ and allow for proper memory alignment. Thus, in practice, the pre-allocated memory will need to be larger than the minimum required for computing linear convolutions. Large image sizes can lead to diminished performance since the pre-allocated memory will no longer fit in the local cache. In our framework, such issues are captured during the size optimization process (in section II-C). For images (or image blocks), in addition to size requirements and memory alignment, we also require conversion from 8-bit fixed point to floating-point. Conversion, zero-padding, and memory copy are performed using vector operations as highlighted in line 8.

We assume the use of pre-optimized 2D FFT functions. The underlying 2D forward and inverse FFTs are assumed to: (i) use conjugate symmetry for real-valued images and kernels, (ii) utilize the use of multiple cores by allowing us to launch multiple threads, and (ii) use the underlying SIMD architecture within each thread for optimized computation. We also assume vector operations for multiplying the 2D FFTs. After applying the inverse 2D FFT to the product, we use a vector-crop operation to recover the result in its required size (see line 12).

```
 1: function FastParConv($f, g, h, N$, init)
    ▷ Computes full linear 2D convolution of g and h.
    ▷ Inputs:
    ▷   $g$: image (or image block) of size $P \times P$.
    ▷   $h$: convolution kernel of size $Q \times Q$.
    ▷   $N$: output size ($N \geq P + Q - 1$).
    ▷   init: if True initializes variables in memory.
    ▷ Outputs:
    ▷   $f$: linear convolution output ($f = g * h$).

 2:     if init then
 3:         Allocate $f, g, h, H$ with optimal padding
 4:                 and memory alignment.
 5:         $h \leftarrow$ VectorCopyZeroPad($h$)
 6:         $H \leftarrow$ FastParDFT ($h$, MaxPhysicalCores)
 7:     end if

    ▷ Convert 8-bit image (or image block) $g$ into
    ▷ floating-point and move into pre-alloc. memory:
 8:     $g \leftarrow$ ConvertZeroPadCopy ($g$)

 9:     $G \leftarrow$ FastParDFT ($g$, MaxPhysicalCores)
10:     $F \leftarrow$ VectorPoint2PointMult ($G$, $H$)
11:     $f \leftarrow$ FastParInvDFT ($F$, MaxPhysicalCores)

    ▷ Remove extra padding used for memory alignment:
12:     $f \leftarrow$ VectorCrop($f$)
13: end function
```

Fig. 1: Basic algorithm for fast and parallel convolutions on CPUs.

B. Overlap and Add Convolutions

The use of overlap-and-add allows us to use the 2D FFT for large images. The basic idea is to decompose a large image into a small number of blocks that can fit into cache memory and also allow us to use smaller sized FFTs per block. For each block, we compute 2D convolutions as discussed in section II-A. We then combine the results from neighboring blocks to produce the full 2D convolution results (e.g., see [12]).

C. Performance Optimization

The optimization method seeks to determine the optimal image block size that provides the maximum throughput for overlap-and-add. We refer to Fig. 2 for the optimization algorithm.

Initially, we allocate random matrices for the image and the kernel. Here, we are assuming that both the image and the kernel are square for simplifying the description. Then, we time the performance of different block sizes. Yet, since the kernel is significantly smaller than the full image size ($n \ll N$), we ignore the overhead associated with adding

```
1:  function OptBlockSize($N, n$)
      ▷ Algorithm computes image block size for
      ▷  optimal throughput.
      ▷ Inputs:
      ▷   $N$: input image frame size.
      ▷   $n$: kernel size. Assume that: $n \ll N$.
      ▷ Outputs:
      ▷   $n_{opt}$: optimal block size.

      ▷ Use random values for optimization:
2:      $g \leftarrow$ GenRandomMatrix($N$).
3:      $h \leftarrow$ GenRandomMatrix($n$).

      ▷ Search for the optimal block size:
4:      $p_{\max} \leftarrow 0$.
5:      $n_{\max} \leftarrow$ NextPowerOf2($N + n - 1$)
6:      for $i = n$ to $n_{\max}$ do
7:          $r(i) \leftarrow$ Time (FastCpuConv($f, g, h, n + i - 1$))
8:          $p(i) \leftarrow i^2 / r(i)$
9:          if $p_{\max} < p(i)$ then
10:             $n_{\text{opt}} \leftarrow i$
11:             $p_{\max} \leftarrow p(i)$
12:         end if
13:     end for
14:     return $n_{\text{opt}}$
15: end function
```

Fig. 2: Algorithm for computing the optimal image block size for maximal throughput.

the contributions from the neighboring blocks (number of additions is relatively small).

Our approach recognizes that modern 2D FFT algorithms have been developed to cover several possible image sizes. Beyond powers of two, we have libraries that can handle sizes $p_1^{i_1} p_2^{i_2} \ldots p_m^{i_m}$ where $p_1, p_2, \ldots, p_m$ are prime and $i_1, i_2, \ldots, i_m$ are non-negative integers. In what follows, we will assume that the underlying architecture supports $p_1 = 2$ at-least. Then, we search for the optimal size between the Kernel size, for minimum blocks of 1 pixel, and the next available power of two (line 5), for the maximum possible block size.

We define throughput as the rate of the number of processed frame pixels per second, for a constant kernel size. We thus seek to find the optimal block size n_{opt} so that the throughput function $p(.)$ is maximized:

$$\max_{n \leq i \leq n_{\max}} p(i).$$

III. Implementation and Results

A. Implementation Details

We test our proposed method on two CPUs: (i) a standard mobile CPU: Intel i7-4710MQ@2.5GHz with 4 physical cores, 256KB for L1, 1MB for L2, 6MB for L3, and 16GB for the main memory, and (ii) a server CPU: Intel Xeon E5-2630v3@3.2GHz with 8 physical cores, 512K for L1, 2MB for L2, 20MB for L3, and 64GB ofor the main memory. For the software, we used a mix of optimized code (C/C++) based on Intel Integrated performance libraries (IPP) and Intel Math Kernel Libraries (MKL).

For memory management, we used the IPP [13] to get support for SIMD-based vector operations. Each array was aligned on a 64-byte boundary and their sizes were based on the optimal block sizes described in section II-C. Each image was converted to floating point and copied to memory using `ippiScale_8u32f_C1R`. Cropping used `ippiCopy_32f_C1R` while the additions for overlap and add were performed using `ippiAdd_32f_C1R`.

For the actual 2D convolution/cross-correlation, we used the MKL Fast Fourier Transform libraries [10], that we believe to provide the best performance on Intel processors. To accelerate the computation we use the Conjugate Even symmetry property that speeds up the computation and reduces the memory storage in half. Also, we used sizes that are composite numbers of the form $2^a \times 3^b \times 5^c$, $a, b, c \geq 0$. For the point to point complex multiplications we used `vcMul`.

B. Results

We provide running time results in Fig. 3 and throughput results in Fig. 4. We present results for blocks of sizes 1×1 up to 4096×4096.

In terms of the throughput, we note a rise from small convolution kernels to larger kernels, followed by a drop in performance. Upon careful examination, we determined that performance drops are associated with 2D FFT algorithms associated with sizes expressed as $3^a 5^b$ where $a, b \geq 0$. In other words, the parallel 2D FFT library performs significantly better for sizes of $2^i 3^b 5^c$ with $i > 0$ versus sizes with $i = 0$. For small sizes ($N < 640$), the total running time is dominated by CPU I/O. In terms of the optimal block-size, we reach peak performance for 512×512 blocks for i7 and 1024×1024 blocks for the Xeon. Beyond these sizes, cache size limitations seem to impact the performance. Approximately, the Xeon is performing at twice the speed of the i7. Here, we note that the Xeon has ten physical cores versus four for the i7.

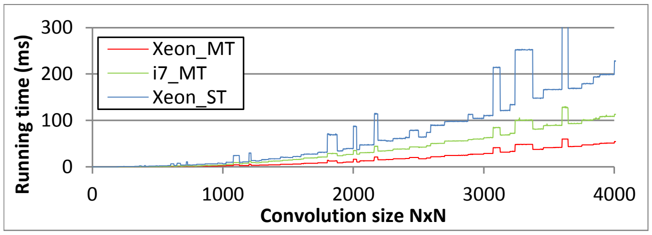

Fig. 3: Running time for 2D-convolution/crosscorrelation. Xeon_MT refers to the use of a parallel FFT using a thread per physical core. Xeon_ST refers to the use of FFT running on a single thread. Similarly, i7_MT refers to the use of multiple threads on the i7.

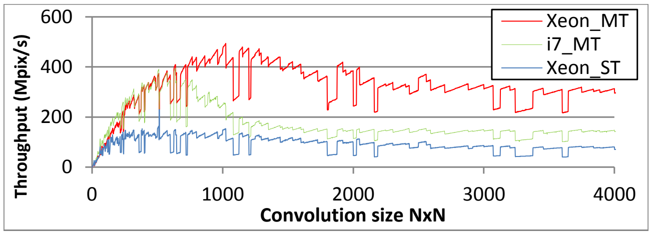

Fig. 4: Throughput results that correspond to the running times given in Fig. 3.

C. *Real-time video processing using overlap-and-add*

We present comparative results for real-time video processing in table I. In table I, we report the output convolved image size in the left column and the convolution kernel size in the right column (as the value of n for $n \times n$ kernels). In the last three rows, we also report results with Tensorflow 2.0. In Table I, we note that processing at 60 fps requires a maximum of 16.7 ms per frame, while 30 fps requires only 33.3 ms per frame.

From the results, it is clear that our proposed approach outperforms Tensorflow for 5×5 kernels and significantly outperforms Tensorflow for 11×11 kernels. In fact, Tensorflow only does better when processing filterbanks of a minimum of eight 3×3 filters.

IV. Conclusions and Future work

We developed an optimal approach for computing convolutions and cross-correlations of large databases of images that can be arbitrarily large. Our approach maximimizes throughput by breaking each image into optimal blocks and then using overlap-and-add to compute the final result. Over each block, we apply a parallelized 2D FFT that runs a thread for each physical core. Our approach significantly outperforms Tensorflow 2.0 for kernels as small as 5×5.

TABLE I: Running times (RT) for real-time time video processing at 30 or 60 fps. For overlap-and-add, the optimal block sizes were found to be: 1024×1024 for Xeon and 512×512 for i7. Each convolution kernel is of size $n \times n$ where n is given in the last column. We use SB for processing images using a single block and MB processing each image using multiple blocks. TF refers to results using Tensor Flow v2.0 [11].

Output Size	RT (ms)	Mpix/s	CPU	Notes
3240 × 3240	33.50	313.36	Xeon	SB, n=11
2160 × 2160	30.92	150.89	i7	SB, n=11
2304 × 2304	15.68	338.55	Xeon	SB, n=11
1600 × 1600	16.10	159.01	i7	SB, n=11
4066 × 2038	17.68	468.69	Xeon	MB, 8 blocks,n=11
4016 × 2008	22.24	362.60	i7	MB, 32 Blocks,n=11
4096 × 2048	15.92	526.53	Xeon	TF, n=3, 8 out channels
4096 × 2048	22.80	367.85	Xeon	TF, n=5, 8 out channels
4096 × 2048	281.92	29.75	Xeon	TF, n=11, 8 out channels
4096 × 2048	214.88	38.98	Xeon	TF, n=3, 1 out channel

V. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1842220.

References

[1] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.

[2] A. C. Bovik, *The essential guide to image processing*. Academic Press, 2009.

[3] ——, *The essential guide to video processing*. Academic Press, 2009.

[4] Y. Dong, Y. Dou, and J. Zhou, “Optimized generation of memory structure in compiling window operations onto reconfigurable hardware,” in *International Workshop on Applied Reconfigurable Computing*. Springer, 2007, pp. 110–121.

[5] D. Mukherjee and S. Mukhopadhyay, “Fast hardware architecture for 2-d separable convolution operations,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 12, pp. 2042–2046, 2018.

[6] C. Carranza, D. Llamocca, and M. Pattichis, “Fast 2d convolutions and cross-correlations using scalable architectures,” *IEEE Transactions on Image Processing*, vol. 26, no. 5, pp. 2230–2245, 2017.

[7] K. R. Rao, D. N. Kim, and J. J. Hwang, *Fast Fourier transform-algorithms and applications*. Springer Science & Business Media, 2011.

[8] C. Carranza, D. Llamocca, and M. Pattichis, “Fast and scalable computation of the forward and inverse discrete periodic radon transform,” *IEEE Transactions on Image Processing*, vol. 25, no. 1, pp. 119–133, 2015.

[9] M. Frigo and S. G. Johnson, “The design and implementation of fftw3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

[10] “Developer Reference for Intel® Math Kernel Library - C.” [Online]. Available: https://software.intel.com/en-us/mkl-developer-reference-c

[11] “Tensorflow website,” https://www.tensorflow.org/, accessed: 2019-12-01.

[12] J. S. Lim, *Two-dimensional signal and image processing*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[13] “Developer Reference for Intel® Integrated Performance Primitives (Intel® IPP) 2019 - Volume 2: Image Processing — Intel® Software.” [Online]. Available: https://software.intel.com/en-us/download/developer-reference-for-intel-integrated-performance-primitives-intel-ipp-2019-volume-2