High Performance Implementation of 2D Convolution using Intel's Advanced Vector Extensions

Hossein Amiri
Department of Computer Engineering
Faculty of Engineering
University of Guilan
Rasht, Iran
Email: sinusee@gmail.com

Asadollah Shahbahrami
Department of Computer Engineering
Faculty of Engineering
University of Guilan
Rasht, Iran
Email: shahbahrami@guilan.ac.ir

Abstract—**Convolution is the most important and fundamental concept in multimedia processing. For example, for digital image processing 2D convolution is used for different filtering operations. It has many mathematical operations and is performed on all image pixels. Therefore, it is almost a compute-intensive kernel. In order to improve its performance in this paper, we apply two approaches to vectorize it, broadcasting of coefficients and repetition of coefficients using Intrinsic Programming Model (IPM) and AVX technology. Our experimental results on an Intel Skylake microarchitecture show that the performance of broadcasting of coefficients is much higher than repetition of coefficients for different filter sizes and different image sizes. In addition, in order to evaluate the performance of Compiler Automatic Vectorization (CAV), and OpenCV library for this kernel, we use GCC and LLVM compilers. Our experimental results show that the performance of both IPM implementations are faster than GCC's and LLVM auto-vectorizations.**

Keywords- Parallel Programming; 2D convolution; Vectorization; SIMDization; AVX2

I. INTRODUCTION

Convolution is one of the most important operations in signal and multimedia processing. The 2D convolution kernel is widely used for filtering such as sharpening, smoothing, and edge detection and it is almost computationally intensive [1, 2, 11-14]. Therefore, an efficient and appropriate implementations for fast filtering is very important [3]. For efficient using of available hardware resources, the designers need a deep understanding of Digital Signal Processor (DSP) algorithms as well as a complete knowledge of the capabilities of the specific microarchitecture [21, 22]. Multimedia extensions such as MMX, SSE, and Advanced Vector Extensions (AVX) are a common approach for dealing with the performance requirements of multimedia applications on General-Purpose Processors (GPPs) [22]. These instruction set extensions provide instructions that process several short elements or subwords packed in a register in parallel in a Single Instruction and Multiple Data (SIMD) fashion. In other words, they provide short vector instructions to process 64-bit, 128-bit, and 256-bit using MMX, SSE and AVX simultaneously.

Intel introduced AVX extensions in 2011 [4]. The AVX is widely used in high performance compilers and libraries [5]. In 2013, AVX2 has been introduced too [6]. The AVX2 exploits SIMD concept for fine-grain parallelism. The majority of the fixed-point arithmetic instructions in AVX2 provide greater performance gain on vertical data processing compared to horizontal processing for parallel data elements. The focus of AVX2 is utilizing integer operations which can be used for vectorizing the multimedia processing algorithms on wide vector registers. However, vectorization is a challenging task.

In order to exploit the AVX2 SIMD instructions, modern compilers such as Gnu Compiler Collection (GCC) and Low Level Virtual Machine (LLVM) infrastructure can utilize the SIMD instructions automatically. On the other hand, Intrinsic Programming Model (IPM) has been developed for explicit vectorization, however, it needs so much programming effort compared to Compiler's Automatic Vectorization (CAV). Furthermore, Open Computer Vision (OpenCV) library is widely used for image processing and computer vision [7]. Hence, a comparative study of these tools is needed. With this insights, our contributions are as follows.

- In order to vectorize the 2D convolution kernel of size 3×3, 5×5, 7×7, and 9×9, two approaches are proposed. Broadcasting of coefficients and repetition of coefficients. These approached have been vectorized using AVX2 technology in IPM. Our experimental results show that the performance of the broadcasting of coefficients is much higher than repetition of coefficients.
- The performance of explicit vectorization is compared to implicit vectorization for 2D convolution kernel for different kernel and image sizes. Our experimental results show that the performance of explicit vectorization is much higher than implicit vectorization.

Our experimental results show that the IPM yields speedups up to 17x over scalar version, but, vertical operations provide much better performance compared to horizontal operations. On the other hand, the alternative tools did not improve the performance significantly such as speedups ranging from 1x to 4.55x, 0.45x to 1.14x, and 1.64x to 4.08x for GCC, LLVM, and OpenCV, respectively. The LLVM CAV ruined the performance for large kernel sizes while GCC CAV gained performance. For hand-unrolled implementations both LLVM and GCC vectorized the algorithm for large kernels. However, the LLVM yielded speedups up to 2.46x over GCC CAV.

This paper is organized as follows. In section 2 background information and related works are described. Our implementations have been explained in section 3 and results are shown in section 4. Finally, our conclusions are drawn in Section 5.

978-1-5386-2585-9/17/$31.00 ©2017 IEEE

II. BACKGROUND INFORMATION AND RELATED WORKS

In this section, we briefly describe convolution operation and related works.

A. Convolution Algorithm

The convolution operation is widely used in image and signal processing [1, 8, 9]. A naive convolution is fairly simple to implement [10]. Its concept is related to matrix multiplication which is computationally expensive, especially a two-dimensional (2D) convolution. In some filters such as Gaussian and Sobel, the kernel can be separated and 2D convolution can be performed as two 1D convolutions which is more cost effective [2]. For both separable and inseparable, 2D convolution can be used. The inputs of the 2D convolution algorithm include an image and filter coefficients which is depicted in Fig. 1. The kernel or windows size is related to the filter and image properties. The most used kernel sizes are 3×3, 5×5, 7×7, and 9×9 in real multimedia processing applications.

A PxQ kernel				An NxM image							
C00	C01	C02	...	X00	X01	X02	...	X0(15)	X0(16)	X0(17)	...
C10	C11	C12	...	X10	X11	X12	...	X1(15)	X1(16)	X1(17)	...
C20	C21	C22	...	X20	X21	X22	...	X2(15)	X2(16)	X2(17)	...
⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋯	⋮	⋮	⋮	⋱

Figure 1. Coefficients and input data of 2D convolution algorithm

$$Y(i,j)=\sum_{k=0}^{P}\sum_{l=0}^{Q}C(k,l)*X(i,j) \qquad (1)$$
$$0<i<N,\ 0<j<M$$

$$Y(i,j)=\sum_{k=0}^{P*Q}C(k/Q,k\%Q)*X\begin{pmatrix}i-(P/2)+(k/Q),\\ j-(Q/2)+(k\%Q)\end{pmatrix} \qquad (2)$$
$$0<i<N,\ 0<j<M$$

Equation (1) shows the 2D convolution operation for a filter of size P×Q which is performed on the input image of X for computing a single output element of Y. Correspondingly, each output pixel needs P * Q multiplications that means for an image of size N × M, number of multiplications is N * M * P * Q. Equation (1) can be transformed to equation (2). It performs the same operations which we used for unrolling the algorithm. Each gray scale pixel can be filled with a number between zero and 255 [13]. It is common to use 16-bit elements for each pixel that provides 16-way parallelism [15]. There are many 16-bit instructions in AVX2 instruction set architecture that can be used in this case, such as "*vpmullw*" and "*vpaddw*". These instructions can operate on 16×16-bit elements simultaneously [16, 17]. The latency of these instructions is less than accessing to the first level of the data (L1D) cache, however, loading a vector from the L1D cache might be slightly slower than scalar loading [18], but its 16-way parallelism is cost effective for implementations.

B. Related Works

In order to provide a fast 2D convolution, instruction-level parallelism, thread-level parallelism, and data-level parallelism has been used on a single core of the processor, particularly for Sobel filter [19]. For utilizing the AVX technology, gcc in -O3 optimization level was used. In [20], a clear vision of 2D convolution theory was explained. Execution times show that for large matrices this operation is very expensive. However, Open Computing Language (OpenCL) and Message Passing Interface (MPI) for parallelization have been used. A convolution engine for fast convolution has been used in [21]. The algorithm broadcasts each coefficient to a vector register. Then, coefficients are multiplied to input data to produce the output pixel. One of the oldest studies on 2D convolution can be found in [11] which uses multiply and accumulate instructions for SIMDizing.

III. VECTORIZATION OF 2D CONVOLUTION

In order to vectorize the 2D convolution kernel, two approaches, broadcasting of coefficient and repetition of coefficient have been proposed which their data flow graphs are depicted in Fig. 2 and Fig. 3, respectively. For computations, we broadcast the coefficients to a vector register. Then, a vertical multiplications by pixels and adding the results, produce 16 convoluted outputs. The data flow graph of this version for a 3×3 kernel is depicted in Fig. 2. It reduced 144 iterations of the naive implementation to one iteration.

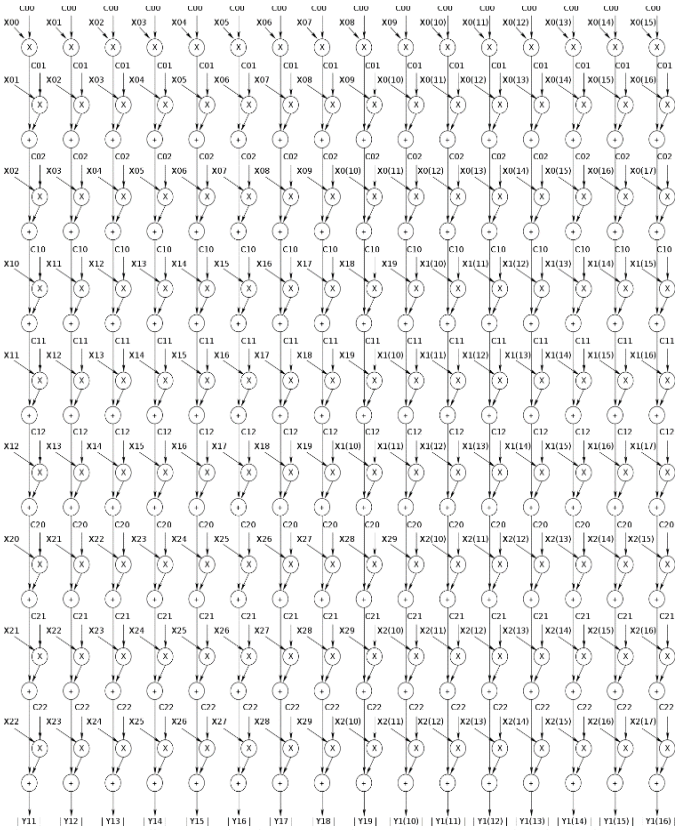

Figure 2. Data flow graph of vectorization of 2D convolution kernel for a 3 x 3 size using broadcasting of coefficients into a vector register and multiply them with image pixels.

In the second approach, all coefficients which are located in a row, for example C00, C01, and C02 for a filter size of 3 x 3 are loaded and repeated into a vector register. Other coefficients are also loaded and repeated into some vector registers. These vector registers are multiplied with image row pixels and intermediate results are added to each other as can be seen in Fig. 3.

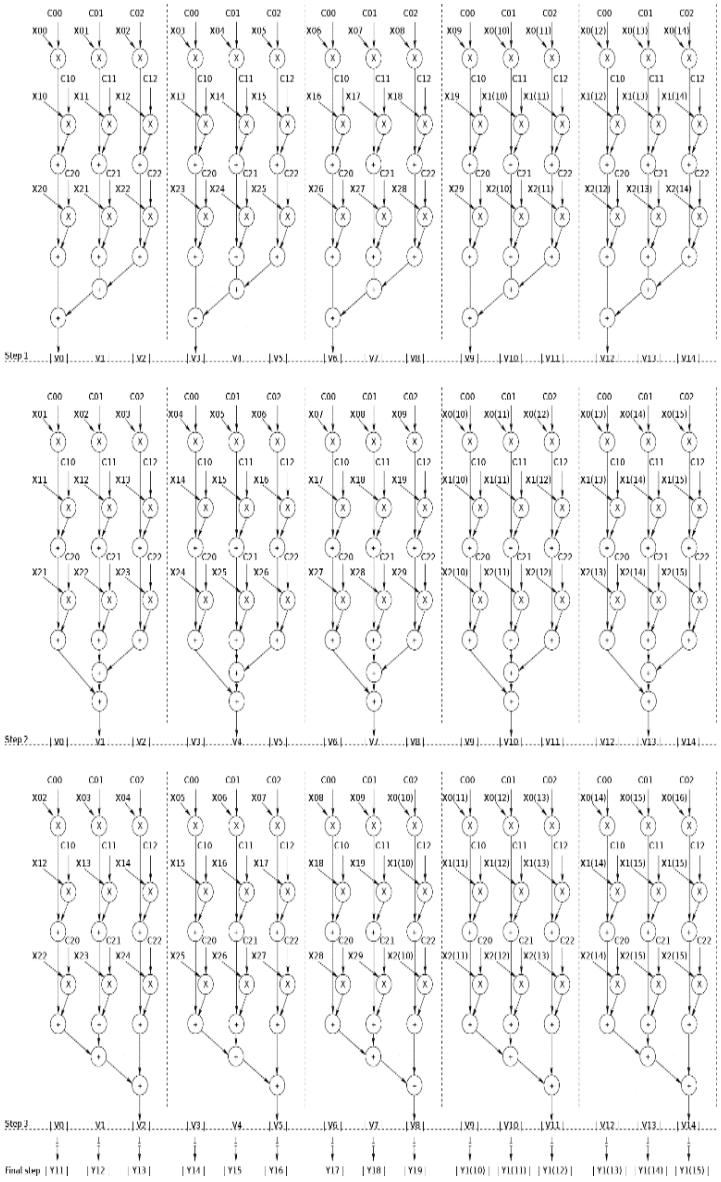

Figure 3. Data flow graph of vectorization of 2D convolution kernel for a 3 x 3 size using repeatation of coefficients into vector registers and multiply them with image pixels.

The input pixels and coefficients are multiplied through a group of packed multiplication instruction such as "vpmullw". As can be seen in Fig. 3, in three steps 15 output pixels are computed. Because of this transformation, it computes 135 iterations of the naive implementation in one iteration.

IV. Performance Evaluation

In this section, we present the platform configuration and experimental results.

A. Environment Setup and Vectorization Tools

Our platform is based on a 2.60 GHz processor of Intel Corei7-6700 HQ with three level of data cache as depicted in Table 1.

TABLE I. PLATFORM SPECIFICATION [24]

Hardware	***CPU***		Intel Corei7- 6700HQ
	Micro-architecture		Skylake
	Register width		Maximum 256 bits
	RAM		12GB
	Cache	***Line size***	64 Bytes
		L1D	32KB, 8-way set associative, fastest latency: 4 cycles, 2×32B load + 1×32B store
		L2	256 KB,4-way set associative, fastest latency: 12 cycles
		L3	Up to 2 MB per core, up to 16-ways, fastest latency: 44 cycles
Software	***Operating system***		Linux Mint 18.1, 64-bit
	Compilers		GCC 6.2.0 and Clang-LLVM 3.8 -O3 optimization level
	Programming tools		Standard C, Open CV, and x86 Intrinsic (x86intrin.h)
	Disable vectorizing		-fno-tree-vectorize -fno-tree-slp-vectorize

In order to compare our implementations, we used GCC for scalar implementation. Both vectorization approaches, for exploiting the capabilities of AVX technology have been implemented using IPM which is called IPM-BC (Broadcasting-Coefficients) and IPM-RC (Repetition Coefficients) at -O3 optimization level while auto-vectorization was disabled by adding "-fno-tree-vectorize" and "-fno-tree-slp-vectorize" to the command line. The IPM approach is one of the most powerful programming models for explicit vectorization [28]. The IPM must be supported in compilers, moreover, compatibility and portability issues must be addressed [25]. For our implementations, we use GCC and LLVM in a particular general purpose processor, the IPM can be used without concerning about the mentioned issues.

For utilizing the CAV for implicit vectorization both GCC [26] and LLVM [27] compilers were used at -O3 optimization level and speedup was measured compared to scalar runtime of each compiler, separately. We also used "filter2D" function of Open CV library to compare our implementation to one of the most used image processing tools [23]. The kernel sizes of 3×3, 5×5, 7×7, and 9×9 have been used in our implementations. All implementations are available in [10].

For our implementation, we converted data flow graphs in Fig. 2 and 3 to Intrinsic functions. In order to warm up the cache and avoid context switching, we executed all programs many times and measured the smallest execution time. Speedup is used to draw the performance improvements which is the best achieved time ratio of a particular program. In addition, the input matrix is filled with real world gray-scale images data and

coefficients are set for different filters. Our experimental result is based on smoothing filters.

B. Experimental Results

Our experimental results are depicted in Figs. from 4 to 7. The speedups of different implementations, explicit vectorizations IPM-BC and IPM-RC, implicit vectorizations GCC and LLVM, and OpenCV over scalar implementation of 2D convolution kernel of size 3 x 3, 5 x 5, 7 x 7, 9 x 9 for different image sizes are depicted in Figure 4, 5, 6 and 7, respectively. As these figures show, the speedups of the explicit vectorization is much higher than other vectorizations and implementations approaches. In addition, the performance of broadcasting of coefficients, IPM-BC approach is much higher than repetition of coefficients, IPM-RC approach.

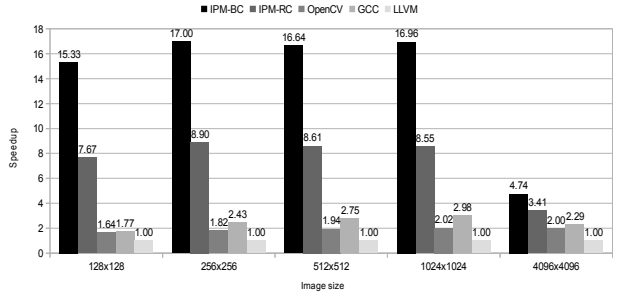

Figure 4. Speedups of different implementations, explicit vectorizations IPM-BC and IPM-RC, implicit vectorizations GCC and LLVM, and OpenCV over scalar implementation of 2D convolution kernel of size 3 x 3 for different image sizes.

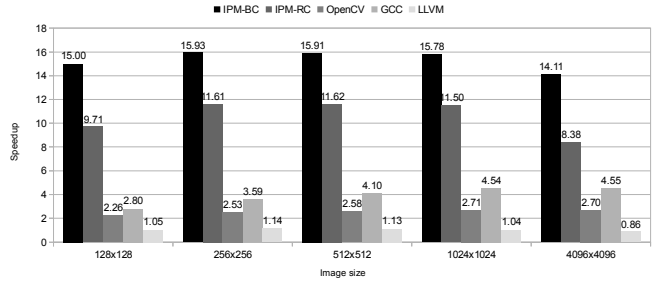

Figure 5. Speedups of different implementations, explicit vectorizations IPM-BC and IPM-RC, implicit vectorizations GCC and LLVM, and OpenCV over scalar implementation of 2D convolution kernel of size 5 x 5 for different image sizes.

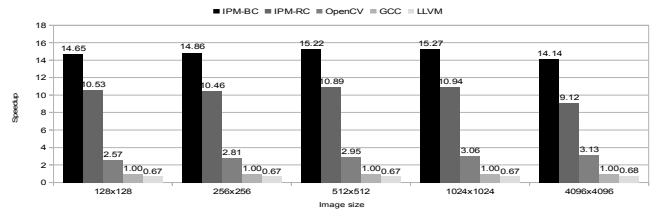

Figure 6. Speedups of different implementations, explicit vectorizations IPM-BC and IPM-RC, implicit vectorizations GCC and LLVM, and OpenCV over scalar implementation of 2D convolution kernel of size 7 x 7 for different image sizes.

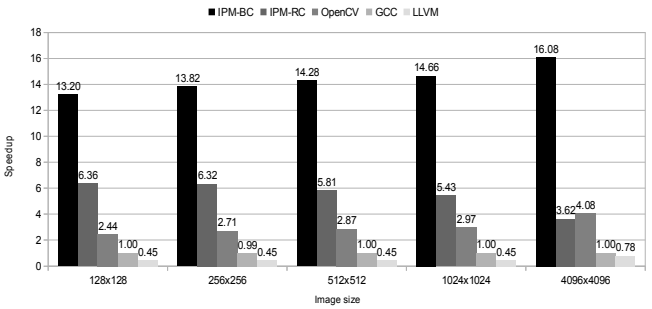

Figure 7. Speedups of different implementations, explicit vectorizations IPM-BC and IPM-RC, implicit vectorizations GCC and LLVM, and OpenCV over scalar implementation of 2D convolution kernel of size 9 x 9 for different image sizes.

The IPM-BC yields speedups mostly near to the vectorization factor of 16. It is because overhead instructions are reduced by broadcasting each coefficient to vector registers compared to IPM-RC approach which coefficients are repeated to vectors and in each step some horizontal addition, permutation, and shift instruction is used to place the output in the appropriate places.

As depicted in Figs. 4 and 5, GCC vectorization improves the performance significantly while LLVM's improvements are not countable. When looking at the generated assembly [10], it seems LLVM's auto-vectorizer generated some instructions working on 128-bit vectors which read coefficients and data inside the inner loop and insert them into the "xmm" registers, however, coefficients must be loaded to some vectors outside the inner loop. This is because LLVM does not gain performance while all kernels are vectorized.

As depicted in Figs. 6 and 7, for large kernels CAV, GCC and LLVM vectorization did not improve the performance, however, LLVM's auto-vectorizer ruined the performance for large kernels because of a large amount of insertion, shuffle and horizontal operations in the bottleneck section. The filter2D function of OpenCV library improved the operation significantly compared to CAVs. The IPM-BC yielded some speedups ranging from 13.20x to 16.08x, while IPM-RC restricted the performance for 9×9 kernel. This is because, each vector was filled with nine coefficients and rest of the vector was zeroed. Thus, it means for each iteration seven zeros are participated in computations. GCC auto-vectorizer did not generate SIMD instructions, and scalar codes were used.

According to our results in figures 6 and 7, the implicit vectorization using GCC and LLVM could not improve the performance of 7×7 and 9×9 kernels of naive implementation. We unrolled the inner loop for these kernel sizes and assessed the compiler's behavior and we obtained new results. The speedups of GCC and LLVM CAV over their scalar implementations with unrolling technique are depicted in Fig. 8. As can be seen, all speedups for different image sizes are more than 2. This means that both GCC and LLVM vectorized the unrolled version of 2D convolution algorithm for filter sizes of 7 x 7 and 9 x 9. In addition, LLVM improvement is significantly better than GCC. The speedup of LLVM over GCC for unrolled version is depicted in Fig. 9. The LLVM is much faster than GCC for unrolled version. This is because, LLVM's auto-vectorizer generate a "vpblendd" instruction to mask the results

in contrast to a bunch of extract instruction in GCC. Moreover, GCC generated assembly is 3x larger than LLVM for the 9×9 kernel which is fully unrolled.

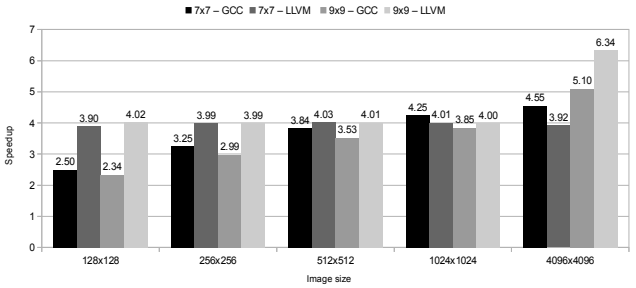

Figure 8. Speedups of implicit vectorizations GCC and LLVM, over their scalar implementation for 2D convolution kernel of size 7 x 7 and 9 x 9 using unrolling technique for different image sizes.

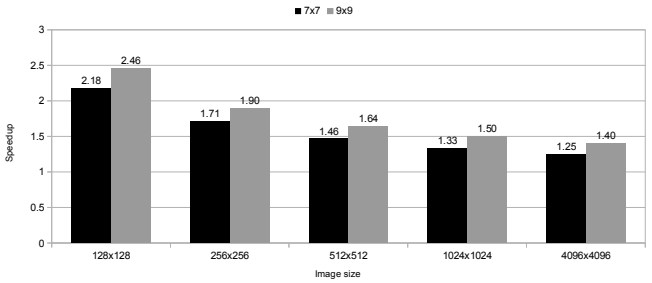

Figure 9. Speedups of LLVM auto-vectorization over GCC auto-vectorization for 2D convolution kernel of sizes 7 x 7 and 9 x 9 using unrolling technique for different image sizes.

V. CONCLUSIONS

The 2D convolution algorithm is an important kernel in many multimedia applications and it is almost computational intensive. In order to improve its performance on general-purpose processors enhanced with multimedia extension, Single Instruction and Multiple Data (SIMD), we have vectorized it using both explicit and implicit vectorization using AVX technology which is a new SIMD extension in Intel architecture. For vectorization, two approaches, broadcasting of coefficients and repetition of coefficients have been proposed. In the first approach, a coefficient is broadcasted into all subwords of a vector register while in the second approach all coefficients in a row of a filter size are repeated in all subwords of a vector register. For explicit vectorization, we have used Intrinsic Programming Model (IPM), while for implicit vectorization we have used auto-vectorization of Gnu Compiler Collection (GCC) and Low Level Virtual Machine (LLVM) at -O3 optimization level. Moreover, OpenCV, one of the most used image processing library been also used. We evaluated different implementations of 2D convolution for 3×3, 5×5, 7×7, and 9×9 kernel sizes. The IPM version with Broadcasting of Coefficients (IPM-BC) and Repetition of Coefficients (IPM-RC) were implemented to exploit 16-way data-level parallelism. Our experimental results show that the performance of explicit vectorization is much higher than implicit vectorization. In addition, the performance of the IPM-BC is higher than IPM-RC. The CAV cannot vectorize the larger kernel sizes of 7 x 7 and 9 x 9. In order to vectorize these sizes, their code should be unrolled. Both GCC and LLVM vectorize the unrolled version and the LLVM vectorization is faster than GCC vectorization for large sizes such as 7 x 7 and 9 x 9.

REFERENCES

[1] Pavel K, David S (2013) Algorithms for Efficient Computation of Convolution. Design and Architectures for Digital Signal Processing 179–208. doi: 10.5772/51942

[2] Parker JR (2011) Algorithms for Image Processing and Computer Vision, 2end ed. Wiley

[3] Carranza C, Llamocca D, Pattichis M (2017) Fast 2D Convolutions and Cross-Correlations Using Scalable Architectures. IEEE Transactions on Image Processing PP:1–16. doi: 10.1109/TIP.2017.2678799

[4] Intel Corporation (2011) Intel Advanced Vector Extensions Programming Reference.

[5] Gepner P, Gamayunov V, Fraser DL (2011) Early performance evaluation of AVX for HPC. International Conference on Computational Science 4:452–460. doi: 10.1016/j.procs.2011.04.047

[6] Intel Corporation (2015) Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture. 1:5–28.

[7] OpenCV library. http://opencv.org/. Accessed 1 Apr 2017

[8] Chu N, Gac N, Picheral J, Mohammad-Djafari A (2014) 2D Convolution Model Using (IN)VARIANT Kernels for Fast Acoustic Imaging. In: Proceedings of the 5th Berlin Beamforming Conference. HAL, Berlin, Germany, pp 1–15

[9] Lian X (2017) Video and Image Analysis Using Local Information. Ph. D. dissertation, University of California

[10] Amiri Hossein (2017) Intel SIMD Technologies. In: github. https://github.com/sinusee/Intel-SIMD-Technologies. Accessed 18 Mar 2017

[11] Jamro E, Wiatr K (2001) Implementation of Convolution Operation on General Purpose Processors. Proceedings of 27th Euromicro Conference 2001: A Net Odyssey 410–417. doi: 10.1109/EURMIC.2001.952482

[12] Kyo S, Okazaki S, Kuroda I (2003) An Extended C Language and a SIMD Compiler for Efficient Implementation of Image Filters on Media Extended Micro-Processors. Proceedings of Advanced Concepts for Intelligent Vision Systems 234–241.

[13] Gonzalez RC, Woods RE (2002) Digital Image Processing, Second. Leonardo. doi: 10.2307/1574313

[14] Kim CG, Kim JG, Lee DH (2014) Optimizing Image Processing on Multi-core CPUs with Intel Parallel Programming Technologies. Multimedia Tools and Applications 68:237–251. doi: 10.1007/s11042-011-0906-y

[15] Nguyen H, John LK (1999) Exploiting SIMD parallelism in DSP and multimedia algorithms using the AltiVec technology. Proceedings of the 13th international conference on Supercomputing 11–20. doi: 10.1145/305138.305150

[16] Intel Corporation Intel Intrinsics Guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/. Accessed 29 Jan 2017

[17] Intel Corporation (2015) Intel 64 and IA-32 Architectures Software Developer's Manual.

[18] Anger Fog (2016) Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.

[19] Zekri AS (2016) Optimizing image spatial filtering on single CPU core. Multimedia Tools and Applications 1–31. doi: 10.1007/s11042-016-4266-5

[20] Pande A, Chandna R (2013) Matrix Convolution using Parallel Programming. International Journal of Science and Research 2:286–291.

[21] Qadeer W, Hameed R, Shacham O, Venkatesan P, Kozyrakis C, Horowitz MA (2013) Convolution Engine: Balancing Efficiency & Flexibility in Specialized Computing. Proceedings of the 40th Annual International Symposium on Computer Architecture 58:24–35. doi: 10.1145/2485922.2485925

[22] Shahbahrami A, Juurlink B, Vassiliadis S (2008) Implementing the 2-D Wavelet Transform on SIMD-Enhanced General-Purpose Processors. IEEE Transactions on Multimedia 10:43–51. doi: 10.1109/TMM.2007.911195

[23] OpenCV 3.2 - OpenCV library. http://opencv.org/opencv-3-2.html. Accessed 1 Apr 2017

[24] Intel Corporation (2016) Intel 64 and IA-32 Architectures Optimization Reference Manual. doi: 10.1535/itj.0903.05

[25] Pohl A, Cosenza B, Mesa MA, Chi CC, Juurlink B (2016) An Evaluation of Current SIMD Programming Models for C++. Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing 1–8. doi: 10.1145/2870650.2870653

[26] Free Software Foundation GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF). https://gcc.gnu.org/. Accessed 29 Jan 2017

[27] Clang clang: a C language family frontend for LLVM. https://clang.llvm.org/. Accessed 29 Jan 2017

[28] Hassan SA, Hemeida AM, Mahmoud MMM (2016) Performance Evaluation of Matrix-Matrix Multiplications Using Intel's Advanced Vector Extensions (AVX). Microprocessors and Microsystems 47:369–374. doi: 10.1016/j.micpro.2016.10.002