Improving the Addweighted Function in OpenCV 3.0 Using SSE and AVX Intrinsics

Panyayot Chaikan and Somsak Mitatha

***Abstract*—This paper presents a new algorithm for improving the speed of OpenCV's addWeighted function for blending images. We propose two implementations: one using the SSE (Streaming SIMD Extension), the other employing the AVX (Advanced Vector Extension), which increases the function's speed by 3.49x and 5.77x respectively. The multi-core version of our algorithm utilizes load balancing to distribute loads between user threads while keeping the correct memory alignment for each SIMD instruction type. This approach improves the function's speed by 23.08 times compared to its original implementation in the OpenCV library.**

***Index Terms*—Image blending, multicore programming, AVX.**

I. Introduction

Single-instruction-multiple-data (SIMD) processing is part of many computer architectures, such as Intel's SSE and AVX instructions [1], ARM's NEON [2], and the PowerPC's AltiVec [3], and can drastically increase computation speeds. For example, F. Gerneth proposed an implementation of the FIR filter using Intel SSE instructions [4]. J.Y . Liu *et al.* increased the performance of the Shallow Water equations and Euler equations by means of AVX and OpenMP [5]. J. Frances *et al.* speeded up Finite-Difference Time-Domain (FDTD) processing using AVX and OpenMP [6]. B. L. Gal *et al.* took advantage of AVX to map the successive cancellation (SC) decoding of polar codes [7]. C. C. Chi *et al.* utilized AVX instructions for accelerating High-Efficiency-Video-Coding (HEVC) decoding [8].

OpenCV 3.0 has the ability to utilize the SSE and AVX instruction sets [9], and some of its operations can run faster if properly vectorised. For example, G. Mitra et al. used SSE to augment the speed of OpenCV's image binarization and convolution operations [10]. We propose to increase the speed of OpenCV's add Weighted function by using SSE and AVX intrinsics, combined with the distribution of loads in a multi-core environment; speed-ups of more than 23 times are obtained.

The paper is organized as follows: Section II introduces the SIMD capabilities of the x86 architecture, and Section III describes OpenCV's addWeighted function. In Section IV, we present our proposed add-and-weight algorithm, and the load distribution mechanism for a multi-core architecture. Experimental results are given in Section V, and Section VI concludes the paper.

Manuscript received October 23, 2015; revised December 30, 2015.

Panyayot Chaikan is with the Department of Computer Engineering, Faculty of Engineering, Prince of Songkla University, Thailand (e-mail: panyayot@coe.psu.ac.th).

Somsak Mitatha is with the Department of Computer Engineering, Faculty of Engineering, King Mongkhut's Institute of Technology Ladkrabang, Thailand (e-mail: kmsomsak@kmitl.ac.th).

II. The SIMD SSE and AVX Instructions

The SSE is a SIMD instruction set designed by Intel as an extension to the traditional x86 architecture. Depending on the type and amount of data that the SSE registers support, it allows one instruction to operate on multiple data simultaneously. In the x86's 64-bit long mode, there are sixteen 128-bit SSE registers, named XMM0-XMM15 [1]. Each of these registers can hold i) two double precision floating point numbers; ii) four single precision floating point numbers; iii) sixteen 8-bit integers; iv) eight 16-bit integers; v) four 32-bit integers; or vi) two 64-bit integers. There are four main ways to utilize SSE and AVX instructions: i) write assembly code using these SIMD machine instructions; ii) use SSE or AVX inline assembly in C or C++ [11]; iii) use the compiler's intrinsic functions; or iv) use the compiler's support for auto-vectorization [10].

The syntax of the SSE intrinsic functions follow the pattern _mm_<*operation*>_ <*suffix*> [10], where the *operation* can be the load, store, arithmetic, or logical operations, and the *suffix* is the type of data used by the operation, e.g., *_mm_add_epi8* and *_mm_add_epi32* stand for adding 8-bit and 32-bit integers.

The AVX has sixteen 256-bit registers, called YMM0-YMM15, which are a superset of the XMM registers because each XMM is actually the lower-half of the corresponding YMM register. The syntax of the intrinsic functions for the AVX has the pattern __mm256_<*operation*>_<*suffix*>. Since the AVX doubles the size of the SSE registers, it is in theory two times faster than the SSE.

The Intel SSE operations employ a 16-byte memory alignment boundary so data must start at memory addresses divisible by 16. In the Intel AVX, this memory alignment requirement is relaxed so that data can be kept in unaligned addresses but with some performance reduction. For this reason, keeping data in 32-byte memory alignment is recommended for the best performance [1].

III. The Addweighted Function

The most prominent use of the addWeighted function is for blending images. If S_1, S_2 are the input images and D is the output image, then D's (x, y) pixel is calculated using:

$$D(x, y) = S_1(x, y) \times \alpha + S_2(x, y) \times \beta + \gamma \qquad (1)$$

DOI: 10.7763/IJET.2017.V9.943

where

α = a weight for the first image,
β = a weight for the second image,
γ = a scalar added to each sum [12].

IV. Our Proposed Algorithm

This section begins with our proposed algorithm to perform add-and-weight using SIMD implementations, then the load balancing algorithm for the multi-core implementation is presented.

A. Our SIMD Implementations of the Add-and-Weight Operation

Our add-and-weight function utilizes two SIMD instruction types: the SSE and AVX.

Our SSE algorithm starts by reading sixteen pixels from the source images S_1 and S_2 to the 128-bit XMM registers. The lower eight 8-bit pixels from each register are converted into eight 16-bit integers, using the *_mm_unpacklo_epi8* instruction, and these eight pixels from each image are multiplied by a pre-calculated α*128 and β*128 simultaneously. The two multiplicands are summed, and the result added to a pre-calculated γ*128. The results are divided by 128 by shifting their bits to the right by seven positions. These lower eight result pixels (in 16-bit integer form) are stored in a variable named D_1. The higher eight pixels from S_1 and S_2, that are already in the XMM registers, are converted into eight 16-bit integer by means of the *_mm_unpackhi_epi8* instruction, processed in the same way, and stored in D_2. The D_1 and D_2 variables are converted from 16-bit integers back to 8-bit integers using the *_mm_packus_epi16* function, and then the image's fully processed 16 pixels are stored in the destination image.

After all 16 pixels in this register have been manipulated, the process repeats by reading from S_1 and S_2. This goes on until there are less than 16 pixels left in the image, and the remainder is processed using conventional code.

Parallelism is achieved by utilizing SSE parallel integer operations: *_mm_add_epi16*, *_mm_mullo_epi16,* and *_mm_srli_epi16*, as shown in Fig. 1. The α, β and γ floating point values are multiplied by 128 before being converted to 16-bit integers to reduce the error of floating point to integer conversion.

The AVX implementation of our algorithm works in the same way as the SSE version, but the AVX register size is doubled and uses the type *__m256i*. The intrinsic functions that begin with *_mm* in the SSE are changed to *_mm256*, (e.g., the SSE *_mm_add_epi16* becomes *_mm256_add_epi16*).

We implemented our algorithm using integer calculations for two reasons. The first is that more integer operands can be computed simultaneously than floating point values in SIMD registers. For example, sixteen 16-bit integer operands can be executed simultaneously in the YMM registers but only eight single precision floating point numbers. The second reason is that integer operations have lower computational latency than floating point values [1].

B. Multi-core Programming for SIMD

Our load balancing mechanism tries to distribute equal loads to the cores while retaining the memory size alignment for each SIMD instruction. The algorithm assumes that the start of the memory storing the image pixels is aligned to the memory required by the SIMD instructions (divisible by 16 and 32 for SSE and AVX respectively).

Let "/" be an integer division operation, N_{thread} the number of threads that are running in the multi-core environment, and P_{total} the number of pixels in the image. If all the threads are assigned an equal load, then the number of pixels processed by each thread is

$$W_{balanced} = P_{total} / N_{thread}. \quad (2)$$

However, memory alignment is required for SIMD calculations, so we cannot directly use equation (2) to calculate the load for each thread. Let A be the number of bytes needed for memory alignment for each SIMD instruction (16 and 32 for SSE and AVX respectively), and let T_{id} be the ID of each thread ranging from 0 to N_{thread}-1, then the load of all the threads except the last one is

$$W_{non_last_thread} = W_{balanced} + A - (W_{balanced} \% A) \quad (3)$$

where "%" is a modulus operation. The load of the last thread ($T_{id} = N_{thread}$-1) is

$$W_{last_thread} = P_{total} - (N_{thread} - 1) * W_{non_last_thread}. \quad (4)$$

All the threads except the last one will use SIMD instructions to process their image pixels. For the last thread, the load involves a SIMD part and a FPU part. The SIMD work is determined by

$$W_{simd_last_thread} = \begin{cases} 0 & \text{if } (W_{last_thread} < A) \\ W_{last_thread} - (W_{last_thread} \% A) & \text{otherwise.} \end{cases} \quad (5)$$

The remaining pixels of the last thread that cannot be processed by SIMD instructions must be handled by FPU instructions. The amount of work is

$$W_{fpu_last_thread} = W_{last_thread} \% A. \quad (6)$$

Although each thread knows how many pixels it must process, each thread also needs to know the beginning address of its image pixels for performing the add-and-weight calculation. The starting memory address, called S_{start}, of thread ID = T_{id} will begin at

$$S_{start} = I_{start} + (T_{id} * W_{non_last_threads}) \quad (7)$$

where I_{start} stands for the starting memory address of the image pixels.

For the last thread, any pixels that must be processed by

the FPU start at location

$$S_{FPU_start} = S_{start} + W_{simd_last_thread}. \quad (8)$$

```
__m128i *pA, *pB, *pD;
__m128i A, B, D1, D2, alfa, beta, zero;
float a1, a2, a3;
unsigned short af1, af2, af3;
a1 = thread_params[thread_id].ulfa * 128.0f;
a2 = thread_params[thread_id].beta * 128.0f;
a3 = thread_params[thread_id].gamma*128.0f;
af1 = (unsigned short)a1;
af2 = (unsigned short)a2;
af3 = (unsigned short)a3;
alfa = _mm_set1_epi16(af1);
beta = _mm_set1_epi16(af2);
gamma= _mm_set1_epi16(af3);
zero = _mm_setzero_si128();

/* pA, pB, and point to the beginning memory address of the image
pixels in S1, S2, and D */

for (i = 0; i < N_loop; i++)
{
   A = _mm_unpacklo_epi8(*pA, zero);
   B = _mm_unpacklo_epi8(*pB, zero);
   A = _mm_mullo_epi16(A, alfa);
   B = _mm_mullo_epi16(B, beta);
   D1 = _mm_add_epi16(A, B);
   D1 = _mm_add_epi16(D1, gamma);
   D1 = _mm_srli_epi16(D1, 7);
   A = _mm_unpackhi_epi8(*pA, zero);
   B = _mm_unpackhi_epi8(*pB, zero);
   A = _mm_mullo_epi16(A, alfa);
   B = _mm_mullo_epi16(B, beta);
   D2 = _mm_add_epi16(A, B);
   D2 = _mm_add_epi16(D2, gamma);
   D2 = _mm_srli_epi16(D2, 7);
   *pD = _mm_packus_epi16(D1, D2);
   pA++;
   pB++;
   pD++;
}
//Then process the remaining pixels (if any).
```

Fig. 1. Our add-and-weight algorithm implemented using SSE intrinsic functions.

Image pixels are loaded into SIMD registers 16 bytes at a time for the SSE, and 32 for the AVX. The number of loops that the SIMD operations must perform in each thread is defined by

$$N_{loop} = \begin{cases} W_{non_last_thread} / A & \text{if } \left(T_{id} \neq N_{thread} - 1 \right) \\ W_{simd_last_thread} / A & \text{otherwise.} \end{cases} \quad (9)$$

V. Experimental Results

The SSE and AVX intrinsics were tested in nine configurations using 1, 2, 4, 6, 8, 10, 12, 14, and 16 threads. The machine was a 2.5 GHz Core i7-4710HQ with four processing cores, and 16 GB RAM; 64-bit Windows 8.1, Microsoft Visual Studio 15, and OpenCV 3.0 were installed. The correct memory alignment of each SIMD configuration was ensured by setting the environment variable *CV_MALLOC_ALIGN* to 16 and 32 for SSE and AVX respectively.

The *cv2.setUseOptimized* function was set to enable SIMD optimization, and the AVX auto-vectorization feature of Microsoft Visual Studio 15 was turned on. However, auto-vectorization was turned off when code using the SSE and AVX intrinsics was compiled. Five different image resolutions were tested, and the results are shown in Table I.

The execution time has two parts : the user CPU time for the add-and-weight operation and the system CPU time [13]. To reduce the effect of the system CPU time, our SSE and AVX functions were run *n* times, and the total elapsed time was divided by *n* to obtain the user CPU time for each SIMD-enabled function. *n* were set to 500,000 for images smaller than 1920x1200, and 250,000 and 30,000 for images sized 1920x1200 and 3648x2736 respectively.

The speed up factors for the multi-core configuration shown in Fig. 2 come from the best result for each SIMD implementation obtained from Table I. When operating on small images, our AVX implementation is more than 23 times faster than the original Addweighted function. However, the performance gain tends to decrease for larger images due to main memory bandwidth limitations. All the pixels of smaller images can fit into the cache, and the AVX and SSE instructions can reach the data quickly. For larger images, repeated access to main memory is required, and peak memory bandwidth is reached. The AVX is 1.83 times faster than the SSE in a multi-core implementation for image size 320×240. When the size is larger, the performance difference between these two SIMD instruction types decreases. For the largest image (3648x2736 pixels), SSE and AVX are only 2.54 and 2.57 times faster than the original addWeighted function.

TABLE I: Computation Time for Each Configuration of The Add-and-Weight Operation Performed on Different Image Sizes

Configurations	Computation Time (micro seconds)				
	320 x240	640 x480	1024 x768	1920 x1200	3648 x2736
OpenCV addWeighted compiled with AVX2 auto-vectorization	30.07	119.70	331.44	944.82	1409.1
SSE (1 thread)	8.61	34.59	116.82	359.3	704.9
SSE (2 threads)	4.43	18.55	56.41	194.4	659.0
SSE (4 threads)	2.51	11.56	31.64	129.9	637.3
SSE (6 threads)	2.54	11.92	32.03	108.1	631.9
SSE (8 threads)	2.38	11.29	30.38	112.7	646.6
SSE (10 threads)	2.55	11.60	31.00	94.7	617.1
SSE (12 threads)	2.43	11.61	30.85	92.2	620.5
SSE (14 threads)	2.52	11.87	30.58	92.4	588.5
SSE (16 threads)	2.45	11.37	30.38	91.2	554.8
AVX (1 thread)	5.21	30.63	88.87	300.4	689.0
AVX (2 threads)	2.88	15.35	44.87	183.9	650.4
AVX (4 threads)	1.55	6.84	24.38	121.9	633.7
AVX (6 threads)	1.52	7.26	22.57	91.7	628.1
AVX (8 threads)	1.30	5.58	20.98	111.6	644.0
AVX(10threads)	1.34	5.76	21.00	73.2	626.6
AVX(12threads)	1.33	5.66	20.20	67.9	612.0
AVX(14threads)	1.35	5.89	18.19	65.2	583.7
AVX(16threads)	1.31	5.73	18.02	64.0	548.5

Fig. 3 shows that for image sizes of 320×240 and 640×480, eight is the optimum number of threads which is twice the number of available CPU cores. However, larger image sizes mean that more threads increase the running speed. The multi-core AVX version follows the same pattern (see Fig. 4).

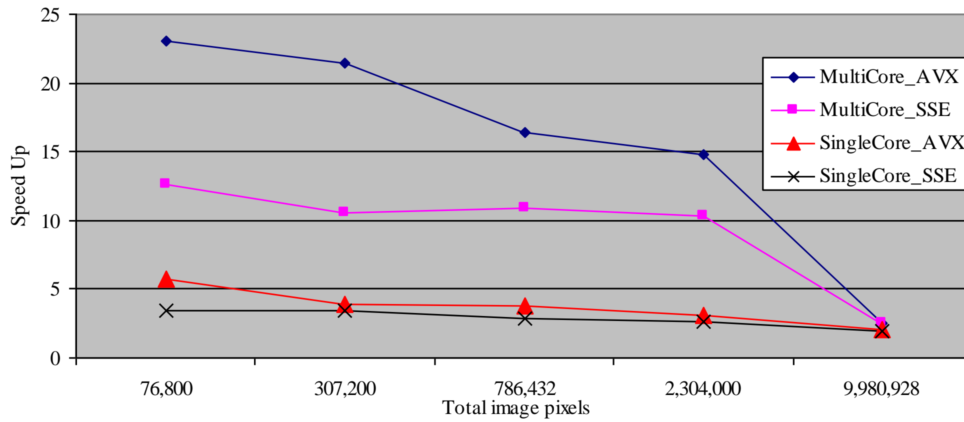

Fig. 2. Speed up factors for our algorithm relative to the original addWeighted function.

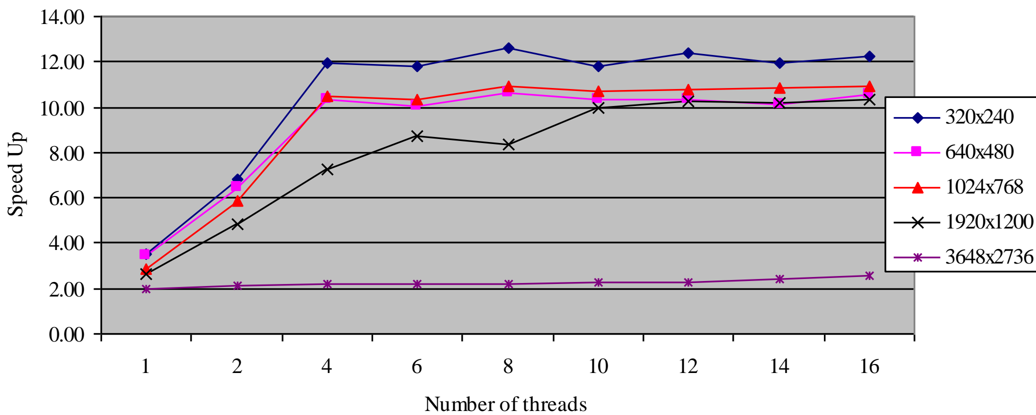

Fig. 3. Speed up factors of our algorithm using SSE intrinsics on different numbers of threads relative to the original addWeighted function.

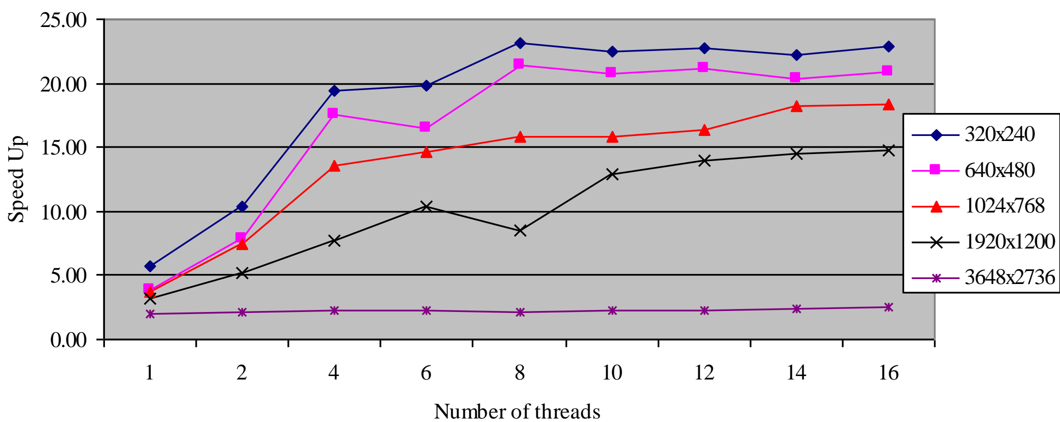

Fig.4. Speed up factors of our algorithm using AVX intrinsics on different numbers of threads compared to the original addWeighted function.

VI. Discussion and Conclusion

Our algorithm exceeds the speed of the addWeighted operation in the OpenCV library in every tested configurations, and even for very large images where peak memory bandwidth has been reached. Although AVX is twice as fast as SSE in theory, our work reveals that a performance gain of this magnitude is not easy to obtain. Memory bandwidth is one of the key factors limiting performance improvement.

Acknowledgement

The authors are grateful to Dr. Andrew Davison for his kind help in polishing the language of this paper.

Refferences

[1] Intel Corporation, *Intel Advanced Vector Extensions Programming Reference*, Ref.#319433-011, June 2011.

[2] ARM Limited, *Cortex-A9 NEON Media Processing Engine Technical Reference Manual*, 2011.
[3] Freescale Semiconductor Limited, *AltiVec Technology Programming Environments Manual*, 2006.
[4] F. Gerneth, *FIR Filter Algorithm Implementation using Intel SSE Instructions: Optimizing for Intel Atom Architecture*, Intel Corporation, March 2010.
[5] J. Y. Liu, M. R. Smith, F. A. Kuo, and J. S. Wu, "Hybrid OpenMP/AVX acceleration of a split HLL finite volume method for the shallow water and euler equations," *Computer & Fluids*, vol. 110, pp. 181-188, 2015.
[6] J. Frances, S. Bleda, A. Marquez, C. Neipp, S. Gallego, B. Otero, and A. Belendez, "Performance analysis of SSE and AVX instructions in multi-core CPUs and GPU computation on FDTD scheme for solid and fluid vibration problems," *The Journal of Supercomputing*, vol. 70, no. 2, pp. 514-526, 2014.
[7] B. L. Gal, C. Leroux, and C. Jego, "Multi-Gb/s software decoding of polar codes," *IEEE Transactions on Signal Processing*, vol. 64, no. 2, pp. 349-359, 2015.
[8] C. C. Chi, M. Alvarez-Mesa, B. Bross, B. Juurlink, and T. Schierl, "SIMD acceleration for HEVC decoding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 5, pp. 841-855, 2015.
[9] Optimization. [Online]. Available: http://docs.opencv.org/master/dc/d71/tutorial_py_opti mization. html#gsc.tab=0
[10] G. Mitra, B. Johnston, A. P. Rendell, and E. McCreath, "Use of SIMD vector operations to accelerate application code performance on low-powered ARM and intel platforms," in *Proc. IEEE 27th International* Parallel *and Distributed Processing Symposium Workshops and PhD Forum*, pp. 1107-1116, May 20-24, 2013.
[11] C. Juan and Y. Canqun, "Optimizing SIMD parallel computation with non-consecutive array access in inline SSE assembly language," in *Proc. International Conf. on Intelligent Computation Technology and Automation*, pp. 254-257, January 12-14, 2012.
[12] Addweighted. [Online]. Available: http://docs.opencv.org/modules/core/doc/operations_ on_arrays.html?highlight=addweighted#addweighted
[13] D. Patterson and J. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 5th edition, Morgan Kaufmann, 2013.

Panyayot Chaikan received the B.Eng. degree in computer engineering, the M.Eng. degree in electrical engineering from King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand, in 1999 and 2002, respectively. He received the Ph.D in computer engineering from Prince of Songkla University, Songkhla, Thailand, in 2010. He is currently a lecturer of the Department of Computer Engineering at Prince of Songkla University. His research interests include image processing, pattern recognition, parallel programming, and embedded systems.

Somsak Mitatha received the B.Ind. degree in television technology, the M.Eng. degree in electrical engineering, and the D.Eng degree in electrical engineering from the King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand, in 1987, 1995, and 2008, respectively. He is currently a lecturer and associate professor of computer engineering with the Department of Computer Engineering. His current research interests include computer hardware design, pattern recognition, hybrid networks, embedded systems, and nonlinear optical communication.