Cross-comparison of the Performance of Sequential Summed Area Table and Box Filter Algorithms with respect to C/C++ Compilers

Ali OZTURK¹, Ibrahim Cayiroglu²

¹ Machine and Metal Technologies, Mus Alparslan University, Turkey (a.ozturk@alparslan.edu.tr)
² Department of Mechatronics, University of Karabuk, Turkey (icayiroglu@karabuk.edu.tr)

Abstract

Summed area table algorithm has been used to accelerate some computer vision and signal processing algorithms. In this study, the performance of the sequential summed area table algorithms and box filter algorithm with and without summed area table algorithm is examined by taking account of effect of C/C++ compilers. Three variants of sequential summed area table algorithm are included into the study. Loop invariant code motion and loop unrolling optimization techniques are applied to one of them. The performance of GNU Compiler Collection (GCC) and Intel C/C++ Compiler (ICC) on both Windows and Ubuntu and Visual C++ (CL) compiler on Windows by using the summed area table and box filter algorithms are compared. Result of the study reveals that Intel C/C++ Compiler (ICC) perform best on Ubuntu with respect to others for sequential summed area table and box filter algorithm. The performance of summed area table calculation which utilizes Viola-Jones Equation by using a scalar accumulator outperforms other summed area table algorithms.

Keywords: Summed Area Table, Integral Image, Box Filter, Optimization, Compiler

1 Introduction

Computer vision algorithms require processing a large amount of input data that is retrieved from an image. Robust and accurate algorithms impose high computational complexity. However, there is a trade-off between robustness and efficiency, i.e., the run-time speed of an algorithm. Efficiency plays an important role in time-critical applications [42] or in real-time applications such as robotics, surveillance and augmented reality. There has been a huge amount of research activity in the literature to keep efficiency as high as possible without compromising the robustness and accuracy. Research activities that are aiming to leverage the efficiency can be divided into two main groups [27]; the algorithmic studies and the hardware architectural studies. Using a Monte-Carlo type speed-up technique to reduce the time-complexity of the least-median-of-squares method for regression analysis
(fitting a model to noisy data) in computer vision [30] and using the summed area table calculation to reduce the time-complexity of the texture mapping algorithm [8] can be given as examples of the algorithmic studies. On the other hand, the hardware architectural studies try to harness as much as possible the power of micro-architecture of modern processors. The objective of these studies is converting the algorithms to parallel architecture to take advantage of the multi-core architecture of the processors [13] and mapping the algorithm as suitable as it to fit the underlying memory hierarchy of the micro-architectures [31, 34, 25].

Summed area table has been widely used to accelerate the processing time of various computer vision algorithms since it is first introduced by Crow [8] for texture filtering. However, its implementation in computer vision algorithms was escalated when Viola-Jones [44] used it as an intermediate transformation within the face detection algorithm. Since then, its most popular implementation is its utilization within Speeded-Up Robust Features (SURF) extraction [3], which is one of the most prominent object detection algorithms. On the other hand, we can mention some other studies that are utilizing the summed area table for reducing the cost of computation of their algorithm. Kasagi et al [22] used it to accelerate the image classification algorithm which utilizes convolutional neural networks (CNNs). Lewis [26] benefits from summed area table computation in template matching algorithm. Yuan [45] proposed an accumulative motion model based on the summed area table for video smoke detection. Chen and Cheng [6] utilized summed area table to accelerate the computation of the singularity analysis in characterizing mineralization and predicting mineral deposit. Venter et al [43] employed summed area table in digital signal processing to detect an object by using electromagnetic waves. As it is summarized here, the summed area table has a vast application domain not only in computer vision but also in signal processing because of its positive effect on the execution performance of algorithms.

Summed area table reduces computational complexity and provides a considerable speed gain. However, its calculation still imposes a performance degradation for, especially, large size images [10]. Therefore, there have been also research activities to accelerate or optimize the summed area table calculation. These studies are focusing on how efficiently the summed area table can be calculated on multicore processors. Optimal parallel summed area table studies implements two successive prefix-sum; row-wise prefix sum and column-wise prefix sum [5, 41]. Parallel prefix sum or scan is one of the most important parallel primitives for parallel algorithms for building data structure, sorting [16] as well as the summed area table. Parallel algorithm studies for summed area table computation mainly tries to harness the power of modern GPUs [21, 35]. These studies deal with problems such as shared memory bank conflicts, cache misses as well as uncoalesced or strided global memory access [37]. Apart from studies, those are utilizing the parallel prefix sum on GPUs, targeting to the real-time embedded vision system, Ehsan et al [10] proposed a method to calculate summed area table in a manner of row parallel way by decomposing the recursive equations of Viola-Jones [44]. They also deal with the storage problem [4] of summed area table in real time embedded vision system.

In the context of obtaining high performance from an algorithm, compiler optimization can also be considered to be one of the main research streams. Modern programming languages have been increasingly developing towards to more abstract level so that they don’t require to be aware of the details of the underlying micro-architecture by providing high-level logic and mathematics tools to generate algorithms [29]. Consequently, an increasing level of abstraction from the micro-architectures puts an increasing strain on the compiler to generate best assembly code from the high-level code written by the programmers. However, C language still remains one of the closest programming languages to the machine language whilst C++ has been moving towards to be more abstract level [39].

Compilers are building executable from the source code by implementing some combination of optimization for the target micro-architecture. Of course, the performance of the compilers varies from each other. Code developers sometimes choose a compiler only for their convenience without being aware of the dominant effect of the compilers on software performance. So that, comparison of
different compilers according to the run-time performance of their executable is important and have been studied and analyzed in the literature. Machado et al [28] statistically analyzed the level of the optimization of GCC and ICC with the multithreaded algorithms on two distinct but compatible processors (Intel Xeon and AMD Opteron). They evaluate the performance with the execution time of the algorithm which achieves in run-time on Ubuntu operating system. However, their search space is very large and it is not easy to come up with a conclusion even with using statistical analysis. Karna et al [20] analyzed the compiler effect on the context of software reliability under UNIX based Fedora operating system by using some metrics such as time for compilation, generated code size and so on. The compilers they included in their study are Intel C/C++ Compiler (ICC), GNU Compiler Collection (GCC), Low Level Virtual Machine (LLVM), Sun Studio, and PGI Workstation. Algorithms from the SPEC CPU2006 are used for the benchmarking. They concluded that GCC shows more reliable performance with respect to other compilers and PGI outperforms ICC on Fedora operating system. Almomany et al [1] evaluated the performance of some algorithms from the PARSEC benchmark suites according to GCC and ICC on Ubuntu operating system. They concluded that ICC outperforms GCC in terms of the run-time speed-up of compiler’s executable. Han [15] evaluated the performance of some algorithms from SPEC CPU2006 benchmark suite according to different optimization flag of GCC on Windows operating system. The research on this paper is not enough to provide any conclusive evidence. Fourment and Gillings [12] evaluated some standard bioinformatics algorithms according to six different programming language on both Windows and Linux operating system. They observed that the performance of C/C++ is fastest and there is no clear evidence that distinguishes the performance on Windows and Linux. However, they do not take account of any possible effect of C/C++ compilers and their optimization option flags on the performance.

The primary objective of this study is to measure and compare the performance of sequential summed area table algorithms and box filter calculation with and without using summed area tables. Three different sequential summed area table algorithms are given; Algorithm 1 is using computation of Viola-Jones equations, Algorithm 2 is utilizing the reduced form of Viola-Jones equations, Algorithm 3 is a variant of Algorithm 1, which is utilizing a scalar accumulator with Viola-Jones equations.. Some classic optimization techniques, loop invariant code motion and loop unrolling, are applied to one of them. Performance of algorithms is measured and compared. Raw execution time measurement for each repetitive iteration is filtered by using Singular Spectrum Analysis before taking the average of each iteration. Measurement are carried out with GNU Compiler Collection (GCC), Intel C/C++ Compiler (ICC) on both Windows and Ubuntu and also Visual C++ Compiler (CL) on Windows operating system. So that this study provide an insight into the performance of three distinct compiler (GCC, ICC, and CL) as well as what makes different about working on Windows and Ubuntu in terms of the performance. Facciola et al [11] tested the performance of three different versions of the sequential summed area table algorithms with some C/C++ compilers on various processors. They reported that the algorithm of Viola-Jones Equations using a scalar accumulator outperforms other algorithms for all compilers. However, they did not include the algorithm of the reduced form of Viola-Jones Equations. Moreover, a cross-comparison of the performance of the compilers on the algorithms was not reported. As we see some researcher or code developer refer to different summed area table algorithms, this study also can be a guide for the selection of a summed area table algorithm in terms of the performance.
2 Summed Area Table

2.1 Definition

A summed-area table of a gray scale image is an array (or a matrix) in which each entry value representing the sum of all intensities above and to left of the corresponding pixels of the images [32, 8, 17] (see Fig. 1 (b)). More formally, a summed area table $S$ of an image $I$ can be expressed by

$$S(x, y) = \sum_{i \leq W} \sum_{j \leq H} I(i, j)$$

where $W$ and $H$ respectively are width and height of the image. Once the summed area table has been obtained, sum of all intensities over an arbitrary rectangular region of image can be easily recovered by taking a sum and two differences with only four memory access from the summed area table array references [44]. Computational complexity of this calculation is irrelevant to size of the rectangular region [3, 19] and the calculation can be realized in $O(1)$ time [35]. More specifically, let $I_1$ is an arbitrary rectangular region of image $I$, or $I_1 \subseteq I$. And, $S_1$ is the rectangular region of summed area table $S$, $S_1 \subseteq S$, which is corresponding to $I_1 \subseteq I$. Then, sum of all intensities over $I_1$ can be calculated by

$$\sum_{i,j} I_1(i, j) = P_1 - P_2 - P_3 + P_4$$

where $P_1$, $P_2$, $P_3$ and $P_4$ are the values on the border of $S_1$ (see Fig. 1 (a)).

2.2 Sequential Algorithms and Optimization

Sequential summed area table computation can be formulated with equations of Viola-Jones [44] as follows;

$$P(x, y) = P(x, y - 1) + I(x - 1, y - 1)$$

$$S(x, y) = S(x - 1, y) + P(x, y)$$

where $I$ is input image, $P$ is row-wise prefix sum (or scan), $S$ is summed area table. It should also be noted that $P(x, 0) = S(0, y) = 0$, $x = 1...H$, $y = 1...W$, $W$ and $H$ respectively stand for width and height of the input image. Equation 3 and equation 4 can be reduced to only one equation [33, 6]. If we input equation 3 into equation 4, we obtain;

$$S(x, y) = S(x - 1, y) + P(x, y - 1) + I(x - 1, y - 1)$$

On the other hand, by rearranging equation (4);

$$P(x, y - 1) = S(x, y - 1) - S(x - 1, y - 1)$$

Than, if we input equation (6) into equation (5), finally we obtain;

$$S(x, y) = S(x - 1, y) + S(x, y - 1) - S(x - 1, y - 1) + I(x - 1, y - 1)$$

where $x = 1...H$, $y = 1...W$, $W$ and $H$ respectively stand for width and height of the input image. Furthermore, width and height of the summed area table are respectively expanded to $(W + 1)$ and $(H + 1)$. It should also be noted that all entries of summed area table are initialized to the zero so that entries in first row and the column remain zero after computation. This case can be expressed as $S(0, y) = S(x, 0) = 0$. 

4
Figure 1: (a) Sum of all intensities over an arbitrary rectangular region of the image can be easily recovered by taking a sum and two differences from the summed area table with only four memory access [44, 23]. Time requiring for this calculation does not depend on the size of rectangular region [3] and the calculation can be realized in $O(1)$ time [19]. (b) Each entry of summed area table is the sum of all intensities above and to the left of the corresponding pixels of the images. This figure illustrates the sequential summed area table computation according to Equation 7. Green and blue entries of $S$ count up twice pixels in the green traced area of $I$, so that green entry of $S$ which corresponds with the green traced area of $I$ is subtracted according to Equation 7.
Viola-Jones equations (equation (3) and (4)) and their reduced form, equation (7), produces same result. However they describe two different algorithm. Their memory access patterns are different. Equation (7) becomes more easier to illustrate summed area table calculation. Fig. 1 (b) illustrates summed area table calculation according to equation (7).

Algorithm 1 and Algorithm 2 shows serial summed area table computation according to respectively Viola-Jones equations and their reduced form. Algorithm 1 utilizes an intermediate array to store the row-wise prefix sum of an input image (this is denoted by $P$). However, this is wasting memory resources and hinders the performance. Instead of that, a scalar accumulator [11] can be used to store row-wise prefix sum for the only current iteration. Algorithm 3 utilizes this consideration.

Loop invariant code motion and loop unrolling are commonly employed common sense optimization techniques. Loop invariant code motion or elimination of common sub-expression is an optimization of that aim is saving time by calculating loop invariant expressions and assigning them to temporary variables before a piece of code inside a loop that uses those parts is executed [14]. Loop unrolling is another optimization technique that is attempting to increase the execution performance by minimizing the cost of loop overhead such as testing the termination conditions, updating loop counter variables as well as hiding latency of reading data from memory [38]. To overcome this loop overhead, the necessary code within the loop can be rewritten multiple times, and accordingly, the counters and conditions can be updated. There is a trade-off between the loop unrolling factor, that is the number of times the loop is unrolled, and the code size. The best unrolling factor can only be determined by measurement [24].

Algorithm 4 is optimized version of Algorithm 3 by using loop invariant code motion and loop unrolling techniques. There are two loops of which outer loop is indexing rows with variable $y$, and the inner loop is indexing columns with variable $x$. In Algorithm 4, outer loop dependent calculations those are using variable $y$ are moved out from the inner loop. These calculations are assigned to variables ($a, a1, b, b1, c,$ and $c1)$. So that they are calculated before inner-loop starts. This is implementation of loop invariant code motion technique. Furthermore, both loops are unrolled two times. Two scalar accumulator variables ($\text{sum}$ and $\text{sum0}$) are defined and are assigned to zero before the inner loop starts because the inner loop which is indexing columns is unrolled two times. So that, when the code inside the inner loop is executed, two summed area table variables are calculated for every two adjacent rows. Therefore, for each iteration, four summed area table variables are calculated in the total. This loop unrolling procedure is shorten loop iteration number four times. After these two optimization techniques, we can expect some performance gain from Algorithm 4 with respect to Algorithm 3.

**Algorithm 1 Summed Area Table: Vectorized Viola-Jones Equations**

**Input:** $I$: one dimensional array for input image data with size $W \times H$

**Output:** $S$: one dimensional array for summed area table data with size $(W + 1) \times (H + 1)$

1: Initialize all summed area table data to zero
2: Define a one dimensional array for $P$ with size $(W + 1) \times (H + 1)$ and initialize to zero.

3: for $y = 1; y <= H; y ++$ do
4:     for $x = 1; x <= W; x ++$ do
5:         $P[x + y \times (W + 1)] \leftarrow I[(y - 1) \times W + x - 1] + P[x - 1 + y \times (W + 1)]$
6:         $S[x + y \times (W + 1)] \leftarrow S[x + (y - 1) \times (W + 1)] + P[x + y \times (W + 1)]$
7:     end
8: end
### Algorithm 2 Summed Area Table: Reduced Form of Viola-Jones Equations

**Input:** \( I \): one dimensional array for input image data with size \( W \times H \)

**Output:** \( S \): one dimensional array for summed area table data with size \((W + 1) \times (H + 1)\)

1. Initialize all summed area table data to zero

2. for \( y = 1; \ y \leq H; \ y++ \) do
3.  for \( x = 1; \ x \leq W; \ x++ \) do
4.     \( S[x + y \times (W + 1)] \leftarrow S[x + (y - 1) \times (W + 1)] \)
5.     \(+S[x - 1 + y \times (W + 1)]\)
6.     \(-S[x - 1 + (y - 1) \times (W + 1)]\)
7.     \(+I[x - 1 + (y - 1) \times W]\)

### Algorithm 3 Summed Area Table: Viola-Jones Equation Using a Scalar Accumulator

**Input:** \( I \): one dimensional array for input image data with size \( W \times H \)

**Output:** \( S \): one dimensional array for summed area table data with size \((W + 1) \times (H + 1)\)

1. Initialize all summed area table data to zero

2. for \( y = 1; \ y \leq H; \ y++ \) do
3.     \( sum \leftarrow 0 \)
4.     for \( x = 1; \ x \leq W; \ x++ \) do
5.         \( sum \leftarrow I[(y - 1) \times W + x - 1] + sum \)
6.         \( S[x + y \times (W + 1)] \leftarrow S[x + (y - 1) \times (W + 1)] + sum \)
7.         \( S[x + 1 + y \times (W + 1)] \leftarrow S[x + 1 + (y - 1) \times (W + 1)] + sum \)

### Algorithm 4 Summed Area Table: Optimized Version of Algorithm-3

**Input:** \( I \): one dimensional array for input image data with size \( W \times H \)

**Output:** \( S \): one dimensional array for summed area table data with size \((W + 1) \times (H + 1)\)

1. Initialize all summed area table data to zero

2. for \( y = 1; \ y \leq H; \ y+ = 2 \) do
3.     \( sum \leftarrow 0 \)
4.     \( sum1 \leftarrow 0 \)
5.     \( a \leftarrow W \times (y - 1) \)
6.     \( a1 \leftarrow W \times y \)
7.     \( b \leftarrow (W + 1) \times y \)
8.     \( b1 \leftarrow (W + 1) \times (y + 1) \)
9.     \( c \leftarrow (W + 1) \times (y - 1) \)
10.    \( c1 \leftarrow b \)
11.   for \( x = 1; \ x \leq W; \ x++ = 2 \) do
12.     \( sum \leftarrow I[a + x - 1] + sum \)
13.    \( S[x + b] \leftarrow S[x + c] + sum \)
14.    \( sum \leftarrow I[(a + x] + sum \)
15.    \( S[x + 1 + b] \leftarrow S[x + 1 + c] + sum \)
16.    \( sum1 \leftarrow I[a1 + x - 1] + sum1 \)
17.    \( S[x + b1] \leftarrow S[x + c1] + sum1 \)
18.    \( sum1 \leftarrow I[(a1 + x] + sum1 \)
19.    \( S[x + 1 + b1] \leftarrow S[x + 1 + c1] + sum1 \)
20.
3 Application of Summed Area Table to Box Filter

![Convolution Mask](image)

\[ p_f = A p_{11} + B p_{12} + C p_{13} + D p_{21} + E p_{22} + F p_{23} + G p_{31} + H p_{32} + L p_{33} \]

Figure 2: Illustration for the convolution operation by using a 3x3 convolution mask. The origin of the convolution mask is the location of \( E \) and the weights \( A, B, ..., L \) are the values of \( h(-k, -l) \) where \( k, l = -1, 0, 1 \).

Image filtering can be simply defined as preprocessing the image to obtain a more convenient form for the further image processing operation. Linear filtering is an operation utilizing a local operator or neighborhood operator \([40]\) in which a pixel value of the output image is calculated by using some collection of the pixel values in the predetermined vicinity of the corresponding input pixel:\(^1\)

For such a filter, the output \( g(x, y) \) is the convolution of \( f(x, y) \), which is the raw entries of the input image, with the weight kernel or convolution mask \( h(x, y) \). This convolution operation is denoted by the operator \( \otimes \) and is defined:\(^2\)

\[ g(x, y) = f(x, y) \otimes h(x, y) = \sum_{k=1}^{n} \sum_{l=1}^{m} f(k, l) h(x - k, j - l) \] (8)

This convolution operation is illustrated in Figure 2 using a 3x3 kernel or convolution mask.

Box filter which is also called moving average or mean filter is the simplest form of the linear filter in which a pixel value is substituted with the average value of the pixel values in a \( K \times K \) window centering the corresponding pixel location. For the box filter, convolution operation (Equation 8) can be expressed as follows;

\[ g(x, y) = \frac{1}{M} \sum_{k=1}^{n} \sum_{l=1}^{m} f(k, l) \] (9)

where \( M \) is the total number of pixels in the window. In case of the box filter, for a \( K \times K \) window,
convolution mask can be expressed as below:

\[ h = \frac{1}{K^2} \begin{bmatrix} 1 & 1 & \ldots & 1 \\ 1 & 1 & \ldots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \ldots & 1 \end{bmatrix} \]  

(10)

where all entries are identical to \( 1/K^2 \). This convolution mask has an averaging effect on all intensities over the window.

Figures 3: Effect of box filter calculation with Lena image. To make the effect of box filter more apparent, the image of the sub-window surrounding the eyes of Lena image is given below each image. Noisy image is produced with salt and paper noise from the original Lena image. The size of the image is 1024x1024. 7x7 and 19x19 window is created with respectively \( a_1 = -3, a_2 = 3 \) and \( a_1 = -9, a_2 = 9 \) in Algorithm 5 or in Algorithm 6.

Box filter computation can be written as in Algorithm 5. In this algorithm, size of the window is \((a_2-a_1)x(a_2-a_1) \in \mathbb{N}^+\), and \( a_1 = a_2 \). \( y \) and \( x \) indexes are starting from the minus value of \( a_1 \), so that the pixel value which is being processed is centered with convolution mask. Two innermost loops are acting inside the window where all intensities are accumulated when those two loops are terminated. When two innermost loops is terminated, the calculated sum is averaged by the size of the window and written into the location in the array which is allocated for the filtered image data. One limitation of this algorithm is that it does not calculate convolution on the left and bottom edge of the image. These calculations\(^3\) are ignored at here. Increasing the window size results in a greater amount of noise elimination, but on the contrary results in reduction on the image details and features (see Figure 3).

The time complexity of the Algorithm 5 is \( O((KxK)x(WxH)) \) where \( WxH \) is the size of the image and \( KxK \) is the size of the window. Therefore, for a given image at any size, increasing the window size also increases the time complexity. Here is where the summed area tables come in handy. The moving average of each window can be calculated with the pre-calculated summed area table of a given image. Two innermost loops in Algorithm 5 calculates the average value of intensities

inside the window. So that these loops can be replaced with the calculation which is expressed in Equation 2 (consider Figure 1). Box filter computation with the summed area table is given in Algorithm 6. This algorithm reduces the time complexity from $O(K \times K \times (W \times H))$ to $O(W \times H)$.

### Algorithm 5 Box Filter Computation without Summed Area Table

**Input:** $I$ : one dimensional array for input image data with size $W \times H$

**Output:** $F$ : one dimensional array for filtered image data with size $W \times H$

1. for $y = -a_1; y <= (H - a_2); y + +$ do
2.  for $x = -a_1; x <= (W - a_2); x + +$ do
3.    $\text{sum} \leftarrow 0$
4.    for $yy = a_1; yy <= a_2; yy + +$ do
5.      for $xx = a_1; xx <= a_2; xx + +$ do
6.        $i \leftarrow x + xx$
7.        $j \leftarrow y + yy$
8.        $\text{offset}1 \leftarrow i + j \times W$
9.        $\text{sum} \leftarrow \text{sum} + I[\text{offset}1]$
10.       $\text{intensity} \leftarrow \text{sum} / ((a_2 - a_1) \times (a_2 - a_1))$
11.      $\text{offset}2 \leftarrow x + a_1 + (y + a_1) \times W$
12.    $F[\text{offset}2] \leftarrow \text{intensity}$

### Algorithm 6 Box Filter Computation with Summed Area Table

**Input:** $S$ : one dimensional array for Summed Area Table with size $(W + 1) \times (H + 1)$

**Output:** $F$ : one dimensional array for filtered image data with size $W \times H$

1. for $y = -a_1; y < (H - a_2); y + +$ do
2.  for $x = -a_1; x < (W - a_2); x + +$ do
3.    $\text{sum} \leftarrow S[x + a_1 + (y + a_1) \times (W + 1)]$
4.    $-S[x + a_1 + (a_2 - a_1) + (y + a_1) \times (W + 1)]$
5.    $-S[x + a_1 + (y + a_1 + (a_2 - a_1)) \times (W + 1)]$
6.    $+S[x + a_1 + (a_2 - a_1) + (y + a_1 + (a_2 - a_1)) \times (W + 1)]$
7.    $\text{intensity} \leftarrow \text{sum} / ((a_2 - a_1) \times (a_2 - a_1))$
8.    $\text{offset} \leftarrow x + a_1 + (y + a_1) \times W$
9.    $F[\text{offset}] \leftarrow \text{intensity}$

### 4 Material and Methods

The experiments is performed in only one machine: Intel (R) Core(TM) i7-4710HQ CPU with physical cores at 2.5GHz and 16GB of RAM and its system type is x64. Windows 10 and Ubuntu 16.04.05 LTS are installed on same machine.

Compile option -O3 turns on high level optimization for both GCC and ICC. For Visual C++ (CL) compiler, /O2 flag optimizes code for maximum speed. Performance of the algorithms are tested with and without these compiler options.
Algorithm execution time is measured by using `sys/time.h` header file [7] on Ubuntu and by using `Windows.h` header file - i.e based on Windows high-performance timers\(^1\) on Windows. Execution time of an algorithm is measured inside a loop that iterates 1000 times and the execution time for each iteration is saved. Saved data is filtered by using Singular Spectrum Analysis on Matlab. After filtering, mean value of data is used to show the performance of algorithms.

The superiority of the singular spectrum analysis (SSA) over other filtering algorithms such as Wigner function, discrete wavelet transform, splines as well as Butterworth filters is reported in the literature [2]. A short description of the SSA algorithm can be found in this paper [36]. The output of SSA algorithm relies on two parameters; \( L \): the window length and \( r \): the number of largest eigenvalues those are explained the greater amount of variation and trend in the raw data with respect to remaining eigenvalues.

Figure 4 show the raw and filtered execution time for each iteration of `integral` function of OpenCV library. For this computation of SSA, \( L \) is 100 and \( r \) is 1 because first eigenvalue constitutes %99 of all eigenvalues.

![Figure 4: Raw and filtered execution time of `integral` function of OpenCV](image)

Only one input data size used in this study. Random `double` type numbers between the range of 0 – 255 with the size of 1024x1024 are created and used for all algorithms and for all tests in this

study. For box filter calculation, the window size selected as $a1 = -7$ and $a2 = 7$ in Algorithm 5 and Algorithm 6.

Performance result of the compilers are labelled as follows; if none of the optimization flag is used, then gcc, icc and cl are used to show the results of GCC, ICC and CL. If the optimization flag is used, then gccO3, iccO3 and clO3 are used to show the results of GCC, ICC and CL.

5 Result

5.1 Performance of Summed Area Table Algorithms

Figure 5 and Figure 6 shows the performance of Summed Area Table Algorithms, those are given in Sequential Algorithms and Optimization Section, when they are compiled with different compilers with and without their optimization flags on both Windows and Ubuntu. From the perspective of these figures, the compilers with their optimization flag produce far better result with respect to the compilers without optimization flag except Intel Compiler (ICC). Performance improvement of ICC with its compile option O3 relative to ICC without any optimization flag is not notable in these measurements. However, ICC produces approximately close performance to GCC with most advanced optimization flag (O3) even when it compiled without any optimization flag. Microsoft Visual C++ (CL) compiler with its most advanced optimization flag, O2, produces worst results with respect to other compilers when they are compiled with their most advanced optimization flag (O3).

Performance of Microsoft Visual C++ (CL) compiler with compile option O2 approximates the performance of other compilers only in Algorithm 4 which is optimized version of Algorithm 3. The optimization techniques applied in Algorithm 4, loop invariant code motion and loop unrolling techniques, play a prominent role in Microsoft Visual C++ (CL) compiler. In GCC and ICC with compile option of O3, the effect of optimization on Algorithm 4 is not notable. The effect of optimization techniques is also notable in GCC when it is compiled without any optimization flag. However, the performance of GCC and ICC with optimization flag O3 in Algorithm 3 is close to their performance in Algorithm 4, slightly better than the performance of Microsoft Visual C++ (CL) in Algorithm 4, and far better than the performance of GCC without any optimization flag in Algorithm 3 and Algorithm 4. This indicates that GCC and ICC compiler intrinsically include the optimization techniques applied in Algorithm 4, and it is unnecessary to apply these optimization techniques by utilizing these compilers.

According to Figure 7, which is the reduced form of Figure 5 and Figure 6 by omitting result of GCC without optimization flag on both Windows and Ubuntu and also results of Microsoft Visual C++ (CL) compiler, it can be said that the compilers, GCC and ICC, produce better result on Ubuntu with respect to Windows. However the performance differences is very small between Ubuntu and Windows.

Figure 8 shows the relative performance of Algorithm 1, Algorithm 2, Algorithm 3, and Algorithm 4. Algorithm 1 is using computation of Viola-Jones equations (Equation 3 and Equation 4). Algorithm 2 is utilizing the reduced form of Viola-Jones equations (Equation 7). Algorithm 3 is a variant of Algorithm 1, which is utilizing a scalar accumulator with Viola-Jones equations. Algorithm 4 is optimized version of Algorithm 3 with loop invariant code motion and loop unrolling techniques. From the figure, it is clear that Algorithm 3, which is using a scalar accumulator with Viola-Jones equations, produces a better result with respect to its vectorized version (Algorithm 1) and the algorithm which is using the reduced form of Viola-Jones equations (Algorithm 2).
Cross-comparison of the Performance of Sequential Summed Area Table and Box Filter Algorithms with respect to C/C++ Compilers

![Algorithm-1](image1)

![Algorithm-2](image2)

**Figure 5:** Result of Summed Area Table Calculation with Algorithm 1 and Algorithm 2. The performance of GCC and ICC with and without their optimization flag \((O3)\) on both *Ubuntu* and *Windows* and result of *CL* with and without its optimization flag \((O2)\) on windows are given.
Figure 6: Result of Summed Area Table Calculation with Algorithm 3 and Algorithm 4. The performance of GCC and ICC with and without their optimization flag ($O3$) on both Ubuntu and Windows and result of CL with and without its optimization flag ($O2$) on windows are given.
Figure 7: Result of Summed Area Table Calculation with Algorithm 1, Algorithm 2, Algorithm 3 and Algorithm 4. The performance of GCC with its optimization flag (O2) on both Windows and Ubuntu and the performance of ICC with and without its optimization flag on both Windows and Ubuntu are given.
Figure 8: Result of Summed Area Table Calculation of Algorithm 1 to Algorithm 4. Only the performance of GCC with its optimization flag (O3) on Ubuntu and the performance of ICC with and without its optimization flag on Ubuntu are given.

5.2 Box Filter Computation

Figure 9 shows the performance of box filter algorithms; Algorithm 5: box filter computation without the summed area table, Algorithm 6: box filter computation with the summed area table. From the figures it is again obvious that all compilers with their most advanced optimization flag produce a considerable amount of speed-up except Intel Compiler (ICC). However, ICC with or without optimization flag can reach, even outperform the performance of GCC with optimization flag O3.

From the Figure 10, which is the reduced form of Figure 9 by omitting results of GCC and CL without their optimization flag, it is understood that the performance of the compilers with respect to each other differs for Algorithm 5 and Algorithm 6. On Algorithm 5, CL shows close performance to GCC. At this point, we recall that the time complexity of Algorithm 5 is $O((KxK)x(WxH))$ and the time complexity of the Algorithm 6 is $O(WxH)$. Hence there are considerable differences in the performance of Algorithm 5 and Algorithm 6. On Algorithm 5 where the mean execution time is high, it is more clear (because it is less affected from the measurement noise) that ICC produces approximately two times better result on both Windows and Linux with respect to GCC and CL. For Algorithm 6, where the time complexity is reduced, the pattern of the figure is changed. CL produces the worst results with respect to others for this algorithm. ICC produces approximately same result with GCC on Windows, however on Ubuntu produces 1.6 times better result with respect to GCC.

Speed-up obtained with Algorithm 6 with respect to Algorithm 5 reaches up to 477 times according to compiler, compiler option and operating system (see Figure 9 and see Figure 10). Table 2 shows the performance of Algorithm 3 and Algorithm 6 with respect to CL with compile option O2 and GCC and ICC with compile option O3 on both Windows and Ubuntu. It is clear that ICC with compile option O3 on Ubuntu produces best result in total. Its performance is %25 better than ICC with compile option O3 on Windows and GCC with compile option O3 on both Ubuntu and Windows, and %100 better than CL with compile option O2 on Windows.

Figure 11 shows that raw and filtered execution time data for each iteration of summed area table calculation (Algorithm 3) and box filter calculation by using summed area table (Algorithm 6) when the algorithms are compiled by ICC with O3 compile option on both Windows and Ubuntu. From
the raw and filtered data, it is clear that ICC on Ubuntu not only produces better performance but also less fluctuates with respect to all data of Windows.

<table>
<thead>
<tr>
<th>OS</th>
<th>Compiler</th>
<th>Algorithm 3</th>
<th>Algorithm 6</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>clO2</td>
<td>2.2</td>
<td>3.8</td>
<td>6</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>iccO3</td>
<td>1.4</td>
<td>1.5</td>
<td>2.9</td>
</tr>
<tr>
<td>Windows</td>
<td>iccO3</td>
<td>1.5</td>
<td>2.5</td>
<td>4</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>gccO3</td>
<td>1.4</td>
<td>2.5</td>
<td>3.9</td>
</tr>
<tr>
<td>Windows</td>
<td>gccO3</td>
<td>1.5</td>
<td>2.5</td>
<td>4</td>
</tr>
</tbody>
</table>

6 Discussion and Conclusion

Efficiency, i.e, the run time speed of the computer vision algorithms is very important for real-time applications such as robotics, surveillance and augmented reality. The algorithmic and hardware architectural studies have been conducted to leverage the performance of computer vision algorithms. Summed area table which can be categorized into the algorithmic studies has been used to accelerate some computer vision and also signal processing algorithms in the literature. Some researchers also conducted research into the hardware architectural studies to further accelerate the summed area table algorithm. In this paper, the performance of the sequential summed area table algorithms and box filter algorithm with and without summed area table algorithm is examined by taking account of the effect of C/C++ compilers. So, this paper is also be considered to be a comparative study of C/C++ compilers. The performance of GNU Compiler Collection (GCC) and Intel C/C++ Compiler (ICC) on both Windows and Ubuntu and Visual C++ (CL) compiler on Windows by using the summed area table and box filter algorithms are compared.

The result of the present paper demonstrated that: (i) The performance of summed area table calculation which utilizes Viola-Jones Equation by using a scalar accumulator (Algorithm 3) outperforms other algorithms of summed area table algorithms. (ii) Intel C/C++ Compiler (ICC) with (or without) its compile option O3 outperforms other compilers in total of the performance of summed area table algorithm (Algorithm 3) and box filter algorithm (Algorithm 6). (iii) Performance of Intel C/C++ Compiler (ICC) on Ubuntu is better than the performance of it on Ubuntu with respect to total performance of summed area table algorithm (Algorithm 3) and box filter algorithm (Algorithm 6) (iv) Loop invariant code motion and loop unrolling optimization techniques are not necessary to accelerate algorithms if the GCC and ICC used with O3 compile option.

Summed area table provides a considerable speed gain in box filter calculation. However application of summed area table to the linear filters are limited to box filters. Using non-uniform filters such as the filter with Gaussian weighting can enhance the object detection algorithms [9]. Hussein at al [18] extended the summed area table calculation so that a wide class of non-uniform filters can be computed with it. In the future, we will examine performance of such studies in the context of object detection.

References

Figure 9: Result of Box Filter Calculation
Cross-comparison of the Performance of Sequential Summed Area Table and Box Filter Algorithms with respect to C/C++ Compilers

Figure 10: Result of Box Filter Calculation
Figure 11: Raw and filtered execution time data for each iteration of summed area table calculation (Algorithm 3) and box filter calculation by using summed area table (Algorithm 6) when the algorithms are compiled by ICC with O3 compile option on both Windows and Ubuntu.


