MULTI-THREADED COMPUTATION OF THE SOBEL IMAGE GRADIENT ON INTEL MULTI-CORE PROCESSORS USING OPENMP LIBRARY

Ahmed Sherif Zekri[1,2]

[1]Department of Mathematics & Computer Science, Beirut Arab University
P.O. Box 115020, Riad El Solh, 11072809 Beirut, Lebanon
[2]Department of Mathematics & Computer Science, Alexandria University
MoharramBek, Alexandria 21511, Egypt

ABSTRACT

Performance of applications executed on multi-core processors is not boosted by just dividing the work among a team of threads and assigning them blindly to the CPU cores. Factors such as data access patterns in memory, the way of allocating the threads to the physical cores, and how the data are partitioned among the threads significantly affect performance. In this paper, we target the acceleration of the Sobel image gradient computing which is important in segmenting images for further processing in computer vision and image analysis applications. We present a multi-threaded algorithm using the standard OpenMP threading library to parallelize the computations using two Intel multi-core processors. The effects of the parallelization factors on the performance of the proposed algorithm are evaluated using different image resolutions to draw accurate conclusions. Our results showed a maximum attained speedup closer to the number of physical cores in the CPU, which is the maximum theoretical value.

KEYWORDS

Sobel Filter, algorithm parallelization, OpenMP threading library, static loop scheduling, image partitioning.

1. INTRODUCTION

Current general-purpose processors offer parallel processing capabilities at low cost. Nowadays, all manufactured CPU chips in desktop and smart-phones are multi-core, i.e., having more than one processing core inside. For example, the Intel Core i7 CPU is composed of four physical cores integrated in one microprocessor chip which can execute up to eight independent tasks or logical threads at the same time; the Intel Hyper-threading technology that permits two threads to execute concurrently on the same physical core.

Many applications especially in digital image processing are inherently parallel. In these applications, usually the pixels are scanned by rows where some linear and/or non-linear operation is applied at each pixel. The computations involve the neighbor pixels to the current one. It is clear that processing can be done on pixels at different divisions of the image, simultaneously. Therefore, multi-core processors can definitely be exploited to perform these computations in parallel.

In this paper, we show that using a multi-threading library, such as OpenMP, for exposing the parallelism of the spatial image filter kernel can significantly boost performance of important

DOI:10.5121/ijcsit.2016.8207

image processing applications such as image visual enhancement, detection of objects, and high-level image analysis operations leading to ultimate goal of computer vision applications in robots and autonomous systems. We investigate the parallel execution of the important image-filtering kernel used in many applications such as the detection of objects' borders or simply edges in digital images. The most common technique used to determine edges is to convolve a filter of fixed size window with the pixels surrounding the current pixel to determine if the current pixel belong to an edge or not. For example, the Sobel and Prewitt edge operators have special 3 x 3 masks/filters to detect the intensity changes between neighbor pixels in the horizontal, vertical, and/or diagonal directions. At each pixel, weighted summation of the mask coefficients and the eight neighbor pixels of the current pixel element-wise multiplied to get a scalar value. Using the final values, the gradient vectors can beestimated using some mathematical formula, discussed in the next section. The calculations at each pixel require twice as nine integer multiplications and additions in addition to two absolute value operations to find the net edge value at the current pixel. For larger mask sizes, the number of computations increases. Therefore, the main problem with the gradient computation is the time-complexity especially if executed sequentially on one core of a multi-core CPU. Hence, our main objective is to use all the available cores of the CPU to accelerate the computation of Sobel image gradient.

Since the determination of the edges information in the whole image is inherently a parallel computation, the image pixels can be divided among the cores of the CPU so that each core determines the edges in its assigned part of the image. If the execution proceeds with minimal inter-thread communications, then substantial performance improvements are obtained. This reduction of execution time is a major benefit to many on-line image processing applications such as robotics vision, object tracking, medical diagnostics, to name a few.

The rest of the paper is organized as follows. Section 2 presents the sequential algorithm to determine the edges of objects in a gray-level image. Section 3 introduces our multi-threaded parallel algorithm of the Sobel sequential algorithm. Section 4 describes the OpenMP implementation of the proposed parallel algorithm. Section 5 describes the experimental results on a multi-core processor with two Intel multi-core processors, a Core i7 CPU having four physical cores and a Core 2 Duo CPU having dual cores. We analyzed the performance based on image partitioning, the mapping of the logical threads to the physical cores of the CPU, which is known as the processor affinity, and the effect of using different number of threads on the available physical cores. Section 6 presents the related works of our research. Section 7 concludes the paper and outlines our future plans.

2. SEQUENTIAL EDGE DETECTION ALGORITHM

Consider a gray-level input image of n rows and m columns. Each pixel at location (x,y), has an intensity value between 0 (Black) and 255 (White). Let the input image be assigned to a two-dimensional matrix I of the same size. The idea of the Sobel operator is to find the value of the edge at each pixel by approximating the calculation of the gradient vector. The gradient vector is composed of two components along the x-axis and y-axis of the image, as defined in equation (1) below.

$$\nabla f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{pmatrix} \qquad (1)$$

Therefore, the magnitude of the gradient vector can be calculated as follows:

$$|\nabla f| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \qquad (2)$$

Algorithm 1: Sequential Algorithm

Input:

An $n \times m$ gray-scale input image is read using any available image processing library into a two-dimensional array $I(x, y)$, $0 \leq x \leq m-1$, $0 \leq y \leq n-1$.

Output:

A binary $n \times m$ image $O(x, y)$ is calculated from the magnitude of the image gradient.

Compute:

for $i \leftarrow 1$ **to** $n-2$ **do**
  for $j \leftarrow 1$ **to** $m-2$ **do**
    At pixel $I(i, j)$, calculate the horizontal and vertical approximations of the gradient S_x and S_y given by equations (5) and (6), respectively.
    Find the net magnitude of the gradient given by equation (7).
  end
end

Listing1: The Sequential Algorithm.

Sobel algorithm approximates the calculation of the gradient components at a pixel *I(x,y)* by calculating two weighted sums S_x and S_y in the x-dimension (vertical) and y-dimension (horizontal), respectively. Each weighted summation is the result of convolving a specially designed 3 x 3 matrix with the current pixel *I(x,y)* and its eight neighbors*I(x-1,y-1)*, *I(x-1,y)*, *I(x-1,y+1), I(x,y-1)*, *I(x,y+1)*, *I(x+1,y-1)*, *I(x,y), and I(x,y+1)*. These 3 x 3 masks are given in equations (3) and (4) below:

$$M_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \qquad (3)$$

$$M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \qquad (4)$$

The weighted sums are computed in equations (5) and (6) below:

$$S_x(x, y) = \sum_{i=-1}^{1} \sum_{j=-1}^{1} W_x(i, j) I(x+i, y+j) \qquad (5)$$

Where the weight $W_x(i,j)$ corresponds to the element of the Sobel mask $M_x(i+1,j+1)$, and

$$S_y(x,y) = \sum_{i=-1}^{1}\sum_{j=-1}^{1} W_y(i,j) I(x+i, y+j) \quad (6)$$

Where the weight $W_y(i,j)$ corresponds to the element of the Sobel mask $M_y(i+1,j+1)$.

The Algorithm of computing the edges of an input image is described in Listing 1. The input to this sequential algorithm is the gray-level n x m matrix I. The output is a binary image of black and white intensities. The white pixels represent the edges of the objects calculated by the Sobel operator while the black pixels constitute the background. In the two nested loops at the compute step, the two weighted sums (5) and (6) are calculated at each pixel of the input image. Note that the pixels at the four borders of the image are not included since not all the nine neighbors will be available. Hence, these pixels are set to black in the output image.

The magnitude of the gradient vector given at equation (2) is approximated, as many image-processing applications do, by using the absolute values instead of the square root and the squaring of the gradient components. This is given in equation (7) below [1].

$$|\nabla f| \approx |S_x| + |S_y| \quad (7)$$

3. PARALLEL ALGORITHM

In this section, we start by presenting some important factors, which can significantly affect the parallelization of algorithms on shared-memory multi-core processors. Then, we present our multi-threaded algorithm to parallel Sobel's algorithm, and explains its main stages.

3.1 Parallelization Problems

The algorithm discussed in this section targets shared-memory systems, specifically, single-chip CPUs composed of multi-cores. Therefore, the data processed by the cores or threads are assumed stored in one global shared memory, which is the RAM modules in desktop computers. The main problem arise is how the shared data will be partitioned (logically) and assigned to the available threads such that the communication between the threads is minimized.

Most of the low-level image processing operations, including the detection of object's edges, are inherently parallel. The same operation can be applied to almost all pixels of the image, if the processing in any pixel requires only pixel values in a small neighborhood around the target pixel. For instance, in Sobel algorithm, the calculation of the edge value on a given pixel depends on the values of the eight neighbor pixels surrounding it; this is because the size of the conventional Sobel masks are 3 x 3. When using larger size masks, the neighborhood size of the current pixel increases and more pixels are involved in the calculation. Fortunately, as we can see that this pattern of data dependency is only imposed in the reading of the pixel values not on their update since the new values are usually written to another output image. Hence, taking this pattern of data dependency into consideration leads to employing the most appropriate data-partitioning scheme.

A good partitioning scheme should take into consideration the following three factors:

The pattern of data access of the running algorithm. The first factor is imposed by the structure of the algorithm itself, which is the responsibility of the programmer. Usually if the programmer is aware of some loop optimization techniques [8] that can be applied at the source-code level; it could help the compiler generate efficient codes.

The layout of data into memory. The underlying language compiler controls the second factor. When using a C/C++ compiler, the layout of data is the well-known row-major pattern. However, other language such as FORTRAN or MATLAB, the arrays are laid out in column-major fashion.

The data cache line size. The third factor is a hardware feature imposed by the processor vendor based on the sizes of the cache memories at different levels of the memory hierarchy and the underlying architectural features implemented in the physical cores. Given a cache line size, if one thread is updating one pixel value located in some cache line, other threads working on pixels in the same cache line are paused from updating their pixels even the threads are updating different pixels. This phenomenon is known as false sharing, and it can cause severe performance degradation if data is not partitioned among the cores in a proper way. The effect of sharing a cache line between two cores will definitely affect performance especially if the threads are assigned to two different cores where the cache line will be ping ponged between the L1 data caches of both cores. This restriction guides us to partition the image among the working team of threads in a way that can avoid this effect. We will elaborate on this important issue in Section 5.

3.2 The Algorithm

Now, we present a multi-threaded implementation of the sequential Sobel algorithm presented earlier in Algorithm 1. We concentrated, in both the sequential and parallel algorithms, on the most time consuming part that is the calculation of the gradient approximations at each pixel. The algorithm description is given in Algorithm 2 below.

Algorithm 2: The Multi-threaded Algorithm

Input:

An input matrix $I(x, y)$ of n rows and m columns,
A team of logical threads $N_T > 1$.

Output:

A gradient matrix $O(x, y)$ of n rows and m columns.

Divide:

Partition the input matrix I into I_j blocks, $0 < I_j < N_B - 1$.
Partition the output matrix O into O_j blocks, $0 < O_j < N_B - 1$.

Assign:

Assign the threads $T_i, 0 < i < N_T - 1$ to at least one pair of blocks (I_j,O_j), $0 < j < N_B - 1$.

Compute:

Each thread T_i computes the magnitude of the gradient, $|\nabla I|$, for all pixels in each assigned block I_j and updates the pixels of the corresponding output block O_j.

Listing2: The proposed Multi-threaded Algorithm.

The input to the algorithm is an input matrix *I* of size *n* x *m*, and a team of logical threads created and managed by the multi-threading library.

The output is an *n* x *m* matrix *O* representing the edge pixels computed by the team of threads.
The division step is responsible for partitioning the input and output matrices into blocks which are to be assigned to the available team of threads. The partitioning can be implemented in different ways. For instance, the image is partitioned into blocks each containing a number of rows (or columns). Also, the partitioning can be performed through rows and columns

simultaneously resulting into rectangular blocks or checker-board partitions. This latter partitioning scheme is also known as tiling. Figure 1 shows an example of rows blocking and one possible assignment of threads to the blocks.

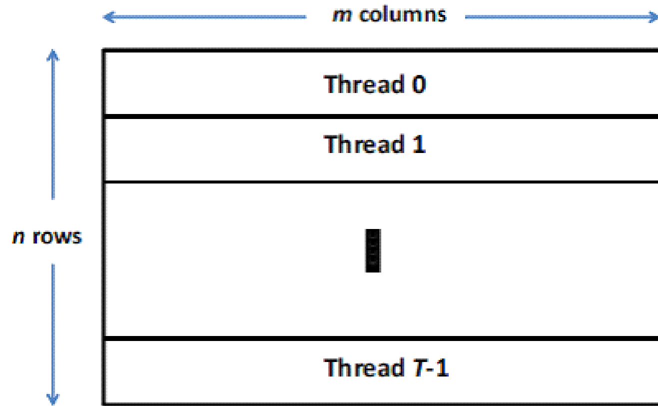

Figure 1. An image with *n* rows and *m* columns partitioned horizontally into N_B blocks. A number of threads, *T* are assigned to the blocks in the order shown, $T=N_B$.

The assignment step is responsible for allocating each pair of blocks (I_j,O_j) to one of the threads. Actually, this scheduling problem has an exponential number of different assignments. We will show how the threading library controls this assignment either by the operating system affinity scheduling or by controlling the environment variables of the OpenMP library at runtime to specify a specific scheduling strategy.

The last step in Algorithm 2 is the actual Compute step that calculates the gradient pixels and updates the output edges image. In this step, each thread computes the gradient at each pixel of the assigned block(s) j and updates the corresponding block(s) Oj in the output matrix. In the next section, we evaluate the implementation of Algorithm 2 on a quad-core processor.

4. DESCRIPTION OF THE OPENMP IMPLEMENTATION

We used the standard OpenMP library to implement our parallel multi-threaded algorithm, i.e., Algorithm 2. The OpenMP directives are used to partition the image among a team of threads in addition to determining the ordering of assignment of the partitions to the threads. The code in Listing 3 below shows the function 'sobel' that takes an *N1* x *N2* input matrix *a* containing the image pixels and computes the gradient in matrix *b*, of the same size as *a*. Note that both the gradients in x- and y-axis are merged in one function as opposed to common libraries such as OpenCV and CImg.

The method of parallelization is summarized as follows. First, a team of threads is created using the OpenMP directive *#pragma omp parallel* as shown in lines 8 and 9 where the number of threads, *nthrd*, is specified. The task of computing the weighted sums *sx* and *sy*, and the gradient values *b[i][j]* at line 19 is divided among the threads using a 'for' work-sharing OpenMP directive. Since the 'for' directive is inserted immediately before the outer loop, the partitioning is applied to the outer loop only. This means the image will be divided into blocks of rows, as shown in Figure 1. Note that both the creation of the team of threads and the work-sharing directives are combined in one directive (see line 8). This combination reduces the parallelization overhead and hence produces better performance than declaring each directive separately [6]. While the 'for' work-sharing directive divides the work among the team of threads, it can also

control how the partitions of the outer loop are allocated to the threads by adding the 'static' clause as shown in line 8 which means the scheduling is static and defined before the threads start to execute.

```
1  \label{listing1}
2  void sobel(DATA **a, DATA **b, int N1, int N2,  int nthrd, int bs)
3  {
4  int i,j,l,k;
5  int mx,my,bs;
6  int mx[]={1,0,-1,2,0,-2,1,0,-1};
7  int wy[]={1,2,1,0,0,0,-1,-2,-1};
8  #pragma  omp parallel for schedule(static, bs) num_threads(nthrd) \
9           default(none) shared(a,b,N1,N2,bs) private(i,j,mx,my,mx,my)
10 for(i=1;i<N1-1;i++)
11   for(j=1;j<N2-1;j++)
12     {
13          gx= a[i-1][j-1]*mx[0]+a[i-1][j]*mx[1]+a[i-1][j+1]*mx[2]\
14                   +a[i][j-1]*mx[3]+a[i][j]*mx[4]+a[i][j-1]*mx[5]\
15                   +a[i+1][j-1]*mx[6]+a[i+1][j]*mx[7]+a[i+1][j+1]*mx[8];
16          sy= a[i-1][j-1]*my[0]+a[i-1][j]*my[1]+a[i-1][j+1]*my[2]\
17                   +a[i][j-1]*my[3]+a[i][j]*my[4]+a[i][j-1]*my[5]\
18                   +a[i+1][j-1]*my[6]+a[i+1][j]*my[7]+a[i+1][j+1]*my[8];
19          b[i][j]=abs((int)sx)+abs((int)sy);
20     }
21 return;
22 }
```

Listing 3: The C++ code including the OpenMP directives.

The listing shows one possible scheduling which is defined as 'static'. In this type of scheduling the size of each block is specified, *bs*, and the blocks of the outer loop are assigned to the team of threads in a round-robin fashion. However, if the block size is not given, the outer loop iterations will be divided evenly among the threads so that each thread is assigned one partition only. Other scheduling strategies can be used in OpenMP, see [6] for more details. Although a scheduling strategy is specified in the code, the order of assigning the threads to the physical cores can affect the performance. This is because the CPU micro-architecture dictates the arrangement of the caches and which cores share which data cache. This effect is shown in the next section. We will also show the numbering of the physical cores can be advocated using the core affinity feature in the OpenMP library.

5. Performance Evaluation

In this section, we evaluate the performance of the multi-threaded implementation of the Sobel gradient computing algorithm. In the experiments, we used two multi-core processors: a quad-core processor and a dual-core processor.

5.1 Experimental Setup

We used the OpenMP threading library, which is a standard for parallelizing codes on multi-core processors and SMP machines. In our implementations of both the sequential and parallel gradient computing algorithms in this paper, we used the GNU GCC g++ 4.8.2 compiler to generate the executable codes, which run under the operating system Ubuntu Linux 14.04 64-bit.
We tested all the C++ codes on **(i)** a Dell notebook with Intel Core i7 2670QM CPU @ 2.20 GHz. This processor has four physical cores where each one can run two logical threads simultaneously, a feature known as hyper-threading, and **(ii)** a Dell desktop PC with an Intel Core

2 Duo E6750 processor running @ 2.6GHz with two physical cores that can execute only two logical threads; the hyper-threading technology is not implemented in this processor.

The run times reported in our results measure the execution times of computing the image gradient using Sobel filter using the sequential and parallel algorithms given in Listing 1 and Listing 2, respectively. The OpenMP implementation of the parallel computing algorithm is implemented in Listing 3. We used the OpenMP timing function *omp_get_wtime()* to measure the execution time in seconds to all the reported experiments. The run times are the average of 100 runs.

The test images we used have resolutions of 256 x 256, 512 x 512, 1024 x 1024, and 2048 x 2048. The pixel values are of type integer.

5.2 Results and Discussion

5.2.1 The effect of the number of used cores

First, we show the effect of using different number of physical cores on the performance of the multi-threaded algorithm. For a fixed number of cores, we used an equal number of threads for the execution. That is, one thread for each core. Figure 2 shows the execution time of the sequential and multi-threaded implementations. The experiments are done on a range of images with sizes 256 x 256, 512 x 512, 1024 x 1024, and 2048 x 2048. It is clear from the figure of each image size that the execution time decreases as we increase the number of cores. However, when using one thread and one physical core (out of the available four), the execution time actually increased by around 9% due to the overhead of creating and managing the thread by OpenMP runtime. However, the percentage of this overhead decreases as the image size increases. By using two cores and more, the total execution time increases as the image size increases, and the speedup obtained is increasing and get closer to the number of cores used, as shown in Table 1.

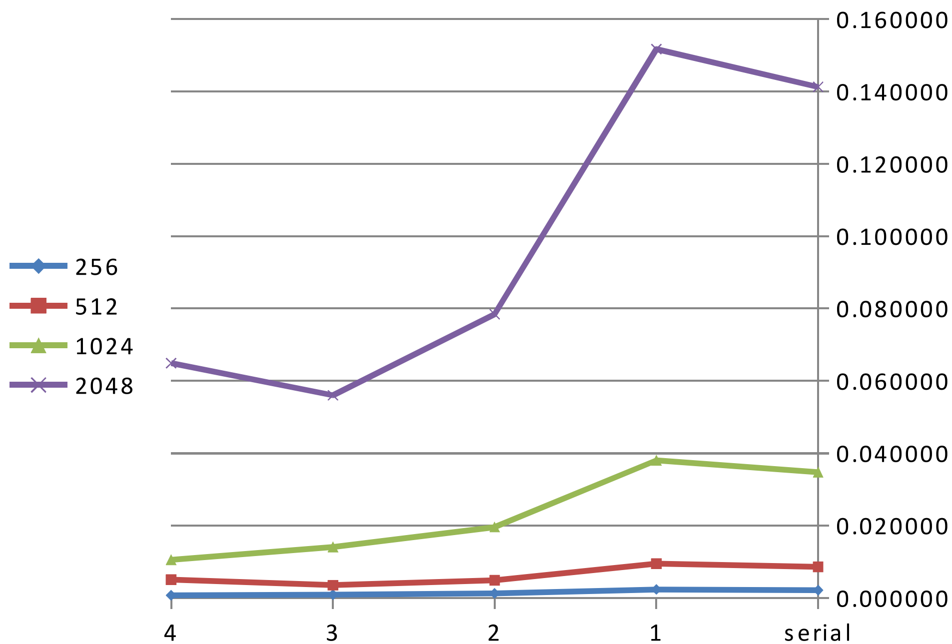

Figure 2. The execution time in seconds of the multi-threaded algorithm using 1, 2, 3, and 4 cores (x-axis) of the Intel Quad-core i7 processor. The results are for square images of sizes 256, 512, 1024, and 2048.

Table 1: The execution time in seconds and the speedup of the OpenMP implementations relative to the sequential implementation. The number of physical cores used equals the logical threads used. The image size is 2048 x 2048.

#physical cores	Execution Time(sec)	Speedup
serial	0.138168	1.00
1	0.151348	0.91
2	0.078349	1.76
3	0.055968	2.47
4	0.042086	3.28

Figure 2 also shows that using one thread for parallelization is not worth due to the effect of the parallelization overhead. This shown in each curve of the figure as the speedup is less than one.

5.2.2 The Effect of Increasing the Number of Logical Threads

For a given number of physical cores used in execution, we run our codes with varying number of threads to see the effect on performance. We divided the input images into a number of blocks equals the same number of the created threads. Each thread is assigned a block in the image. Figure 3 and Figure 4 show the speedup measurement of the multi-threaded implementations on both the Core 2 Duo processor and the core i7 processor on a range of image sizes, 256 x 256, 512 x 512, 1024 x 1024, and 2048 x 2048. In this experiment, all the available cores in each processor are employed in the execution. The allocation and scheduling of the threads on the cores are left to the default strategy in OpenMP (i.e., static, see next section). The numbers of logical threads are varied from 1 to 16. It can be seen from Figures 3 and 4 that the maximum speed-up approaches the expected ideal value as the image size increases. These speedups equal 4.0 for the Quad-core Core i7 processor and 2.0 for the Dual-core Core 2 Duo processor. It can be concluded from the speedup curves that for image sizes larger than 256x 256 using the maximum number of logical threads that can run simultaneously by the processor (eight in case of the Core i7 and 2 for the Core 2 Duo) is recommended to get the full parallelization speed of the multi-threaded Sobel gradient computations proposed in this paper. The main reason for the non-contention of the threads running on the same core simultaneously is that the algorithm accesses the pixels sequentially as it is laid out into memory. This access pattern ease the mission of the data cache pre-fetchers, which can easily predict the access pattern and help in pre-fetching the cache line that are going to be used in the next loop iterations. These results in reduced cache miss rates and hence enhanced performance.

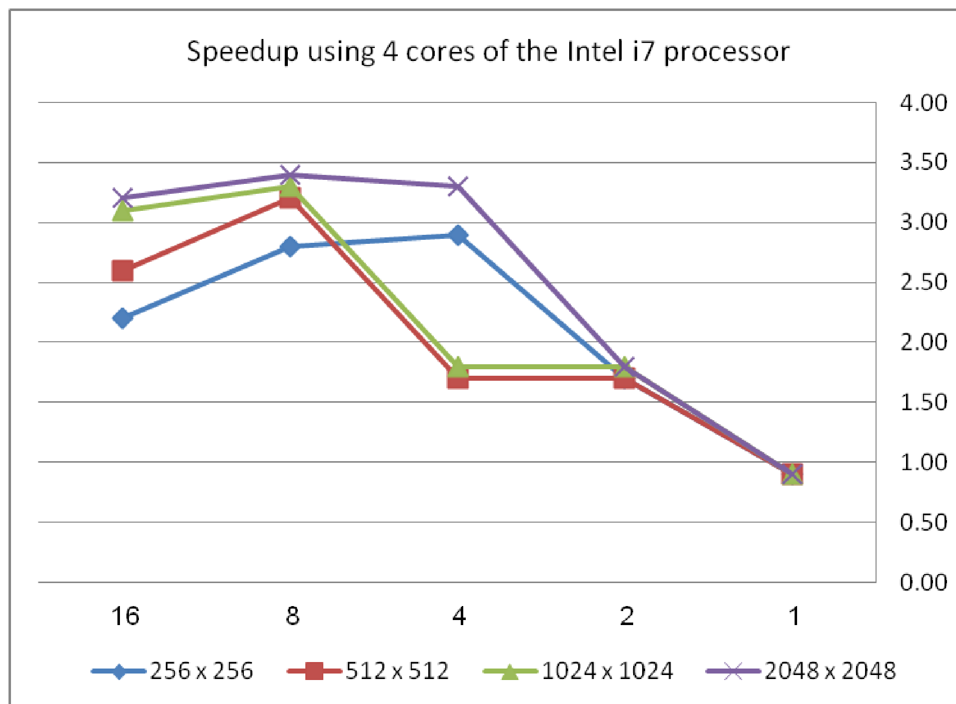

Figure 3. The speedup of the multi-threaded algorithm compared with the sequential algorithm using different number of logical threads (x-axis) and four cores of the Intel Quad-core i7 processor. The results of four different image sizes are shown.

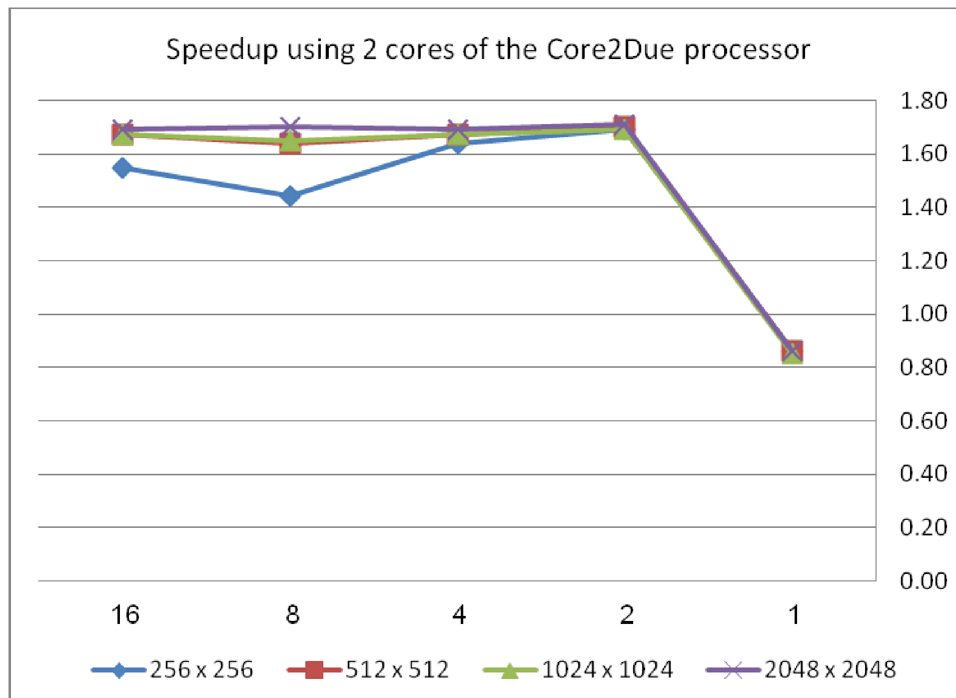

Figure 4. The speedup of the multi-threaded algorithm compared with the sequential algorithm using different number of logical threads (x-axis) and two cores of the Intel Core2Due processor. The results of four different image sizes are shown.

Table 2: The speedup of comparing the execution time of our OpenMP parallel codes with the sequential code. The speedup is measured with varying: the number of logical threads used (Column 2), how the threads are assigned to the physical cores (Column 3) numbered 0,1,2 and 3, the number of physical cores used (Column 1) according to the thread assignment criteria, the execution time in Micro-sec (Column 4) and the speedup (Column 5). The image size is 512 x 512. The average sequential execution time was 8560 Micro-sec.

#physical cores	# Logical Threads	Thread Assignment	Execution Time	Speedup
1	1	'0'	9434	0.91
	2	'00'	9418	0.91
2	2	'01'	4877	1.75
	2	O.S.	4851	1.76
	4	'0101'	4887	1.75
	4	'0011'	4901	1.75
4	4	'0123'	2613	3.28
	4	'0213'	2700	3.17
	4	O.S.	2619	3.27
	8	'01230123'	2674	3.20
	8	'00112233'	2661	3.22
	8	O.S.	2694	3.18

Table 3. The speedup of the performance of our OpenMP parallel code due to changing the number of logical and physical cores, and how the threads are assigned to the physical cores. Image size is 1024 x 1024. The average sequential execution time was 35810 Micro-sec.

#physical cores	# Logical Threads	Thread Assignment	Execution Time	Speedup
1	1	'0'	37909	0.94
	2	'00'	38421	0.93
2	2	'01'	19829	1.81
	2	O.S.	19611	1.83
2	4	'0101'	19851	1.80
	4	'0011'	19682	1.82
4	4	'0123'	10578	3.39
	4	'0213'	10934	3.27
	4	O.S.	11968	2.99
4	8	'01230123'	10573	3.39
	8	'00112233'	10644	3.36
	8	O.S.	10384	3.45

5.2.3 The Effect of Thread Scheduling Criteria

After dividing an image into blocks and assigning them to threads, the next step is allocated the threads to physical cores for execution. Actually, the allocation of the threads and their scheduling may be determined completely by either the underlying operating system or the OpenMP runtime. Definitely, the choice of the scheduling strategy affects the performance.

In general, if we chose four threads to execute the algorithm, and the image is divided evenly into four blocks, there will be a maximum of 4! ways to assign the threads to the blocks. For example, if we have four cores 0, 1, 2, and 3, and we have four threads, we denote the assignment pattern '0123' to show that the four threads are assigned to cores 0, 1, 2, and 3, in the same order. This type of scheduling is called 'round-robin' scheduling. Also, the assignment pattern '0213' means that the first thread (block) is assigned to core0, the second to core2, the third to core1, and finally the last one to core3.

There are cases where the number of available cores is less than the number of threads. In this case, each core will be assigned more than one thread to execute. For example, if the number of available physical cores is two, then the assignment '0101' means that threads 0 and 2 are assigned to core 0 while threads 1 and 3 are assigned to core 1 (round-robin fashion). Table 2 shows the execution time and speedup comparison of the OpenMP implementations using different number of logical threads with varying the number of physical cores and the thread assignment pattern.

Assigning a thread to a specific core is called processor binding. We controlled the thread assignment to the physical core through the so-called processor affinity feature. In Linux, this feature can be enforced through setting the environment variables *OMP_PROC_BIND* and *GOMP_CPU_AFFINITY*, see [7] for details.

Note that if processor affinity is not specified the assignment of the threads to the cores is done at runtime by the operating system, as we indicate by using the pattern O.S. in the third column of the table. In this scheduling strategy, neither thread binding nor core affinity are specified.

6. Related Works

The computing of the image gradient is the most time consuming part of Sobel algorithm to detect edges in images. Sobel algorithm has been implemented on shared-memory and distributed-memory machines. On distributed memory systems, the image is divided into blocks that are distributed on distant processor where inter-node communications are used to interchange data and results. For example, on a Beowulf cluster the standard message-passing interface MPI was applied to parallelize the Sobel algorithm using the inter-communication between the nodes of the system [4]. On multi-core processors, a shared-memory system, an algorithm for implementing Sobel operator using MPI library to distribute data and collect results between the cores of the CPU was implemented in [3]. This work using message passing to communicate through shared memory. Using the MFC multi-threading library, a parallel implementation of Sobel color image detection on an Intel multi-core processor is presented in [5].

In [9] a number of image processing kernels, including the image filter, were hand optimized on exploiting the hardware architectural features such as vectorization. They achieved near 4.0 speedup on Intel core i7 processor when the image size approaches 2048 x 2048. In [10] the authors studied the effect of vectorization feature of Intel and ARM processors on a number of image processing kernels including Sobel filtering. They obtained a speedup little above 2 for a 3264 x 2448 image on an Intel core i7 processor.

In this paper, we present a parallel implementation of Sobel algorithm on a multi-core Core i7 Intel processor using the C++ language and the standard threading library, the OpenMP. This paper differs from the above related works in that we studied the effects of different factors on the performance of Sobel filter, namely: varying the number of logical threads used in the parallel team and the assignment of threads to the physical cores using different strategies. Other related

works are studying the variation of the number of threads on performance without studying the affinity of the processors that we consider.

7. CONCLUSIONS

We have presented a parallel multi-threaded algorithm to compute the Sobel filter using the standard OpenMP threading library for multi-core CPUs. The image data is divided based on the layout of the pixels in memory in order to optimize the use of cache prefetchers. We studied the effects of increasing the number of logical threads used for parallelization in addition to their mapping policies to the available physical cores. Different core assignments are tested and their impact on the performance is reported. The best results obtained when using eight logical threads assigned to four different physical cores is a speedup of 3.5 compared to the sequential algorithm, and 3.7 compared to using one thread. The ideal speedup is 4 which is not obtained due to the parallelization overhead to manage the threads on the physical cores.

Since the image data is laid out in memory in row-major order, and each thread is assigned a block of the input image composed of contiguous rows, the pre-fetching feature of the data cache is best exploited to hide the cache miss penalty and reduces thread stalls. The hyper-threading feature implemented inside each core has a performance benefit for larder size matrices since the amount of the processed exceeds the overhead to switch between the threads on the same core. Our ongoing research is to combine vectorization with OpenMP benefits to get higher performance on different image kernel operations.

ACKNOWLEDGEMENT

This research is supported by Beirut Arab University.

REFERENCES

[1] Gonzalez, Rafael C. “Digital Image Processing”, Pearson Education, Inc., publishing as Prentice Hall, 2008.

[2] SOBEL, I., An Isotropic 3×3 Gradient Operator, Machine Vision for Three – Dimensional Scenes, Freeman, H., Academic Pres, NY, 376-379, 1990.

[3] Noor E. Abdul Khalid and et. al. "Analysis of parallel multicore performance on sobel edge detector." In *Proceedings of the 15th WSEAS international conference on Computers*, Nikos Mastorakis, ValeriMladenov, Zoran Bojkovic, FragkiskosTopalis, and KleanthisPsarris (Eds.). World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 313-318, 2011.

[4] Nazleeniharon, Ruzaini Amir, IzzatdinA,Aziz, Low Tan Jung, SitiRohkmah, " Parallelization of Edge Detection Algorithm using MPI on Beowulf Cluster", Innovations in Computing Sciences and Software Engineering., pp 477-482, 2010

[5] A. Kamalakannan and G. Rajamanickam, "High Performance Color Image Processing in Multicore CPU using MFC Multithreading," International Journal of Advanced Computer Science and Applications, Vol. 4, No. 12, 2013

[6] OpenMP Application Program Interface. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf, downloaded Mar 22, 2016.

[7] Bind threads to specific CPUs. "https://gcc.gnu.org/onlinedocs/libgomp/GOMP_005fCPU_005fAFFINITY.html", retrieved Mar 22, 2016.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011

[9] D. Kim, V. W. Lee and Y. K. Chen, "Image Processing on Multicore x86 Architectures," in *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 97-107, March 2010.

[10] Gaurav Mitra, Beau Johnston, Alistair P. Rendell, Eric McCreath, and Jun Zhou (2013). Use of SIMD Vector Operations to Accelerate Application Code Performance on Low-Powered ARM and Intel Platforms. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum* (IPDPSW '13), Washington, DC, USA, 1107-1116, 2013

AUTHOR

Ahmed S. Zekri

Received both B.Sc. and M.Sc. in computer science from Department of Mathematics & Computer Science, Faculty of Science, Alexandria University. He has a Ph.D. degree in computer science and engineering from The University of Aizu, Japan 2008. Presently he is an Assistant Professor at Beirut Arab University, on leave from Alexandria University. His research interests include parallel and distributed computing, performance evaluation of parallel image processing and cryptography algorithms, job scheduling and management in cloud data centers, and high performance computing.