Working towards Efficient Parallel Computing of Integral Images on Multi-core Processors

Nan Zhang
Department of Computer Science and Software Engineering
Xi'an Jiaotong-Liverpool University
Suzhou, China P.R.
nan.zhang@xjtlu.edu.cn

***Abstract*—This paper presents a novel multi-threaded parallel algorithm for computing integral images on multi-core processors. At each stage of the design, we evaluated several approaches, well-established as well as newly proposed. According to the results of the evaluations and our analysis the best suitable solutions have been identified, from which the parallel algorithm was synthesised. Tests were made and showed that on systems with fast FSB, e.g., 1333MHz, when running with two threads bound on distinctive processors sharing the same L2 cache the implementation of the algorithm could run at a speed twice as fast as that of the best known sequential reference. Tests also revealed that L2 cache size, system bus speed, micro-architecture and topology of the processor all had their share in determining the performance of the implementation relative to the sequential reference.**

***Index Terms*—Parallel processing, Multi-core computing, Image processing, Integral image.**

I. Introduction

Integral image, formerly known as summed area table [1], was first introduced into the area of computer vision by Viola and Jones [2] in their face detectors to reduce computational complexity. With the use of the integral image, Haar-like features can be computed at any scale in constant time. This improvement made their detector be able to run at a higher frame rate comparing to previous approaches. Following this trend, the same authors later applied the integral image to their classifiers to detect pedestrians [3]. Bay et al. built their image feature extraction and representation algorithm SURF (Speeded-Up Robust Features) [4] upon the integral image and gained significant speedup than its predecessor SIFT (Scale Invariant Feature Transform) [5]. In their algorithm responses of box filters and Haar filters are computed on the integral image rather than directly on the original picture. This makes the computation of the responses be able to complete in constant number of operations regardless of the size of the interested area.

The integral image has become a useful and popular vision tool which helps to accelerate certain image feature computation algorithms. Because of that, integral image itself has recently attracted much attention, especially with the emergence of non-traditional computing platforms. Kisačanin presented his work [6] in which the computing of integral images is optimised for a media processor used in embedded systems. Messom and Barczak [7] designed an algorithm for computing integral images on GPUs, and tested their design on an ATI HD4850. Terriberry et al. [8], in their endeavour to develop a GPU-based SURF algorithm for NVIDIA cards, evaluated three GPU-based methods for computing the integral image, including a novel approach they invented.

The purpose of this paper is to find an efficient multi-threaded algorithm for computing integral images on multi-core processors. The current author has investigated, evaluated and compared quite a few methods, well-established as well as newly proposed. Experiments have shown that the task of designing an efficient parallel algorithm of computing the integral image for multi-core processors is far from a trivial one, for several platform-dependent factors, such as bus speed, cache effect, micro-architecture of the computing unit and topology of the processor, all have their share in selecting the best suitable approach. Some conclusions drawn from this study are applicable to parallel computing on symmetric multi-core processors in general.

II. Sequential reference

An integral image $I_\Sigma(x, y)$ at the position (x, y) of image I is to sum up all the pixels within a rectangular region formed by the origin and pixel $I(x, y)$. Formally, this can be defined by the formula

$$I_\Sigma(x, y) = \sum_{i=0}^{i \le x} \sum_{j=0}^{j \le y} I(i, j). \tag{1}$$

It is commonly known from the literature, e.g., [6], that there are three sequential algorithms to compute the integral image.

The first method first sums up pixels in the first row. The integrated value for any other pixel is calculated as the sum of all the preceding pixels in the same row, the pixel itself and the integrated value of the pixel at the same column one row above. For an image I of size w (width) $\times$ h (height), this can be expressed as

$$\begin{aligned} I_\Sigma(x, 0) &= \sum_{i=0}^{i \le x} I(i, 0), && x \in [0, w-1] \\ I_\Sigma(x, y) &= \sum_{i=0}^{i \le x} I(i, y) + I_\Sigma(x, y-1), && x \in [0, w-1] \\ & && y \in [1, h-1]. \end{aligned} \tag{2}$$

If we ignore the overhead of updating loop index and pixel position, this computation takes $2wh$ addition operations.

978-1-4244-6349-7/10/$26.00 ©2010 IEEE

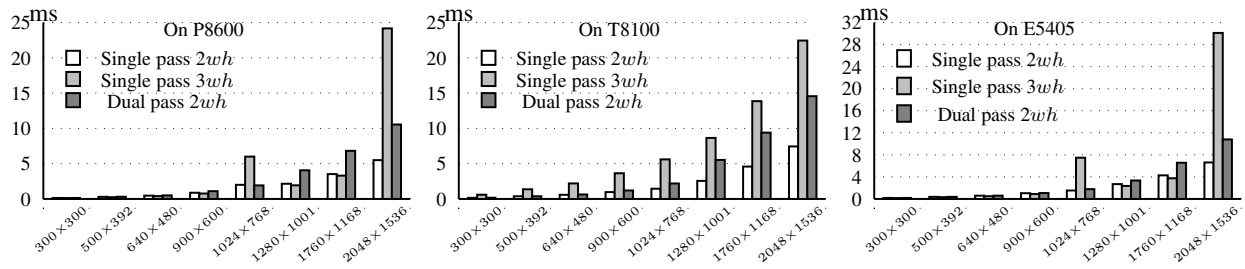

Fig. 1: Performance test results of the three commonly known serial algorithms for computing the integral image on the three testing machines.

The second method first computes the integrated values for pixels in the first row and then the first column. The integrated value for any other pixel is calculated as the sum of all the pixels in the same row up to the pixel in question and the integrated value of the pixel one row above at the same column subtracting the integrated value of the pixel at one row above and one column before. For an image I of $w \times h$ this can be defined as

$$\begin{aligned} I_\Sigma(x,0) &= \sum_{i=0}^{i\le x} I(i,0), && x \in [0, w-1] \\ I_\Sigma(0,y) &= \sum_{j=0}^{j\le y} I(0,j), && y \in [0, h-1] \\ I_\Sigma(x,y) &= I_\Sigma(x-1,y) + I_\Sigma(x,y-1) && x \in [1, w-1] \\ &\quad -I_\Sigma(x-1,y-1), && y \in [1, h-1]. \end{aligned} \tag{3}$$

Ignoring the overhead of computing the indexes, this algorithm takes about $3wh$ additions and subtractions.

The third approach differs from the previous two in that the computation is done in two separate double loops. It is to sum up pixels along the rows, followed by a summation along the columns. If I^r_Σ is used to denote the intermediate results from the row-wise summation, for an image I of $w \times h$, this computation can be defined as

$$\begin{aligned} I^r_\Sigma(x,y) &= \sum_{i=0}^{i\le x} I(i,y), \qquad x \in [0, w-1],\ y \in [0, h-1] \\ I_\Sigma(x,y) &= \sum_{j=0}^{j\le y} I^r_\Sigma(x,j), \quad x \in [0, w-1],\ y \in [0, h-1]. \end{aligned} \tag{4}$$

This method like the first one needs $2wh$ additions but, of course, incurs more overhead.

We have implemented all the three methods to compare their performance. But before presenting the results we shall first state the configurations of the testing platforms. The tests were made on three machines, two of which were equipped with single Intel Core Duo processor, and the other one double Intel Xeon Quad Core processors. All the systems were running Linux. Their detailed relevant information are summarised in Table I.

To measure the execution times we have adopted the K-best scheme [9], where the parameters were set as $M = 20$, $K = 5$, and $\epsilon = 0.001$. The compiler we used was the Intel C/C++ compiler 11.0 for Linux with optimisation options -O3 (optimisation for speed) and -ipo (whole program optimisation) enabled. The time durations were measured in milliseconds (ms). All the durations were wall clock times. The two-dimensional image data were stored in row-major order in memory.

TABLE I: Hardware and software configurations of the three machines used in the testing.

CPU	L2 Cache	FSB	Kernel
T8100@2.1G	3MB	800MHz	2.6.24 32-bit
P8600@2.4G	3MB	1066MHz	2.6.27 64-bit
E5405@2.0G	4×6MB	1333MHz	2.6.27 64-bit

The results of the testings for the three sequential approaches are presented in Fig. 1

The results are not correlated to their computational complexity, but are determined by the efficiency of the cache utilisation of each method. From the results we may conclude that the approach of the single-pass with $2wh$ additions is the best solution among the three. The dual-pass method clearly needs more computation time than the other two. Although the approach with $3wh$ additions outperformed the one with $2wh$ in a few cases, its results were far from stable. Another reason against the $3wh$ approach is that the associative law for addition does not generally hold for floating point arithmetic, which will cause the $3wh$ approach less precise. Therefore, the $2wh$ single-pass solution is chosen as the sequential reference from which ratios of parallel speedup are later derived. The dual-pass method is selected as the basis on which the parallel algorithm is developed.

III. Parallel computing of the integral image

The parallel algorithm is built on the dual-pass sequential computation where multiple threads are created to sum up the rows and then the columns in parallel. The algorithm takes an argument t which is the number of co-existing threads to integrate the rows and columns. To get better performance, instead of blocking the main thread of the program and creating t working threads to do the summation , we only create $t-1$ working threads while assigning an equal fraction of the task to the main thread .

As for the t co-existing threads, they can be left to the Linux kernel for processor assignment. However, experience taught us that on shared last level cache multi-core processors binding the threads to distinctive processing cores gives better performance. So in the algorithm we explicitly bind thread i

to processing core $(i \bmod c)$[1], where $i \in [0, t-1]$, c is the number of physical cores available in the system, and in the test we had $t = c$.

For t threads to complete the row-wise summation in parallel, the image is partitioned equally into t sub-regions along its rows, and each of the sub-regions is assigned to a thread.

The task of summing up a single row of pixels is a classic problem known as prefix sum or scan.

The simplest way (Fig. 2a) of doing it is to add the value of each pixel to its successor down to the last one. However, there are two drawbacks with this solution. First, this method puts a heavy usage on the system bus even in the case of a medium-sized image. We have tested its bus utilisation ratio using Intel VTune™for Linux by

$$\text{bus ratio} = \frac{2 \times \text{BUS\_TRANS\_ANY.ALL\_AGENTS}}{\text{CPU\_CLK\_UNHALTED.BUS}}, \quad (5)$$

where the numerator is the count of bus cycles used for transferring bus transactions of any type and the denominator is the count of total bus cycles consumed, and it was found that for the 900×600 image used in the testing the ratio was more than 70% on the system equipped with the P8600 whose FSB was 1066MHz. This figure is obviously undesirable whereas a ratio above 60% is considered bad. Later we will see, in fact, that system bus speed is a bottleneck to the efficiency of the parallel algorithm. Secondly, there is true data dependency in the method where the result of an instruction is required for subsequent instructions. Intel Core microprocessor has three distinctive ALUs in each core [10]. However the data dependency found in the method prevents them to be fully loaded.

To address the dependency problem we developed a novel multi-pass approach where the iteration is unrolled by a degree of 6, and the sequence of the addition operations are re-arranged so that parallel processing is possible. Fig. 2b shows the procedure. Take the first 6 pixels a_0 to a_5 as an example. In the first pass, the value of a_0 is added to the value of a_1, the value of a_2 is added to that of a_3 and the value of a_4 is added to a_5. As these operations do not dependent on each other they can be done in parallel by the ALUs. In the second pass a_1's updated value is added to a_2 and a_3 in parallel. In the third pass, a_3's updated value is added to a_4 and a_5. Finally the updated value of a_5 is added to a_6 to carry the sum to the next round of computation. Comparing to the serial method, this approach enhances parallelism but requires slightly more addition operations.

To address the bus utilisation problem we may consider to load multiple pixel data from memory into registers where they are manipulated and then written back to memory in one go. Fig. 2c illustrates such a solution we developed with the help of Intel SSE2 instructions. This method unrolls the loop by a degree of 4 and loads 4 pixel data into xmm0 register in one go. The content of the xmm0 register is then copied into xmm1 register, where the values are left-shifted by 4 bytes and then added back to the xmm0 register. This repeats two more times until we have the correct values in the xmm0 register, which are then written back to memory. This method relieves the heavy burden the first approach places on the system bus but requires much more operations.

Finally we cannot ignore the classic prefix sum algorithm intended for parallel processing presented by Blelloch [11]. The algorithm comprises a up-sweep stage and a down-sweep stage, as illustrated in Fig. 2d on a 9-element array. The up-sweep and down-sweep step are presented in Algorithm 1 and 2 respectively. Comparing with the ones presented in [12], two minor mistakes were corrected and certain inefficiencies were removed from our version.

Algorithm 1 The up-sweep step for an array of w elements.

Require: An array of w elements, denoted by $a_0[0]$ to $a_0[w-1]$.
Ensure: Intermediate results stored in buffers a_d, where d is from 1 to $\log_2 w$

```
for d ← 1; d ≤ log2 w; d ← d + 1 do
    for i ← 0; i ≤ w/2^d − 1; i ← i + 1 do
        a_d[i] ← a_{d−1}[2i] + a_{d−1}[2i + 1]
    end for
end for
```

Algorithm 2 The down-sweep step for an array of w elements.

Require: The intermediate results produced by the up-sweep step.
Ensure: The array with all its elements summed up.

```
for d ← log2 w − 1; d ≥ 0; d ← d − 1 do
    for i ← 1; i ≤ w/2^d − 1; i ← i + 1 do
        if i mod 2 ≠ 0 then
            a_d[i] ← a_{a+1}[i/2]
        else
            a_d[i] ← a_d[i] + a_{d+1}[i/2 − 1]
        end if
    end for
end for
```

We have implemented all the four methods to evaluate their performance. Various code optimisation techniques were applied, such as moving loop-invariant code out of the loop, replacing base 2 exponential by shifting and eliminating if-else branching, to give each method a favourable result. As the results have shown (Fig. 3a-c) that the method of the enhanced parallelism yielded the best result. The strictly sequential integration came a little behind because of its simplicity of computation. The other two performed notably poorer [2].

To sum up the output of the row-wise integration along the column by multiple threads the whole area of the pixel data can either be partitioned column-wise or row-wise . To sum up the data along the column a naive implementation will demonstrate bad access patterns with poor locality. The same column-wise summation can be done with better locality by applying a trick known as eliminating strided access [13] where a group of data items in the same row is added to data items at one row below. An extra benefit of this altered access pattern is it permits SSE additions to be applied to the situation.

[1]The modulo operation is used to allow the number of threads employed exceeding the number of physical cores in a system with hyperthreading support.

[2]On the 32-bit system, in programming the approach with the SSE2 support we had to use the un-aligned load and store which are slower than their aligned counterpart used in the implementations for the 64-bit systems.

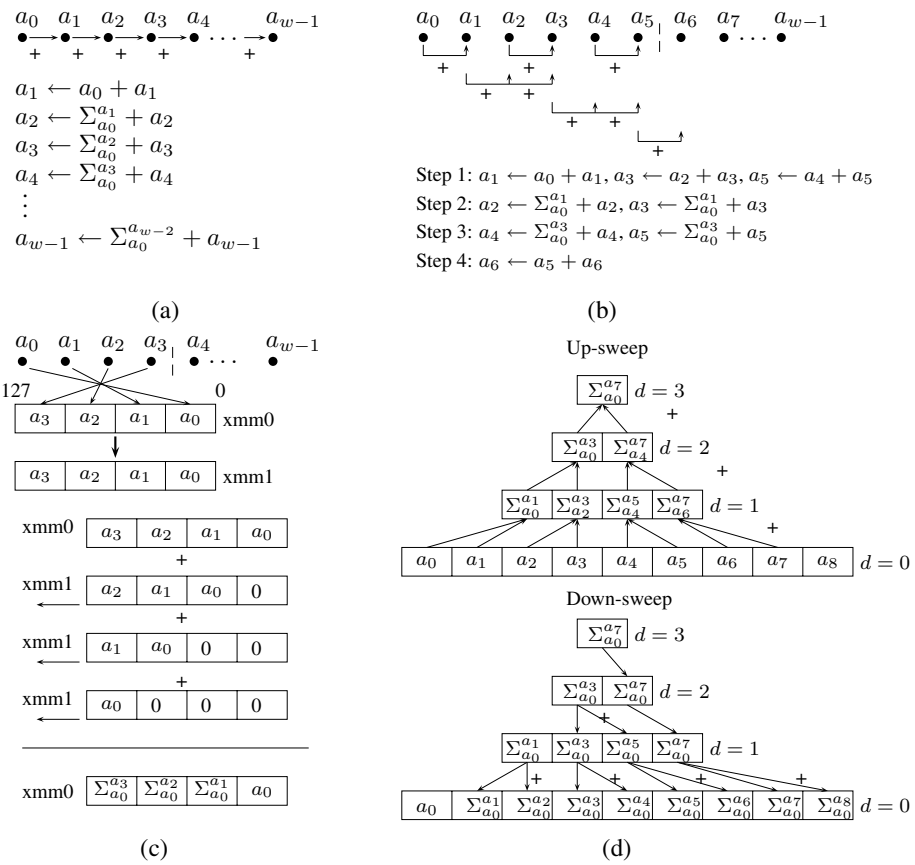

Fig. 2: Four different ways of summing up one row of pixels.

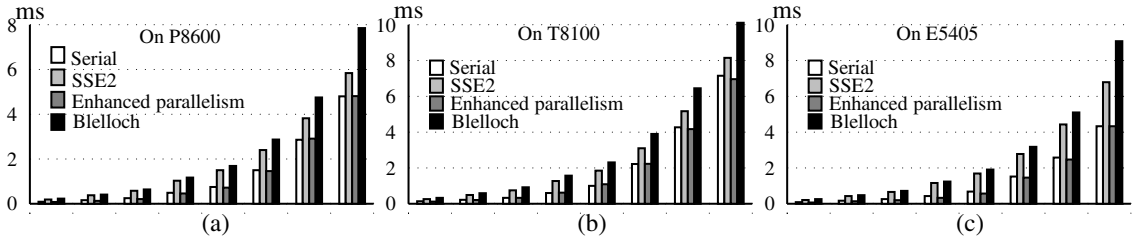

Fig. 3: Performance comparison of the four methods for parallel row-wise summation.

IV. Test and performance

Synthesising the results of the discussion above we choose the best approaches for computing the integral image in parallel on multi-core processors. Regarding to the overall strategy of thread management we choose to create $t - 1$ threads while assigning the main thread an equal portion of task. During the computation all the t threads are bound to distinctive processing cores. In the phase of the row-wise integration we use the method of the enhanced parallelism (Fig. 2b) and in the stage of the subsequent column-wise integration we choose the column-wise approach with the strided access eliminated .

We have tested the speedup ratio of the parallel algorithm with respect to the sequential reference we have selected. The results on all the three systems are shown in Fig. 4c, from which we can see that speed of the system bus is a key to the success of the algorithm. In the processing of the 900×600 image on the E5405 we observed superlinear speedup which must have been caused by the enhanced parallel addition in the row-wise summation.

Another factor which has great impact on the performance of the algorithm but has not been discussed so far is the topology of the processor. Fig. 4a-b illustrates the topologies of the processors we used in the tests.

We have tested the speedup ratios of the algorithms with the two fastest row-wise summation methods on different combinations of the cores in the system equipped with double Intel Xeon E5405. The results in Fig. 4d-e have shown that

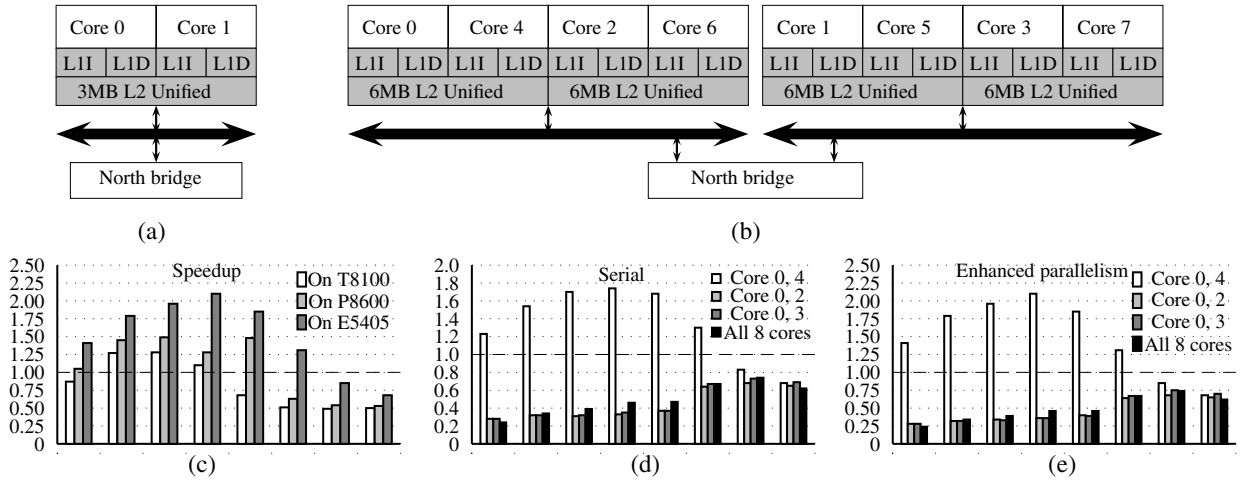

Fig. 4: Topologies of the processors used in the tests and their influence on the performance of the parallel algorithm, (a) for the T8100 and the P8600 and (b) for the double E5405. The numeric identities of the cores are assigned by Linux operating system. Subfigure (c) shows the speedups on the three testing machines.

only when running on two cores sharing the same L2 cache did the computations show efficient speedup, and in this respect using more cores had no help. Time spent on populating cold caches made the algorithm in practice inefficient.

V. Discussion and conclusion

In this paper we have presented our work in pursuing of an efficient parallel algorithm for computing integral images on multi-core processors. The algorithm we propose is a dual-pass procedure where the pixel data are summed up along the rows followed by a summation along the columns. Both the summations are computed by multiple threads in parallel. Concerning the overall strategy of thread management, the row-wise summation and the column-wise summation each has several options available. Through the analysis and the evaluation we have made the choices to maximise the efficiency of the design. The reasons behind the choices were stated.

In regard to the speedup ratio to the sequential reference the algorithm was tested using images of various sizes. The results showed that the algorithm performed much better on the system with a faster FSB, e.g., 1333MHz, than on a system with a slower FSB, e.g., 1066MHz or 800MHz. The row-wise summation method we have chosen simple though uses the system bus heavily. We tested the method using Intel VTune^{TM}and found that the method's L2 cache missing rate (event MEM_LOAD_RETIRED.L2_LINE_MISS) grew much faster than the increasing rate of the image size, a phenomenon that was not found in the sequential computation. Through testing we have eliminated false sharing as a possible cause to the problem. So, we think this is worth further investigation.

We then turned our attention to the impact that the topology of the processor has on the performance of the algorithm. We tested the implementation on the E5405 system which had 8 physical processing cores. We found that the algorithm only performed well when running on two cores sharing the same L2 cache. When running on two cores backed by different L2 caches, the parallel algorithm became less efficient than the sequential reference, even when 8 threads were activated to use all the 8 cores.

References

[1] F. C. Crow, "Summed-area Tables for Texture Mapping," in *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*. ACM, 1984, pp. 207–212.

[2] P. Viola and M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," in *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 1, 2001, pp. 511–518.

[3] P. Viola, M. J. Jones, and D. Snow, "Detecting Pedestrians Using Patterns of Motion and Appearance," in *Proceedings of the 9th International Conference on Computer Vision (ICCV'03)*, 2003, pp. 734–741.

[4] H. Bay, A. Ess, T. Tuytelaars, and L. V. Gool, "Speeded-Up Robust Features (SURF)," *Computer Vision and Image Understanding (CVIU)*, vol. 110, no. 3, pp. 346–359, 2008.

[5] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, Nov. 2004.

[6] B. Kisačanin, "Integral Image Optimizations for Embedded Vision Applications," in *Proceedings of the 2008 IEEE Southwest Symposium on Image Analysis and Interpretation*, Mar. 2008, pp. 181–184.

[7] C. Messom and A. Barczak, "Stream Processing of Integral Images for Real-Time Object Detection," in *Proceedings of the 9th International Conference on Parallel and Distributed Computing, Applications and Technologies*, Dec. 2008, pp. 405–412.

[8] T. B. Terriberry, L. M. French, and J. Helmsen, "GPU Accelerating Speeded-Up Robust Features," in *Proceedings of the 4th International Symposium on 3D Data Processing, Visualization and Transmission*, Jun. 2008.

[9] R. E. Bryant and D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2002, ch. 9, p. 671.

[10] *Intel 64 and IA-32 Architectures Software Developer's Manual Volumn 1: Basic Architecture*, Intel Corporation, Jun. 2009.

[11] G. E. Blelloch, "Prefix Sums and Their Applications," Carnegie Mellon University, Tech. Rep., 1990.

[12] S. Sengupta, A. E. Lefohn, and J. D. Owens, "A Work-Efficient Step-Efficient Prefix Sum Algorithm," in *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, May 2006, pp. D–26–27.

[13] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing*, 2nd ed. Addison Wesley, Jan. 2003, ch. 2, p. 21.