NEW DOMAIN DECOMPOSITION METHOD FOR QUALITY-AWARE PARALLEL H.264 VIDEO CODING

***Mohammed Faiz Aboalmaaly*[1], *Rosni Abdullah*[2], *and Ali Kattan*[3]**

[1]National Advanced IPv6 Centre,
[2]School of Computer Sciences,
Universiti Sains Malaysia, Minden, 11800, Penang, Malaysia

[3]Information Technology Department,
Ishik University, 100 Meter St., Erbil, Iraq

E-mail: [1] essa@ nav6.usm.my, [2] rosni@cs.usm.my, [3] ali.kattan@ishik.edu.iq

ABSTRACT

In this paper, we present a new parallel granularity called "tiling" to parallelize the H.264 codec. The new parallel granularity, which has the same granularity level as the parallel slice-level H.264 codec, is based on decomposing the entire video frame into tiles by utilizing a new inherently parallel 2D domain decomposition method. To assess the proposed approach, its parallel scalability, bit rate, and parallel impact on visual quality (peak signal-to-noise ratio) are compared with those of other approaches. Empirical results show significant improvements in encoding time as compared to the serial and the parallel slice-level approaches. In terms of peak signal-to-noise ratio and bit rate, certain results improved, a few were comparable, while a few were discouraging, when compared to the results of the other approaches. However, addressing the limitations of the proposed method is highlighted as future work.

Keywords*: *Video bit rate, video compression efficiency, speed-up, OpenMP, 2D domain decomposition method, parallel scalability, parallel impact.

1.0 INTRODUCTION

The digital revolution has made the use of multimedia applications widely spread. New multimedia services are continuously being introduced by the industry. This phenomenon is strongly related to the rapid development of hardware and software for computing devices, such as servers, laptops, and smart phones.

In terms of online multimedia, services must be capable to facilitate the exchange of multimedia contents within an expected period of time. Thus, the efficiency of compression techniques is important to deliver these online multimedia services over the Internet. For example, among several video compression techniques that have been introduced, some have become an industry standard. H.264 is one of the most popular international standards for video compression. It is commonly used as video codec for high-definition (HD) videos [1]. H.264 is a hybrid block-based video codec that involves prediction, transformation and quantization, filtering, and entropy coding [2]. It outperforms previous video coding standards in terms of compression efficiency. However, this improved compression efficiency comes at the expense of complexity [3, 4]. This complexity is mainly attributed to the introduction of new coding features, such as variable block size, multiple reference frames, and quarter-pixel accuracy. Technically, a H.264 encoder requires computations that are more than one order of magnitude when compared to previous video coding standards such as H.263, MPEG-2, and MPEG-4 Part 2. Moreover, it requires approximately two to four times of additional computations in the decoder side compared to these previous video coding standards [5, 6].

When H.264 was designed, there was no explicit consideration of the revolution in parallel hardware architectures. Fortunately, the upcoming High Efficiency Video Coding (HEVC) [7] standard has addressed this limitation by its obvious support of parallelization [8]. However, addressing such a limitation in a new video coding standard cannot be directly backward-compatible with earlier video coding standards, i.e., H.264, due to the scope of standard. Moreover, in several scenarios, shifting to a newer video coding method may not be feasible for many organizations. This unfeasibility is clearly the reason for the use of several video transcoders [9-11] among video coding systems, such as H.264 and earlier video coding standards. In addition, it motivates researchers to optimize current video coding systems (such as the one in this paper).

Numerous studies, such as [12-16] and others have strived to address the complexity bottleneck of the H.264 encoder. In general, the directions of these works [12-16] can be classified into few categories. The first category overcomes the complexity by utilizing special hardware as accelerator. The application specific integrated circuit (ASIC) and field programmable gate array (FPGA) are examples of accelerators in this hardware-oriented category. Hardware-oriented methods have shown good performance efficiency. However, their limitation is the difficulty in reconfiguration when compared to the software-oriented approach, which is the second category. In the software-oriented category, the high complexity of H.264 is made more algorithmically digestible on general-purpose processing elements. The second category can be further splitted into two approaches: complexity reduction and parallel computing. In H.264, a complexity reduction algorithm is used by removing some of the coding features of H.264, which are subjectively deemed as redundant [17]. On the other hand, the parallel computing approach is applied on H.264 using the data-level approach, the task-level approach, or a combination of the two. The parallel computing approach has been addressed in a remarkable number of studies because of the ongoing widespread use and affordability of parallel hardware, such as multicore and x.

In some scenarios, both complexity reduction and parallel computing approaches may compromise the visual quality or bit rate of the encoded video. For example, in terms of complexity reduction, the early termination or skipping of any of the search-based steps of H.264 encoding, such as inter-prediction, intra-prediction, and motion compensation, will likely result in visual quality degradation. However, complexity reduction-based solutions are additionally susceptible to an increase in bit rate. On the other hand, as will be detailed later, leveraging parallel computing in video encoding would also have, in some cases, affected the video quality and bit rate. Nevertheless, parallelizing H.264 video coding as an approach is forming the vast majority of studies classified under the software-oriented approach when compared to the complexity reduction approach. This preference alludes to the better practicality of parallel computing solutions over the complexity reduction solution in the H.264 codec.

In this paper, we employ parallel computing to design a new parallel-friendly H.264 video encoder. The major concept behind this work differs from other parallel studies in that it introduces a new parallel granularity for the H.264 encoder called “tiling,” which offers improved or comparable outcomes in terms of speedup, video quality, and bit rate as compared to other parallel approaches. The remainder of this paper includes a review of the literature, an illustration of the quality-aware parallel H.264 encoder, a detailed description of our experimental results, discussion, and a conclusion.

2.0 BACKGROUND AND RELATED WORKS

Parallel computing has enabled the wide adoption of H.264 [18]. Since its inception, several parallel-based attempts have been introduced to make the H.264 encoder parallel-friendly, thereby rendering H.264 video encoding more applicable to specific situations. As mentioned in Section 1, the employment of parallelization in H.264 encoding has exhibited different directions. These parallel directions can be categorized according to the flavors of the parallel computing itself. Based on the type of parallelism, parallelizing the H.264 encoder can be achieved by using the data-level or task-level approach. The task-level approach involves two or more independent tasks running concurrently on different processing elements, while the data-level approach divides the overall data size into smaller pieces for simultaneous processing.

As indicated in ample previous literature, such as [3, 19], parallelizing the H.264 codec using the data-level approach has shown better outcomes in terms of speed-up when comparing to the task-level counterpart. This is because of the inherent dependency and unequal workloads of the encoding stages, which limit parallel scalability and efficiency when the task-level approach is employed. Hence, parallelizing the H.264 encoder using the data-level approach is the main focus of this section.

2.1. Overview of H.264 Encoder Parallel Granularities

Parallelizing the H.264 encoder based on the data-level approach is integrally featured in different types based on the relative size of the parallel unit (see Fig. 1). Different possible granularities that can be chosen to parallelize the H.264 encoder include, from largest to smallest, group-of-pictures (GOP), short GOP-level, frame-level, slice-level, macroblock (MB)-level, short MB-level, and block-level.

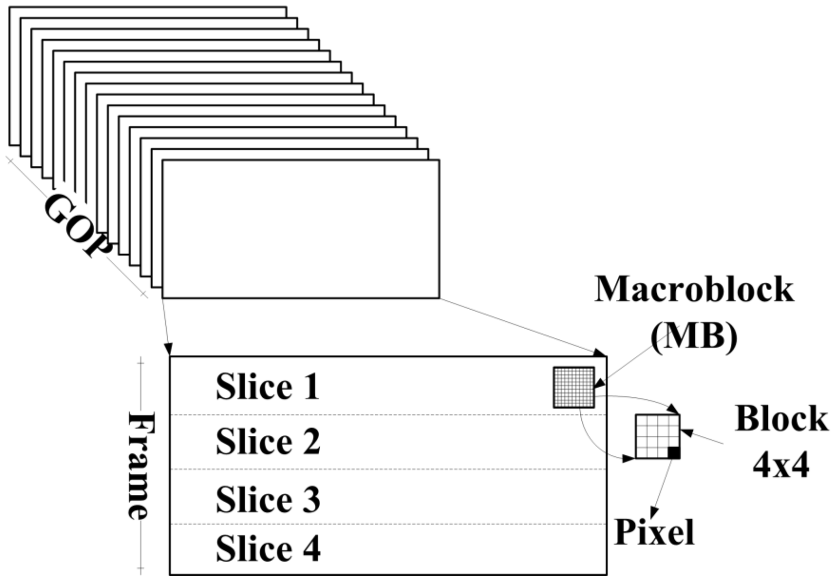

Fig. 1. Data levels parallelism of H.264 codec [12].

GOPs are a coding-independent unit. Therefore, the GOP level is easy to implement; however, it has long latency [20] and large memory requirements [3]. Thus, parallelizing the GOP level is inappropriate for shared memory architecture because of limited on-chip memory [21]. Frame-level coding does not increase bit rate. However, complex interdependencies, which are caused by very flexible usage of reference pictures, limit its parallel scalability [12, 22, 23]. Moreover, this level of coding is associated with large memory requirements. Slice-level coding has been associated with minimal synchronization cost, normal memory requirements, and good scalability performance [3]. The only drawbacks of slice-level coding are the bit rate increment and visual quality degradation when the number of slices increases [12]. MB-level and block-level coding incur no bit rate decrement; nevertheless, both are associated with high synchronization costs because of the small-sized parallel unit, dependency among them [24], and poor scalability [3], which render their incompatibility with the current trend of multicore.

Given this background, parallel granularity in video coding could potentially reflect the performance of a parallel system in terms of scalability, synchronization cost, and memory requirement. In terms of parallel H.264, slice-level is a trade-off method among all the considered performance metric. Moreover, it is the most universal parallelization method employed to parallelize the H.264 codec [24].

2.2. Related Works

Researchers have proposed several parallel-based H.264 encoders, such as the hierarchical parallelization approach for the H.264 encoder introduced in [25]. The hierarchical approach suggests that GOP-level parallelism and slice-level parallelism can be combined to overcome the latency problem of using only GOP-level parallelism. Using the Message Passing Interface (MPI) and multi-threaded parallelism, the work presented in this paper parallelizes the H.264 encoder on a cluster machine. Synchronization is the main problem that produces the loss in the encoding speed-up. We believe this problem is caused by the double layer of parallelism, which introduces several points of barriers.

Moreover, a frame-level parallelism for the H.264 encoder was proposed in [26]. The frame-parallel encoding scheme is based on encoding picture frames that share no data dependency. Up to only three concurrent encoding frames could be reached because of the dependency among frames. Although a reduction of 66% in system bandwidth was reported, no time measurements were shown.

An adaptive slice control scheme was proposed in [24] to parallelize the H.264 encoder. The encoder decides the number of slices before encoding each time segment on a per-frame basis. Using a four-core machine, a speed-up of 3.03 times encoding speed was achieved over the serial implementation. However, the proposed solution is workable when the encoding complexity over some parts of the frame (motion) is significantly different from other parts (low motion). When each frame shows normality in the complexity of encoding among slices, the solution will show no speed-up gain, and the proposed solution will cause extra overhead in deciding the number of slices, which leads to extra encoding time.

At the parallel unit level, a parallel algorithms based on Intel Hyper-Threading architecture for H.264 encoder was proposed [12]. The concept involves splitting a frame into several slices, which are processed by multiple threads. The resulting speed-ups are ranged from 3.1 to 3.7 times on a system with four Intel Xeon processors and Hyper-Threading disabled.

Zhuo and Ping proposed a parallel algorithm with a wavefront-based technique relied on the analysis of data dependencies in the H.264 baseline encoder [13]. Data were mapped onto different processors at the granularity of frames or MB rows, and the final speed-ups were up to 3.17 times on a software simulator with four processors. This method of data partitioning with the wavefront-based technique avoids decrement on the compression ratio by splitting frames into slices. However, in the motion estimation of the H.264 encoder, the search center is the predicted motion vector (PMV). This leads to the data dependencies introduced by ME, which are not confined by the search range However, their analysis is unsatisfactory because this method confines the search center at the position of (0, 0).

At a finer level, a MB region partitioning technique was proposed [27] to explore parallelization at the MB level. A one-dimensional (i.e., vertical) partitioning is submitted to the frame, and it maps each partition to different processors. Then, a wavefront-based technique is adopted. However, to avoid data dependency, processors start to encode data one by one after a short time. The process takes place during the time in which a processor encodes a row of MBs in an MB region and transfers required reconstructed data to the next adjoining processor, which will propagate synchronization overhead. This synchronization will become a crucial issue if the workload of each MB region is significantly unequal. Simulation results of four processors showed a speed-up up to 3.33 times when compared to the sequential reference encoder JM 10.2 by using a CIF (352 × 288) video sequence. Finally, using data-parallel hardware, such as GPUs, several works have ported parts of the encoding/decoding stages of H.264 to be processed by a GPU. Motion estimation (ME) is the main part that is ported to the GPU. Hence, GPUs are used as accelerators. In [28], a contiguous diagonal parallelization was proposed by changing the data dependency at the MB level to increase the level of parallelism. By using the Compute Unified Device Architecture (CUDA), real-time encoding was achieved with an 8.2% increment in bit rate when comparing to the reference encoder JM 16.0.

The observations reported by each work, regardless of its outcomes and based on its chosen granularity, has been designed to suit a particular case. The capacity of the target hardware and other factors affects the selection of the parallel granularity. As a conclusion, no solution could prove that it outperforms other solutions due to the utilization of different performance metrics.Moreover, in terms of parallelizing the H.264 encoder, synchronization has been the main performance bottleneck; avoiding the synchronization by design, will significantly increase the performance of the parallelizing H.264 encoder. Finally, in all works, the parallel granularities that are used are the defined syntax elements in H.264 terminology. However, the manner in which these elements are processed in parallel varies among the studies.

3.0 PARALLEL TILING ALGORITHM

The idea of this algorithm is to decompose a two-dimensional space (such as 2D arrays) into rectangular shapes called tiles. Each of these tiles can be simultaneously processed in parallel. Tiling is not a new term in parallel computing [29, 30], but the way it has been designed in this work, which represents its novelty and simplicity, differentiates it from previous works. In fact, the differentiation is achieved by considering the parallel software applicalabilities during the design phase of the algorithm and not as a post stage which is a common trend in most of the algorithms. In this section, the proposed parallel tiling algorithm is detailed.

3.1. Determining Tile Size

First, the parallel tiling algorithm starts by acquiring the number of tiles (n) that requires to be disintegrated from a two-dimensional space ($p \times q$). Generally, the number of tiles depends on the number of threads in a parallel system. The horizontal dimension is denoted by p, and q refers to the vertical dimension. Once the number of tiles is obtained, this number is employed as input for a prime factorization algorithm as a second step. The second step outputs the factors of n. The generated factors are used as dominators to the horizontal and vertical dimensions of the space ($p \times q$) to determine the horizontal and vertical dimensions of the tile ($T_p \times T_q$).

There are three scenarios for the generated factors: the number of factors is equal to two, the number of factors is more than two, or the number of factor is one (the number of tiles is prime). When the number of factors is more than two, a proposed factor reduction step is employed to reduce the number of factors to two, thereby ensuring the desired number of tiles. This step is achieved by multiplying the extra factors to reduce them to two.

Several possible results can be obtained. However, the best case is when the summation of the two factors is the smallest among other possible cases. Adding such a criterion ensures less length of the total tile boundaries. Moreover, when the number of reduced factors are exactly two, no factor reduction step is required; the factors will be directly used as dominators to determine the horizontal and vertical dimensions of the tile ($T_p \times T_q$). Finally, when the number of reduced factors is one (the number of tiles is prime), this factor (prime number) will then be used as a dominator for one of the dimensions of the space (particularly the horizontal dimension), and the second dominator will be set to one. In the third scenario, the parallel domain decomposition will turn into one-dimensional parallel domain decomposition rather than two-dimensional domain decomposition.

Once the two final factors are obtained, the horizontal and vertical dimensions of the tile ($T_p \times T_q$) are now ready to be determined by using integer division to divide the horizontal and vertical dimensions of the original space to these factors. For example, in the standard video dimension, a HD video 720p (1280 x 720), has 1280 horizontal number of pixels and 720 vertical number of pixels. By reviewing the dimensions of standard video sizes, it is concluded that the horizontal number of pixels is always more that of the vertical number of pixels. Therefore, to further reduce the total number of tile boundaries, if the values of the two dominators (two final factors) are dissimilar, larger value will be used to divide the horizontal dimension and the smaller number will be used to divide the vertical dimension.

Examples for 3, 4, 8, 16 tiles are demonstrated to determine the size of each tile (see Table 1). When the number of the tiles is 3, the prime factorization step will generate 3 (the number itself), because 3 is a prime number. The prime factorization step for 4 tiles will generate 2 and 2. Next, the prime factorization step will generate three factors, i.e., 2, 2, and 2 when considering 8 tiles as an input. Finally, 16 tiles will generate four factors, i.e., 2, 2, 2, and 2. No factor reduction is required for 3 and 4 tiles because the number of factors is either exactly two or less than two. On the other hand, a factor reduction step is employed for 8 tiles, for which the result is 2 and 4, where the 4 is calculated from multiplying 2 and 2. In the case of 16 tiles, two outputs can be generated from the factor reduction step: 2 and 8, and 4 and 4. As previously mentioned, the best case here is 4 and 4 because the sum of these two numbers is eight, which is smaller than the sum of the other set of factors (i.e., the sum of the other set of factors, 2 and 8, is 10). The final desired dominators (two final factors) will be denoted by d_p and d_q, where $d_p \geq d_q$ is always true. Once d_p and d_q are determined, T_p and T_q will be equal to p / d_p and q / d_q, respectively.

Table 1. Size of each tile for four different numbers of tiles in each scenario

Size of the space (1280 × 720) p = 1280, q = 720	*Number of tiles*							
	3 (3 × 1) $d_p = 3, d_q = 1$		4 (2 × 2) $d_p = 2, d_q = 2$		8 (4 × 2) $d_p = 4, d_q = 2$		16 (4 × 4) $d_p = 4, d_q = 4$	
	T_p	T_q	T_p	T_q	T_p	T_q	T_p	T_q
	426	720	320	360	320	360	320	360

3.2. Parallel Processing of Tiles

Once the size of each tile is determined, each tile is assigned to one thread for parallel processing. To dynamically achieve this assignment, a new mathematical formula is suggested in this paper. This formula is designed by using division (/) and modulus (%) operations to explicitly assign a single tile to an individual thread. Besides, the proposed mathematical formula requires to retrieve the ID of a thread in a parallel region. In fact, most, if not all, parallel libraries as well as parallel APIs have a built-in facility to retrieve the thread ID for each thread within a parallel region. Values of this variable start from zero to n-1 where n is the number of threads specified to solve a task in parallel. The following pseudo code illustrates the parallel tiling algorithm:

```
Parallel Tiling Algorithm:
read (n)
/*Size of Tiles*/
factors[i] = prime_factorization(n)
if (sizeof(factors[i] > 2))
{
        reduced_factors[2] = factors_requction(factors)
        if (reduced_factors[0] ≥ reduced_factors[1])
                        dp = reduced_factors[0];
                        dq = reduced_factors[1];
        else
                        dp = reduced_factors[1];
                        dq = reduced_factors[0];
        Tp = p / dp;
        Tq = q / dq;
}
else
if (sizeof(factors[i] = 2))
{
        if (factors[0] ≥ factors[1])
                dp = factors[0];
                dq = factors[1];
        else
                dp = factors[1];
                dq = factors[0];
                Tp = p / dp;
                Tq = q / dq;
}
else
if (sizeof(factors[i] = 1))
{
        dp = factors[0];
        dq = 1;
        Tp = p / dp;
        Tq = q / dq;

}

/* Parallel Processing*/
threadID = get_thread_ID( );
for (i = (threadID % dp) * Tp; i < ((threadID % dp) + 1) * Tp; i++)
for (j = (threadID / dp) * Tq; j < ((threadID / dp) + 1) * Tq; j++)
parallel_processing(space[p, q]);
```

In the above pseudo code, *threadID* is an integer variable used to store the ID of each thread in a parallel region.

As shown in the previous pseudo code, parallel processing is represented in the two "for" loops, where the proposed mathematical formula is embedded in the body of the "for" loop. Table 2 shows the range of allocation for each thread inspired from the data of Table 1.

Table 2. Tile range of each thread

Number of Tiles = 3		
threadID	Horizontal length	Vertical length
0	[0, 426]	[0, 720]
1	[426, 852]	[0, 720]
2	[852, 1278]	[0, 720]
Number of Tiles = 4		
threadID	Horizontal length	Vertical length
0	[0, 640]	[0, 360]
1	[640, 1280]	[0, 360]
2	[0, 640]	[360, 720]
3	[640, 1280]	[360, 720]
Number of Tiles = 8		
threadID	Horizontal length	Vertical length
0	[0, 320]	[0, 360]
1	[320, 640]	[0, 360]
2	[640, 960]	[0, 360]
3	[960, 1280]	[0, 360]
4	[0, 320]	[360, 720]
5	[320, 640]	[360, 720]
6	[640, 960]	[360, 720]
7	[960, 1280]	[360, 720]
Number of Tiles = 16		
threadID	Horizontal length	Vertical length
0	[0, 320]	[0, 180]
1	[320, 640]	[0, 180]
2	[640, 960]	[0, 180]
3	[960, 1280]	[0, 180]
4	[0, 320]	[180, 360]
5	[320, 640]	[180, 360]
6	[640, 960]	[180, 360]
7	[960, 1280]	[180, 360]
8	[0, 320]	[360, 540]
9	[320, 640]	[360, 540]
10	[640, 960]	[360, 540]
11	[960, 1280]	[360, 540]
12	[0, 320]	[540, 720]
13	[320, 640]	[540, 720]
14	[640, 960]	[540, 720]
15	[960, 1280]	[540, 720]

As observed from Table 2, in some situations it is possible that parallel processing will not cover the entire two-dimensional space. To avoid such a defect, some tiles are of different sizes, specifically the tiles at the row-bottom, right-most, and corner (see Fig. 2) in the original two-dimensional space if the start point (0, 0) was selected to be at the top left. Therefore, the pseudo code of the parallel processing part shown above will be replaced by the pseudo code shown below:

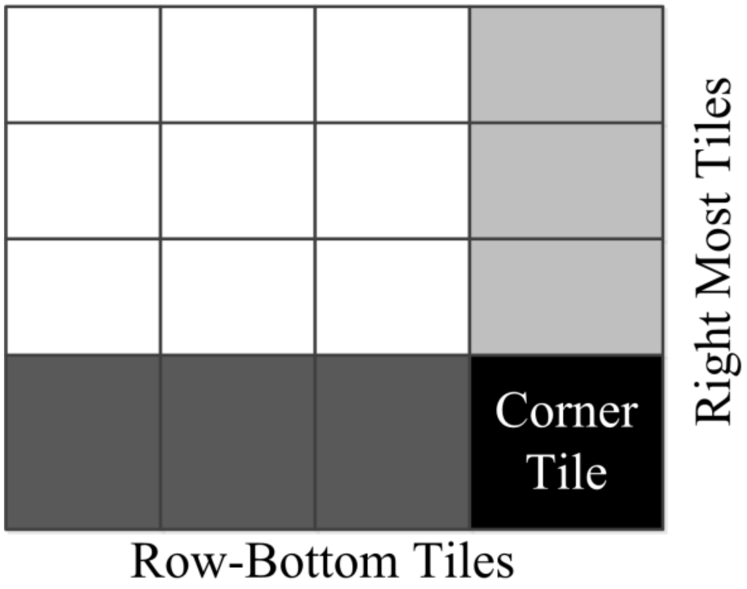

Fig. 2. Different possible tile sizes.

```
/* Parallel Processing*/
threadID = get_thread_ID( );
/* normal tiles*/
for (i = (threadID % dp) * Tp ; i < ((threadID % dp) + 1) * Tp; i++)
for (j = (threadID / dp) * Tq ; j < ((threadID / dp) + 1) * Tq; j++)
parallel_processing(space[p, q]);
/* right most tiles*/
for (i = (threadID % dp) * Tp ; i < p; i++)
for (j = (threadID / dp) * Tq ; j < ((threadID / dp) + 1) * Tq; j++)
parallel_processing (space[p, q]);
/* row-bottom tiles*/
for (i = (threadID % dp) * Tp ; i < ((threadID % dp) + 1) * Tp; i++)
for (j = (threadID / dp) * Tq ; j < q; j++)
parallel_processing(space[p, q]);
/* corner tile*/
for (i = (threadID % dp) * Tp ; i < p; i++)
for (j = (threadID / dp) * Tq ; j < q; j++)
parallel_processing(space[p, q]);
```

As shown in the pseudo code earlier, parallel processing is broken into three types to cover all the possible scenarios of tile sizes. Therefore, the parallel processing part of the algorithm can cover the entire two-dimensional space regardless of the input size.

Finally, it is worth mentioning that the parallel tiling algorithm is designed such that even if the hardware architecture is made of one processing element, it can still work and provide correct results. Taking the same example of Table 1, where p = 1280, q = 720, and n = 1 (*threadID* = 0), d_p and d_q will therefore both be set to 1 (because no prime factorization or factor reduction is required). Accordingly, T_p and T_q will be equal to 1280 and 720, respectively. The mathematical formula after applying these numbers will show that the thread with the ID that equals zero (the only thread) can cover from 0 to 1280 for the horizontal dimension and from 0 to 720 for the vertical dimension, which is the same size as the original two-dimensional space. Thus, hardware-driven backward compatibility has been guaranteed with the proposed tiling algorithm.

3.3. Time Complexity of the natively Parallel Tiling Algorithm

When we are trying to find the complexity of an algorithm, the interest is not in the exact number of operations that are being performed. Instead, the interest is in the relation of the number of operations to the problem size. Thus, with regard to the parallel algorithm proposed in this paper, the pre-processing stage, where the sizes of tiles are determined, is neglected, while the focus is given to the time complexity of the parallel processing part. However, unlike conventional sequential algorithms, where the time complexity is represented by the *big-O* notation, it is determined with regard to the problem size only. The time complexity of parallel algorithms is usually determined with regard to n and p, where p is the number of processors. In the case of our proposed tiling algorithm, enrolling p is inevitable due to the inherent presence of parallelism in its design. Therefore, the role of p needs to be investigated.

Generally, the parallel processing part will take $O\ (n \times m)$, where n and m are the dimension of the 2D array. For simplicity, we assume that n and m are equal, hence, the time complexity will equal to $O\ (n^2)$. By using a number of processing units equal to p, the time complexity will equal to $O\ (n^2/p)$. However, since p is a constant and it does not necessarily scale as the size of the 2D array scales, it is then, neglected. Therefore, the time complexity of the parallel algorithm will remain the same $O\ (n^2)$. However, as empirically proved in Section 5.2, this is not necessarily means that their actual execution time is equal.

4.0 TILE-LEVEL PARALLEL H.264 ENCODER

Because videos are examples of two-dimensional data, the target domain of the parallel tiling algorithm matches the requirements intended to accelerate the process of applications that utilize such data. In this section, the parallel tile-level H.264 encoder, which is based on the inherently parallel tiling algorithm explained earlier, is proposed.

4.1. Defining Tiles

In the proposed design, tiles can be defined as an optional feature at the encoder side that can be used to explore parallelism. In addition, it is fundamentally correct that tiles will break the dependency at their boundaries in the same manner as those at slice boundaries. Moreover, once required, tiles can coexist with slices because the latter are initially proposed for packetization needs and are not only for parallel purposes.

As mentioned previously, H.264 is a block-based video coding standard in which MBs are the data units used for transformation, intra-frame, and inter-frame prediction processes. Hence, to comply with this fact, tiles must be adjusted to be comprised of a specific number of MBs. In terms of H.264, 16×16 is the largest size of MB that can be used for inter-frame prediction. However, possible partitions of a 16×16 MB are used for transformation, intra-frame, and inter-frame prediction processes. Therefore, a 16×16 MB is selected to be the unit size such that each tile can include a number from it and not be an arbitrary size. This requirement is easy to achieve by determining the size of the tile based on the size of an MB. However, because of the standard video resolution size, not all tiles will have the same number of

MBs; that is, tiles at the row-bottom and right-most of the original video frame may have a fraction of MBs or a different sizes of MBs compared to other tiles in the frame. Nevertheless, having a fractional MB size at the frame boundary is a typical case, even in the H.264 standard itself.

4.2. Parallel Encoding with Tiles

The idea of utilizing the parallel tiling algorithm in video encoding following the H.264 standard is based on independent encoding for each tile during transformation, inter-frame prediction, and intra-frame prediction processes.

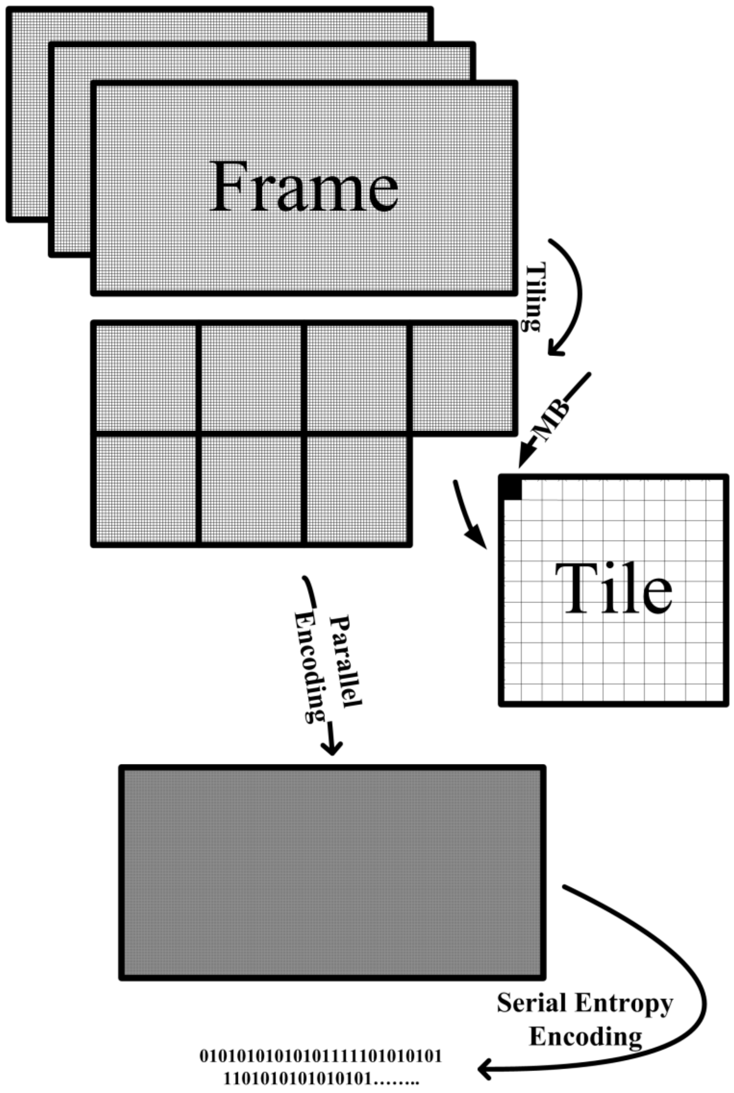

Fig. 3. Tile-level parallel H.264 encoder.

In entropy encoding, the original frame is treated in serial due to the raster-scan order and probability nature of this stage, which make parallelization unsuitable for the proposed parallel tiling algorithm (unless further research is conducted). Fig. 3 shows a video frame partitioned into a number of tiles; each tile can be independently and simultaneously encoded. In other words, each tile can be considered as a standalone frame in transformation, inter-frame prediction, and intra-frame prediction processes. In addition, each tile is encoded by one processing core. Moreover, in inter-prediction process, tiles will predict motions by motion vector (MV) from the analogue tile at the reference frame. However, if a neighboring MB belongs to another tile, its MV will be zero. In intra-frame prediction process, the tile boundaries are handled in the same manner as those of the frame boundaries of the H.264 encoder. Finally, the parallelization process is terminated once it reaches the entropy encoding process.

4.3. Tile-Level Advantages and Limitations

Fundamentally, the tile-level has several advantages over some other known parallel H.264 encoder approaches. Tiles are designed to be encoding-independent units, similar to slices. Therefore, no synchronization is required among them. In other words, no data exchange is required to be transferred between processors that encode different tiles. Besides, no waiting time is required when the wavefront-based technique for parallelizing H.264 encoder is applied. However, the tile-level continues to have an advantage over the slice-level in terms of fewer boundary lengths that break the dependency. Therefore, fewer MVs will be equal zero, which achieves better compression efficiency and lower bit rate. Moreover, because of their relative size, tiles are not associated with large memory requirements and long latency, which is similar to the GOP and frame-level approaches. Furthermore, tiles are not as small as MBs. Therefore, no high synchronization is required when parallelization is employed, and no encoding dependency among tiles is required, which is inevitable at the MB-level approach. Nevertheless, as a parallel approach for the H.264 encoder, tiling can still cause degradation in video quality and a slight increment in bit rate when compared to the serial implementation of the H.264 encoder. This is because a particular tile is still required to break the dependency of MBs belonging to different tiles. Therefore, to inspect the applicability of the parallel tile-level H.264 encoder, considerable comparisons with other approaches are required, which are provided in the next section.

5.0 EXPERIMENTAL SETUP, IMPLEMENTATION, RESULTS, AND DISCUSSION

To experimentally determine tile-level ranking among parallel approaches, a comparison with various tile-level parallelization approaches for H.264 was conducted. The criteria of this comparison were selected based on several parallel granularities that have been previously used to encode H.264 in parallel. Hence, MB-level (wavefront-based) [13] and MB region partitioning [27] were selected. However, the slice-level [24], which has the same parallel granularity as the tile-level, as well as the multi-threading solution proposed in [12], were also included in the comparison.

All other studies selected for the comparison were tested with four processors, which is the same number of processors used in our hardware platform (a quad-core Intel Core i7-2600K processor (four cores, 8 M cache)). Moreover, because the speed-ups in [12, 13, 24, 27] were evaluated in percentage as compared to the serial implementation, a more realistic comparison was provided regardless of the different computation powers of any of these processors. Additionally, four HD 1080p video test sequences were selected (see Table 1). The main reason for selecting these particular video sequences was to emphasize different types of motion and content.

Table 3. Video test sequences

Test Sequence	*Number of Frames*	*Size (MB)*	*Format*
Pedestrian	217	1112 MB	YUV 4: 2: 0
Blue Sky	375	644 MB	YUV 4: 2: 0
River Bed	250	742 MB	YUV 4: 2: 0
Rush Hour	500	1483 MB	YUV 4: 2: 0

However, as will be presented, we were unable to provide a direct side-by-side comparison that matches the settings of our study with all of the other considered studies because some of their experimental setups were not detailed in the papers. For example, not all of the studies, especially those employing a standard or modified slice-level parallelization, have shown the relative bit rate increment or the video quality degradation when the number of slices is increased. However, there are few facts that can be relied on to avoid such a gap; these facts suggest that the size of the slice header is fixed. Hence, the standard slice-level can be used to estimate the bit rate increment of other studies that used a modified slice-level approach.

5.1. Implementation

Because OpenMP is a cross-platform, open-source technology that has been used extensively in several parallel solutions, it is used to implement the parallel H.264 tile-level encoder. OpenMP is the de facto standard of parallelization on shared memory architectures. Moreover, it is a directive-oriented API in which each directive is added to an existing code to explore parallelism. Regardless of its wide range of applications, OpenMP is tailored for large array-based applications, such as video coding, wherein each frame is an image, and each image can be represented in memory as a multi-dimensional array [31].

OpenMP 2.0 is one of the parallel APIs that has a built-in facility to retrieve the ID of the thread inside a parallel region. This attribute makes it possible to use OpenMP to implement the proposed approach.

Tiles are not syntax elements in the H.264 standard; therefore, to add it to the encoder, a code modification is made to the JM 18.5 reference encoder. Particularly, the *lencod.c* file has undergone significant code modifications while modifications to other files were minor. We decided to embed the code in this file instead of adding a new file, such as *tile.c* and *tile.h*, to reduce the time required to start from scratch and to eliminate the chance of producing errors. The encoding flow after adding the tile level was basically a decision-based encoding whereby the encoder decides, based on user input, whether to follow the parallel tiling encoder, the parallel slice-level encoder, or the conventional serial encoder.

5.2. Results and Discussion

To obtain a clear result, the proposed tile-level parallel H.264 encoder was evaluated against three metrics: speed-up, peak-to-signal-noise-ratio (PSNR), and bit rate.

Among all parallel approaches used in the comparison, the speed-up was evaluated with reference to the serial encoding time of the JM reference encoder. Table 2 shows the speed-up achieved by the parallel approaches, including the proposed tile-level, over the serial implementation of the JM reference encoder.

Table 4: Parallel evaluation of the proposed approach; processors = 4

Test Sequence	*Serial Time, JM 18.5*	*Parallel Time*	*Speed-up (times)*
Pedestrian	45.3 s	12.2 s	3.7
Blue Sky	22.7 s	6.3 s	3.6
River Bed	25.6 s	7.85 s	3.26
Rush Hour	53.2s	15.1 s	3.52

As illustrated in Table 4, the proposed approach exhibited a significant speed-up as compared to the JM 18.5 sequential reference encoder and good speed-up as compared to the other parallel approaches (outperformance) when compared to the results of other studies that have been discussed previously in section 2.2, except for the results in [12], where it was comparable. However, the speed-up of the proposed method did not reach the theoretical expectation of linearity. It achieved near-linear speed-up because of the overhead caused by the extra computation of the proposed technique. Because of the lack of hardware availability, no experiments were conducted to inspect the proposed approach with a higher number of processors, and no hyper-threading technology was used to provide more consistence results.

Additionally, we evaluated the influence of the tile-level approach using various PSNR and bit rate, which is expressed as a percentage between the reference encoder JM 18.5 and this approach. However, none of the studies that utilized the slice-level had evaluated the PSNR. Moreover, only one study, i.e., [12], had studied the effect of the bit rate. Therefore, we relied on our standard implementation of the slice-level as an alternative once required, while the values of these two metrics were directly taken from [13, 27]. Accordingly, Table 5-9 shows the difference in bit rate and PSNR of tile-level and slice-level approaches when the number of tiles and slices was equal to 2, 4, 6, 8, and 10 respectively.

Table 5: Video quality and compression efficiency results (slices = tiles = 2).

Test Sequence	*Reference Encoder, JM 18.5*		*Proposed Tile-based*	
	ΔPSNR	*ΔBit rate*	*ΔPSNR*	*ΔBit rate*
Pedestrian	-0.72 dB	+3.18%	-0.23dB	+2.12%
Blue Sky	-0.65 dB	+2.28%	-0.19dB	+0.99%
River Bed	-0.69 dB	+2.78%	-0.53dB	+2.15%
Rush Hour	-0.81 dB	+2.98%	-0.41dB	+1.26%

Table 6: Video quality and compression efficiency results (slices = tiles = 4).

Test Sequence	*Reference Encoder, JM 18.5*		*Proposed Tile-based*	
	ΔPSNR	*ΔBit rate*	*ΔPSNR*	*ΔBit rate*
Pedestrian	-0.93 dB	+5.13%	-0.43 dB	+3.11%
Blue Sky	-0.83 dB	+3.48%	-0.34 dB	+1.52%
River Bed	-1.23 dB	+3.98%	-0.67 dB	+2.87%
Rush Hour	-0.92 dB	+4.15%	-0.62 dB	+2.41%

Table 7: Video quality and compression efficiency results (slices = tiles = 6).

Test Sequence	*Reference Encoder, JM 18.5*		*Proposed Tile-based*	
	ΔPSNR	*ΔBit rate*	*ΔPSNR*	*ΔBit rate*
Pedestrian	-1.24 dB	+8.24%	-0.61 dB	+5.21%
Blue Sky	-0.98 dB	+6.11%	-0.51 dB	+3.17%
River Bed	-1.95 dB	+5.67%	-0.99 dB	+4.02%
Rush Hour	-1.58 dB	+5.43%	-0.87 dB	+3.52%

Table 8: Video quality and compression efficiency results (slices = tiles = 8).

Test Sequence	*Reference Encoder, JM 18.5*		*Proposed Tile-based*	
	Δ*PSNR*	Δ*Bit rate*	Δ*PSNR*	Δ*Bit rate*
Pedestrian	-1.65 dB	+10.69%	-0.92 dB	+7.58%
Blue Sky	-1.78 dB	+8.62%	-0.85 dB	+5.03%
River Bed	-3.45 dB	+8.59%	-1.69 dB	+6.19%
Rush Hour	-3.19 dB	+7.93%	-1.48 dB	+5.64%

Table 9: Video quality and compression efficiency results (slices = tiles = 10).

Test Sequence	*Reference Encoder, JM 18.5*		*Proposed Tile-based*	
	Δ*PSNR*	Δ*Bit rate*	Δ*PSNR*	Δ*Bit rate*
Pedestrian	-2.16 dB	+12.43%	-1.27 dB	+8.61%
Blue Sky	-2.37 dB	+10.49%	-1.16 dB	+5.97%
River Bed	-4.07 dB	+9.82%	-2.45 dB	+7.03%
Rush Hour	-3.92 dB	+10.07%	-1.88 dB	+6.47%

When compare the results of Tables 5-9 and the relavent results of the related work dissuced in section 2.2, it is evident that the proposed tile-level approach failed to achieve better results in terms of PSNR and bit rate compared to [13] and [26]. This is because the MB-level approach did not incur bit rate increment, while it achieved better bit rate reduction as compared to [12] and [23] for approximately 50% reduction in several scenarios. Similarly, the visual quality of the tile-level approach was proved to be better than [12] and [23] based on the PSNR obtained.

In conclusion, none of the studies outperformed our method for all of the considered performance metrics. Therefore, employment of our proposed method can be considered as a trade-off method because it achieves a better or comparable speed-up when compared to other parallel approaches. At the same time, it outperforms some of these parallel approaches in terms of the PSNR and generated bit rate. However, as we indicated earlier in this work, the capacity of the target platform would affect the performance of these approaches in terms of speed-up; therefore, it is possible to obtain variations in results.

6.0 CONCLUSIONS AND FUTURE WORK

A new parallel granularity for the H.264 encoder was presented in this paper. Through tiling, the proposed approach demonstrated either faster or comparable encoding time as compared to other parallel approaches. On the other hand, in terms of PSNR and bit rate, the proposed parallel approach, in some scenarios, failed to outperform some of the parallel approaches. When related works used a standard or modified slice, the proposed approach performed better.

As future works, we have identified few directions. The first is inspired by contributing a type of algorithmic-level approach, in which further optimization of the algorithm can be accomplished to make it faster and simpler. Similarly, dynamic scheduling can be introduced in application areas where computational load differs in parts of the data domains in order to reach a similarity in terms of per-processor computation. In terms of implementation, because this work was implemented on OpenMP, inclusion can be made to support other parallel APIs and libraries, such as POSIX threads [32] and Open MPI [33]. The second direction would be exploring the possibility of further enhancing the visual quality of the tile-level approach by designing a customized deblocking filter at the tile boundaries. Moreover, entropy coding was excluded from being run in parallel in this work, whereas a few studies, such as [34, 35], have tried to parallelize the entropy encoding process in H.264. Therefore, there is ample room for investigating the possibility to include the entropy encoding process in the parallel loop without introducing a significant amount of extra computation, thereby introducing a parallel-friendly H.264 encoder. Thirdly, native hybrid parallelism, where the CPU and other types of processors such as the GPU are utilized together to further reduce the encoding time of the proposed parallel H.264

tiling approach, can also be investigated. However, as a post stage, such an investigation is not considered new and has been presented in several literatures such as [36].

REFERENCES

[1] Y. H. Zhao and R. N. Yin, "A system design of H.264 HD encoding used in broadcast television systems," *International Conference on System Science, Engineering Design and Manufacturing Informatization (ICSEM)*, 2011, pp. 152-155.

[2] J. Ostermann, *et al.* "Video coding with H.264/AVC: tools, performance, and complexity," *IEEE Circuits and Systems Magazine*,2004, pp. 7-28.

[3] S. Jo, *et al.*, "Exploring parallelization techniques based on OpenMP in H.264/AVC encoder for embedded multi-core processor," *Journal of Systems Architecture*, vol. 58, 2012, pp. 339-353.

[4] J. F. Franche and S. Coulombe, "A multi-frame and multi-slice H.264 parallel video encoding approach with simultaneous encoding of prediction frames," in *2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet)*, 2012, pp. 3034-3038.

[5] T. H. Tu, *et al.*, "Batch-pipelining for multicore H.264 decoding," *Journal of Visual Communication and Image Representation*, vol. 23, 2012, pp. 742-752.

[6] S. Saponara, *et al.*, "Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications," *EURASIP J. Appl. Signal Processing*, 2004 pp. 220-235.

[7] G. J. Sullivan, *et al.*, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, 2012, pp. 1649-1668.

[8] F. Bossen, *et al.*, "HEVC complexity and implementation analysis," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 22, 2012, pp. 1685-1696.

[9] E. Peixoto and E. Izquierdo, "A complexity-scalable transcoder from H.264/AVC to the new HEVC codec," *Image Processing (ICIP), 2012 19th IEEE International Conference on*, 2012, pp. 737-740.

[10] E. Peixoto, *et al.*, "H.264/AVC to HEVC video transcoder based on dynamic thresholding and content modeling," *Circuits and Systems for Video Technology, IEEE Transactions on*, 2013, pp. 1-1.

[11] T. Shen, *et al.*, "Ultra fast H.264/AVC to HEVC transcoder," *Proceedings of the 2013 Data Compression Conference*, 2013. pp 241-250

[12] C. Yen-Kuang, *et al.*, "Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures," *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004, pp. 63-73.

[13] Z. Zhuo and L. Ping, "A highly efficient parallel algorithm for H.264 video encoder," in *2006 IEEE International Conference on Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings.*, 2006, pp. 489-492.

[14] N. Wu, *et al.*, "Streaming HD H.264 encoder on programmable processors," *Proceedings of the 17th ACM international conference on Multimedia,* Beijing, China, 2009. pp 371-380.

[15] Y. Hu, *et al.*, "Joint rate-distortion-complexity optimization for H.264 motion search," in *Multimedia and Expo, 2006 IEEE International Conference on,* 2006, pp. 1949-1952.

[16] N. Keshaveni, *et al.*, "Design and implementation of integer transform and quantization processor for H.264 encoder on FPGA," in *Advances in Computing, Control, & Telecommunication Technologies, 2009. ACT '09. International Conference on,* 2009, pp. 646-649.

[17] Gorur, P. & Amrutur, B. (2014) Skip Decision and Reference Frame Selection for Low-Complexity H.264/AVC Surveillance Video Coding. Circuits and Systems for Video Technology, IEEE Transactions on. 24(7) pp 1156-1169.

[18] J. X. WANG, *et al.*, "On Parallelizing H.264/AVC rate-distortion optimization baseline profile encoder," *Journal of Information Science and Engineering,* vol. 26, 2010, pp. 409-426.

[19] Y. Takeuchi, *et al.*, "Scalable parallel processing for H.264 encoding application to multi/many-core processor," in *Intelligent Control and Information Processing (ICICIP), 2010 International Conference on*, 2010, pp. 163-170.

[20] J. C. Fernandez and M. P. Malumbres, "A Parallel implementation of H.26L video encoder (research note)," presented at the *Proceedings of the 8th International Euro-Par Conference on Parallel Processing,* 2002, pp 830-833.

[21] H. K. Zrida, *et al.*, "High level optimized parallel specification of a H.264/AVC video encoder," *International Journal of Computing & Information Sciences,* vol. 9,2011, pp. 34-46.

[22] B. Jung and B. Jeon, "Adaptive slice-level parallelism for H.264/AVC encoding using pre macroblock mode selection," *Journal of Visual Communication and Image Representation*, vol. 19, 2008, pp. 558-572.

[23] M. Roitzsch, "Slice-balancing H.264 video encoding for improved scalability of multicore decoding," *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software,* Salzburg, Austria, 2007, pp 269-278.

[24] Z. Lili, *et al.*, "A dynamic slice control scheme for slice-parallel video encoding," in *19th IEEE International Conference on Image Processing (ICIP)*, 2012, pp. 713-716.

[25] A. Rodriguez, *et al.*, "Hierarchical parallelization of an H.264/AVC video encoder," in *Parallel Computing in Electrical Engineering, 2006. PAR ELEC 2006. International Symposium on*, 2006, pp. 363-368.

[26] C. Yi-Hau, *et al.*, "Frame-parallel design strategy for high definition B-frame H.264/AVC encoder," in *IEEE International Symposium on Circuits and Systems*, 2008, pp. 29-32.

[27] S. Sun, *et al.*, "A highly efficient parallel algorithm for H.264 encoder based on macro-block region partition," in *High Performance Computing and Communications*. vol. 4782, R. Perrott, *et al.*, Eds., Springer Berlin Heidelberg, 2007, pp. 577-585.

[28] F. Takano and T. Moriyoshi, "GPU H.264 motion estimation with contiguous diagonal parallelization and fusion of macroblock processing," in *Consumer Electronics (ICCE), 2013 IEEE International Conference on*, 2013, pp. 19-20.

[29] K. Minwoo, *et al.*, “Parallel transpose of matrix multiplication based on the tiling algorithms,” in *Circuits and Systems (MWSCAS), 2011 IEEE 54th International Midwest Symposium on*, 2011, pp. 1-3.

[30] V. Bandishti, et al., “Tiling stencil computations to maximize parallelism,” in High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for, 2012, pp. 1-11.

[31] G. G. Slabaugh, *et al.*, “Multicore image processing with OpenMP,” *IEEE Signal Processing Magazine,* vol. 27, 2010, pp. 134-138.

[32] F. Garcia and J. Fernandez, “POSIX thread libraries,” *Linux Journal*, vol. 2000, p. 36, 2000.

[33] Open MPI: Open Source High Performance Computing. [accessed 12th April 2013] Available: http://www.open-mpi.org/.

[34] G. Pastuszak, “A high-performance architecture of the double-mode binary coder for H.264.AVC,” *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 18, pp. 949-960, 2008.

[35] Z. Huibo, *et al.*, “A two-way parallel CAVLC encoder for 4K×2K H.264/AVC,” in *ASIC (ASICON), 2011 IEEE 9th International Conference on*, 2011, pp. 75-78.

[36] B. Wang, et al., "Parallel H.264/AVC Motion Compensation for GPUs using OpenCL", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, 2014, pp. 1-8.