Advanced Video Coding on CPUs and GPUs: Parallelization and RD Analysis

Svetislav Momcilovic, Aleksandar Ilic, Nuno Roma, and Leonel Sousa

INESC-ID / IST-TU Lisbon, Rua Alves Redol, 9, 1000-029 Lisboa, Portugal

{Svetislav.Momcilovic,Aleksandar.Ilic,Nuno.Roma,Leonel.Sousa}@inesc-id.pt

Keywords: GPU, video coding, hybrid CPU/GPU system, load balancing

1 Introduction

Increasing need for high quality video communication and video streaming, and tremendous growth of video content on Internet stimulated development of highly efficient compression methods. H.264/AVC is the newest international video coding standard, which achieves compression gain of about 50% comparing the previous standards, keeping the same quality of reconstructed video [1]. However, such compression efficiency is payed by dramatic increase of the computational demands, which makes the video coding hard to be achieved in real-time on single-core Cental Processor Units (CPUs).

The commodity computers of the latest generation, equipped with both multi-core CPUs and many-core Graphical Processing Units (GPUs), are able to achieve high performance in various signal processing algorithms. GPUs' architectures usually consist of hundreds of cores, especially adapted to exploit fine-grained parallelism. As such, GPUs have proven to be highly efficient in exploiting data-parallel parallelism and have been frequently applied to implement complex signal processing applications. On multi-core CPUs, on the other hand, data-parallelism can be exploited either at a lower level, by using vector instructions, or at a higher level, by concurrently running multiple threads on different cores. The simultaneous exploitation of all these different parallelization models involving both the CPUs and the GPUs conducts to complex but rather promising challenges that are widely attractive and worth to be exploited by the most computational demanding applications. However, even though these devices are able to run asynchronously, efficient parallelization models are needed to exploit such computational power of concurrently running devices . These models must guarantee respecting of data dependencies in the parallelized algorithm, aiming in the same time to achieve load balanced execution on processing devices.

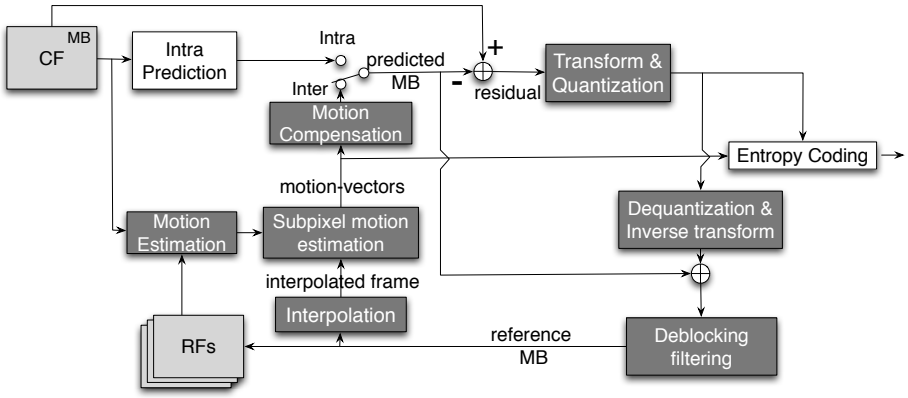

Fig. 1. Block diagram of the H.264/AVC encoder: inter-loop.

According to the newest H.264/AVC video standard [2], each frame is divided in multiple Macroblocks (MBs), which are encoded using either inter-mode or intra-mode (Figure 1). In the most computationally demanding and most frequently applied inter-mode, the best-matching predictor of each MB is searched within already encoded Reference Frames (RFs). This process, denoted by Motion Estimation (ME), allows a further division of the 16×16 (pixels) MB into MB partitions, as small as 4×4. Search procedure is further refined by interpolating the RFs with half-pixel and quarter-pixel precision. Then, the integer transform is applied on the residual signal, which is quantized and entropy coded, before it is sent alongside with the motion vector to the decoder. The decoding process, composed of the dequantization, the inverse integer transform and motion compensation, is also implemented on the feedback loop of the encoder in order to reconstruct the RFs. A deblocking filter is finally applied to improve the visual aspect of the reconstructed frame.

GPUs are commonly used in the latest generations of commodity computers as co-processing devices of the CPU. Their architectures consist of hundreds of cores, especially adapted to exploit fine-grain parallelism [3]. As such, GPUs are highly efficient in executing data-parallel signal processing applications and have been frequently used to implement parallel video coding. In fact, the Compute Unified Device Architecture (CUDA) programming interface, designed for NVIDA GPUs, has proved to allow efficient parallelizations of video coding algorithms based on multi-threading In an usual scenario, pure data-level parallelization models are considered, where the same algorithm is applied on multiple chunks all over the video frame.

According to CUDA [4] thread hierarchy, each thread block in the GPU typically performs all the computations related to a given data-chunk. However, data-dependencies between different chunks often restrict the number of simultaneous thread blocks, thus limiting the exploited level of the GPUs computation power. In order to further improve performance, the newest GPU architectures offer the possibility to execute concurrently up to 4 kernels on a single GPU. In such a way, an additional *task-level* parallelization model can now be exploited within the same algorithm, by concurrently executing several kernels with data independent tasks. On the other hand, on multi-core CPUs data-parallelism can be exploited either at a lower level, by using vector instructions, or at a higher level, by concurrently running multiple threads on different cores. Furthermore, following the asynchronous execution of the GPU kernels, the task parallelism can be also exploited by executing the data-independent tasks on both CPU and GPU.

In the newest NVIDIA GPU architecture the threads are created, managed, scheduled, and executed in groups of 32 parallel threads called *warps*. a warp executes one common instruction at a time, so full efficiency is achieved when all the threads within a warp have the same execution path. However, the existence of data dependent conditional branches, can cause the divergency of the threads over the execution path, causing the sequential execution of each branch path. This branch divergence can significantly decrease the performances of General Purpose GPU (GPGPU) programs that include branches.

Recently, several proposals have been presented to implement parallel video coding algorithms based on GPUs [5–9]. All of these works mostly exploit data-level parallelism and most of them only focus on a single module of the prediction-loop of the H.264/AVC encoder. However, there are still no proposals that effectively consider a parallelization approach, where entire H.264/AVC inter-loop is parallelized on both CPU and GPU .

2 Parallel Video Coding Algorithms for the CPU and the GPU

To efficiently exploit the computational power offered by GPUs, it is important to adopt a number of thread blocks greater than the number of multiprocessors, so that all multiprocessors have at least one block to execute. Furthermore, there should be multiple active blocks per multiprocessor, to guarantee that the GPU is kept busy with the blocks that are not waiting for a thread synchronization. The number of threads per block is usually set to a value between 128 and 256 (or even 384 for Fermi architecture), in

order to hide the latency caused by register dependencies. Nevertheless, the grid dimensions and the thread block size are highly dependent on the characteristics of the implemented algorithm, such as data dependencies. Considering the dimension of a MB (16×16 pixels) and the number of MBs per frame (e.g. 396 for a 352×288 frame, 1620 for a 1280×720 frame), all the operations related to a MB are performed within a single thread block, making that all MBs of a given frame are processed at once.

Concerning the CPU implementation, the number of cores that are usually available imposes a coarser-grain parallelization model. On the core level, the parallelization is exploited by dividing large loops between the multiple threads, distributed among the different cores, by using the OpenMP API [10]. However, the parallelization on such a level frequently does not provide sufficient speedup to achieve real-time video coding for the larger video formats. On the other hand, vector instruction set extensions, such as SSE [11], provide instructions that simultaneously process multiple successive data elements in parallel. However, to efficiently exploit such extensions, a redesign and vectorization effort often has to be made in order to apply the same instruction to a set of neighboring pixels.

In this section it is presented a brief description of the considered implementations for the main modules of the encoder in the two considered platforms: CPU and GPU.

2.1 Full Pixel Motion Estimation

The ME module, as it is implemented in the JM reference software imposes several data dependencies that significantly decrease the parallelization potential of this module. On the other hand, according to the computational complexity of the ME, the efficient parallelization of this module is crucial for overall performance of the video encoder.

2.2 Fast vs. Exhaustive Search

As it was explained in the section 1, the exhaustive Full-Search Block-Matching (FSBM) algorithm [12] requires the processing of a large number of the best matching candidates for several MB partitions and multiple reference frames, where computationally hard Sum of Absolute Differences (SAD) operation needs to be applied to compute the matching distortion. Considering its computational complexity, the FSBM algorithm is very time consuming when it is serially performed on single core CPU, and often requires several hours to encode very short video sequences.

In order to deal with the high computational complexity of the exhaustive FSBM algorithm, several fast search algorithms were developed [13, 14]. The most of this algorithms initially analyze various best matching predictors in order to define both the Search Area (SA) center and the early stopping criterion. Following it, the best matching candidates are selected according to the predefined set of patterns applied around the SA center. The patterns are moved toward the direction of the minimal distortion until either the best candidate is found in the center of the pattern or the early stopping criterion is satisfied. Even though the fast algorithms significantly decrease the computational load of the ME (by examining only a subset of the best matching candidates), the required processing time is still very large to meet the real-time video coding limit on single core CPUs.

One of the most important data dependencies, regarding eventual GPU parallelization of the ME algorithms, is imposed by the computing of the median vector of the best matching candidates of the left, up, and right-up neighbors of the currently processed block. This vector is not only used to determine the SA center, but also to compute the displacement of the best matching candidates, called Motion Vector (MV), which cost is added to SAD value when computing the matching distortion [2]. However, in order to compute this vector, the best MVs of the neighboring blocks need already to be be found. Consequently, the processing of the neighboring blocks can not be performed at the same time, and only a half of the blocks located in the same anti-diagonal can be processed in parallel (see Fig. 2). This number of block is, however, not sufficient to keep all the GPU multi-processors busy, and efficiently exploit its computation power.

In the contrast to the FSBM, the most used fast search algorithms, such as Enhanced Predictive Zonal Search (EPZS) [13] and Unsymmetrical-cross Multi-Hexagon grid Search (UMHexagonS) [14] apply not only the median predictor, but also a larger set of the predictors that include ones from the left, up and right-up neighbors when defining the SA center. Moreover, in the case of the EPZS algorithm, the distortion values found for this predictors are also used in order to make both search pattern decision and the adaptive threshold value for early stopping criterion. Consequently, we can say that the application of this predictors in the mentioned fast algorithms (specially EPZS) is more crucial than in the case of FSBM.

Furthermore, the computational complexity of the most of the fast algorithms highly depend on the video content, which leads to unbalanced distribution of the computational load among different MBs. This fact additionally limits the efficiency of parallel

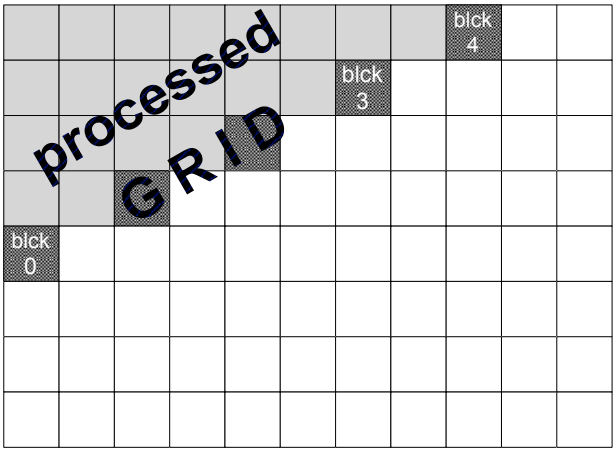

Fig. 2. A GPU grid for eventual parallelization of a fast search algorithm

multi-thread processing, due to the fact that the last finished thread block determines overall performance. This is even more true when considering the parallel processing of different best matching candidates. Due to the fact that the dependency on the video content causes different algorithm paths of the fast algorithms, the processing of different candidates in different CUDA threads will cause branch divergency, and force the GPU to serially execute the instructions within the same warp.

Considering all these limitations and the fact that no efficient fast algorithm is proposed for GPU architectures, we chose the FSBM algorithm for both CPU and GPU parallelization. Even though this algorithm is very slow and impractical for single-core CPUs , we will show herein that parallelization of this algorithm on both multi-core CPU and GPU platforms can lead to very high video coding performance.

One of the strategies that can be applied to decrease the computational complexity of the FSBM, is to reuse the SAD values for the smallest MBpartitions to hierarchically compute the SAD values for larger ones [15]. However, this is only possible when the same predictor is used to determine the SA center for all the MB partitions.

Parallel Algorithm for GPU In the proposed parallelization of Full-Search Block-Matching with Sum of Absolute Differences reusing (FSBMr), each individual MB is processed in a single thread block. Within a thread block, the best matching candidates are examined in different threads. For example, in the case of search are of 32×32

pixels, since each thread block comprises 16×16 threads, each thread examines 4 candidates.

An efficient way to decrease the latency of accessing the current frame and the reference frame samples is the exploiting of the local GPU caches, e.g. shared memory. In the case of ME, the both MB and SA samples need to be accessed several times during the search process, which can cause a significant decrease of the overall performance. The MB samples also need to be accessed when computing the distortion of each best matching candidate. The maximal performance improvement can be achieved when entire SA and the MB are loaded into shared memory, from where they be used for all related computations. However, in the case of large SAs (e..g. 128×128), the entire SA can not feet into the shared memory, and have to be loaded in several steps. In order to achieve the scalability of the approach regarding the SA size, and to keep the most efficient solution for smaller SAs, we implemented two algorithms, where appropriate one is selected according to the available shared memory and required SA size. In the first one, both MB and the SA are cached at once, while in the second one the SA is cached and processed in the portions of $B_x \times B_y$ best matching candidates.

In order to maximally reuse the samples once loaded into shared memory, we used the strategy presented in Fig. 3 and Fig. 4. Namely, the size of the SA partition that will be loaded and examined at once ($B_x \times B_y$) is chosen according to the available shared memory and the preferable number of threads per block (usually 16×16). Due to the fact that each thread processes single best matching candidate, and the maximal size of the MB partition and its candidate is N×N (N=16), the size of the area that need to be loaded is $(B_x + N - 1) \times (B_y + N - 1)$. This means that we have the overlapping area between the neighboring partitions, which size is $(B_y + N - 1) \times N$ pixels for horizontal and $(B_x + N - 1) \times N$ for vertical neighbors. Therefore, we adopt the approach where the direction of selecting the SA partitions that will be processed in the following step alternates for every partition row (see Fig. 3. Such approach leads to the maximal data reuse, as it can be seen from the examples of two typical data reusing situations in Fig. 4.

Algorithm 1 presents the processing performed in a single thread. Different RFs are selected serially within a thread (see Algorithm 1, line *0.1*). Initially, MB and SA pixels are cached into the shared memory. As it was explained above, in general case the SA can not fit into the shared memory, and the processing and caching of the best matching candies is performed in two loops - vertical (line *0.2*) and horizontal one (line *0.3*).

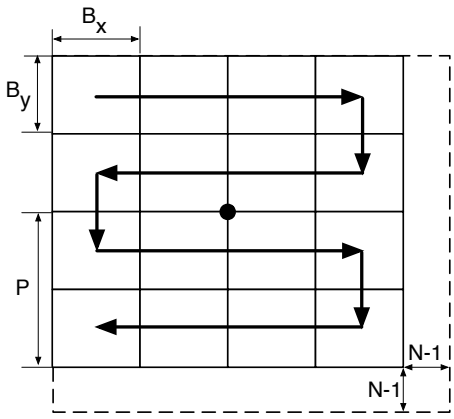

Fig. 3. Caching of the SA samples in proposed algorithm.

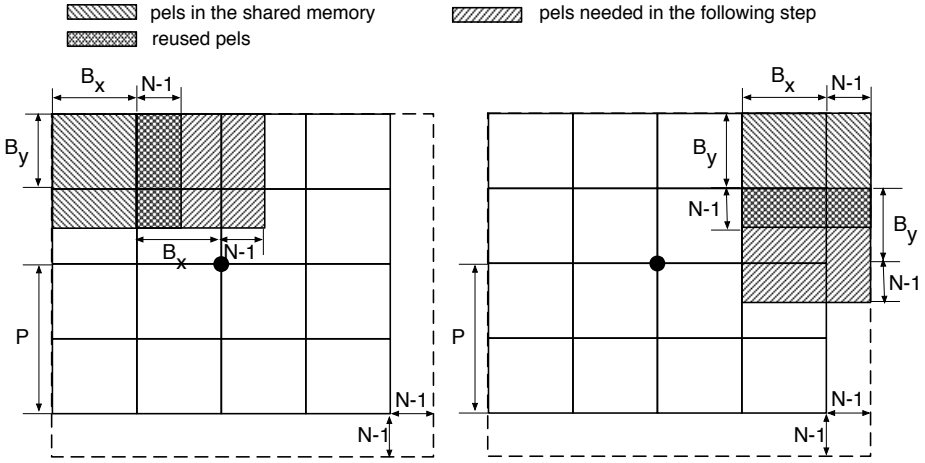

Fig. 4. Data reuse in the SA caching in proposed algorithm.

Each thread (*th*) reads a different pixel of the MB and a part of the SA (see Algorithm 1, lines *0.0-0.2*). For example, if the entire SA of (32×32) pixels is loaded and 256 threads are employed, each thread reads one pixel of the 16×16 pixels large MB, and 4 pixels of the SA.

The FSBMr search algorithm is started by computing the SADs for 4×4 Sub-macroblocks (SBs) (Algorithm 1, line *1.1*). The SAD values obtained for the 4×4 MB partitions are then stored in registers and reused for hierarchical computing of SADs of larger MB partitions (Algorithm 1, line *1.4*). In the same step the motion vector cost

Algorithm 1 GPU motion estimation iteration on single thread level

```
0.0 read MB[th]
0.1 for all RFs
0.2     for i = 0..(2 * P + 1)/B_y
0.3         for j = 0..(2 * P + 1)/B_x
0.4             read SA[th][i][j]
0.5             synchronize threads
0.6             for all candidates per thread
0.7               check_candidate(mv)
0.8             end for
0.9         end for
0.a     end for
0.b         end for
0.c for st = 0 to 40
0.d     update minSMV[th/41][(st+th) mod 41]
0.e     synchronize threads
0.f end for
0.g perform reduction of 6 minSMV lines
0.h send results

1.0 procedure check_candidate(mv)
1.1     compute 4×4 SADs
1.2     update smv_th
1.3     for all SB modes
1.4         calc. SADs hierarchically
1.5         update smv_th
1.6     end for
1.7 end
```

is separately added to each SAD in order to compute the distortion value. The computed *smv* values are used to update the minimal distortion obtained in each thread for the processed MB partitions (*smv_th* in Algorithm 1, lines *1.2* and *1.5*). In order to record in one single instruction the minimum of two SADs and the related motion vectors (MVs), they were concatenated in pairs (SAD — MV). In practice, the SAD value is shifted left by 16 positions, and motion vector is added to the value called *smv*. Consequently, since the SAD value is placed in most significant 2-bytes, whenever the minimal SAD is changed the attached MV is automatically kept updated. However, as soon as the best candidate within the thread is found, the candidates found on different threads need to be compared.

At the end of this step, the 41 best candidates are found in every thread, each of which is related to a different MB partition, and stored in *smv_th* vector. In the reduction process (Algorithm 1, lines *0.c - 0.f*) for each element of this vector the minimum among the threads need to be found. Since the pixel values are no longer needed, the corresponding positions in shared memory were reused for the reduction process. Since each thread can update one of 41 positions at a time, the minSMV matrix of dimensions 41×6 was allocated in shared memory, so each of 256 threads can write at one of its positions. In a loop of 41 steps, each thread updates the minimum at position $(step + thread_{number})\ mod\ 41$ (Algorithm 1, lines *0.c - 0.f*). In such a way, every thread updates a different position at a moment, keeping the maximal number of shared memory banks busy. At the end of this process, there will be 6 remaining candidates for each MB partition, so the minimal one is found by applying the usual reduction tree [16].

Parallel Algorithm for CPU For CPU full-pixel ME, the parallelization on the thread level is applied on each loop of MBs rows. The vectorization, however, is a more challenging task. The main points that need to be taken into account are the pixel transfers into the XMM vector registers and the SAD computation. While the MB needs to be loaded once into XMM registers, the pixels of the best matching candidates need to be exchanged. Hence, in order to prevent eventual performance drop, the candidates are examined in a column major order, instead of the usual row major. In such a way, for every next candidate in the same row, only one vector needs to be updated, while the rest of them are reused.

The SAD computation is based on SSE4 instruction MPSADBW, which operates on 4-byte wide chunks and produces eight 16-bit SAD results. Each 16-bit SAD result is formed from overlapping pairs of 4-byte fields in the destination with the 4-byte field from the source operand. These destination fields start from the 8 consecutive positions. In such a way, it is possible to compute eight 4×4 SADs with only 4 instructions, giving an average of 0.5 instructions per 4×4 SAD. The hierarchical computation of the SAD values is performed by applying the vector addition operation. The obtained $MV - SAD$ pairs are packed in the same way as in the case of GPU implementation. This approach significantly decrease the complexity of obtaining the minimal distortion motion vector, due to the fact that only the vector instruction can be applied, and no branches are needed.

2.3 Interpolation and Sub-pixel Refinement

Parallel algorithm for GPU After finding the motion vectors for the best matching blocks in the RFs, the refinement process needs to be done using the interpolated pixels. Considering the amount of memory currently available on GPUs, the most efficient solution is to perform the interpolation on whole frames prior to the refinement process.

There is no data dependencies between interpolation and full-pixel ME. As a consequence, the kernels for these two algorithms can be executed in parallel streams, while the sub-pixel refinement has to wait for both of them to proceed with the processing. The grid and thread block size for the interpolation algorithm is the same as in the case of the full-pixel ME. In the first step, each thread computes 2 half-pixels, by applying the 6-tap filter in both horizontal and vertical direction (Figure 5, pixels e and f). A threads synchronization step needs to be done prior to the computing of the diagonal half-pixel, by applying the 6-tap filter on either of the two directions (pixel g). Shared

memory is not used in the case of interpolation, since most of the pixels are only used once within a thread block. After the additional thread synchronization, a quarter pixel interpolation is done by applying a linear filter. In this step, 12 additional pixels are produced: pixels ae, eb, fg and gh in the horizontal direction, pixels af, fc, eg and gi in the vertical direction and pixels fe, fi, eh and gd in the diagonal direction.

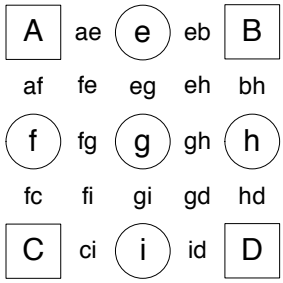

Fig. 5. Interpolation within a thread block

The sub-pixel refinement is done for each MB partition, starting from the motion vector position obtained in the full-pixel resolution. Since the initial best matching candidates are different for all the MB partitions, hierarchical SAD computation schemes are not possible. However, since the sub-pixel refinements for all the MB partitions are data-independent, they can be done in concurrent streams. Accordingly, since the refinement of the MB partitions of the same MB needs the same SAD computing function ($SAD_{m\times n}$), seven separate kernels are used, each one for each MB partition mode.

In the sub-pixel refinement process 25 best matching candidates are examined for each MB partition. In the proposed approach, the size of the thread block is $25\times n$, where n is the number of the MB partitions in each mode. As an example, for the 4×4 mode there will be 25×16 threads in a blocks. However, while the MB pixels are cached in shared-memory, the SA pixels cannot be cached, since the SAs for different MB partitions can be located not only in a different part of the frame, but also in different RFs. The reduction process is performed using a reduction tree, by finding the minimum among the 25 best matching candidates for each MB partition within the thread block.

As soon as all the sub-pixel refinement kernels finish, two simple kernels are applied in order to find the best mode and compute the residual signal. The best mode is found by simply comparing the sum of the cost values of the several modes, representing the contribution of the SAD cost and the motion vector cost.

Parallel algorithm for CPU Just as ME, the interpolation and the sub-pixel refinement on the CPU side is computed using both thread-level parallelization on the level of MB-rows and the logical and arithmetical vector instructions. However, in the interpolation process, due to the fact that 1-byte pixels are used, the 6-tap filtering cannot be directly applied due to overflow of the addition and shift-left operations. Therefore, the PMOVZXBW vector instruction is initially applied, in order to extend the pixels to 2-bytes. The interpolated frame is also conveniently prepared for vectorized sub-pixel refinement. Namely, the best matching candidate pixels used for the SAD calculation with sub-pixel resolution are distanced 1 full-pixel from each other, which means 4 quarter-pixels apart. However, the SAD vector instruction can be only applied on the pixels placed in successive memory positions. Therefore, instead of storing each interpolated frame in a single 2D array, it is stored in 4 *interpolated subframes*, where the sub-pixels with the same distance from the full-pixel position are stored in the same subframe. This is also computationally simpler. For example, both pixels f and h in Figure 5 are computed by applying vertical half-pixel interpolation (6-tap filter). Consequently, these pixels are, in the general case, the output of the same vector operation and can be directly stored in such a interpolation subframe.

In the case of the sub-pixel refinement, only MPSADBW instruction can not be applied for SAD calculation (as opposed to the full-pixel ME), since the consecutive candidates are not consecutive in memory. The PSADBW is used instead. The PSADBW is an SSE2 instruction which works on eight 2-byte chunks, being capable of processing 4 pixels wide chunks. However, since the XMM registers are 16 bytes wide, it is possible to examine two neighboring MB partitions in parallel, giving an average of 2 instructions per 4×4 SAD. The rest of the algorithm is the same as in the case of the full-pixel ME.

2.4 Transform and Quantization

Parallel algorithm for GPU The motion compensation and the residual are jointly computed by subtracting from each of the best-mode MB partitions its best-matching candidate. In order to fit the format of the residual signal to the implementation of the 4×4 integer transform, the residual frame is written in a format of $(FRAME\_SIZE/4)\times 4$, so that $4 \times FRAME\_WIDTH$ rows are stored row by row. When the frame is formatted in such a way, the implementation of the 4×4 integer DCT is scalable, so different thread block-size/grid-size ratios can be tested in order to find the best execution

setup. In the proposed algorithm, 4 threads perform the computation of one 4×4 block. When the transform is performed in vertical direction, each thread computes a single column, while in the horizontal direction they compute different rows. After testing different solutions, it was decided to use 64 threads to compose a thread block, which means that sixteen 4×4 blocks are processed in a single thread block.

The quantization and its inverse are preformed straight-forwardly, where each thread computes a single pixel, and all the MB pixels are computed within a single thread block.

These two algorithms are performed within the same kernel, in order to reuse already computed pixel offsets. However, the threads must be synchronized and their outputs are written in different output buffers.

Parallel algorithm for CPU In the considered implementation, the mode decision and residual computation are performed serially due to the low computational requirements and complex vectorization. The output residual is saved in a 2-bytes per pixel format, in order to prevent overflow in the integer transform and its inverse. As usual, the transform is performed in two steps (vertical and horizontal), and after each of them the transpose of 4×4 MB partitions is performed. Both the transform and the transpose operate on two 4×4 MB partitions at once. In the case of (de)quantization, the vectorization is done straight-forwardly, and the multi-thread parallelization is done on the outer-most row major loop.

2.5 Deblocking Filtering

Parallel algorithm for GPU Data dependencies between the neighboring MBs are a serious problem when fine grain parallelism is required. In fact, since each MB cannot be filtered until its left and upper neighbors are processed, it can be shown that the maximum number of MBs that can be filtered in parallel is equal to the number of MBs within a single anti-diagonal (see Figure 6), by applying a wavefront computational approach [17]. This pattern seriously limits the attained performance, since there should be at least as much thread blocks as cores on GPU for maximum efficiency.

In the presented optimized implementation, each thread filters one pixel of the vertical edge. Considering four 16 pixels vertical edges in each MB, only 64 threads operate in a single thread block. Since the filtered vertical pixels are subsequently used for the processing of the horizontal edges, they are written back to shared memory in a trans-

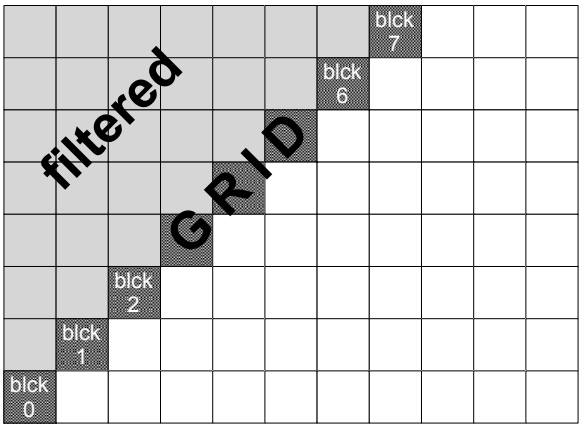

Fig. 6. Deblocking filtering on GPU - wavefront model

posed order. In such a way, the bank conflicts will be minimized, as it was the case for the vertical processing. The algorithm is repeated as many times as the number of MBs within a column. Despite all the considered optimizations, this algorithm cannot avoid branch divergence, but it takes advantage of adaptivity to reduce the computation.

Parallel algorithm for CPU Due to the high adaptivity of the filter, the vectorization of this module represents the most difficult task on the CPU side. Just as in the GPU, the deblocking is performed in two steps, by filtering the vertical and the horizontal edges. In the first step, the loaded vectors need to be transposed, in order to apply vector instructions. Apart from the transpose of the MB vectors, the 3 transposed vectors from the left neighbor also need to be computed, in order to filter the left-most edge. Due to the 16 byte size of the XMM registers and the 2-byte pixels' format, vertical filtering is done in 4 sub-steps, one for each 8×8 MB partition. Prior to each edge filtering step, the boundary strength is checked in order to verify if any filtering needs to be done. The procedure for the horizontal filtering is similar, with the difference that the transpose operations are not needed.

Although the edge filtering is the same for both cases, its high adaptivity causes that different operations need to be applied for different edge pixel values. Consequently, the vectorization can only be done if all the possible branches are executed first and the correct ones are selected afterwards, according to the observed branch conditions.

For that purpose, the PBLENDVB vector instruction is used. This instruction selects the elements from either of the two input vectors, according to a predefined pattern generated by the comparison instructions.

References

1. Wiegand, T., Schwartz, H., Kossentini, F., Ulivan, G.S.: Rate-constrained coder control and comparison of video coding standards. IEEE Trans. on Circuits and Systems for Video Tech. **13**(7) (July 2003) 668–703
2. Ostermann, J., et al: Video coding with H.264/AVC: tools, performance, and complexity. IEEE Circuits and Systems Magazine **4**(4) (2004) 7–28
3. Garland, M., Le Grand, S., Nickolls, J., Anderson, J., Hardwick, J., Morton, S., Phillips, E., Zhang, Y., Volkov, V.: Parallel computing experiences with cuda. Micro, IEEE **28**(4) (july-aug. 2008) 13 –27
4. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2) (2008) 40–53
5. Schwalb, M., Ewerth, R., Freisleben, B.: Fast motion estimation on graphics hardware for H.264 video encoding. Multimedia, IEEE Transactions on **11**(1) (Jan. 2009) 1–10
6. Momcilovic, S., Sousa, L.: Development and evaluation of scalable video motion estimators on GPU. In: IEEE Workshop on Signal Processing Systems (SiPS). (October 2009)
7. Kung, M.C., Au, O., Wong, P., Liu, C.H.: Intra frame encoding using programmable graphics hardware. In: PCM'07: Proceedings of the multimedia 8th Pacific Rim conference on Advances in multimedia information processing, Berlin, Heidelberg, Springer-Verlag (2007) 609–618
8. Obukhov, A., Kharlamovl, A.: Discrete cosine transform for 8x8 blocks with CUDA. NVIDIA research report, NVIDIA, Santa Clara, CA (February 2008)
9. Shen, G., Gao, G.P., Li, S., Shum, H.Y., Zhang, Y.Q.: Accelerate video decoding with generic GPU. Circuits and Systems for Video Technology, IEEE Transactions on **15**(5) (may 2005) 685 – 693
10. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation). The MIT Press (October 2007)
11. Intel: Intel SSE4 Programming Reference. (2007)
12. Huanga, S.Y., Tsai, W.C.: A simple and efficient block motion estimation algorithm based on full-search array architecture. Signal Processing: Image Communication **19** (2004) 975–992
13. Tourapis, A.M.: Enhanced predictive zonal search for single and multiple frame motion estimation. In: Visual Communications and Image Processing. (2002) 1069–1079
14. Chen, Z., Xu, J., He, Y., Zheng, J.: Fast integer-pel and fractional-pel motion estimation for H.264/AVC. In: Journal of Visual Communication and Image Representation. (October 2005) 264–290
15. Ates, H.F., Altunbasak, Y.: SAD reuse in hierarchical motion estimation for the H.264 encoder. In: Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005. (ICASSP'05). Volume 2., IEEE Signal Processing Society (March 2005) ii/905–ii/908 Vol. 2
16. Harris, M., Blelloch, G., Maggs, B., Govindaraju, N., Lloyd, B., Wang, W., Lin, M., Manocha, D., Smolarkiewicz, P., Margolin, L., et al.: Optimizing parallel reduction in cuda. Proc. of ACM SIGMOD **13** (2007) 104–110
17. Aji, A.M., Feng, W.c., Blagojevic, F., Nikolopoulos, D.S.: Cell-swat: modeling and scheduling wavefront computations on the cell broadband engine. In: CF '08: Proceedings of the 5th conference on Computing frontiers, New York, NY, USA, ACM (2008) 13–22

18. ITU-T: JVT Reference Software unofficial version 17.2, http://iphome.hhi.de/suehring/tml/download. (2010)