Journal of Real-Time Image Processing (2020) 17:887–902
https://doi.org/10.1007/s11554-018-0834-4

ORIGINAL RESEARCH PAPER

Enhanced multicore–manycore interaction in high-performance video encoding

Giuliano Grossi[1] · Pietro Paglierani[2] · Federico Pedersini[1] · Alessandro Petrini[1]

Received: 16 March 2018 / Accepted: 19 October 2018 / Published online: 2 November 2018
© Springer-Verlag GmbH Germany, part of Springer Nature 2018

Abstract
This paper presents an efficient cooperative interaction between multicore (CPU) and manycore (GPU) resources in the design of a high-performance video encoder. The proposed technique, applied to the well-established and highly optimized VP8 encoding format, can achieve a significant speed-up with respect to the mostly optimized software encoder (up to ×6), with minimum degradation of the visual quality and low processing latency. This result has been obtained through a highly optimized CPU–GPU interaction, the exploitation of specific GPU features, and a modified search algorithm specifically adapted to the GPU execution model. Several experimental results are reported and discussed, confirming the effectiveness of the proposed technique. The presented approach, though implemented for the VP8 standard, is of general interest, as it could be applied to any other video encoding scheme.

Keywords Video coding · VP8 · CPU–GPU interaction · Hybrid/heterogeneous architectures · Parallel processing · NVIDIA CUDA

1 Introduction

Today, video data transmission covers about 75% of the global Internet traffic, and this percentage is expected to grow in the near future [16]. Video processing-based functions, and in particular higher-resolution video encoders, have thus become key functions for network operators, in their continuous attempt to offer innovative services and attract customers.

To respond to the increased interest in cloud-based video services, video encoders have been recently implemented as virtual network functions (VNFs), which are software appliances running on commodity servers in the cloud. Video-processing VNFs have also attracted the attention of network operators in the multi-access edge computing context, since their aim is to offer innovative video services to the users by running software appliances at the network edge [11].

High-resolution video encoders, however, are not only highly compute-intensive workloads; often, they are also subject to strict time constraints, to minimize transmission latency and guarantee an adequate quality of service to the users. Conversely, the video decoding process is usually much less demanding in terms of compute resources; hence, researchers have mainly focused their attention on video encoding schemes [23].

To cope with the huge need of high-performance computing resources brought about by video applications, HW implementations of some encoding algorithms are now available [2, 22]. Nonetheless, owing to the high variety of available video encoding schemes and the operators' urge for flexibility in service management, software video encoders running on commodity servers in the cloud are often preferred to proprietary solutions based on bespoke hardware [17].

The open source community has further contributed to the success of software video encoders. In particular, the open source X264 [8] and WebM [2] projects, implementing the widely used H.264 and VP8/VP9 schemes, respectively, can obtain an impressive compute performance. For

This work was undertaken under the EU FP7-ICT (7th Framework Programme—Information and Communication Technologies) T-NOVA Project, partially funded by the European Commission under the Grant Agreement No. FP7-ICT-619520.

✉ Giuliano Grossi
giuliano.grossi@unimi.it

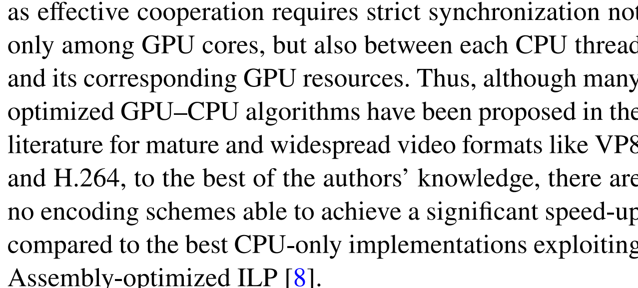
[1] Dipartimento di Informatica, Università degli Studi di Milano, Via Celoria 18, 20133 Milan, Italy

[2] Italtel S.p.A. Research and Development, Castelletto di Settimo Milanese, Milan, Italy

instance, X264 can achieve up to a ×1000 speed-up compared to the JM H.264 standard reference encoder [20, 23]. Such results have been obtained by thoroughly exploiting the optimized SSE/MMX/AVX assembly instruction set, which enables instruction-level parallelism (ILP) on register vectors up to 512-bit-wide. Further performance improvements can be achieved when multi-core x86 CPU architectures are available, as video encoding algorithms efficiently scale on multiple cores, and can therefore greatly benefit from multithreading capabilities [26].

Another very promising approach to further accelerate software video encoders is based on the exploitation of the manycore architectures, which are nowadays commonly provided by off-the-shelf graphical processing units (GPUs). Most of such techniques exploit a CPU–GPU cooperative approach [15]. They usually run the most demanding tasks of the encoding process (typically, motion estimation, ME), on the GPU, while the CPU carries out the remaining functional blocks. According to the results reported in literature, GPU-based schemes can achieve significant compute performance improvements with respect to CPU-only software implementations. However, most of the algorithms proposed in literature are compared to the corresponding standard reference encoder, which is typically not optimized at all (such as the JM encoder for H.264), and often implements the still prohibitive (from the computational point of view) full search ME on CPU [20]. Conversely, to the best of the authors' knowledge, only few works compare the achieved compute performance to the one provided by the fully accelerated version (that is, a version exploiting efficiently all ILP features, like MMX/SSE/AVX) of x264 or WebM VP8 encoders. In particular, in [20, 23] GPU-accelerated H.264 encoders have been compared to x264, while [17, 24] report some preliminary results about a GPU-accelerated VP8 version. Such works clearly highlight that, when compared to the fully accelerated version of x264 or WebM VP8, the known collaborative GPU-based techniques do not exhibit significant compute performance improvements [21]. For instance, [20] reports a speed improvement of only 20% with respect to x264.

Furthermore, when the exploitation of manycore GPU resources is combined with implementations that effectively use ILP-Assembly instructions and the multicore architecture, the achieved performance gain is usually modest. For instance, a comparison of the GPU-accelerated x264 encoder (using OpenCL-based motion estimation techniques [21]) to the x264 CPU-only version [23] yields disappointing results. Experiments performed by the authors with `libvpx` [1, 2] have confirmed the difficulties in achieving a significant computation speed-up when combining multicore (CPU) with manycore (GPU) resources [23]. In fact, the use of CPU multithreading [23] further complicates the achievement of efficient synergy in GPU–CPU collaborative algorithms,



as effective cooperation requires strict synchronization not only among GPU cores, but also between each CPU thread and its corresponding GPU resources. Thus, although many optimized GPU–CPU algorithms have been proposed in the literature for mature and widespread video formats like VP8 and H.264, to the best of the authors' knowledge, there are no encoding schemes able to achieve a significant speed-up compared to the best CPU-only implementations exploiting Assembly-optimized ILP [8].

This paper proposes an encoding scheme that combines the use of manycore GPU's with CPU ILP-based acceleration techniques in an efficient way, obtaining significant speed-up with respect to the best benchmarks. To this purpose, the paper analyzes the problem of effectively combining highly optimized software encoders with GPU resources. The well-established open source VP8 WebM encoder [1] has been used as benchmark. This encoder, in fact, exploits both the full set of optimized Assembly instructions and the CPU multicore architecture. The presented modified VP8 scheme, implemented in CUDA C running on NVIDIA GPU resources, can achieve up to ×6 speed-up with respect to the mostly optimized version of the reference VP8 WebM encoder (`libvpx`), with minimum degradation of visual quality.

The techniques presented in this paper, applied to VP8, could be potentially used also in more recent coding schemes, such as VP9 and HEVC. In particular, the proposed approach can guarantee sequential raster-order processing of video frame macroblocks by the CPU, introducing minimum latency. For this reason, we can adopt the well-known rate-distortion optimized (RDO) techniques for motion estimation [15, 28], widely used in recent coding schemes, thus overcoming one of the most challenging drawbacks in ME implementations on GPU's [15]. The developed algorithm has been archived in the WebM Project repository, and is thus available for further investigations to the research community [4].

The paper is structured as follows: Sect. 2 gives an overview of the VP8 encoding scheme and of the literature on efficient computation of the ME in the VP8 encoding standard. Section 3 describes the proposed algorithm. Section 4 presents the achieved experimental results. Finally, some conclusions are drawn in Sect. 5.

2 Overview and related work

2.1 Overview of the VP8 encoding scheme

Figure 1 illustrates the functional diagram of the VP8 encoding scheme. At this level of detail, this diagram is common to all modern video encoding standards. Each frame of the video stream is divided into square blocks of pixels

(called macroblocks, MB), each consisting of 16 × 16 luminance pixels and two 8 × 8 blocks of chrominance pixels. The macroblocks are processed sequentially (left-to-right, top-down); for each block MB_i, the best predicting macroblock (i.e., the one maximally similar to MB_i) is chosen among the previously encoded MB. Then, the image difference between MB_i and the best predicting macroblock, called the residue, is transformed using either the discrete cosine transform (DCT) or the Walsh–Hadamard transform (WHT), and the transformed coefficients are quantized and encoded, producing the final coded video bitstream [2, 12]. As Fig. 1 shows, the best predicting MB can be found either through intra-frame prediction, meaning that it is chosen among the already coded MBs of the same frame, or through inter-frame prediction, that is, chosen among neighboring MBs belonging to a previous frame. To this aim, the algorithm selects some of the previous frames (called the last frame, the golden frame, and the alternate reference frame) during the encoding process, to consider them as possible sources of the inter-frame predicted macroblock. In case of inter-frame prediction, the optimal inter-frame macroblock is searched in one of the reference frames, by means of a motion estimation (ME) procedure, which finds the 16 × 16 window that is maximally similar (in the sense of minimal pixel-by-pixel luminance/chrominance difference) to the currently analyzed macroblock. The result of this search is a motion vector (MV) representing the hypothetical motion of the macroblock from the reference (previous) frame to the current frame.

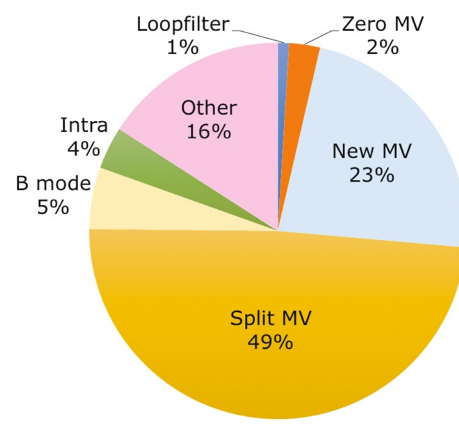


Fig. 2 Average encoding times (in ms) for the major functional blocks of the `encodingFrame()` function of the VP8 reference encoder for a sequence of 100 frames in Full HD resolution. Functional blocks involving motion estimation (NewMV and SplitMV) are responsible for over 70% of the encoding frame time

Once the best predicting MB is found, the pixel-by-pixel difference between MB_i and its best prediction, called the residue, is transformed (through DCT or WHT) and the resulting coefficients are quantized and finally entropy coded through an arithmetic encoding scheme, together with all information describing the MB. The coded MB is then sent

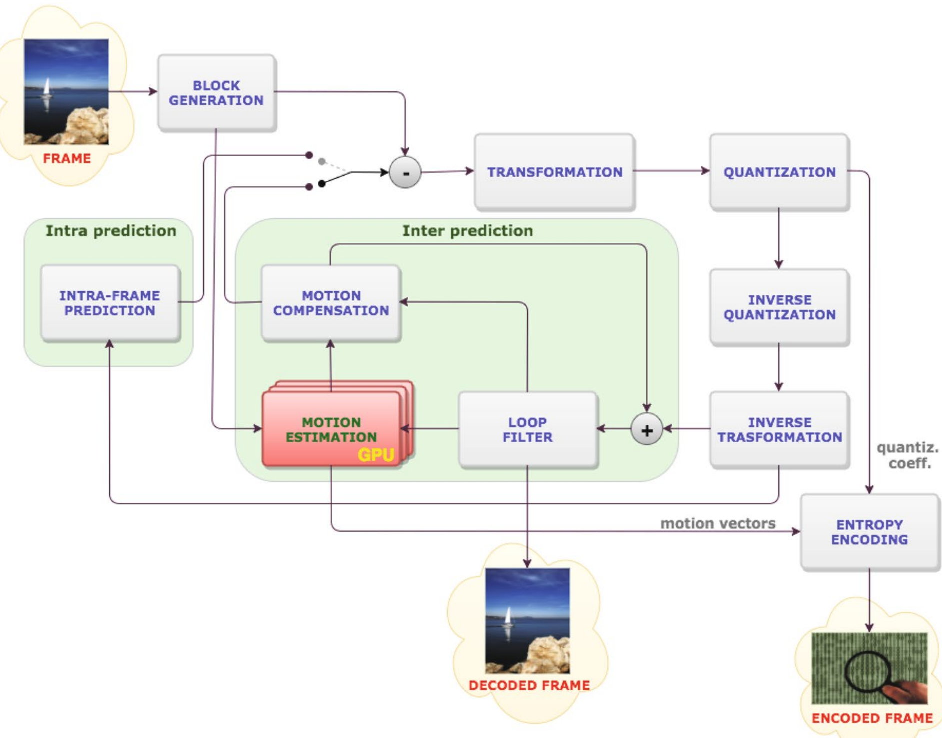


Fig. 1 Block diagram of the VP8 encoding scheme. Motion estimation plays a major role in the inter-frame prediction and it is one of the most computationally intensive tasks of the whole encoding process

(or stored) as part of the coded stream but, at the same time, it undergoes the inverse process, that is the same decoding process carried out at the receiver side, and possibly taken as reference for intra-frame or inter-frame prediction.

In most current encoding schemes, like AVC/H.264, motion estimation is the most demanding functional block, in terms of computational effort [20, 28]. Since all modern encoding schemes resort to motion estimation in a very similar way, it is reasonable to assume that ME is the most demanding function also in VP8. To verify this assumption, we carried out some profiling of the reference VP8 encoder, by measuring the execution time of each major functional block involved in the encoding of a test sequence. Figure 2 reports a summary of this experiment: the functional blocks NewMV and SplitMV, which make use of motion estimation, take more than 70% of the encoding frame time, on average.

Besides that, motion estimation is one of the few tasks that can be massively parallelized (each MB can be computed independently and therefore in parallel with the others). This is the reason why we chose, in our approach, to delegate the motion estimation tasks to the GPU, while keeping the inherently sequential tasks on the CPU, similarly as in many GPU–CPU cooperative approaches to high-performance video encoding.

2.2 Related work

Accelerating ME can provide significant compute performance improvements, with reasonably low implementation efforts. The WebM VP8 software-only encoder available at [1], used in this work as the reference VP8 software encoder, suitably combines several techniques to perform ME in an effective and highly optimized way.

As in most video encoding standards, in VP8 ME is carried out in the luminance plane of the YUV color space [2, 12, 13]. As seen in the previous section, three different reference frames are used to evaluate the motion vector (MV) of a macroblock (MB): the last frame, the Golden Frame and the alternate reference frame [12, 13]. Five different motion estimation modes are available: ZEROMV, NEARESTMV, NEARMV, NEWMV and SPLITMV. ZEROMV indicates no motion of the current MB with respect to the reference frame, while NEARESTMV and NEARMV re-use the two most similar MVs among those of the already coded neighboring MBs in the current frame. NEWMV, conversely, implies the calculation of a new MV for the current MB (a 16×16 pixel block), while SPLITMV divides the MBs in sub-blocks of different pixel sizes (4×4, 8×8, 16×8 or 8×16), each possibly with its own MV.

To exploit both spatial and temporal correlation among frames, VP8 processes MBs in sequential, raster-scan order. This constraint also holds for most recently proposed video codecs, such as H.264, HEVC or VP9, in which this sequential order is exploited by advanced RDO-ME techniques [28]. In RDO-ME, the algorithm first finds a set of candidate motion vectors and then selects for transmission, among all candidates, the MV that minimizes a weighted function of distortion and rate. This algorithm therefore needs the MVs of the preceding MBs, as in VP8, to evaluate the so-called predicted motion vector (PMV). The need to proceed in sequential order causes an unavoidable dependency between the MBs of the same frame, and represents one of the major obstacles to the parallelization of ME.

To accelerate ME, the reference VP8 encoder can use multithreading [2], so as to exploit data parallelism on the CPU. To overcome the MB dependency constraint, in this implementation [1] ME is performed along the frame diagonal: the first CPU thread starts processing the macroblocks of the first row; the second thread starts processing the second row as soon as the neighboring MBs have been processed by the first thread, and so on, for all the available threads. As in X264, MV calculation is based on a modified diamond search technique [29]. In the algorithm, the initial position is estimated from the motion vectors of the neighboring MBs, which are combined to give a predicted MV [15, 20] (in the following, this predicted MV will be referred to as the PMV from diamond search, or PMVDS, because its role is conceptually different from that of PMV). A large-sized diamond is moved around in the reference frame until a low energy area is found. Successively, a subpixel search is performed, up to quarter-pixel resolution.

The above ME approach results in a highly optimized and effective ME procedure, that provides a very good trade-off between computational efficiency and high visual quality. Designing a GPU-accelerated ME VP8 encoding algorithm that outperforms the WebM reference encoder, is a complex activity that poses various challenges, briefly described in the following:

– Many GPU-based techniques address ME parallelization for H.264 or HEVC [19, 25]. In particular, in [28] a two-step procedure is proposed, which performs a MV coarse search in the first step and a refined search in the second step. The coarse search supplies the PMV to the RDO–ME algorithm, which selects the MV to be transmitted. However, the introduction of such a scheme into the VP8 encoder would significantly alter its highly optimized software architecture, thus implying non-negligible integration efforts. Also, since such algorithm performs full search ME, it is questionable whether its use in VP8 would lead to an actual speed-up against the software-only version [15]. Similar considerations hold for most ME algorithms performing full search, as discussed in Sect. 1.

- A different approach to ME parallelization suggests to replace the CPU ME procedure with an equivalent ME function running on the GPU, processing MBs sequentially. This approach has been adopted in X264 for the H.264 codec, achieving a performance gain up to 55% [21]. However, the degree of parallelism that can be achieved by processing a single MB is very low, and can thus provide a quite disappointing performance. Further tests carried out using the ME technique suggested in [21] are reported and discussed in [23]; they show that in most cases the achieved gain is close to 5%.
- The off-diagonal approach to process MBs proposed in [15] can be successfully applied in a multicore CPU, with tens of parallel cores, but would not allow to fully exploit the massive parallelism of a GPU, which provide thousands of cores.
- In ME procedures running on highly parallel architectures, the search for the initial MV cannot be initialized to the PMVDS value, since different MBs are processed independently and simultaneously [15]. This may affect the performance of the ME process, in particular when the MV search area within the reference frame is small. This loss can be compensated using a wider MV search area, however, at higher computational costs.

The proposed GPU-based technique processes MBs in parallel and is able to overcome the described difficulties, leading to significant speed improvements, as shown by the experimental results presented in Sect. 4. In particular, our technique can be straightforwardly introduced in the existing reference VP8 implementation [1], preserving its highly optimized software architecture. The key features of the proposed approach are the exploitation of task streaming techniques and the development of a modified diamond search-like algorithm, as described in the next section.

3 The proposed cooperative CPU–GPU VP8 encoding

3.1 The cooperative approach: general architecture

The key idea of the proposed approach is to exploit the GPU computing power to perform the ME of the current frame with respect to the three reference frames, thus relieving the CPU (running the mainstream encoding process) from such burdensome task. The goal is to provide the CPU with the computed motion vectors as fast as possible, while the MBs are processed by the CPU in sequential order.

Naturally, as discussed in Sect. 2, during this cooperative process two conflicting requirements could arise. On one hand, the GPU should concurrently process as many MBs as possible, to fully exploit data parallelism and enhance the compute performance. On the other hand, to avoid latencies in the CPU–GPU synchronization, only small chunks of MBs should be processed in parallel by the GPU, so that their MVs could be quickly returned to the CPU.

To cope with such opposite needs, in this approach the ME procedure running on the GPU is split into sub-tasks, which are launched sequentially as CUDA streams, each working on a subset of N macroblocks. This cooperative ME approach can be summarized as follows:

- Each time a new frame (the current frame) is available for encoding, it is copied into the GPU's device memory, by performing a synchronous copy (the CPU waits until the copy is completed before continuing with the next instruction). This ensures that the full frame is available as the ME starts in the GPU.
- As soon as the frame copy has completed, the GPU starts computing the ME between the current frame and the three reference frames. The GPU performs ME on groups of N macroblocks in parallel, and copies the results back into the CPU memory. The ME kernels on the GPU are started asynchronously, i.e., the CPU launches their execution on the GPU, and immediately continues to the next instruction. From this point on, GPU and CPU work in parallel and independently.
- The CPU proceeds with the sequential encoding process, one MB after the other. Each time this algorithm needs a MV for a MB (e.g., to evaluate the opportunity of inter-frame coding), it can simply resort to the already available GPU-computed MVs, thus saving the time that would be otherwise necessary for ME computations.

It is clear that the overall interaction between CPU and GPU must be suitably designed, to minimize the CPU latencies that could occur if waiting for MVs to become available.

3.2 Using streams to minimize latencies

As said before, it is of crucial importance that the MVs computed by the GPU are available before the CPU requires them. For this reason, the GPU kernel for ME has been structured in such a way as to produce the first results (the MVs of the initial frame MBs) as soon as possible. This has been accomplished using CUDA streams.

The structure of this CPU–GPU cooperation is shown in Fig. 3. The asynchronous launch of the three ME kernels (the same GPU kernel, launched on each of the three frame pairs: current frame vs. last frame, current vs. golden, and current vs. AltRef) causes the start of the three streams in sequence, beginning with the first grid of each stream (grid 0), working on the initial MBs. After completion of one grid, the next one is immediately scheduled in the GPU execution queue. Even if the GPU does not guarantee the

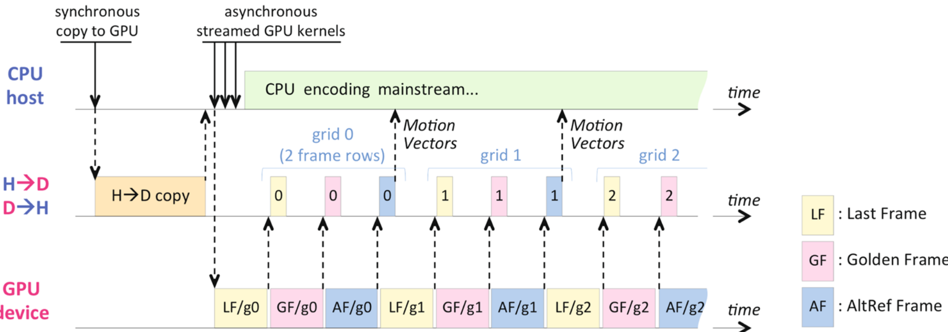


Fig. 3 Temporal diagram of the CPU–GPU cooperation scheme: after a new frame is copied from the CPU's (host) to the GPU's (device) memory, the three ME kernels start in the GPU, organized as streams, to provide the MVs needed by the CPU at the beginning of the encoding process, therefore avoiding CPU latencies

sequential order among different CUDA streams launched in parallel, the order shown in Fig. 3 is respected because each grid takes 100% of the CUDA resources, therefore only one grid is executed at a time. Consequently, all subsequent grids are scheduled and executed, for the three reference frames in sequence, in order of grid number (last frame/grid 0 → golden frame/grid 0 → AltRef frame/grid 0 → last frame/grid 1 → …), as shown in Fig. 3. To verify these temporal assumptions, they have been experimentally verified by analyzing the timing results provided by the CUDA profiler (`nvvp`).

Each kernel grid is composed of N CUDA blocks; the computation of the MV for each MB is assigned to a block, consisting of 128 CUDA threads running in parallel, on the 128 GPU cores of a SM. Each kernel grid terminates with an asynchronous copy of its computed motion vectors back into the host memory. This operation is scheduled and carried out in parallel with the following grid execution, as the device-to-host (D2H) copy engine works independently (and concurrently) from the processing cores. This scheme is called two-way concurrency in CUDA devices [5, 14] (not considering here the concurrency of the CPU). Theoretically, three-way concurrency would be also achievable, by fragmenting the initial synchronous copy of the current frame into streams, and thereby exploiting concurrent operation of the host-to-device (H2D) copy engine. In practice, however, we have experienced, by means of experimental tests, that the two-way concurrent scheme leads to higher speed, due to the additional overhead of the H2D management and to the fact that the time necessary for the entire H2D synchronous copy is short, compared to the overall encoding time.

As soon as the synchronous D2H copies are completed by the first grid (grid 0), for all the 3 reference frames, the encoding mainstream on the CPU can read the MVs of the first macroblocks, computed by the GPU within this grid. Referring to Fig. 3, if, in the time between the start of the frame encoding process and the completion of grid 0, the encoding process on the CPU needs the MV for one of these initial macroblocks, the CPU process must wait for the D2H copies of grid 0 to complete. The occurrence and duration of this possible latency, t_W, depends on the time at which the CPU needs the first MV. The worst case, corresponding to the maximum value for t_W, occurs when the CPU encoding thread requests the MV of the very first macroblock (the up-most and left-most one) using as reference the AltRef frame (processed at last by the GPU). In this case, t_W would be slightly less than the total GPU execution time for grid 0, as the timing diagram in Fig. 3 shows. After then, under the reasonable assumption that the GPU grid execution time $t_{g,GPU}$ is significantly smaller than the time needed by the CPU to process the same set of macroblocks, $t_{g,CPU}$, no CPU latency can occur anymore, as the GPU will be at least one grid ahead, and the CPU will therefore always find already-computed MV's.

As a consequence, the size of the grid N (being N the total number of macroblocks processed at once) strongly influences the processing speed: smaller grids increase the overhead for scheduling and launching, thus reducing the speed, whereas bigger grids increase the time needed for the generation and transmission to the CPU of the initial results (the first set of motion vectors), which increases the occurrence of the described CPU latency, as well as the maximum value of its duration, t_W. The optimal grid size is therefore device-dependent, as it mainly depends from the t_g, CPU$/t_g$, GPU ratio. For this reason, the optimal size N has been determined experimentally; for the hardware adopted in this work, the maximum speed has been obtained with grids containing two rows of macroblocks ($N = 144$ for full HD resolution, $N = 288$ for 4 k), rather independent of the frame resolution.

Each grid, launched as a stream instance, is composed of CUDA blocks, organized to run in parallel on the whole GPU device. The adopted device (NVIDIA GeForce 980GTX) is composed of 16 streaming multiprocessors (SM), each containing 128 processing cores. The grid is made of a number of blocks multiple of 16, so that 16 macroblocks are scheduled in parallel. Each macroblock is then processed by a CUDA block in one SM. This choice provides several advantages:

– All SMs perform exactly the same task, thus leading to full efficiency in the exploitation of the GPU device.

– The access to the data is perfectly parallelized among different SMs, as each SM works on a different macroblock.
– No data interchange is needed among different SMs, as each MB is processed independently from the others.

3.3 SM-optimized diamond search

As said before, each stream grid is composed of N CUDA blocks. Each of them is scheduled on one SM. Each block performs the motion estimation procedure on one MB, therefore our device is able to process 16 MBs perfectly in parallel. Each block is composed of 128 threads to have a one-to-one match between threads and processing cores within each SMX streaming multiprocessor. This ensures the real parallelism of all the threads, which are thus executed SIMD-like.

The search for the best motion vector is based on the classic diamond search algorithm, adopted as a standard in many encoding schemes, including VP8 and H264. In its original proposal [29], the diamond search algorithm adopts a diamond-shaped search space, according to the statistical distribution of the motion vectors, empirically obtained by analyzing a large set of videos taken as standard [10]. Figure 4 shows the originally proposed diamond search space.

It has to be considered, however, that the analysis giving this result was carried out on videos in CCIR (720×480) and CIF (360×240) pixel resolutions. In contemporary video resolutions, such as "Full HD" (1920×1080 pixel), it is reasonable to scale the search space accordingly (that is, 3× to 5×). For this reason, we propose an adapted diamond-shaped search space shown in Fig. 5. Our search space extends up to ±16 pixel horizontally and ±12 pixel vertically. This search space covers, in proportion, a larger area than in the original case (accordingly to the scale ratio). A larger area is needed because our aim is to implement a one-shot search instead of an iterative search, as in the original paper [29]. Eventually, we have implemented both solutions (one-shot and iterative). A comparison of their performance, shown in the experimental results, has evidenced that the two approaches yield different trade-offs between processing speed and image quality.

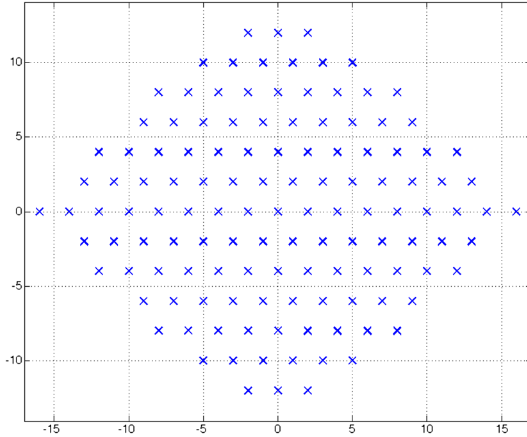


Fig. 5 The proposed diamond-shaped search space, optimized for contemporary video resolutions, like full HD (1920×1080 pixel). The search space consists of 127 vectors; this enables a motion search optimized for the 128 processor cores of a CUDA SMX streaming multiprocessor

The motion estimation algorithm is composed of the following main steps.

3.3.1 Absolute difference computation

For each of the 256 pixels of the considered macroblock, a thread computes the absolute difference between its luminance and that of the corresponding pixel in the reference frame (last, golden, or AltRef). Each SM thread computes the absolute difference for 4 pixels and sum them: more precisely, thread j, where $j = 0 \cdots 63$, computes pixels $j + 64k$, with $k = 0 \cdots 3$, as shown in Fig. 6. In this way, the 128 threads of a SM compute the differences for 2 vectors of the map at the same time. All these differences are computed and stored in a 128×64 difference matrix located in the shared memory; the matrix element DM(i, j) contains the

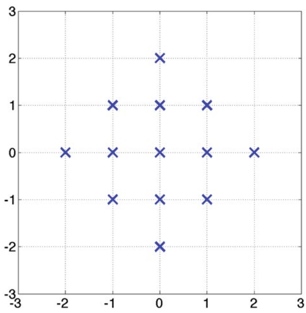


Fig. 4 Original diamond-shaped search space, as proposed in [29] for CCIR (720×480) and CIF (360×240) video resolutions

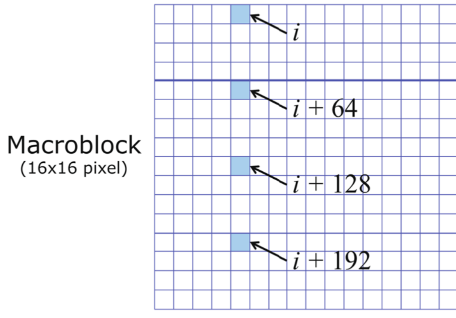


Fig. 6 Computing the absolute difference: GPU thread i computes the sum of the absolute values of the luminance difference (current frame − reference frame) for the four pixels: $i + 64k$, $k = 0, 1, 2, 3$

sum computed by thread j [that is, the SAD (sum of absolute differences) of the 4 pixels $j + 64k$, with $k = 0 \cdots 3$], for the ith vector of the search map in Fig. 5.

Since the shared memory is local to each streaming multiprocessor, all threads within a block will share the same DM, thus they are all cooperating to the motion estimation for the same MB, but there will be a different DM in each of the 16 blocks processed in parallel, corresponding therefore to 16 independently and simultaneously computed macroblocks.

3.3.2 Sum of absolute differences

Each thread computes the sum of the absolute differences (SAD) for the ith displacement vector. This is carried out using a vector reduction approach, executed sequentially by each thread over the ith row of DM, and in parallel over all the displacement vectors. This algorithmic approach involves the execution of 63 subsequent CUDA multiply-and-accumulate (MAC) instructions, whereas a conventional iterative loop to compute the sum would execute 64 MAC instructions. Nevertheless, in the proposed approach the optimized access to the *DM* matrix in the shared memory leads to a significantly lower computation time. The final sum of the absolute differences (SAD), for each displacement vector i is stored in the first column of the DM, i.e., DM(i, 0).

3.3.3 Minimum search

The search for the minimum SAD value among the 127 computed values is carried out through binary vector reduction in parallel by a subset of the available processing cores within each SM, taking just seven subsequent MAC instructions (as $\log_2 128 = 7$).

3.3.4 Search refinement

The vector with the minimum SAD has been found among all motion vectors of the displacement map shown in Fig. 5. As the figure shows, this map is characterized by two-pixel resolution, as the minimum distance between neighboring points is mostly two pixels. For this reason, the above described search is followed by a second, finer search, in which the best displacement is found in the direct neighborhood of the previously determined displacement, this time with pixel resolution. The algorithm for this refinement phase is the same as that described above, except for the reduced displacement map with one-pixel resolution, composed of 15 vectors only (for the sake of efficiency, 16 threads are used, one per vector plus one remaining inactive), as shown in Fig. 7. This map will be centered on the previously determined MV, therefore the resulting best vector represents an additional displacement, that will be added

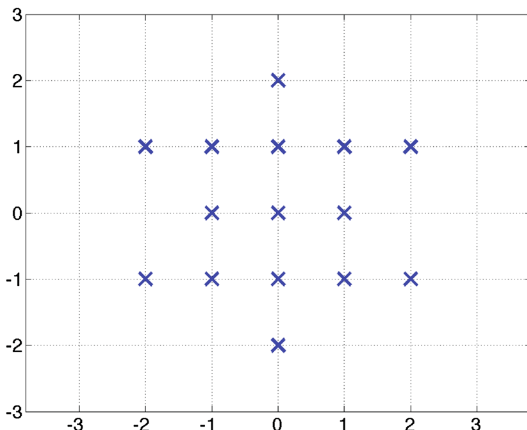


Fig. 7 The displacement map for the motion search refinement. This map is centered on the vector obtained through the main motion vector search, using the displacement map of Fig. 5, and allows to obtain the best motion vector with 1-pixel resolution

to the vector obtained in the main search. At the end, the final MV (the one characterized by the lowest SAD) is stored back into the GPU's device memory, together with its SAD value representing the MV cost.

3.4 Assembly-optimized SAD

The most demanding part of the motion estimation procedure is the initial computation of the SAD for each vector of the search map. During this procedure, as described in Sect. 3.3.2, each GPU thread computes the absolute difference of four pixels (as shown in Fig. 6) and stores the sum of these differences. This operation can be further optimized by exploiting CUDA's PTX ISA (parallel thread execution instruction set architecture), the low-level instruction set of the CUDA cores [7]. In particular, we can resort to PTX SIMD instructions, which perform, within each single GPU thread, the same arithmetic operation in parallel on multiple data. SIMD4 instructions, in particular, consider a 32-bit-wide word as a set of four independent 8-bit operands. In this particular situation, we can exploit the PTX-SIMD4 instruction `vabsdiff4()` which computes the absolute difference of four byte-wide pairs of operands (regarded as unsigned integer values) and sums them into a 32-bit register, as schematized in Fig. 8: by means of this instruction, it is possible to condensate the entire four-pixel SAD step, described in Sect. 3.3.2, into one single execution of `vabsdiff4()`, provided that the luminance values of the four considered pixel pairs are organized as four adjacent bytes within two 32-bit registers. For this reason, this approach is maximally efficient if applied to four adjacent pixels, which can be loaded into a 32-bit-wide local variable (or a local 32-bit register) with a single access to the device memory.

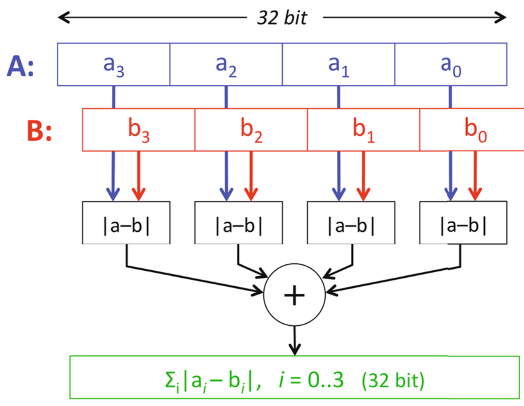


Fig. 8 Computation diagram of the PTX SIMD4 instruction `vabsdiff4()`

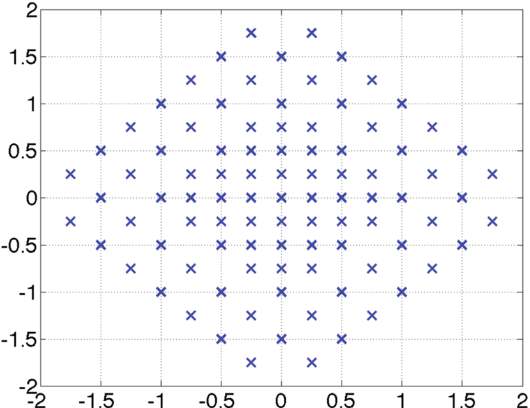


Fig. 9 The 93-point sub-pixel diamond-shaped search space, employed in the motion estimation refinement procedure that exploits the GPU's texture memory interpolation capabilities

After then, the four-pixel SAD is directly computed by calling `vabsdiff4()`. The procedure works as follows:

SAD Computation:
for all threads $tid \leftrightarrow MV(tid) \in$ {Search Map} **do**
  for all sets of 4 adjacent pixel $4i \in$ Macroblock **do**
    $A \leftarrow$ `(int32)` $CF[\ 4i\ ]$
    $B \leftarrow$ `(int32)` $RF[\ 4i + MV(tid)\ ]$
    $DM[tid][i] \leftarrow$ `vabsdiff4`(A, B)
  end for
end for

where CF and RF are the current frame and the reference frame, respectively. This algorithm takes 64 iterations per macroblock, exactly as much as the standard SAD, but in this case, besides the advantage of performing 4 absolute differences and 3 sums with a single assembly instruction, each thread makes only two 32-bit-wide accesses to the device memory (to CF and RF) at each iteration, instead of eight 8-bit-wide accesses/iteration of the approach in Sect. 3.3.2. As the presented experimental results will show, the increased efficiency of the optimized SAD, compared to the former, leads to speed improvements that become notable only in conditions of frequent call of the motion estimation procedure by the encoding process (depending on the encoding parameters like the speed factor, etc.).

3.5 Exploiting the texture memory for sub-pixel estimation

In VP8, MVs have quarter-pixel precision [12]. This approach gives a better visual quality by increasing the signal-to-noise ratio (SNR), but requires pixel interpolation from the current reference frame. To this end, VP8 synthesizes each pixel by exploiting a 6-tap interpolating filter, both horizontally and vertically.

Our ME algorithm implements sub-pixel motion search as a refinement of the integer MV returned by the previous steps. This approach relies on the usage of the GPU texture memory for the synthesis of the interpolated pixels. The reference and input frames are stored as textures, so that the required sub-pixel luminance values are obtained by exploiting the hardware interpolation capabilities of the GPU.

To this purpose, we have implemented a texture-based search employing the sub-pixel diamond-shaped search map shown in Fig. 9, which is composed of 93 search vectors in the interval $[-1.75, 1.75] \times [-1.75, 1.75]$ with spatial resolution of 0.25 or 0.5 pixel—the former in the most internal area of the diamond and the latter along the edges.

3.6 CPU–GPU architecture tuning

Generally, in GPU programming, the most challenging part of a development is the task of tailoring the algorithm to the available GPU architecture in such a way as to reach the maximum performance. There is, however, no general rule to achieve significant speed-ups on GPU; the optimization and parallelization choices that yield the best performance can be quite different from algorithm to algorithm.

A commonly suggested optimization approach in CUDA programming is to maximize the GPU occupancy, defined as the ratio between the actual number of threads actively running and the maximum number of threads that can be scheduled on a GPU. A high occupancy value generally means a high level of parallelism, thus leading to high performance. High occupancy is normally obtained by permanently having a large number of threads scheduled on the cores; this way, whenever a thread execution stops because of a global-memory access, another thread can be processed by the core, thus avoiding to waste time for memory latencies (the so-called latency hiding [14]). In this approach, a

fast switching among threads is obtained by partitioning the shared memory, to provide shared memory space to threads running on the same SMX but belonging to different blocks. For this reason, it is suggested to keep the allocated shared memory space as small as possible.

On the other hand, the shared memory is a precious resource to drastically reduce the latency of threads waiting for global-memory access, and the efficiency of some algorithms is seriously compromised by a limitation of the shared memory space. The need to find a good trade-off between occupancy and use of shared memory is known in literature [18].

Our algorithms actually represent a typical exception to the described 'high-occupancy' rule: the need for massive parallel work on independent 256-pixel macroblocks takes such a great advantage from large shared memory space, that the best results in terms of speed are achieved by maximizing the shared memory space, even at cost of a lower occupancy. This peculiar behavior can be experimentally shown by measuring both occupancy and speed of the different approaches (using the standard CUDA profiling tool `nvprof`). We compared two different setups related to kernels implementing the techniques described in Sect. 3.3: in the first setup, the kernel exploits all the available space of shared memory within a CUDA block, whereas in the second kernel the use of shared memory is reduced, to maximize the GPU occupancy (MAX-OCC).

The experiments have shown that MAX-OCC indeed obtains a much better occupancy, but the MAX-SM setup achieves a significantly higher speed. As an example, Table 1 reports a comparison of the obtained results, in terms of occupancy and speed, for one of the sequences used for the experimental tests (sequence Rush Hour, see Table 2 in Sect. 4) using the GeForce GTX 980 device. The results in Table 1 confirm that our "unorthodox" approach, which maximizes the deployment of shared memory, in spite of being characterized by a much lower level of occupancy with respect to the more "canonical" MAX-OCC approach ($\sim$ 12% vs $\sim$ 72%, in average), is approximately $\times$4.4 faster, therefore it represents the better optimization approach to this algorithm.

4 Experimental results

To give a detailed account of the GPU-accelerated behavior with respect to quality and speed, we tested the encoder with several different configurations, considering both mono- and multi-thread CPU settings. The speed/quality performance of any encoder strongly depends on the encoding parameters. For this reason, to perform significant experiments, three reasonable speed/quality trade-offs have been selected:

- GPU-FAST, which implements the optimized techniques described in Sect. 3.3;
- GPU-SPLITMV, which adds to the previous approach a sub-block motion estimation that exploits the assembly-optimized SAD described in Sect. 3.4;
- GPU-TEX, which performs both sub-block ME, as in GPU-SPLITMV, and sub-pixel ME by exploiting the hardware sub-pixel linear interpolation provided by the GPU texture memory, as described in Sect. 3.5.

All these modes are available in our version of the open source encoder software released in [4] as fork of the WebM project, and can be selected by setting the `cuda-me` command line parameter to 1, 2, or 3, respectively.

To assess speed and quality, the encoding time (ET) and the peak signal-to-noise ratio (PSNR) were calculated. The PSNR was obtained by comparing the original video sequence to the one obtained by (a) encoding it with the considered encoder and (b) decoding it with the WebM VP8 reference decoder. The ET is defined as the execution time of the overall encoding process.

4.1 General setup

For the experiments, we have selected five representative video sequences (selected among the most commonly used in literature [27]) from the `Xiph.org` video repository [10]. Their main features are summarized in Table 2. All the test sequences shows moving fields, either due to camera or object motion, hence triggering the ME during the encoding of each frame. As reference encoder [1], we have used the latest stable release of `libvpx` (1.6.0), compiled with `gcc` 5.4.0 and `yasm` 1.3.0; all hardware accelerators were enabled during compilation, and were therefore available at

Table 1 Occupancy vs speed with different exploitations of the shared memory—kernel: GPU-FAST; test sequence: Rush hour (see Sect. 4)

GPUFAST setup kernel	Occupancy %			Time (s)		
	Average	Min	Max	Average	Min	Max
MAX-OCC	72.42	69.61	75.37	17.23	14.87	24.75
MAX-SM	12.43	12.25	12.49	3.90	3.73	4.13

Table 2 Summary of sample clips' specifications

Clip	# frames	Resolution	Ratio	Format
Old town	500	1080p	16:9	YUV420
Pedestrian area	375	1080p	16:9	YUV420
Rush hour	500	1080p	16:9	YUV420
Tractor	690	1080p	16:9	YUV420
Big buck bunny	90	2160p	16:9	YUV420

Old town: Camera on helicopter, slow constant and non-uniform motion, highly detailed. Pedestrian area: Fixed camera, big sprites fastly moving in front of the camera, chaotic object movements. Rush hour: Fixed and tilting camera, big and small sprites slowly moving, highly detailed and noisy. Tractor: Camera following a big moving sprite, panning, zooming chaotic object movement. Big buck bunny: CGI sequence, panning, fast moving sprite, chaotic object movement, scene change

execution time. To the best of the authors' knowledge, such encoder is the best performing VP8 encoder available when this paper was written. The code running on the GPU was compiled with `nvcc` 8.0.44 and the NVIDIA CUDA Toolkit was the latest available at the moment of writing (8.0) [6].

The tests have been performed on a workstation with two Intel Xeon E5-2620v3 CPU's, each with six physical cores, 2.40 GHz clock, 64 GB RAM memory, 2 TB disk space and Linux Ubuntu 16.04 as operating system. This machine was equipped with an NVIDIA GeForce GTX980 GPU, featuring 2048 Maxwell cores with Compute Capability 5.2 and 4 GB of video memory. To execute the tests and post-process the results, we followed the official contribution guidelines of the WebM Project, using the provided scripts and automated tools [9].

4.2 Single-thread performance analysis

A first set of tests was performed by assigning a single CPU core to the entire VP8 encoding process, both in the reference and in the proposed encoder. Each video sequence in Table 2 was processed both by the reference encoder and by the three GPU-accelerated encoders, for different values of `target-bitrate`, while all the other encoder control parameters remained unchanged. In VP8, the `target-bitrate` parameter sets, the objective output bitrate (in kbps) that the encoder should achieve [3]. In the tests, this parameter took values in the range 2000–8000 kbps with steps of 2000 kbps for the full HD sequences; in the range 5000–25,000 kbps with steps of 5000 kbps for the 4 k sequence. The complete set of parameters used in the encoding session is as follow:

Fig. 10 Encoding performance achieved by the different GPU kernels on the test sequences in Table 2. On the left: average image quality, expressed as average PSNR (dB), as function of the data rate. On the right: average encoding frame time (s)

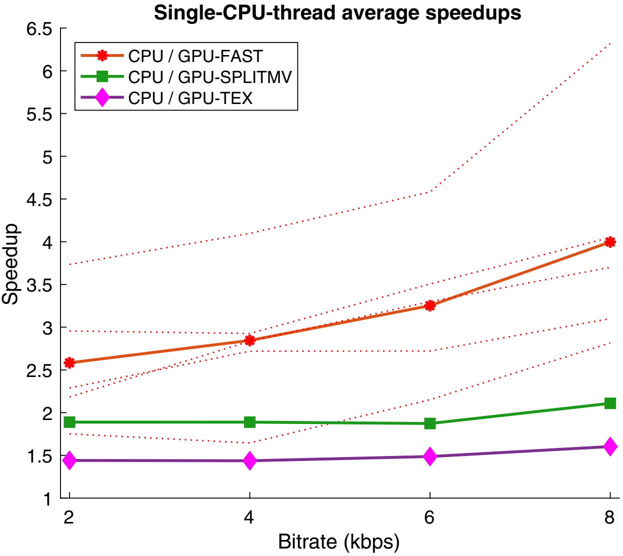


Fig. 11 Solid lines represent the average speed-ups achieved by the three GPU-accelerated encoder versions with respect to the CPU reference encoder, running on the five sequences. Dotted lines show the individual speed-up achieved by the sole GPU-FAST on all sequences

Table 3 Average speed-ups achieved by the three kernels: GPU-FAST, GPU-SPLITMV, and GPU-TEX on three different CUDA devices: GeForce GTX 980 (Maxwell), Tesla K40 (Kepler) and Tesla P100 (Pascal)

Speed-up: GPU architecture comparison				
GPU architecture (CC)	#cores (cores/SM)	FAST	SPLITMV	TEX
Maxwell (5.2)	2048 (128)	×3.10	×1.90	×1.48
Kepler (3.5)	2880 (192)	×2.77	×1.63	×1.25
Pascal (6.0)	3584 (64)	×2.94	×1.61	×1.33

The first column reports the GPU architecture type (together with its compute capability, CC), the second one reports the total number of cores (and cores per SM). The following three columns report the average speed-up achieved by the three kernels, averaged over all the considered speed/quality (bitrate) settings and over all five test sequences

GPU-FAST by red lines, GPU-SPLITMV by yellow lines, and GPU-TEX by violet lines.

The PSNR curves show that the quality provided by GPU-TEX is very close to the one achieved by the reference encoder for every test sequence, while GPU-FAST and GPU-SPLITMV yield slightly lower values than the reference encoder; however, these small differences are hardly perceivable during the playback of the encoded sequences.

The three versions of the proposed encoder always exhibit significantly lower ET values than the reference encoder. The GPU-FAST version achieves on average a ×4 speed-up, with a peak value of ×6.3 in one test, while GPU-SPLITMV and GPU-TEX show an average speed-up of ×2 and ×1.5, respectively. As expected, the complexity increase of GPU-SPLITMV and GPU-TEX kernels affects the overall speed-up of the entire encoding process. However, this drawback is partially balanced by the exploitation of the Assembly-optimized SAD, which decreases the SAD computation time up to 75%. The average speed-ups achieved by the three GPU accelerated kernels computed on the pool of test clips of Table 2 are shown in Fig. 11.

To test the efficiency of the technique against different GPU architectures (e.g., different number of cores per SM), we also tested the proposed kernels on three different CUDA architectures, namely Maxwell (on a GeForce GTX 980—compute capability: 5.2), Kepler (Tesla K40—compute capability: 3.5) and Pascal (Tesla P100—compute capability: 6.0). Table 3 reports the average speed-ups achieved by the overall encoding processes, distinct for GPUs and kernels. In particular, the average values have been computed taking into account all video sequences listed in Table 2 and all bitrates chosen in the previous test. As Table 3 shows, the performance of the three devices is comparable, with a slight predominance of the Maxwell architecture. This predominance, in spite of the lower clock rate and memory bandwidth of the Maxwell device, is a consequence of the algorithmic design choices, which tailored the kernel to this architecture, such as the use of 128-thread blocks on a 128 cores per SM architecture to encode a single macroblock.

```
vpxenc inputFile.y4m -o outputFile.webm  --cpu-used=0 --target-bitrate=<br>
--auto-alt-ref=1 --lag-in-frames=25 --kf-min-dist=0 --kf-max-dist=99999
--static-thresh=0 --min-q=0 --max-q=63 --arnr-type=3 --arnr-maxframes=7
--arnr-strength=3 --drop-frame=0 --best --psnr --codec=vp8
```

Figure 10 shows the PSNR (left column) and the ET (right column), as a function of the `target-bitrate`, for the reference and the three versions of the proposed VP8 encoder, considering all video sequences listed in Table 2. Reference encoder CPU is denoted by cyan dash–dot lines,

4.3 Multi-thread performance analysis

The proposed GPU–CPU cooperative encoder has also been tested in a multi-thread environment, to assess its scalability and to verify the effectiveness of the cooperation among the

Fig. 12 Performance achieved in multi-thread environment by the GPU-FAST kernel on the considered test sequences. Left: average PSNR (dB) as function of the data rate. Right: average frame time (s)

GPU resources and multiple CPU cores. To this aim, we performed the same tests described in the previous section, using the same hardware configuration as the single-thread test, but varying the number of CPU threads assigned to the process. Hyperthreading was active in all the following tests, with each thread pinned to a processing core.

To assess the performance in a multi-thread environment, we have carried out two different experiments. Since the main focus of this work is the computation speed, the main experiment is focused on a detailed analysis of GPU-FAST, our fastest encoder. In addition, experiments have been carried out with the other two encoders, GPU-SPLITMV and

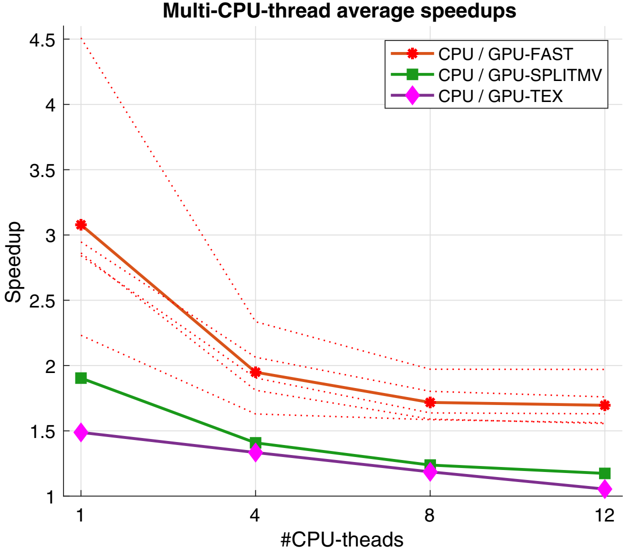


Fig. 13 Solid lines represent the average speed-ups achieved by the three GPU-accelerated encoder versions with respect to the CPU reference encoder. Values have been computed averaging on both different encoding bitrates and the five sequences. Dotted lines show the individual speed-up achieved by the sole GPU-FAST for all sequences

GPU-TEX, to evaluate how their speed varies over the number of available CPU cores.

Figure 12 shows the results achieved by GPU-FAST, in terms of PSNR (left column) and ET (right) versus the datcolumna rate, when launching 1, 4, 8, and 12 CPU threads. The reference encoder performances (dash–dot lines) are compared with the GPU-FAST encoder (solid lines), both in a multi-threaded environment. Each line, both dash–dotted and solid, denotes a test run where a different number of CPU threads was assigned to the process, specifically 1, 4, 8 and 12.

The plots show that, as in the single-thread case, the GPU–CPU approach keeps the image quality unchanged, while obtaining a substantial speed-up. They also confirm that the GPU-FAST kernel scales well in a multi-thread environment, as the computation speed is roughly proportional to the number of threads. These graphs indicate a sort of "equivalence" between GPU–CPU and CPU-only architectures: considering, for instance, the execution time, the graphs suggest that a system with two CPU cores and one GPU device can provide approximately the same performance as a four-CPU core system.

Figure 13 shows the results of the speed-up tests for the three developed kernels, reported in a comparative way. The values reported in the plots represent the speed-up factor (defined as the processing time ratio, with vs without GPU) averaged over the different encoding data rates, for all the considered test sequences. The plots show that, as the number of CPU cores increases, the three kernels actually lose performance, with respect to the CPU-only algorithm. The most significant loss occurs for GPU-FAST, where the speed-up factor decreases by approximately 45% going from 1 to 12 CPU cores. The performance loss with many CPU cores available can be explained as a consequence of the system architecture, where the communication channel between the host and the GPU (the PCI-Express bus) is shared by all CPU cores, thus causing increasing communication latencies as the number of CPUs grows.

5 Conclusions

In this paper we present the design and the implementation of a VP8 video encoder that exploits a cooperative interaction between multicore (CPU) and manycore (GPU) resources. As the experimental results have shown, the proposed technique can achieve, in the best conditions, up to ×6 speed-up with respect to the mostly optimized CPU-only version of the VP8 WebM encoder [1], with minimum degradation of the visual quality. Moreover, the presented results show how the obtained performance improvements remain so significant also when multiple CPU cores are available, proving the effectiveness of the multicore/manycore cooperation and the good scalability of the proposed approach.

Among all possible directions for future research work, we are interested in further investigating the performance in multi-thread environments, which can be considered as the standard situation in HPC scenarios.

Considering possible further software development, the encouraging results obtained in terms of speed-up vs. quality suggest to extend the proposed approach to the most recent encoding standards, such as VP9, AOMedia AV1, and HEVC. Despite the significant differences between these new standards and VP8, the way they all resort to motion estimation is quite similar, therefore such extensions could represent a reasonable effort yielding significant improvements.

References

1. `libvpx` code repository. https://chromium.googlesource.com/webm/libvpx (2016)
2. WebM: an Open Web Media Project. https://www.webmproject.org (2016)
3. WebM project—VP8 encode parameter guide. https://www.webmproject.org/docs/encoder-parameters (2016)
4. Italtel-Unimi—github repository. https://github.com/Italtel-Unimi (2017)
5. NVIDIA CUDA C programming guide, Version 8.0. NVIDIA Corp. http://docs.nvidia.com/cuda/cuda-c-programming-guide (2017)

6. NVIDIA CUDA Toolkit 8.0, NVIDIA Corp. https://developer.nvidia.com/cuda-toolkit (2017)
7. NVIDIA, Parallel Thread Execution ISA—Application Guide, v5.0, NVIDIA Corp. http://docs.nvidia.com/cuda/pdf/ ptx_isa_5.0.pdf (2017)
8. `libx264` project and code repository. http://www.videolan.org/developers/x264.html (2017)
9. WebM Project—Contribution guidelines. Tech. rep. https://chromium.googlesource.com/webm/contributor-guide (2017)
10. Xiph.org video test media (derf's collection). https://media.xiph.org/video/derf (2017)
11. Albanese, A., Crosta, P., Meani, C., Paglierani, P.: Gpu-accelerated video transcoding unit for multi-access edge computing scenarios. In: The Sixteenth International Conference on Networks (ICN2017), Venice, 23–27 April, pp. 143–147 (2017)
12. Bankoski, J., Koleszar, J., Quillio, L., Salonen, J., Wilkins, P., Xu, Y.: VP8 data format and decoding guide (rfc 6386). http://www.rfc-editor.org/info/rfc6386 (2011)
13. Bankoski, J., Wilkins, P., Xu, Y..: Technical overview of VP8, an open source video codec for the web. In: 2011 IEEE International Conference on Multimedia and Expo, pp. 1–6 (2011)
14. Cheng, J., Grossman, M., McKercher, T.: Professional CUDA C Programming. Wiley, Indianapolis, Indiana (2014)
15. Cheung, N.M., Fan, X., Au, O.C., Kung, M.C.: Video coding on multicore graphics processors. IEEE Signal Process. Mag. **27**(2), 79–89 (2010)
16. CISCO Corporation: The Zettabyte Era: Trends and Analysis. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html (2017)
17. Comi, P., Crosta, P.S., Beccari, M., Paglierani, P., Grossi, G., Pedersini, F., Petrini, A.: Hardware-accelerated high-resolution video coding in virtual network functions. In: 2016 European Conference on Networks and Communications (EuCNC), pp. 32–36 (2016)
18. Hayes, A.B., Li, L., Chavarría-Miranda, D., Song, S.L., Zhang, E.Z.: Orion: A framework for gpu occupancy tuning. In: Proceedings of the 17th International Middleware Conference, Middleware '16, pp. 18:1–18:13. ACM, New York (2016)
19. Jiang, W., Wang, P., Long, M., Jin, H.: A novel parallelized motion estimation algorithm for GPU based video encoding. In: 2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM), pp. 1–8 (2016)
20. Ko, Y., Yi, Y., Ha, S.: An efficient parallel motion estimation algorithm and x264 parallelization in cuda. In: Proceedings of the 2011 Conference on Design Architectures for Signal Image Processing (DASIP), pp. 1–8 (2011)
21. Marth, E., Marcus, G.: Parallelization of the x264 encoder using opencl. In: International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2010, Los Angeles, July 26–30, 2010, Poster Proceedings, p. 72:1 (2010)
22. NVIDIA Corporation: High performance video encoding with NVIDIA GPUs. In: 2016 GPU Technology Conference. http://on-demand.gputechconf.com/gtc/2016/ presentation/s6226-abhijit-patait-high-performance-video.pdf (2016)
23. Paglierani, P., et al.: Network functions implementation and testing. Tech. rep., T-NOVA Project Deliverable D5.31 http://www.t-nova.eu/results (2015)
24. Paglierani, P., Grossi, G., Pedersini, F., Petrini, A.: GPU-based VP8 encoding: Performance in native and virtualized environments. In: 2016 International Conference on Telecommunications and Multimedia, TEMU 2016, pp. 1–5 (2016)
25. Radicke, S., HaHn, J.-U., Wang, Q., Grecos, C.: Many-core HEVC encoding based on wavefront parallel processing and GPU-accelerated motion estimation, In: Obaidat, M., Holzinger, A., Filipe, J. (eds.) E-Business and telecommunications: 11th international joint conference, ICETE 2014, Vienna, Austria, 28–30 August 2014. Communications in computer and information science, vol. 554, pp. 393–417. Springer (2015)
26. Sankaraiah, S., Shuan, L.H., Eswaran, C., Abdullah, J.: Performance optimization of video coding process on multi-core platform using gop level parallelism. Int. J. Parallel Program. **42**(6), 931–947 (2014)
27. Shah, N.N., Dalal, U.D., Prajapati, P.H.: Multi-point search pattern for fast search motion estimation of high resolution video coding. In: I. J. Image, Graphics and Signal Processing (IJIGSP), pp. 60–68 (2015)
28. Shahid, M.U., Ahmed, A., Martina, M., Masera, G., Magli, E.: Parallel h.264/AVC fast rate-distortion optimized motion estimation by using a graphics processing unit and dedicated hardware. IEEE Trans. Circuits Syst. Video Technol. **25**(4), 701–715 (2015)
29. Zhu, S., Ma, K.K.: A new diamond search algorithm for fast block-matching motion estimation. IEEE Trans. Image Process. **9**(2), 287–290 (2000)