On the Engineering of a Component for Distributed Motion Estimation

Authors omitted for blind review

Abstract—**The increasing availability of computational resources is enabling the construction of optimized encoders for H.264 and other video standards. In such scenario, keeping the same interface between the encoding stages is highly desirable since it allows for the use of new optimized versions of algorithms while keeping intact the overall design of the encoder. In this paper we present the design and implementation of a component for *motion estimation* which keeps the same interface for all of its three implementations: Cell BE, Multicore IA32, and dedicated hardware. Motion estimation is an important stage for optimization since it consumes around 90% of the total encoding time of raw video into H.264. The proposed component is evaluated according to encoding time and PSNR, showing that it is possible to develop an optimized component while keeping a single interface among distinct implementations.**

Index Terms—**I.4 [Image Processing and Computer Vision]: Compression (Coding)**
I.3 [Computer Graphics]: Parallel processing
D.2 [Software Engineering]: Modules and Interfaces

I. Introduction

Motion Estimation (ME) is a technique used during video compression to explore temporal redundancy in video sequences. Temporal redundancy arises from the fact that neighboring frames often share similar regions of pixels. Therefore, the goal of ME is to *estimate* the shifting of such similar regions across neighbor frames, thus allowing for difference-based encoding. In block-based ME, the displacement of similar regions is represented by *motion vectors*, which are computed by *Block-Matching Algorithms* (BMAs). Standards like the ISO MPEG series and the ITU-T H26x are examples of encoders that use ME to improve the compression ratio of video streams [1]. In fact, about 90% of the total encoding time in a H.264 encoder is spent in the ME stage [2]. Consequently, ME optimization is a relevant issue for H.264 and video encoding in general.

Many strategies have been proposed for ME optimization. These strategies can be divided in three categories: algorithmic optimizations [3], [4], [5]; parallelization of algorithms [6], [7]; and dedicated hardware implementations [8], [7]. The search for new methods to optimize the ME process is an important issue to enable either the construction of real-time H.264 encoders or their implementation in devices with limited computational resources, since these optimizations aim to reduce ME complexity. Furthermore, the wide range of computer architectures available for the deployment of video encoders spans a myriad of specific optimized implementations of encoding algorithms [6], [7], [8]. In such scenario, keeping a homogeneous interface for encoding and decoding components allows the use of different optimized versions of algorithms (e.g. ME) without the need of modifying the whole encoder/decoder design, while maintaining compatibility with video coding standards.

In this paper, we show how a careful design process can be used to produce a ME component which has the same interface for different implementations. Our *Distributed Motion Estimation Component* (DMEC) uses an optimization strategy based on picture partitioning, executing ME in parallel on distinct functional units. In order to demonstrate the reusability provided by DMEC's interfaces, we provide implementations using three distinct computer architectures: the Cell Broadband Engine (Cell BE), an Intel IA32 Multicore Architecture, and a dedicated hardware generated using high-level synthesis (HLS). All implementations of DMEC have the same interfaces and can be used without modifications by an H.264 encoder. For each implementation of DMEC, we have evaluated the encoding time and objective quality, which is measured by the Peak Signal-to-noise Ratio (PSNR).

The remaining of this paper are organized as follows: Section II makes an overview of issues and strategies for ME optimization; Section III presents DMEC and how it was designed; Section IV describes the implementation of DMEC; Section V evaluates DMEC both in isolation and integrated with the standard H.264 encoder; Section VI closes the paper with our final considerations.

II. Strategies for ME Optimization

ME is a technique employed to explore the similarity between neighboring pictures in a video sequence. Figure 1 illustrates the ME process for the neighboring pictures *A* and *B*. By searching for similarities between these two pictures, it is possible to determine which blocks from picture *A* are also found in picture *B*. Such displacement of picture blocks is encoded by *motion vectors* (represented by the small arrows in the bottom side of Figure 1). There are several strategies to optimize the execution time of ME: fast-search algorithms, macroblock subsampling, sample truncation, multi-resolution ME, subsampled motion-field estimation, and parallel and hardware implementations of algorithms.

Fast-search algorithms are BMAs that look only in specific positions of the search window [4], [5]. The search window defines the region of the reference frame that is scanned for a macroblock partition similar to the current one. Only the motion vectors that correspond to the match with the lowest *motion cost* are chosen. The main drawback of this approach is that, since some positions of the search window are discarded, it is possible to find suboptimal motion vectors.

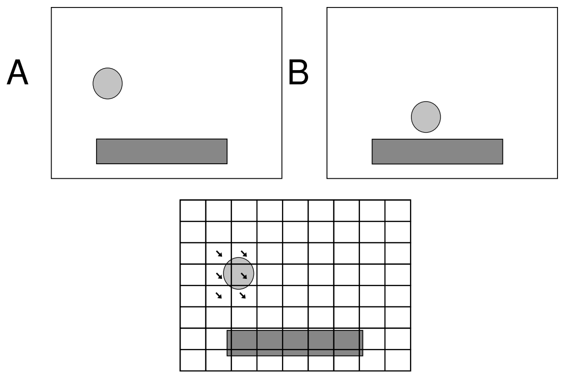


Figure 1. Motion Estimation

Two other strategies to optimize ME during block-matching are *macroblock subsampling* and *sample truncation*. Macroblock subsampling takes into consideration only a macroblock partition (i.e. some samples of a macroblock) while the matching for a position of the search window is being performed. Sample truncation is performed by ignoring the least significant bits of a sample. These strategies have been used separately [9] (subsampling), [6] (truncation), and integrated [3] (subsampling and truncation).

Multi-resolution ME is the strategy in which the motion vectors are computed for distinct resolutions of the same frame. Motion vectors computed in a more coarse level can be successively refined until the finest level (higher resolution). If the search is performed sequentially as in [10], the time of ME can be increased due to the dependencies between distinct levels. On the other hand, if the search is executed in parallel for each resolution level, as in [6], hardware functional units need to be replicated. A similar technique is *subsampled motion-field estimation* [9], which is based on the assumption that motion vectors of neighboring blocks tends to be similar. Thus, for each block, only a set of motion vectors (a motion-field) is computed, while the others are interpolated.

Other strategies for optimizing motion estimation are based on finding parallelism in ME stages, especially in the block-matching algorithms, in order to execute them simultaneously. These parallel strategies commonly have been the base for dedicated hardware implementations. The *Sum of Absolute Differences* (SAD) is a metric of error used in block-matching algorithms. This technique is frequently parallelized using functional units in hardware [6], [7], Hardware implementations of shared buffers for frame data are also common [7], [8].

III. DMEC

The Distributed Motion Estimation Component (DMEC) was developed following a domain engineering process. This process resulted in the isolation of aspects such as synchronization and communication from the ME algorithm itself. The central element of DMEC is described by the interface *PictureMotionEstimator* shown by Figure 2. Such interface describes entities that are responsible to perform ME of a whole picture (or a partition of a picture). An object of the type *PictureMotionEstimator* knows all block modes of the video standard in use (e.g. H.264), and works according to a specific error metric, such as SAD. Such error metric is used to determine the similarity between neighboring pictures. The ME itself is computed by the *match* method that takes the current and the reference pictures (respectively pictures *A* and *B* from Figure 1) and returns their "counterpart" which is composed by the motion vectors and the motion cost.

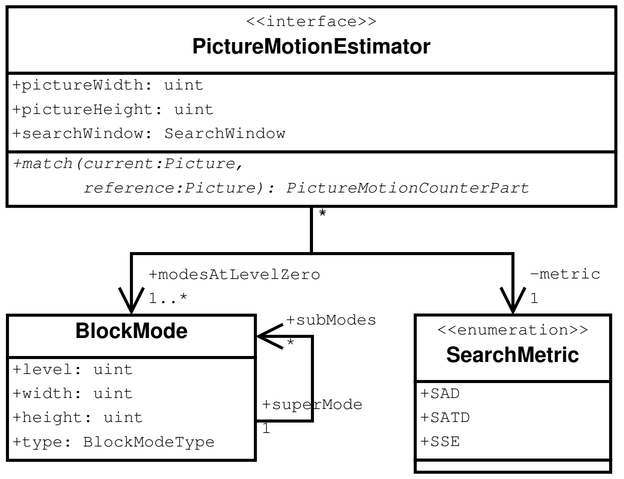


Figure 2. *PictureMotionEstimator* interface

DMEC was designed to be self-contained, thus, the *match* method computes ME for all picture's macroblocks and for all block modes without any dependency from another encoder element. Therefore, the return of the method *match*, which is accessed using the *PictureMotionCounterpart* interface, is a multidimensional vector containing all motion vectors and costs for each macroblock and block mode used in the ME.

A. Parallelization Strategy

The parallelization strategy employed by DMEC is based on data partitioning, in which the unit to be partitioned is the picture. Figure 3 shows all picture partition modes available. All picture partitions dimensions must be multiple to the macroblock dimension (16x16 pixels) to avoid having a macroblock broken between two neighboring partitions. If desired, it is possible to include new partition modes by specifying how the partition should be performed.

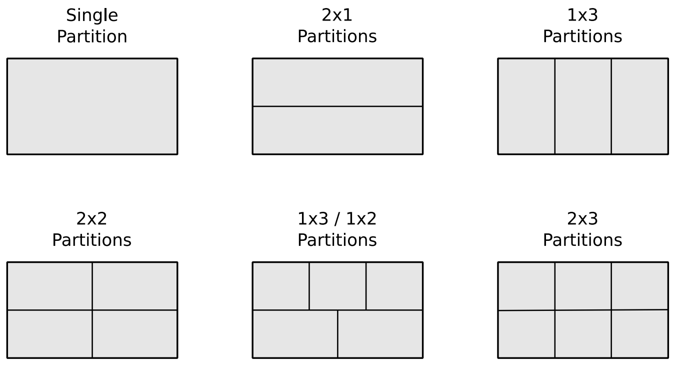


Figure 3. Supported picture partitions modes

In order to improve the performance of ME, each picture partition is then assigned to a *Worker* module which executes in a specific functional unit, such as a core of a multicore processor or a dedicated hardware element. There is also a *Coordinator* module, responsible to define the picture partition for each *Worker* and to provide them with pictures to be processed. The *Coordinator* is also responsible to gather results generated by *Workers* (motion cost and motion vectors) and to deliver these results back to the encoder.

Each *Worker* module computes ME by using a Block Matching Algorithm (BMA), which itself also implements the *PictureMotionEstimator* interface. Thus, as shown by Figure 4, all BMAs follow a single interface, therefore, it is possible to replace a BMA for another, according to the encoder needs.

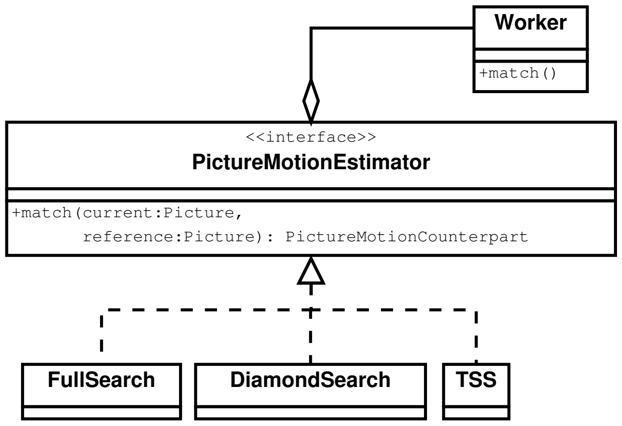


Figure 4. Worker and Block Matching Algorithm

B. Data Transference and Synchronization

We have used a shared memory model for exchanging data between the *Coordinator* and *Worker* modules, although, in practice, such memory can be implemented as a memory block in a single address space (as is the case for our implementation for the IA32 architecture and dedicated hardware), or can be constituted by multiple memory blocks, each one using its own address space (as is the case for our implementation for the Cell BE architecture).

Figure 5 details the interaction between *Coordinator* and *Worker* modules. At the beginning of the ME operation DMEC assumes that all pictures are in the main memory. Using the *TransferenceManager* interface, *Worker* modules can obtain the samples of their picture partitions. In the case of a memory block in a single address space, the operations of *TransferenceManager* are just pointer manipulations. On the other hand, while using memory blocks in distinct address spaces, the operations of *TransferenceManager* are mapped to real memory transferences, such as Direct Memory Access (DMA) requests. Also, by using the *TransferenceManager*, *Worker* modules can publish the computed motion vectors and motion costs. *Worker* modules should wait for a signal from the *Coordinator* module indicating that there are pictures to the processed. Similarly, the *Coordinator* module should wait for the ME results generated by the *Worker* modules. Such synchronization operations are implemented by the *SynchronizationManager* interface, which specifies barrier-like mechanisms, which can be implemented using operating system operations or using directly dedicated hardware elements. In the case of a sequential operation with a single partition containing the whole picture, the operations of *SynchronizationManager* are canceled.

IV. Implementation

This section describes our implementation of DMEC for the architectures Cell BE, Multicore IA32, and dedicated hardware. For each architecture, we present the reasons why we have implemented DMEC for it. We also briefly review the target architecture and explain how data transference and synchronization was implemented, keeping the interfaces described in Section III.

A. Cell BE

The Cell BE is a processor architecture developed by IBM, Sony, and Toshiba, targeting applications with high thread-level parallelism [11]. Our goal on implementing DMEC at Cell BE is evaluating ME in a high performance and distributed architecture.

Cell BE is an example of a "Multi-computer-on-a-Chip" architecture because it is composed by distinct processor units, each one having its own memory address space. The Cell BE architecture incorporates in the same chip nine independent cores: a Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs). The PPE is the main processing unit, responsible for managing all chip resources. The SPEs are dedicated processing units supporting vectorized floating point code execution. Each SPE has its own local memory, called Local Storage (LC). Hence, all communication between cores is performed explicitly though the Element Interconnect BUS (EIB), a high performance bus.

As Cell BE uses memory blocks in distinct address spaces, we have mapped the operations of *TransferenceManager* to DMA requests using the EIB. Using DMA requests, each *Worker* module of DMEC obtain the samples it needs to perform the ME and deliver back the computed ME results to the *Coordinator* module. The local memory of SPEs is limited to 256KB, including data and code. This imposed several limitations. For instance, using a partition mode with six partitions of 640x544 each at a 1080p resolution requires 680KB for data storage. Thus, only the samples that are currently been used by ME computation (or samples that are going to be used in a near future) are kept inside the SPEs local memory. The samples transferences are hidden from the ME algorithm by the implementation of the *Picture* interface. The *Picture*, in this case, has a local buffer used as a cache and obtains sample as needed through the *TransferenceManager*.

Similarly, the implementation of *PictureMotionCounterpart* for Cell BE also uses DMA requests through *TransferenceManager*, in order to deliver the ME results to the main memory. During the ME computation, previously calculated motion vectors can be used to improve the ME for the current blocks. Such motion vectors are called *motion predictors*. The implementation of *PictureMotionCounterpart* for Cell BE

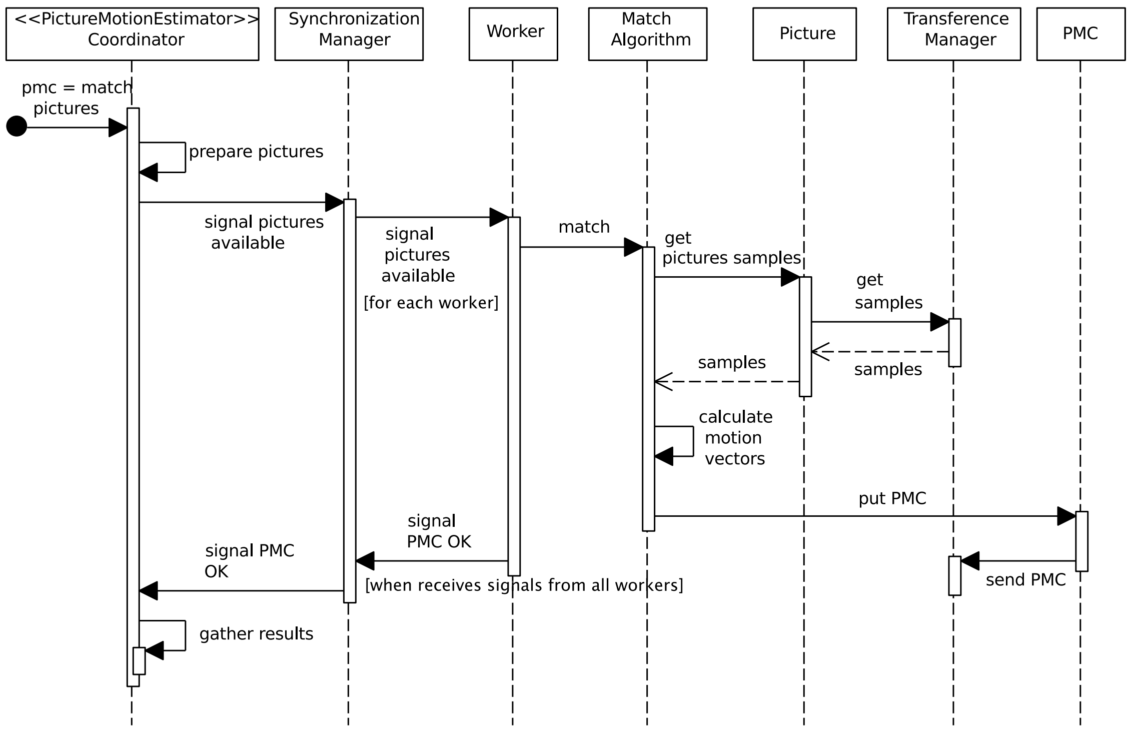


Figure 5. Transference and synchronization between Coordinator and Workers

contains an internal buffer to keep motion predictors inside the local memory of the *Worker* module, avoiding unnecessary DMA transferences between the *Worker* and the *Coordinator* modules.

The *SynchronizationManager* interface, which specifies barrier-like mechanisms, is implemented in Cell BE using the *MailBox*, a hardware resource present on each SPE. By using specific MailBox messages, it is possible to specify when the *Worker* has completed the ME computation as well when there are new pictures to be processed.

B. Multicore IA32

Our goal on implementing DMEC on a Multicore Intel IA32 architecture is evaluating ME for a high performance and Symmetric Multiprocessor (SPM). For our experiments we have used an Intel Core2 Quad, with four symmetric cores.

At the main memory level, all cores on a Multicore IA32 processor share the same memory address space. Thus, for the Multicore IA32 architecture implementation, the operations of the *TransferenceManager* are mapped to direct memory access through pointers. The implementation of *Picture* interface, in this case, is straightforward. It contains the whole picture partition that is going to be used by the *Worker* module.

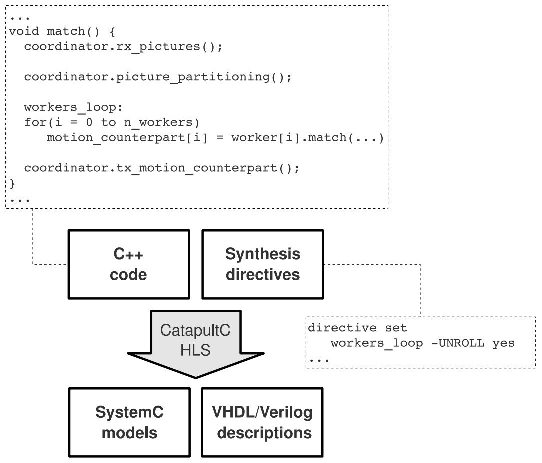
Similarly, the implementation of *PictureMotionCounterpart* for Multicore IA32 also uses the operations of *Transference-Manager*. As there is no practical memory limitation the implementation of *PictureMotionCounterpart* for Multicore IA32, it contains all the motion predictors necessary by the *Worker* modules.

The *SynchronizationManager* interface, which specifies barrier-like mechanisms, is implemented using *pthreads* semaphores of the POSIX thread library.

C. Dedicated hardware

In the last few years, advances in *electronic design automation* (EDA) tools are allowing hardware synthesis from high-level behavioral models. This process is known as *high-level synthesis* (HLS) and allows designers to describe hardware components using languages like C++, and higher-level techniques, such as *Object-Oriented Programming* (OOP). In this work we leverage on HLS technology to obtain a dedicated hardware implementation of DMEC. The main goal of providing hardware DMEC is to show that the same interface defined for the previous implementations can also be used in this case. Additionally, this section highlights the reusability and flexibility of our implementation. Through a simple code refactoring process, we have converted our multicore implementation to a synthesizable behavioral hardware model.

Figure 6 gives an overview of the HLS process and the refactored code. The HLS tool we have used (Calypto's CatapultC [12]) does not work with the concept of tasks and threads, but it extracts parallelism by exploiting the parallelization of *loops*. To cope with such approach, we have serialized the execution of the tasks performed by the *Coordinator* and the *Workers*. The *SynchronizationManager* now defines internal buffers for the pictures and the resulting motion estimation vectors, coordinating the transfer of data between these elements and the main memory. The ME algorithm is implemented in a loop in which each iteration performs the task of one *Worker*. By using *synthesis directives* we can guide the HLS process in order to obtain parallelism. Figure 6 shows that the directive *UNROLL* is set for the *workers_loop*, indicating that this loop should be fully unrolled. In the HLS context, fully unrolling a loop means that all iterations are performed in parallel. There are other forms of loop parallelization. For instance, loops can be pipelined in order to reduce hardware area while keeping the same throughput. To match the multicore behavior in which all workers execute in parallel, we have chosen to fully unroll only the top-lovel loop. Additional directives were set to pipeline inner parts of the ME algorithm whenever possible.



```
...
void match() {
  coordinator.rx_pictures();

  coordinator.picture_partitioning();

  workers_loop:
  for(i = 0 to n_workers)
     motion_counterpart[i] = worker[i].match(...)

  coordinator.tx_motion_counterpart();
}
...
```

```
directive set
   workers_loop -UNROLL yes
...
```

Figure 6. High-level synthesis process for the generation of a dedicated hardware implementation

The output of the HLS process are *register transfer level* (RTL) descriptions in VHDL or Verilog that can be used as input for most RTL synthesis tools. CatapultC also generates cycle- and bit-accurate SystemC models for design verification and evaluation. Figure 7 shows a block diagram of the final RTL implementation generated by CatapultC and its integration with the evaluation infrastructure.

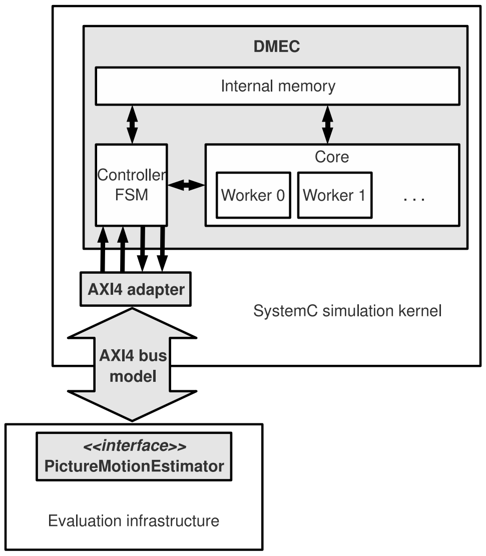


Figure 7. HW DMEC evaluation infrastructure

The final component uses a simple handshaking mechanism for input/output data. This interface is adapted for the AMBA AXI4 bus, a widely used standard for on-chip interconnect of functional blocks. Our performance evaluation is based on simulation of the SystemC model generated by CatapultC. A model of the AXI4 bus model is provided for interfacing the cycle-accurate hardware simulation with the remaining infrastructure.

In the software side, an implementation of the *PictureMotionEstimator* interface serializes the pictures and copies the data to the hardware component internal buffers using a memory mapped interface. Once the ME is complete, the motion vectors are read and used to build the *PictureMotionCounterPart* object which is returned to the test application.

V. Practical Experiments

We have evaluated DMEC in two stages. First, in order to verify how DMEC's performance scales from one to six *Workers* instances, we have evaluated all DMEC implementations in a test case. The test case application mimics the behavior of an H.264 encoder: it provides DMEC with pictures, obtain the ME results (motion vectors and motion cost), and checks if the results are correct. Secondly, in order to assess DMEC influence on the final video quality, we have evaluated all DMEC implementations in the JM H.264 Reference Encoder [13]. The PSNR degradation is computed as the absolute PNSR difference between the original encoder and the optimized ones.

Figure 8 show the speedup of DMEC in there test case application with a different number of *Workers* instances. For such test, we have used an arbitrary set of pictures with a resolution of 1080p. The speedup is normalized to one *Worker* instance (speedup of 1X).

It is worth to mention that for each number of *Worker* instances a different partition mode was used, according to Figure 3. For one *Worker* instance we have used the "Single Partition", for two *Worker* instances we have used the "2x1" partition and so on, up to the "2x3" partition mode (used for six *Worker* instances).

Besides the additional performance obtained by using a higher number of *Worker* instances, the partition mode also has influence on the speedup. The reason of such influence is that, during the partitioning process, the dimensions of the search window shrinks, thus reducing the area of the picture searched for similarities.

In order to evaluate in details the behavior of DMEC for distinct values of encoding bit-rates, we have used the BD-PSNR (Bjøntegaard Delta PSNR) metric using the following values of QP (Quantization Parameter): 16,20,24,28; as described in [14]. It is important to evaluate quality (PSNR) for distinct bit-rates to test whether the approach can be used in distinct scenarios of application. Figure 9 shows the rate-distortions (RD) curves using the original JM encoder and the optimized encoder using DMEC. The video sequence used for this curves was `Crowd Run`, a 1080p sequence with a high amount of motion. Lower values of bit-rate are obtained for higher values of QP since by using higher values for QP more data is discarded, thus increasing the compression ratio.

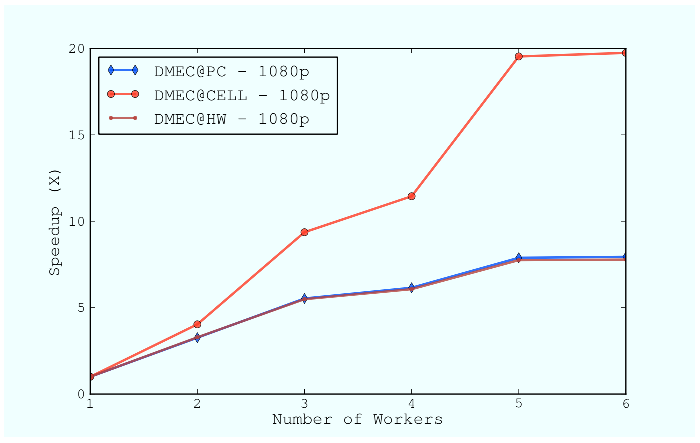


Figure 8. Time performance scalability of DMEC

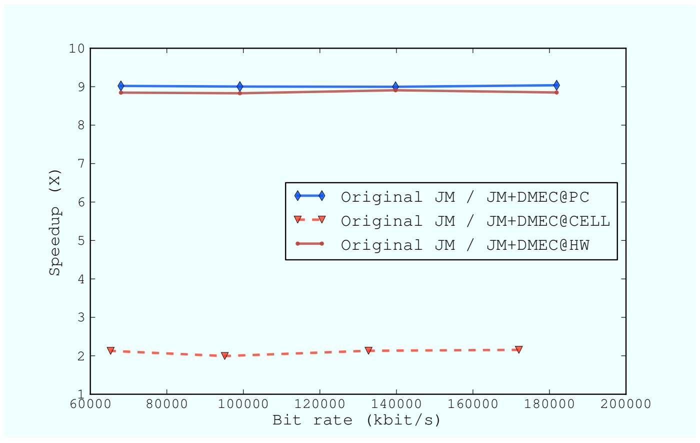


Figure 10. Speedup vs bit-rate of a 1080p sequence

The two curves very near from each other indicates that the DMEC presents a good rate-distortion performance for all the evaluated bit-rates.

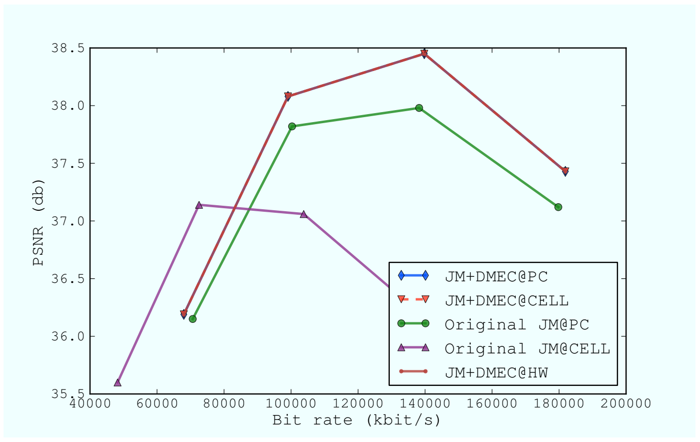


Figure 9. RD curve of a 1080p video sequence

We have evaluated also the speedup obtained in the ME run time while using DMEC for the same QP values we used for BD-PSNR. Figure 10 shows the obtained values while using 6 *Worker* instances. For Muticore IA32, a speedup of around 9 times is obtained for all bit-rate values. For Cell BE this value is about 2 times. A small speedup for the Cell BE, while compared to Multicore IA32 and the dedicated hardware, is due to the memory transferences (picture samples and ME results) which is performed using the DMA requisitions of Cell BE.

VI. Conclusion

In this article, we present the design and the implementation of DMEC, a component for distributed motion estimation. We have shown that, by using a careful design process based on domain engineering, it is feasible to develop a component for ME while keeping the same interfaces for all its implementations. Also, our strategy of ME distribution based on picture partitioning proved to be effective. The visual interdependence between partitions is not significant enough to influence the encoding quality, and allows for a speedup since it enables the simultaneous processing of each picture partition. To demonstrate the flexibility of the proposed interfaces, we have implemented DMEC for the Cell BE, Multicore IA32, and also as dedicated hardware. We also have evaluated DMEC according to encoding time and PSNR, demonstrating an optimized version of ME which keeps the quality of the generated bitstream.

References

[1] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, “Overview of the h.264/avc video coding standard,” *Circuits and Systems for Video Technology, IEEE Trans. on*, vol. 13, no. 7, pp. 560–576, August 2003.

[2] L. Yang, K. Yu, J. Li, and S. Li, “Prediction-based directional fractional pixel motion estimation for h.264 video coding,” in *IEEE International Conference on Acoustics Speech Signal Processing*, 2005, pp. 901–904.

[3] M. K. Ludwich and A. A. Fröhlich, “Optimizing motion estimation for h.264 encoding,” in *XVII Brazilian Symposium on Multimedia and the Web*, Florianópolis, Brazil, Oct 2011, pp. 198–204.

[4] S. Ning-ning, F. Chao, and X. Xu, “An effective three-step search algorithm for motion estimation,” vol. 1, aug. 2009, pp. 400 –403.

[5] S. Zhu, J. Tian, X. Shen, and K. Belloulata, “A new cross-diamond search algorithm for fast block motion estimation,” nov. 2009, pp. 1581 –1584.

[6] C.-C. Lin, Y.-K. Lin, and T.-S. Chang, “Pmrme: A parallel multi-resolution motion estimation algorithm and architecture for hdtv sized h.264 video coding,” vol. 2, apr. 2007, pp. II–385 –II–388.

[7] H. Chang, S. Kim, S. Lee, and K. Cho, “High-performance architecture of h.264 integer-pixel motion estimation ip for real-time 1080hd video codec,” sep. 2009, pp. 419 –422.

[8] H. Yin, H. Jia, H. Qi, X. Ji, X. Xie, and W. Gao, “A hardware-efficient multi-resolution block matching algorithm and its vlsi architecture for high definition mpeg-like video encoders,” *Circuits and Systems for Video Technology, IEEE Trans. on*, vol. 20, no. 9, pp. 1242 –1254, sep. 2010.

[9] B. Liu and A. Zaccarin, “New fast algorithms for the estimation of block motion vectors,” *Circuits and Systems for Video Technology, IEEE Trans. on*, vol. 3, no. 2, pp. 148 –157, apr. 1993.

[10] C.-C. Lin, Y.-K. Lin, and T.-S. Chang, “A fast algorithm and its architecture for motion estimation in mpeg-4 avc/h.264 video coding,” dec. 2006, pp. 1248 –1251.

[11] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, “Synergistic processing in cell’s multicore architecture,” *IEEE Micro*, vol. 26, no. 2, pp. 10–24, Mar. 2006.

[12] Calypto Design Systems, “CatapultC Synthesis,” 2011. [Online]. Available: http://www.calypto.com/

[13] K. Sühring, “H.264/avc jm reference software,” 2011. [Online]. Available: http://iphome.hhi.de/suehring/tml/

[14] G. Bjøntegaard, “Calculation of average PSNR differences between RD-curves,” Mar. 2001. [Online]. Available: http://wftp3.itu.int/av-arch/video-site/0104_Aus/VCEG-M33.doc