University of Castilla–La Mancha

A publication of the

Computing Systems Department

Optimizing H.264/AVC Inter Prediction on a GPU–based Framework*

by

Rafael Rodríguez–Sánchez, José Luis Martínez, Gerardo Fernández–Escribano, José Luis Sánchez, José Manuel Claver and Pedro Díaz

Technical Report **#DIAB-11-01-2** January 2011

(*) This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04. It was also partly supported by The Council of Science and Technology of Castilla-La Mancha under Grants PEII09-0037-2328, PII2I09-0045-9916, and PCC08-0078-9856.

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS
ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE CASTILLA-LA MANCHA
CAMPUS UNIVERSITARIO s/n
02071, ALBACETE, ESPAÑA
Tlf. +34.967.599200, Fax +34.967.599224

Optimizing H.264/AVC Inter Prediction on a GPU–based Framework

Rafael Rodríguez–Sánchez, Gerardo Fernández–Escribano,
José Luis Sánchez and Pedro Díaz,
Instituto de Investigación en Informática
Campus Universitario s/n
02071 – Albacete, España
{rrsanchez,gerardo,jsanchez,pedrodiaz}@dsi.uclm.es

José Luis Martínez
ArTeCS Group
Universidad Complutense de Madrid
28040 – Madrid, España
joseluis.martinez@fdi.ucm.es

José Manuel Claver
Depto. de Informática, Universidad de Valencia
Avenida de Vicente Andrés Estellés, s/n
46100 – Burjassot, Valencia, España
jclaver@uv.es

January 25, 2011

Optimizing H.264/AVC Inter Prediction on a GPU-based framework

Rafael Rodriguez-Sanchez[1], José Luis Martínez[2], Gerardo Fernández-Escribano[1], José L. Sánchez[1], José M. Claver[3] and Pedro Diaz[1].

[1] Instituto de Investigación en Informática de Albacete. Universidad de Castilla-La Mancha.
Avenida de España s/n, 02071, Albacete, SPAIN.
{rrsanchez, gerardo, jsanchez,pedrodiaz}@dsi.uclm.es
phone: +34 967 599200 ext.: 2408, fax: +34 967 599224
[2] Architecture and Technology of Computing Systems Group
Complutense University, Madrid, SPAIN
joseluis.martinez@fdi.ucm.es
[3]Departamento de Informática. Universidad de Valencia
Avenida de Vicente Andrés Estellés, s/n, 46100 Burjassot, Valencia, SPAIN
jose.claver@uv.es

ABSTRACT

H.264/MPEG-4 part 10 is the latest standard for video compression and promises a significant advance in terms of quality and distortion compared with the commercial standards currently most in use such as MPEG-2 or MPEG-4. In order to achieve this better performance, H.264 adopts a large number of new / improved compression techniques compared with previous standards, albeit at the expense of higher computational complexity. In addition, in recent years new hardware accelerators have emerged, such as Graphics Processing Units (GPUs), which provide a new opportunity to reduce complexity for a large variety of algorithms. Most of the GPU-based algorithms found in the literature are aimed exclusively at reducing execution time, but current GPUs suffer from higher power consumption requirements and it is necessary to obtain energy-efficient GPU codes. In this paper we present a detailed procedure to implement the H.264 motion estimation for a GPU, with the aim of reducing time and energy consumption. The results show a negligible drop in rate distortion with a time reduction of over 91% on average and energy consumption 9.53 times better than the reference implementation.

Keywords: Heterogeneous computing; Hardware accelerators; H.264/AVC; Motion Estimation; Inter Prediction.

1 INTRODUCTION

H.264/AVC is the most recent predictive video compression standard that outperforms other previously existing video codecs [1]. The H.264/AVC standard builds on those previous coding standards

This work was supported by the Spanish MEC and MICINN, as well as European Comission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04. It was also partly supported by The Council of Science and Technology of Castilla-La Mancha under Grants PEII09-0037-2328, PII2I09-0045-9916, and PCC08-0078-9856.

to achieve a compression gain of about 50%, largely at the cost of increased encoder [1]. These compression gains are mainly related to the variable and smaller block-size motion compensation, improved entropy coding, multiple reference frames, a smaller block transform and a deblocking filter, among others.

In addition to this, in the past few years new heterogeneous architectures have been introduced in high-performance computing [2]. Examples of such architectures include Graphics Processing Units (GPUs). GPUs are small accelerator devices with hundreds of disparate processing cores which are designed and organized with the goal of achieving higher performance. Although GPUs can be used for general purposes, they come primarily from multimedia and computer or console gaming.

Furthermore, the most important GPU vendors, NVIDIA and AMD/ATI, have provided several tools to facilitate the programming of these devices. CUDA (Compute Unified Device Architecture) [3], introduced by NVIDIA in 2007, is designed to support joint CPU/GPU execution of applications. CUDA is a parallel computing architecture that enables dramatic increases in computing performance by harnessing the power of the GPU. CUDA abstracts both SIMD and task parallelism into thousands of simultaneous threads.

At this point, this paper presents an implementation of a part of the H.264/AVC encoding algorithm in a GPU as a coprocessor to assist the Central Processing Unit (CPU). In fact, the ME (Motion Estimation) developed as part of the H.264/AVC encoding algorithm is executed in parallel in the GPU. The ME algorithm fits well in the SIMD philosophy because ME performs the same program over a large amount of data and, moreover, this is the most time-consuming H.264/AVC coding process. Our ME algorithm is optimized for CUDA architecture by using a large number of threads that can execute on GPU in parallel and can make an efficient use of the shared memory to reduce DRAM access. Moreover, in this work several optimizations and their effect on overall performance are shown. Hence, the baseline (reference) version is optimized in steps till the most optimized version is achieved, which is able to reach a 58x speedup. This work also demonstrates that the GPU-based H.264/AVC encoding algorithm consumes less power than the baseline algorithm running on a CPU. In fact the GPU platforms raise the power consumption but the execution time is much less, which leads to a more efficient use of energy. The implementation of the present approach has been evaluated on an NVIDIA GTX285 GPU platform.

The rest of the paper is organized as follows: Section II contains a brief overview of H.264/AVC and GPU programming. In Section III some related proposals are presented. Section IV describes the approach presented in this paper. In Section V the proposal presented is evaluated. Finally, conclusions are given in Section VI.

2 BACKGROUND

2.1 H.264/AVC

H.264 [4] or MPEG-4 part 10 Advanced Video Coding (AVC) [5] is a compression video standard developed jointly by the ITU-T Video Coding Experts Group (ITU-T VCEG) and the ISO/IEC Moving Picture Experts Group (MPEG). H.264/AVC promises a significant advance in terms of quality and

distortion compared with the commercial standards currently most in use such as MPEG-2 or MPEG-4 [1]. H.264/AVC can be used thus in a variety of applications such as DVD, video-streaming, HDTV, etc., because it exhibits a substantially greater definition of quality with a considerable reduction in the output bit rate of the encoded sequence.

In order to achieve this better performance, H.264/AVC adopts a large number of new/improved compression techniques compared with previous standards. One of these improved features is H.264/AVC inter prediction, which is the procedure for eliminating the temporal redundancy between two or more adjacent frames. The inter prediction supports motion compensation block sizes ranging from 16x16 to 4x4, with many options available. This procedure is known as the tree structure motion compensation algorithm, which is able to search for the optimal matching block by close prediction (where the block size is variable). In a nutshell, inter prediction in H.264/AVC supports motion compensation block sizes ranging from 16x16, 16x8, 8x16 to 8x8; where each of the sub-divided regions is a macroblock (MB) partition. If the 8x8 mode is chosen, each of the four 8x8 block partitions within the macroblock may be further split in 4 ways: 8x8, 8x4, 4x8 or 4x4, which are known as sub-macroblock partitions. Moreover, H.264/AVC also allows intra modes, and a skipped mode in inter frames for referring to the 16x16 mode where no motion and residual information is encoded. Therefore, H.264/AVC allows not only the use of the MBs in which the images are decomposed but also allows the use of smaller partitions from dividing the MBs in different ways. Figure 1 shows the different block sizes in which an MB can be divided, and Figure 2 shows the MB sub-partitions in which the 8x8 partitions can be further divided.

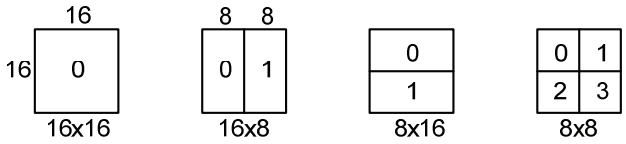


Fig. 1 **MB partitions in H.264/AVC.**

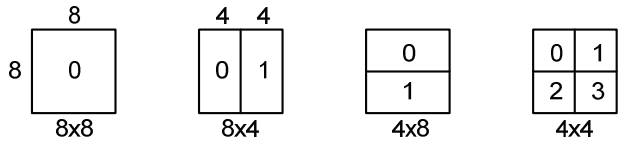
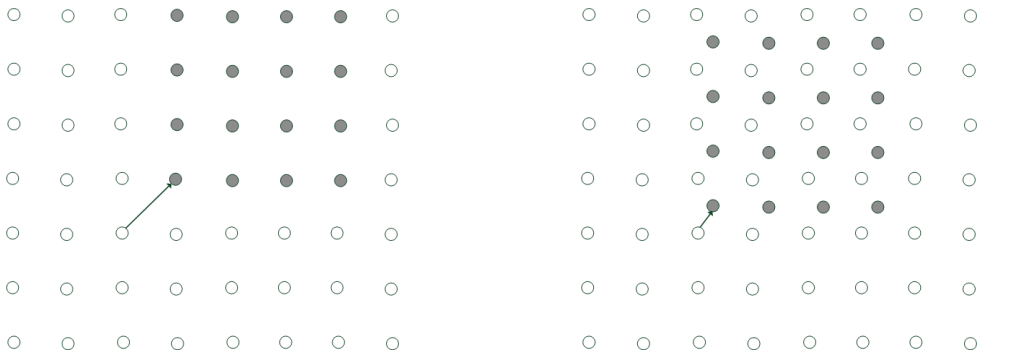


Fig. 2 **8x8 sub-partitions in H.264/AVC.**

For each one of these partitions Motion Estimation is carried out; ME is the process where each MB (or sub-MB) is compared with the previous or following MB in terms of time, looking for a pattern (inside a search area) that indicates how the movement is produced in the sequence.

In order to further improve compression, the H.264 /AVC standard assumes that the best match can be found at a region offset from the current MB by an integer number of pixels. In fact, for many MBs a better match can be obtained by searching a region interpolated to sub-pixel accuracy; for this case, a new

prediction pixel is created by means of an interpolation of its neighbor. Figure 3 shows an example of integer and sub-sample predictions.



Integer prediction: vector (1 , -1) Sub-sample prediction: vector (0.25 , -0.5)

Fig. 3 Example of integer and sub-sample predictions.

2.2 Graphics Processing Units (GPUs)

In the past few years new heterogeneous architectures have been introduced in high-performance computing [2]. Examples of such architectures include Graphics Processing Units (GPUs). GPUs are accelerator devices with hundreds of disparate processing cores which are designed and organized with the goal of achieving higher performance. Although GPUs can be used for general purposes, they come primarily from interactive applications such as multimedia, and computer or console gaming. However, GPUs have recently moved from being exclusively used in graphics applications to being used in what is now called General Purpose Computing on GPU (GPGPU) [6], or in other words, the application of GPUs for applications for which they were not originally designed.

GPU architecture offers a new challenge for engineering since the programming model must be adapted to the available hardware in order to obtain good performance and exploit the full potential of the GPU. This problem has been solved by GPU manufacturers, such as NVIDIA or ATI, by proposing new languages or even extensions for the most commonly-used high-level programming languages.

In this respect, NVIDIA proposes the CUDA parallel computing architecture, which is a software platform for programming massively parallel high-performance computing applications on their company powerful GPUs. NVIDIA has seized upon this opportunity to create a better programming model and to improve the shaders or stream processors. Normally, each stream processor in an NVIDIA GPU can manage many concurrent threads, and these processors have their own FPUs (Float Point Units), registers, and shared local memory [3].

CUDA includes C/C++ software development tools, function libraries, and a hardware abstraction mechanism that hides the GPU hardware from developers, such as an Application Programming Interface (API) which greatly simplifies the programming model. CUDA development tools work alongside a conventional C/C++ compiler, so programmers can mix GPU code with general purpose code for the host CPU. Therefore, CUDA takes the GPU model, which is highly parallel; it divides the data set into smaller

chunks and stores them in on-chip memory, and then it allows multiple thread processors to share each chunk. At this point, the programmers do not write explicitly threaded code, a hardware thread manager handles threading automatically, which is an important feature of CUDA. Figure 3 depicts the hardware model.

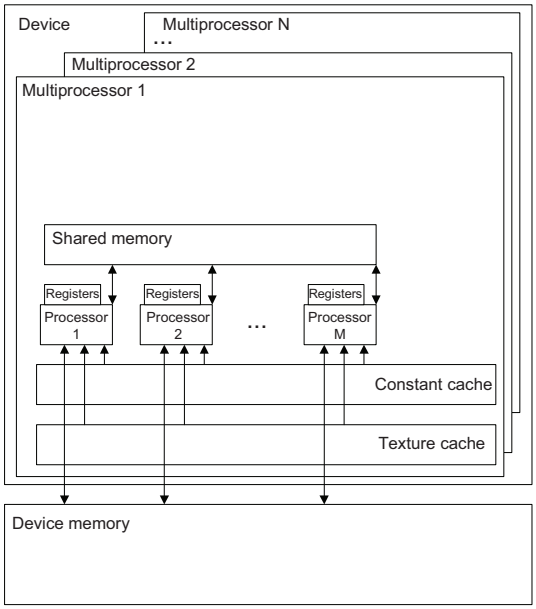


Fig. 3 GPU hardware model.

Thus, the inter prediction algorithm proposed in the H.264/AVC encoding algorithm fits well in the GPU philosophy, and offers a new challenge for the GPUs. The main issue is how to efficiently distribute all the computations over the GPU.

3 RELATED WORK

This section tries to gather all the state of the art available in the literature in the framework of GPU optimization as well as power consumption in GPU-based platforms. Therefore, this related work is divided into two halves: the first one covers some papers which focus on criteria to optimize GPU algorithms, and the second one is focused on power consumption, which is a new statistic introduced to determine how much better a parallel version of an algorithm is, compared with its sequential one.

It is well known that migration from sequential to parallel execution is not trivial, and some problems have already been described, such as bank conflicts or composition of thread blocks. In 2008, Ryoo et al. presented an overview of optimization principles and a performance evaluation of many applications by means of GPU and CUDA. This paper collects some ideas that can be used to adapt traditional serial applications to the new multithreaded GPU environment. One of the selected applications was the H.264 video encoder which achieved a speedup of up to 20.2x [7]; the present work will increase this speedup to 58x by means of a better optimization of the resources available in the GPUs for the H.264 video encoding algorithm. In the same year, Boyer et al. in [8] presented an analysis of programs developed in CUDA; their major conclusion involves both the race conditions and the shared

memory bank conflicts. In this way, the authors propose a technique that avoids both of these problems. Continuing in 2008, some months later, Hartley et al. in [9] tried to parallel biomedical images into a cluster of GPUs and Multicores, obtaining up to 10 TFLOPS. Moreover, in this work, the authors show the performance of the intermediate solutions before achieving the final version, which is a good way to determine how the different GPU conflicts affect the final performance.

More recently, research efforts have not focused only on performance (speedup) and migration from sequential to parallel execution. Studies have appeared that deal with power consumption, which is a new requirement demanded by processor manufacturers. In 2009 Huang et al.[10] presented a trivial algorithm implemented in a GPU but focusing on a full analysis of energy consumption. This study shows that the system including a GPU delivers an energy-delay product that is multiple orders of magnitude better than a traditional one (without a GPU). These results were measured in terms of performance, power and energy characteristics. Also in 2009, Ma et al. in [11] presented another statistical power consumption analysis of a GPU-based framework. In fact, this paper presents a statistical model which is able to determine the power consumption estimation. To the best of our knowledge, these studies are the first that try to measure power consumption in the framework of GPUs. However, as far as we know, there is only one paper which tries to implement the H.264 encoding algorithm in a GPU [12]. In fact, Chen and Hang proposed an implementation of the H.264/AVC motion estimation algorithm using CUDA. The algorithm is based on an efficient block-level parallel algorithm for the variable block size motion estimation in H.264/AVC. However, they do not show any results concerning RD performance and how to deal with the problem of predictors in H.264/AVC. At this point, this paper tries to measure different approximations of the previously proposed algorithm [13] as well as including a power consumption analysis of the present approach.

4 INTER PREDICTION GPU- BASED IMPLEMENTATION

The proposed implementation follows a workflow which is described below. First of all, the reference software source code was analyzed in order to decide which parts are most suitable to be implemented in a GPU. Following the philosophy of GPUs, which is based on Single Instruction Multiple Data (SIMD), not all parts of the encoder are appropriate to be implemented in a GPU. After that, a complete base implementation is obtained and then some optimizations are performed in order to increase the performance by taking advantage of the GPU potential. In this section a general description of the base algorithm implementation is given, followed by a detailed description of each optimization.

4.1 Base implementation

After analyzing the H.264 JM 15.1 reference software [14] and evaluating the time consumption of each module, the results show that the most expensive task in the H.264 reference encoder is motion estimation, which consumes 95.2% of the total time for the encoding process. This value was obtained by means of 18 VGA (640x480 pixels) sequences encoded with different QPs. Moreover, as explained in section 2.1, motion estimation is divided into two main tasks: integer motion estimation and sub-pixel

motion estimation. From our analysis we also conclude that integer motion estimation consumes 89.28% of the total time, so we have focused our efforts on developing an implementation to reduce the time consumption of this part.

The reference integer motion estimation sequentially obtains the Sum of Absolute Differences (SAD) cost for all positions checked inside the search area for all possible partitions/subpartitions defined by the standard for each MB in a frame. Our main idea is to generate all the motion information at the beginning of each frame, dividing integer motion estimation into three tasks: the first one obtains the SAD costs exclusively for the 4x4 subpartitions, the second one obtains the motion information for the other higher partitions/subpartitions by using the data generated by the first task, and the third one performs a reduction to obtain the best match, that is, the motion vector with the lowest cost.

In order to work with a GPU as a coprocessor, the required data must be explicitly transferred into its DRAM memory. Thus, at the beginning of the encoding process some data are transferred from Main memory to GPU DRAM memory. In this base implementation the data that do not change during the encoding process (frame sizes, search area dimensions, search area distribution) are moved into the GPU constant memory at the beginning of the encoding process. On the other hand, at the beginning of coding each frame, the frame itself and the reference frame are transferred to the GPU global memory.

The goal of the first kernel is to obtain the required 4x4 SAD costs, which are needed to build the structured motion tree in the next step. Figure 4 shows the search area for a given MB which contains (2* Search range)^2 positions. The search area positions distribution follows a spiral pattern (defined by the reference software), position 0 corresponding to the center of the search area, motion vector (0,0). A GPU thread is generated for each position in the search area for each MB in a frame. 256 GPU threads are grouped into a GPU thread block, obtaining the SAD costs for correlative positions inside the search area; one thread block calculates the SAD costs for positions 0-255, another for positions 256-511 and so on. GPU threads use 16 registers and no shared memory, obtaining a 100% GPU occupancy factor.

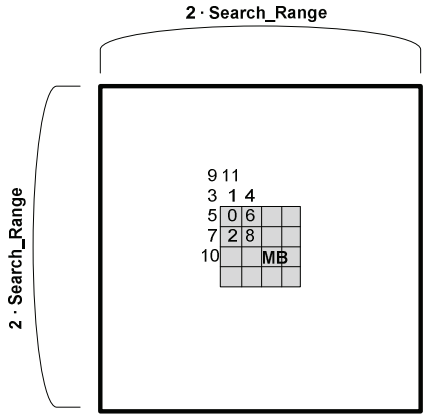


Fig. 4 Base search area.

The SAD calculation is carried out in 4x4 blocks, and therefore each MB is divided into sixteen 4x4 blocks for each search area position, and each GPU thread calculates the 16 4x4 SAD costs of its associated position. These sixteen SAD costs are the basic data to build the structured motion tree in the next step, and they are stored in the GPU texture memory.

The main purpose of the second kernel is to build the structured motion tree, obtaining in this way the SAD costs for all partitions/subpartitions. This kernel also makes a first reduction of the generated data. As input, this kernel takes the motion information of 16 4x4 blocks of a MB for 64 positions. It generates the motion information for all partition combinations, and reduces the amount of information, obtaining the best motion vector for each partition/subpartition of the 64 positions. For this purpose, 64 GPU threads are grouped into a thread block, each of them building the SAD costs for a concrete position for all partitions/subpartitions. Intermediate results are stored in multiprocessor shared memory.

Figure 5 shows how to build the motion information for all partitions/subpartitions from the 4x4 SAD costs generated in the first kernel. In order to obtain the motion information for the 4x8 and 8x4 subpartitions it is only necessary to add 2 4x4 SAD costs, for the 8x8 partitions it is only necessary to add 2 4x8 SAD costs, for the 8x16 and 16x8 partitions it is only necessary to add 2 8x8 SAD costs, and finally to obtain the motion information for the 16x16 partition it is only necessary to add 2 8x16 SAD costs. Intermediate results for all partitions/subpartitions are stored in the multiprocessor shared memory, using a structure composed of an unsigned short with the SAD cost and an unsigned short indicating its associated position (4 bytes).

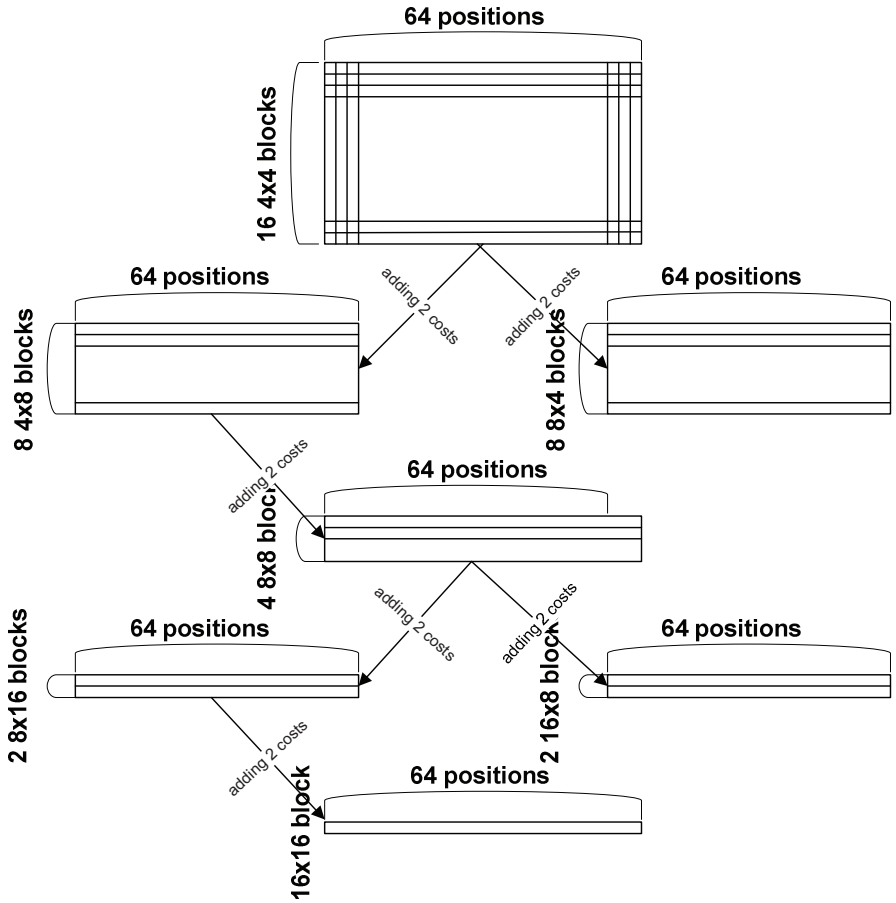


Fig. 5 How to build the SAD partitions/subpartitions.

Once the SAD costs for every partition/subpartition are generated, this information is used to compute the final cost. This calculation adds a penalization for large motion vectors because a 25-pixel-

long vector with cost 50 is not the same as a vector with the same cost but with a length of 5 pixels. The penalization is computed using the following equation:

$$newSAD\_cost = SAD\_cost + K*vector\_bits, \quad (1)$$

where *newSAD_cost* is the penalized SAD cost, *SAD_cost* is the old SAD cost without penalization, *vector_bits* is the motion vector length associated to each position, and *K* is a constant which depends on the QP parameter used to encode the sequence. QP is defined by the JM reference software. Note that the summing of costs for new partitions/subpartitions has to be performed before adding the penalty.

As a starting point a linear reduction was defined to reduce the large dataset generated. Figure 6 shows a generic reduction for the 4x4 sub-partitions in which 16 threads participate (an MB is decomposed into 16 4x4 blocks), each one obtaining the lowest SAD cost of a row.

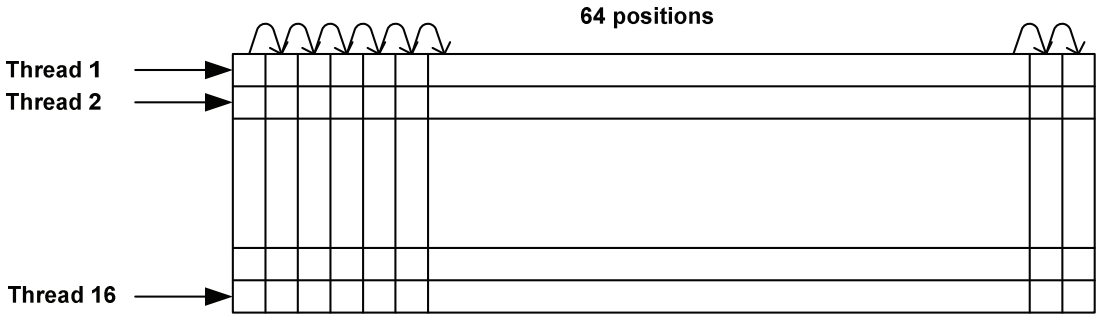


Fig. 6 Linear SAD reduction scheme for 4x4 sub-partitions using 16 threads.

The last kernel follows the same procedure as the one explained for the reduction carried out in kernel 2 but using different data, so no more explanation is needed. This kernel obtains the best SAD cost for each one of the MB partitions/subpartitions of each MB in a frame.

4.2 Optimizations

In this section the most significant optimizations to obtain the final implementations are presented. We comment on these optimizations, grouping them in accordance with the kernel where they have been applied.

4.2.1 Kernel 1 optimizations

This section provides a detailed description of each of the optimizations applied for the first kernel and the justification for their improvements.

Texture memory

One of the main differences between GPU global memory and GPU texture memory is that the second kind of memory is cached, so when multiple accesses to the same memory region occur, memory latency is lower.

In the integer motion estimation algorithm each pixel of the search area is accessed many times because each pixel is used for calculating more than one 4x4 block SAD cost. Therefore, using the GPU texture memory instead of using the GPU global memory is a good way to increase performance.

The caches are small and in order to achieve a good performance they need locality in data accesses, but this does not happen in the base implementation. Due to the spiral search area distribution, consecutive positions do not correspond to consecutive physical positions as shown in figure 4. So we have changed the search area positions distribution in order to achieve higher locality in data accesses. Figure 7 shows the new search area distribution where position 0 corresponds to the top-left corner of the search area and the positions are distributed by rows. As in the base implementation, one thread per position inside the search area is generated and 256 threads are grouped into a GPU thread block, computing N complete search area rows in the same thread block, where N depends on the dimensions of the search area. For example, to code a sequence which uses 32 pixels as search range, N must be equal to 4 (2*Search_Range*4=256). The GPU occupancy factor does not vary with respect to the base implementation.

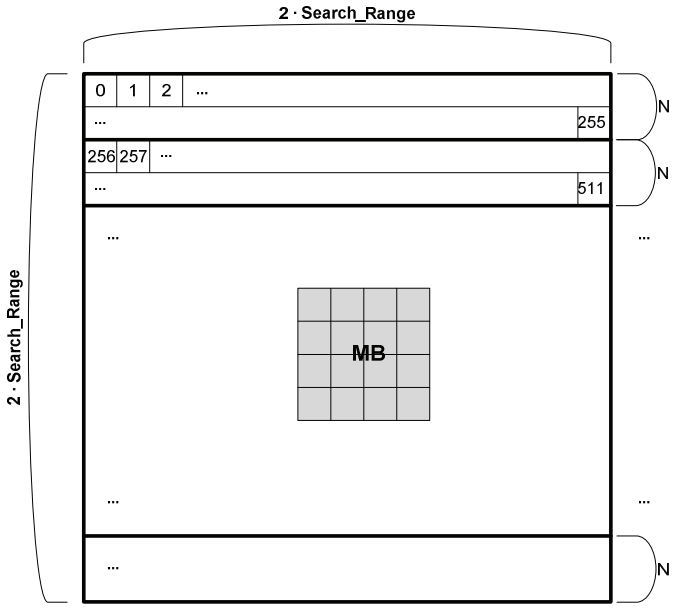


Fig. 7 Custom search area.

Therefore, with this new search area distribution, each thread block defined accesses contiguous data after the redistribution of the search area (increasing locality in data access).

Shared memory

The GPU used in our work has also a 16 KB shared local memory per multiprocessor, which is faster than texture memory. It is known that an access to shared memory without bank conflicts is as fast as a register access. Therefore, we also decided to use this kind of memory.

As a consequence of the new distribution, the required portion of the search area for a concrete thread block is contiguous, so at the beginning of this kernel all the threads can cooperate to copy in shared memory the MB itself and the portion of the search area assigned to the positions. Note that thread blocks only copy the data which they need; they do not copy the full search area.

With this implementation the GPU occupancy factor dropped to 50% because the register usage increased to 21. New thread block sizes such as 192 or 384 can improve this index to 75%, but we decided to maintain 256 threads as the thread block size since from experience it gives us the best performance, and allows us to apply the next optimization.

Using shared memory in this implementation produces bank conflicts because data from MB and the portion of the search area are defined as unsigned short (2 bytes), in accordance with the reference software, and the shared memory banks are organized in words of 4 bytes. When the multiprocessors access two consecutive positions both threads access the same bank at the same time, and the accesses will be serialized.

This kernel uses 21 registers and 3600 bytes of shared memory, giving an occupation of 50%, so GPU occupancy is limited by register usage. We consider changing the memory data type, up-casting from unsigned short to integer, and maintaining 50% of GPU occupancy in order to avoid memory bank conflicts.

4.2.2 Kernel 2 optimizations

This section gives a detailed description of each of the optimizations implemented for the second kernel and the justification for their better performance.

Grouping modes

In the second kernel each thread requires 32 registers to run, which is a constraint. Also, building the SAD costs requires a lot of multiprocessor shared memory, which is the real limitation. Considering that 64 positions require 16*64*4 bytes of multiprocessor shared memory, this results in a 19% GPU occupancy.

As explained above, shared memory consumption is the main limitation in this second kernel, so there are two main strategies: not to use multiprocessor shared memory or to reduce the amount used. If we do not use multiprocessor shared memory, we must use GPU global memory, which is clearly slower. We reject this option, so we opt for reducing the required multiprocessor shared memory.

The base implementation obtains the SAD costs for the 4x4 subpartition and reduces the information using 100% of the allocated multiprocessor shared memory, then builds the SAD costs for the 4x8 subpartition and reduces the information using 50% of the allocated multiprocessor shared memory. To calculate SAD costs for the 8x4 subpartition the same percentage of multiprocessor shared memory is used as for the 4x8 subpartition. The percentages of allocated multiprocessor shared memory usage for the other partitions/subpartitions are shown in figure 8. By analyzing multiprocessor shared memory usage, it is clear that the percentage of use during this second kernel is quite low, so we propose to use half the multiprocessor shared memory in addition to grouping the different modes in the same iteration.

Figure 8 shows the base implementation in the left column, and in the right column the new optimization, in which the allocation of multiprocessor shared memory has been reduced to half and the modes have been grouped to increase multiprocessor shared memory usage. These changes allow GPU occupancy to increase to 38%.

In this last implementation, multiprocessor shared memory usage is 100% in all iterations except in the last one, in which it is 12.5%. But in this case it is possible to allocate an extra row in the data structure (9*64 elements) to include the last iteration in the previous one, thus maintaining GPU usage. So, the new implementation is performed in just 5 iterations.

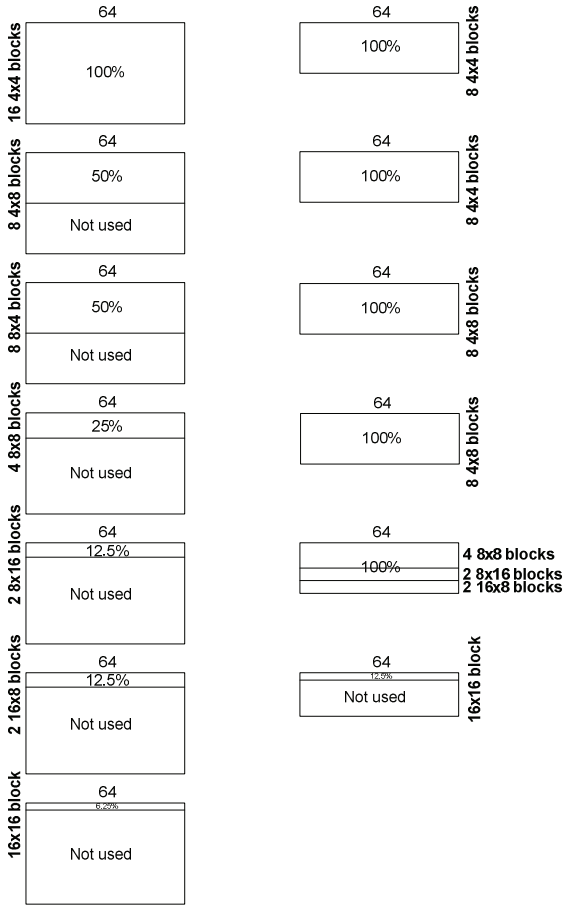


Fig. 8 Shared memory usage.

Motion vector optimization

In previous implementations, the penalization for large motion vectors is calculated when the reduction is performed and it is not stored in multiprocessor shared memory. To build the new partition/subpartition SAD costs the costs without penalization are needed, so the same calculation must be done many times because the penalty for a vector associated to a position P is the same regardless of which partition/subpartition or which blocks inside a partition/subpartition it belongs to. Figure 9 shows the pseudo code for this optimization.

Experimental results show that it is better to calculate the penalization in advance for large motion vectors, update the shared memory with the penalized SAD costs, and reduce the amount of information, although this means more GPU texture memory accesses to calculate the new partition/sub-partition SAD costs.

```
Calculate 4x4 SAD costs from texture memory + penalization
Reduce
Calculate 4x8 SAD costs from texture memory + penalization
Reduce
Calculate 8x4 SAD costs from texture memory + penalization
Reduce
Recalculate 4x8 SAD costs from texture memory
Calculate 8x8 SAD costs using the previous 4x8 SAD costs from shared memory
Calculate 8x16 SAD costs using the previous 8x8 SAD costs from shared memory
Calculate 16x8 SAD costs using the previous 8x8 SAD costs from shared memory
Calculate 16x16 SAD costs using the previous 8x16 SAD costs from shared memory
Add penalization to the last four partitions
Reduce
```

Fig. 9 Pseudo code.

Binary reductions

The linear reduction is very inefficient; typically there are 8 threads, one per row in the structure defined in multiprocessor shared memory, where each thread performs 64 comparisons. However, we have 64 threads in any thread block, so we can use more threads for the reduction process.

Figure 10 shows a generic binary reduction in which each thread involved in the reduction procedure (t_0 to t_{m-1}) performs a reduction for each of the 8 rows in multiprocessor shared memory (b_0 to b_{N-1}). Note that *m* is equal to half of the remaining positions in any of the six iterations needed to reduce from 64 positions to 1. In order to complete the reduction process, six iterations are needed, starting with 64 (2^6) SAD costs per subpartition and finishing with one SAD cost per subpartition, where the SAD-matrix size is reduced by half in each iteration. The reductions are performed with SAD-matrix sizes of 64, 32, 16, 8, 4, and 2 using T strides, such that T=S/2, to obtain the best SAD cost per block. These strides are chosen to avoid local shared memory bank conflicts. The code for the six iterations is unrolled to avoid unnecessary loop climbs. Intermediate results are allocated to multiprocessor shared memory.

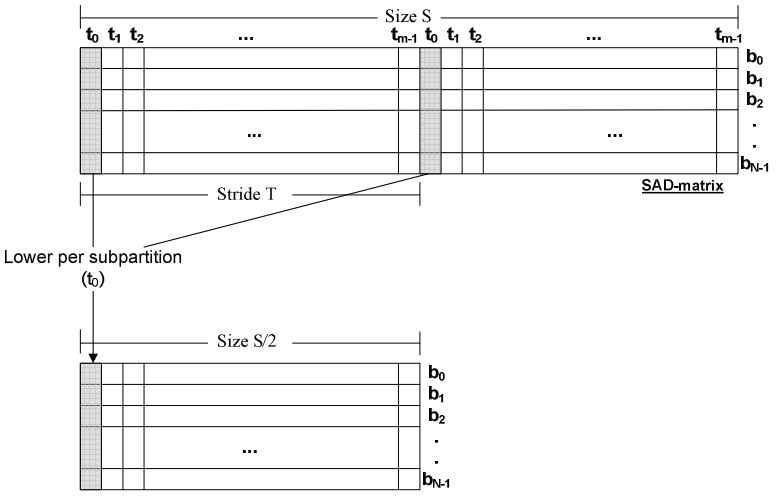


Fig. 10 Binary reduction scheme.

Thread comparisons

Binary reduction is better than linear reduction, but thread usage can be improved. In the first iteration of the reduction 32 threads perform 8 comparisons each (256 in total), in the second one 16 threads carry out 8 comparisons (128 in total), and so on. Finally, one thread performs 8 comparisons. Thus, the number of threads used in the reduction process decreases by half on each iteration and the number of comparisons per thread is constant (8). In order to improve the use of the 64 available threads, a new comparison distribution must be applied.

For the new distribution we developed, in the first iteration of the reduction 64 threads perform 4 comparisons each, in the second one 64 threads perform 2 comparisons, in the third one 64 threads perform 1 comparison, in the fourth one 32 threads perform 1 comparison, in the fifth one 16 threads perform 1 comparison, and finally 8 threads perform 1 comparison. Hence, in this new proposal the number of used threads is constant and equal to the maximum (64), until the number of comparisons is lower than the number of threads. Note that this distribution is implemented avoiding multiprocessor shared memory bank conflicts in order to achieve better performance.

5 PERFORMANCE EVALUATION

In this section we evaluate the behaviour of the different optimizations applied to the base parallel implementation of the H.264 encoder. Firstly, we chose the parameters used in the H.264 encoder configuration and then ran the parallel codes considering several video sequences. We describe the hardware platform used and the way the time and energy consumption data was obtained. Finally, results and comments about them are presented.

5.1 System

In order to show the performance of the optimizations proposed in this paper, all of them were implemented in the H.264 JM 15.1 reference software encoder [14]. The parameters used in the H.264 encoder configuration file were those included in the baseline profile of the mentioned reference software. Only six parameters were changed in the configuration file:

• The number of reference frames was set to 1 in order to keep the complexity as low as possible because higher values imply excessive time consumption.

• RD-Optimization was disabled for the same reason as the NumberReferenceFrames parameter.

• The GOP pattern was fixed to I(11)P.

• The tests were carried out with popular sequences in VGA format (640 x 480), and therefore the SourceWidth and SourceHeight parameters were changed accordingly.

• The parameter FramesToBeEncoded was adjusted according to the sequence, in order to encode the full sequence.

• The Quantization Parameter (QP) called QPISlice and QPPSlice was varied among 28, 32, 36 and 40 according to [16].

The host machine used was an Intel® Core™ 2 Quad CPU Q9300 running at 2.50 GHz with 4GB of DDR2 memory. It included a GPU NVIDIA GTX285, with NVIDIA driver and CUDA support (190.18). The operating system was Linux Fedora 11 x64 with GCC 4.4.

5.2 Metrics

We show results for execution time, quality for the resulting encoded video sequence and power consumption for the whole system. For applications such as the H.264/AVC encoder it is very important to reduce their response time. Therefore, most of the optimizations are aimed at reducing the execution time. However, as a consequence of some of the implementations developed to increase the H.264/AVC encoder performance, such as the redistribution of the search area and data reductions, the resulting H.264/AVC bit stream was changed. The bit stream changes affect the quality and size of the output video sequence, so it is necessary to study these metrics. Note that the changes in the bit streams do not modify the format of the encoded video sequence and the output video sequence can be decoded normally by any H.264/AVC decoder.

On the other hand, current GPUs are composed of a large number of transistors, and they suffer from higher power consumption requirements. Therefore, it is necessary to develop energy-efficient GPU codes. As a consequence power consumption becomes an essential metric in this kind of studies.

5.2.1 Time and Quality

The main metrics of interest are the RD function (bitrate) vs. Quality (PSNR) and the results in terms of ΔTime. These metrics are defined below:

RD function. RD provides the theoretical bounds on the compression rates that can be achieved using different methods. In rate distortion theory, the rate is usually understood as the number of bits per data sample to be stored or transmitted. The notion of distortion is subject to on-going discussion. In the simplest case, and the one actually used in most cases, the distortion can be simply defined as the mean squared error of the difference between the input and the output signals. In the definition of the RD function, the PSNR is the distortion for a given bit rate. The averaged global PSNR is based on Equation 2.

$$\overline{PSNR} = \frac{4 * PSNR_Y + PSNR_U + PSNR_V}{6} \quad (2)$$

The Luminance $PSNR_Y$ is multiplied by four, since the YUV input files are in the format 4:2:0, which is composed of four 8x8 blocks for the luminance component and only two 8x8 blocks for the chrominance components.

ΔTime, ΔPSNR, and ΔBitrate. In order to evaluate the time saved by the fast MB mode decision algorithm, ΔTime, let T_{JM} denote the coding time used by the H.264/AVC JM 15.1 reference software encoder for the motion estimation and compensation process, and T_{FI} the time taken by the proposed algorithm or the mechanism that has been evaluated; ΔTime can then be defined as follows:

$$\Delta Time(\%) = \frac{T_{F1} - T_{JM}}{T_{JM}} * 100 \quad (3)$$

where T_{FI} also includes all the computational cost needed for preparing the residual information required by our proposal.

Note that the bit rate and PSNR differences should be regarded as equivalent, i.e. there is either a decrease in PSNR or an increase in the bit rate, but not both at the same time. The detailed procedures for calculating these differences can be found in the work of Bjøntegaard [15][16], and these have been recommended by the JVT Test Model Ad Hoc Group [17]. The experiments were carried out on the test sequences using four quantization parameters (QP) (i.e., QP = 28, 32, 36 and 40), as specified in Bjøntegaard and Sullivan's common test rule [16]. The YUV files used for comparing the PSNR results are the original YUV file at the input of the H.264/AVC encoder and the one obtained after decoding the H.264/AVC video using an H.264/AVC decoder.

5.2 Energy consumption

With the objective of sampling the current consumed by the whole system (test computer) including the Power Supply Unit(PSU) at a given time. We developed a device capable of transmitting this data in a reliable and easy way to a computer. The principle of this device is based on the analysis of the magnetic field produced by an electric current flowing through a straight conductor and it is capable of sampling and reconstructing the resulting wave, whatever its form, and processing it in order to obtain an average value.

We use a sensor capable of translating these magnetic changes into a proportional voltage level to simplify the procedure. The sensor used is the Allegro Microsystems Inc A1301. The sensor response is given by equation 4, where v is the voltage at the output of the sensor, $y_0 = 2.4610$, $\alpha = 0.4185$, and I is the current flowing through the conductor we want to measure. The constants y_0 and α are obtained by a linear regression:

$$v = y_0 + a * I \quad (4)$$

The sensor output is tied to the analog port of a microcontroller, which is responsible for sampling the voltage data and sending it to the user. The microcontroller used is the Microchip PIC12F683, which operates at 8 MHz. and has an RS232 interface which adjusted its data transfer rate to 115200 bauds.

In order to transfer the sampling data to the host computer we used the Future Technology Devices International Ltd FT232 chip, which makes possible to adapt the RS232 interface to the USB interface. The data is received through a virtual COM port which is created when the installation of the sensor device in the host computer is completed. The software used to collect the data is the *Eltima software* RS232 data logger. Figure 11 shows the basic scheme of our profiler system. Note that two computers were used: The first one (Tested Computer) is the computer on which the H.264 encoder is running and

whose energy usage needs to be measured; the second one (Data Computer) is used in order to collect and process the energy data from the sensor device.

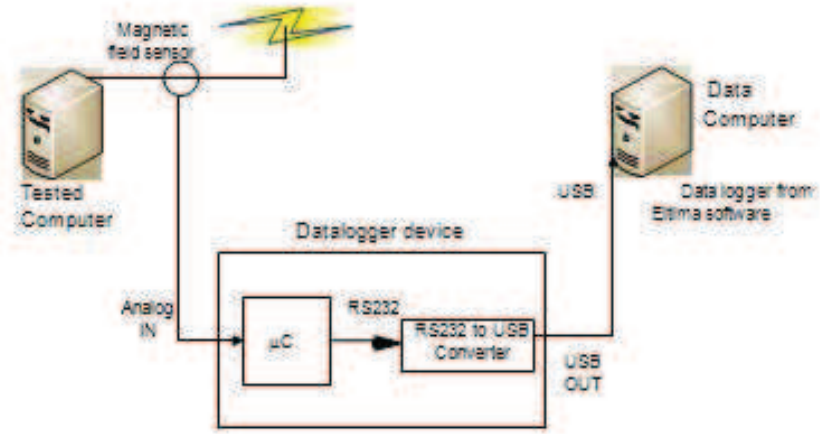


Fig. 11 Profiler system scheme.

Once the voltage data is obtained, it must be processed to obtain the Root Mean Square (RMS) or quadratic mean for a given cycle. When working with periodic and symmetrical waves, such as an electric wave, the RMS value must be calculated in cycles. Our sensor device takes 2K samples per second, that is, 40 samples per cycle. Thus, we calculate the V_{RMS} in a cycle with equation 5.

$$x_{RMS} = \sqrt{\frac{1}{N} * \sum_{i=1}^{N} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + \ldots + x_N^2}{N}} \tag{5}$$

Operating with equation 4 and using the RMS value of the voltage previously obtained, the intensity flowing through the conductor at any cycle can be obtained as:

$$I = \frac{x_{RMS} - 2.4610}{0.4185} \tag{5}$$

Once the data for the intensity is known, we can apply equation 6 to obtain the power consumed by the GPU at any cycle, where V is equal to 230v. Finally, we can obtain the energy consumption applying equation 7, were T is the time consumed by the encoder and P is the average power consumed.

$$P = V \times I \tag{6}$$

$$\mathrm{E = P \times T} \tag{7}$$

5.3 Results

Table 1 shows all the optimizations described in previous sections and their performance in terms of time. The time shown is the average time spent by each kernel after coding all sequence frames of the Tempete sequence in VGA (640x480) format, where the integer motion estimation was performed with the GPU. The results show a remarkable time reduction on each implementation. Each implementation is considered as a starting point to implement the next optimization.

Table 1 Optimization time consumption

kernel	Description	Time	Reduction from the previous optimization
#1	Base implementation	83 ms	
	MB and reference area in GPU texture memory	78 ms	5 ms
	MB and reference area in GPU shared memory	26 ms	52 ms
	Avoiding shared memory bank conflicts	16 ms	10 ms
#2	Base implementation	760 ms	
	Using half of multiprocessor shared memory in addition to grouping the different modes in the same iteration	354 ms	396 ms
	Calculating vector penalizations in advance	58 ms	296 ms
	Binary reduction without bank conflict	50 ms	8 ms
	Better thread usage for binary reduction without bank conflict	47 ms	3 ms

We should also mention that the reference H.264 integer motion estimation algorithm running on the CPU needs 4698 ms per frame on average. This data was obtained in the encoder analysis explained at the beginning of section 4. With our implementation, considering the time consumed by the three GPU kernels and the time consumed by the memory transfers, 81 ms are needed on average per frame, which means a speedup of 58x on average for integer motion estimation.

Table 2 shows the results in terms of ΔTime, ΔPSNR and ΔBitrate for the final optimized version presented in this paper. We also provide the average results for all the video sequences. This should provide a good analysis of the performance of the proposed encoder for all different kinds of video content. Compared with the reference encoder, the proposed approaches have a PSNR drop of, at most, 0.105 dB for a given bitrate, and a bitrate increase of, at most, 2.99 % for a given PSNR in VGA format. This drop in RD performance is negligible if the computational savings are taken into account, which are up to 91% on average for all the tested video sequences (the encoding time is a key feature in the design of real-time H.264/AVC encoders).

Table 2 Time Reduction, ΔPSNR and ΔBitrate of the proposed GPU-based algorithm

Sequence, Format, Number of frames and QPs				ΔBitrate, ΔPSNR and Time Reduction (mean) from H.264/AVC 15.1 reference encoder		
				Proposed (H.264/AVC)		
				Time (%)	ΔPSNR (dB)	ΔBitrate (%)
Canoe	VGA	220	(28,32,36,40)	-92.23	-0.151	4.42
Funfair	VGA	260	(28,32,36,40)	-90.97	-0.116	3.45
Mobile Calendar	VGA	260	(28,32,36,40)	-91.90	-0.148	3.83
Parade	VGA	220	(28,32,36,40)	-91.85	-0.107	2.94
Sgi-ant	VGA	220	(28,32,36,40)	-89.49	-0.053	1.41
Ship	VGA	260	(28,32,36,40)	-87.95	-0.012	0.34
Soft Football	VGA	220	(28,32,36,40)	-93.09	-0.115	3.59
Tempete	VGA	260	(28,32,36,40)	-91.78	-0.138	3.96
Average	VGA	1920	(28,32,36,40)	-91.16	-0.105	2.99

Only the Tempete sequence was chosen for testing the power consumption of the implementations because we discovered that the time consumed for our implementations is independent of the sequence and so it is not necessary to show results for more sequences.

Figure 12 shows the power consumption obtained for coding one GOP (12 frames) using the reference H.264/AVC encoder, the base GPU implementation and the final GPU implementation. Figure 13 is a zoom of figure 12 focused on the two GPU implementations, which corresponds to a range from 0 to 14.5 seconds. Compared with the reference encoder, both GPU implementations consume more energy but for a shorter period of time. The reference encoder consumes 122.55 Watts in 92.62 seconds, but for both GPU implementations the time consumed is shorter, 12.46 and 7.7 seconds for the base and final implementation respectively. 11 consumption peaks can be seen for the GPU implementations (1 GOP is composed of 1 intra frame and 11 P frames where the GPU code is executed). Each GPU peak can be further divided into two components; the first one corresponds to the CPU-GPU memory transfer consuming around 250 Watts, and the execution of the GPU kernels consuming around 200 Watts. The difference between both GPU kernels, as seen in previous results, resides in the kernels execution time. Note that the power consumption for CPU code in the GPU implementations is around 150 Watt, which is higher than for the reference execution because the GPU is always active, waiting for new kernels.

Finally, table 3 shows the average power, the time consumed and the energy consumed in coding one GOP (12 frames) for the reference H.264/AVC encoder, for the base implementation and for the final implementation. The energy consumption for the base implementation is 5.20 times better than the reference implementation and the energy consumption for the final implementation is 9.53 times better.

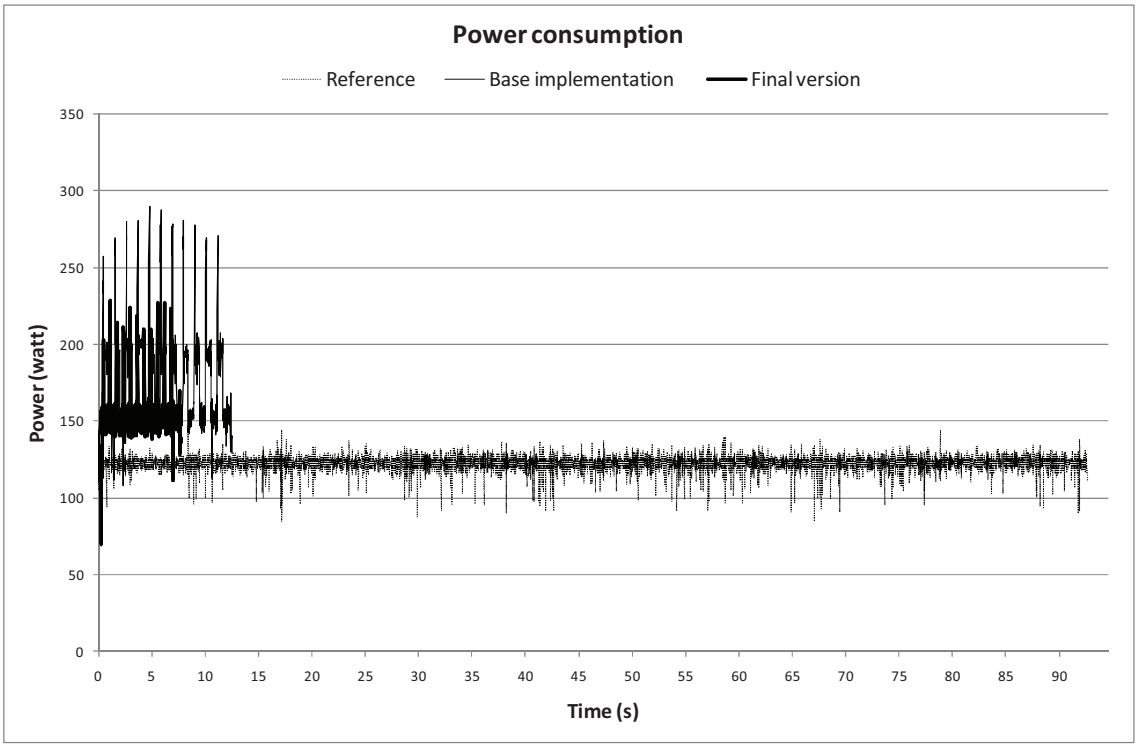


Fig. 12 Algorithm power consumption.

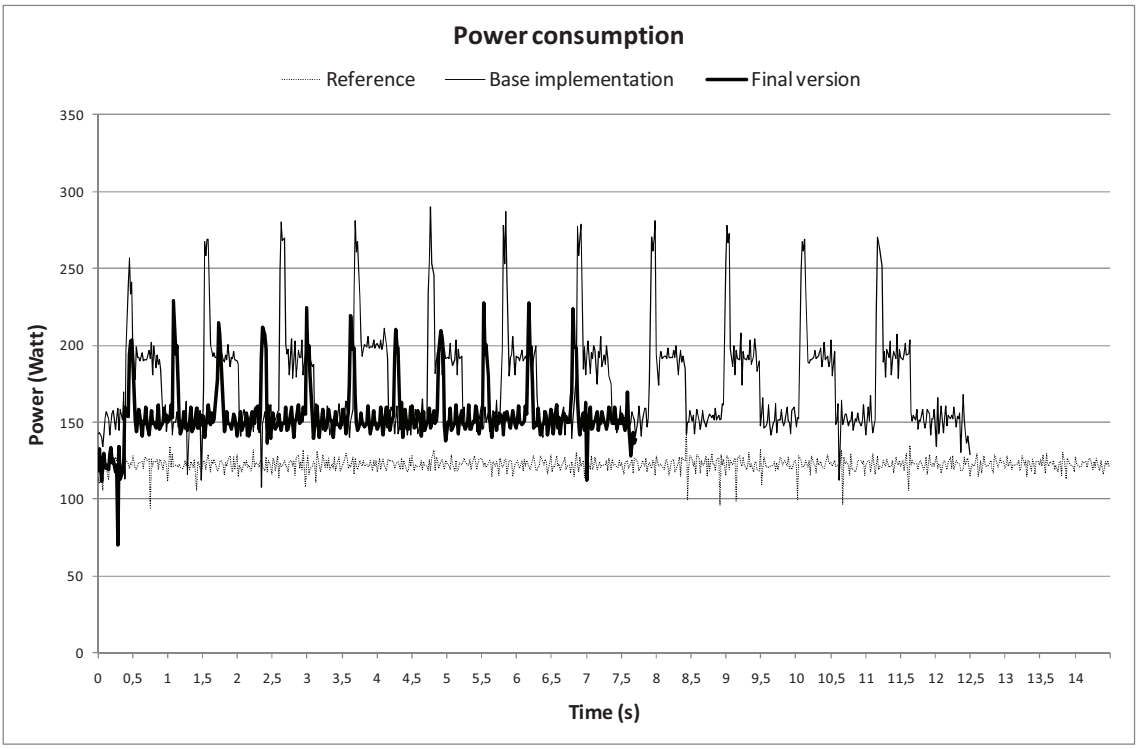


Fig. 13 Algorithm power consumption.

Table 3 Energy consumption

Implementation	Power (Watt)	Time(seconds)	Energy (Jules)	Energy*Time (Jules*sec)
Reference	122.55	92.62	11,350.58	1,051,290.72
Base implementation	174.94	12.46	2,179.75	27,159.69
Final implementation	154.65	7.7	1,190.80	9,169.16

6 CONCLUSIONS

This paper presents a detailed analysis of some optimizations employed to increase the efficiency of a GPU-based algorithm. We started with a sequential H.264 Motion Estimation algorithm, then we implemented a GPU-based algorithm, and finally some optimizations were applied to improve the GPU-based algorithm performance. This paper evaluates performance in terms of time, but also in terms of energy consumption, which is a very important feature due to the GPUs design.

In this way, the proposed optimizations were tested using a large and varied set of video sequences with the aim of evaluating their performance. As a result, the encoder performed very well with all of them, achieving a reduction in computational time of up to 91% with negligible rate distortion loss, and the energy consumption was 9.53 times better than that of the reference implementation.

7 REFERENCES

[1] T. Wiegand, G.J. Sullivan, G. Bjontegaard, A. Luthra: “Overview of the H.264/AVC Video Coding Standard”. In: IEEE Trans. Circuits Syst. Video Technol. 13(7), 560 - 576 (2003).

[2] W.-c. Feng, D. Manocha. "High-performance computing using accelerators". Parallel Computing, vol 33, n. 10-11, pp 645-647, November 2007.

[3] NVIDA, NVIDIA CUDA Compute Unified Device Architecture-Programming Guide, Version 2.3, August 2009.

[4] ISO/IEC International Standard 14496-10:2003, “Information Technology – Coding of Audio - Visual Objects – Part 10: Advanced Video Coding”.

[5] ISO/IEC International Standard 14496-10:2005, “Information technology – Coding of audio-visual objects – Part 10: Advanced Video Coding”.

[6] GPGPU (2007). General-purpose computation using graphics hardware. Available online at: http://www.gpgpu.org.

[7] Shane Ryoo , Christopher I. Rodrigues , Sara S. Baghsorkhi , Sam S. Stone , David B. Kirk , Wen-mei W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, February, 2008, Salt Lake City, UT, USA

[8] M. Boyer, K. Skadron and W. Weimer, “Automated dynamic analysis of CUDA programs”, In Proceedings of 3rd Workshop on Software Tools for MultiCore Systems (STMCS), April, 2008, Boston, MA, USA.

[9] Timothy D.R. Hartley , Umit Catalyurek , Antonio Ruiz , Francisco Igual , Rafael Mayo , Manuel Ujaldon, “Biomedical image analysis on a cooperative cluster of GPUs and multicores”, Proceedings of the 22nd annual international conference on Supercomputing, June, 2008, Island of Kos, Greece.

[10] Huang S, Xiao S, Feng W, “On the energy efficiency of graphics processing units for scientific computing”, In proceedings of the 2009 IEEE international symposium on parallel& distributed processing, IPDPS, pp 1–8, May, 2009, Rome, Italy

[11] Xiaohan Ma and Mian Dong and Lin Zhong and Zhigang Deng, "Statistical Power Consumption Analysis and Modeling for GPU-based Computing", In Proceeding of ACM SOSP Workshop on Power Aware Computing and Systems (HotPower), October 2009, Big Sky, MT, USA

[12] W.-N. Chen, H.-M. Hang, "H.264/AVC motion estimation implementation on Compute Unified Device Architecture (CUDA)", In Proceedings of IEEE International Conference on Multimedia and Expo, ICME, pp. 679-700, June 2008, Hannover, Germany.

[13] R. Rodriguez, J. L. Martínez, G. Fernández-Escribano, J. M. Claver, J. L. Sánchez "Accelerating H.264 Inter Prediction in a GPU using CUDA", In International Conference on Consumer Electronics, ICCE 2010, 10.4-2, pp. 463-464, Las Vegas, USA.

[14] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, Reference Software to Committee Draft. JVT-F100 JM15.1, 2009. Available on-line at http://iphome.hhi.de/suehring/tml/.

[15] G. Bjøntegaard, "Calculation of Average PSNR Differences between RD-Curves", presented at the 13th VCEG-M33 Meeting, Austin (Texas), USA, April 2001.

[16] G. Sullivan and G. Bjøntegaard, "Recommended Simulation Common Conditions for H.26L Coding Efficiency Experiments on Low-Resolution Progressive-Scan Source Material". ITU-T VCEG, Doc. VCEG-N81. September 2001.

[17] JVT Test Model Ad Hoc Group, "Evaluation Sheet for Motion Estimation", Draft version 4, February 2003