IET Computers & Digital Techniques

Research Article

# CUDA memory optimisation strategies for motion estimation

ISSN 1751-8601
Received on 11th July 2017
Revised 26th January 2018
Accepted on 15th May 2018
E-First on 6th September 2018
doi: 10.1049/iet-cdt.2017.0149
www.ietdl.org

*Fatma Elzahra Sayadi*[1] ✉, *Marwa Chouchene*[1], *Haithem Bahri*[1], *Randa Khemiri*[1], *Mohamed Atri*[1]
[1]*Electronics and Microelectronics Laboratory, FSM, University of Monastir, Environment Street, Monastir 5019, Tunisia*
✉ *E-mail: sayadi_fatma@yahoo.fr*

**Abstract:** As video processing technologies continue to rise quicker than central processing unit (CPU) performance in complexity and image resolution, data-parallel computing methods will be even more important. In fact, the high-performance, data-parallel architecture of modern graphics processing unit (GPUs) can minimise execution times by orders of magnitude or more. However, creating an optimal GPU implementation not only needs converting sequential implementation of algorithms into parallel ones but, more importantly, needs cautious balancing of the GPU resources. It requires also an understanding of the bottlenecks and defect caused by memory latency and code computing. The defiance is even greater when an implementation exceeds the GPU resources. In this study, the authors discuss the parallelisation and memory optimisation strategies of a computer vision application for motion estimation using the NVIDIA compute unified device architecture (CUDA). It addresses optimisation techniques for algorithms that surpass the GPU resources in either computation or memory resources for CUDA architecture. The proposed implementation reveals a substantial improvement in both speed up (SU) and peak signal-to-noise ratio (PSNR). Indeed, the implementation is up to 50 times faster than the CPU counterpart. It also provides an increase in PSNR of the coded test sequence up to 8 dB.

## 1 Introduction

Accentuated by the ravenous market exigency for real time and high definition, the programmable graphics processing unit (GPU) has progressed into a highly parallel, multithreaded and many core processors with enormous computational power and very large memory bandwidth leading to a significant disparity between the central processing unit (CPU) and the GPU. Hence, many applications, that process large data sets such as image and video processing, are turning to data-parallel programming model using GPUs, to accelerate the computations [1–4]. In fact, the GPU is well adapted to address problems showing off an intensive parallel computing by mapping data elements to parallel processing threads. Furthermore, recent GPUs have been redesigned to be the most energy-efficient processors in the market. On a per-instruction basis, GPUs are more power efficient than CPUs, which commonly have managed the bulk of computations [5].

In this context, we have chosen as a case study the full-search (FS) motion estimation (ME) algorithm used in most of multimedia streaming applications such as H264 and high-efficiency video coding (HEVC). Compared to H.264/ advanced VC (AVC), HEVC reduces the bitrate by 50% on average while preserving the same visual quality [6, 7]; nevertheless, the HEVC encoding is more complex than H.264. Consequently, classical method, which carries out the encoding in a serial way, could no longer supply a real-time application. The solution is to use a GPU data-parallel computing model to develop software algorithm that transparently scales its parallelism to make use of the increasing number of processor cores [8].

There have been a number of published works using GPU cards for ME as a part of a video coding scheme, see [9–14] for a review. Nevertheless, most of these implementation attempts originated without the introduction of the optimised technique given by NVIDIA.

Although mastering the processing of massively parallel code is an interesting matter, with a basic comprehension of the principles behind the efficient parallel code, we can obtain considerable performance growth compared with a classical computing and a sequential computing of the same algorithms. Hence, in order to ensure maximum benefit in terms of performance increase, we need to be conscious of the bottlenecks related to coprocessor/GPU programming [15, 16].

This paper will discuss and demystify these performance-robbing bottlenecks, and provide simple ways to make these a non-factor in the application such as host/device transfers, data movement and memory optimisations. Indeed, memory optimisations are the most important area for performance. The object is to maximise the use of the hardware by maximising bandwidth which is best served by involving as much speedy memory and as little slow-access memory as possible.

Thus, we propose an efficient GPU-based parallel ME algorithm that reduces the memory access latency by removing the communication of intermediate results as well as by taking full advantage of the on-chip memory. Combined with an efficient use of the texture memories makes the peak signal-to-noise ratio (PSNR) and the speed up even higher.

Other additional factors are skillfully addressed in order to reach an efficient implementation. One instance is choosing a good grid topology in a way that maximises hardware utilisation and limits practises that impedes the free distribution of work and the thread usage.

In addition, by proposing a new parallel reduction (PR) algorithm, we enhance thread usage and minimise thread synchronisations by performing several PR operations simultaneously.

By integrating all these approaches, we have achieved great performance since we obtained the best PSNR while maintaining comparable execution time with that of the literature.

The rest of this paper is organised as follows: Section 2 introduces the architecture and memory spaces of the GPU. Section 3 presents details of the proposed parallel FS algorithm. Section 4 shows balancing applications and optimisation interaction; it gives also performance evaluation, which is followed by the final conclusion in Section 5.

## 2 Architecture and memory spaces of graphics processing units

In the NVIDIA GPU-based architecture, parallelisation is obtained through the execution of tasks in a number of stream processors

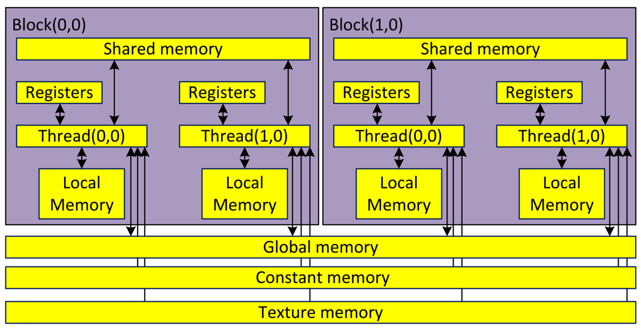
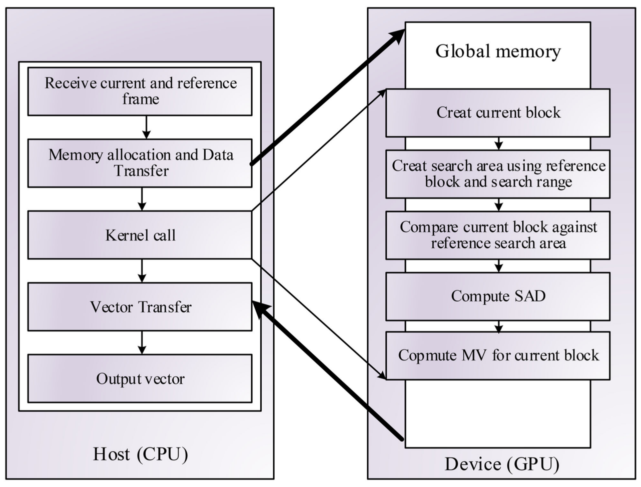


**Fig. 1** *CUDA memory types*



Receive current and reference frame
Memory allocation and Data Transfer
Kernel call
Vector Transfer
Output vector
Host (CPU)
Global memory
Creat current block
Creat search area using reference block and search range
Compare current block against reference search area
Compute SAD
Copmute MV for current block
Device (GPU)

**Fig. 2** *Proposed algorithm flowchart*

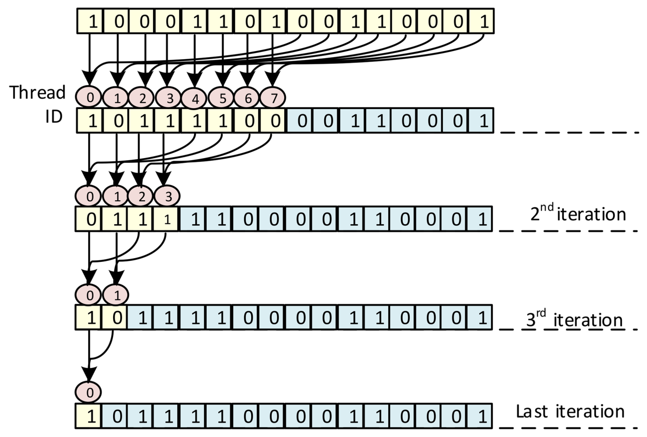


**Fig. 3** *PR process for the summation*

(SPs) or compute unified device architecture (CUDA) cores. The NVIDIA GPU features several CUDA cores organised in streaming multiprocessors (SMPs) that execute in parallel. CUDA cores can be used to execute integer and floating-point instructions. All cores in an SMP execute the same instruction at the same time. This computational paradigm, called single instruction multiple thread, can be considered as an advanced form of the single instruction multiple data paradigm. The code is executed in groups of 32 threads called warps.

The GPU architecture supports several different memory spaces as portrayed in Fig. 1: global, local, texture, constant, shared and register memory [17].

All threads can access global memory, highest throughput can be achieved by accessing consecutive memory locations, that is, with the memory access coalescing.

For each block, shared memory is available for all threads within the block, while registers are the local storage for each SP. Register and shared memory are much faster than device memory that can be used to speed up the access. Constant memory and texture memory are read-only memories accessible for all threads. Texture memory and constant memory are cached memories that can be written by the CPU and read by the GPU. Despite that, the whole GPU memory is available; the CUDA programming environment has some memory restrictions. In fact, the limits of the shared memory per CUDA block and the number of registers are basically two of the most important elements to remember when tuning an algorithm in CUDA. Therefore, incorrectly making use of the available memory in the application can rob of the desired performance.

This paper explores the various kinds of memory on the device and on the host and how best to set up data items in order to exploit the memory effectively. In fact, shared memory, local memory and register memory all have their advantages, disadvantages and need to be used very carefully to save valuable clock cycles. A circumstance such as bank conflicts, memory spilling (too much data being stored in registers and transferred into local memory), improper loop unrolling, as well as the amount of shared memory, all play essential roles in getting a significant performance.

## 3 Parallelisation of the application

This section describes the approaches toward GPU-based parallelism as well as different optimisation strategies. Our goal was to encode video using the ME algorithm in real time. We have used the FS block-matching algorithm based on the sum of absolute differences (SADs).

We begin with an original version of FS motion implementation in order to find the optimal block displacement. This implementation can be applied to both H264 and HEVC encoders by redesigning programme parameters.

The implemented algorithm applies the SAD error criterion which is an extremely fast metric used for block matching in ME for video compression. It takes the absolute value of the difference between each pixel in the original block and the corresponding pixel in the block being used for comparison and sums them up. Hence, it is easily parallelisable, making it readily implementable with parallel programming models such as GPUs.

Our proposed method applies only one CUDA kernel in order to implement (FS) algorithm and to compute the best final motion vector (MV). This differentiates our approach from approaches of [14]. In their work, Zhang *et al.* divide the workload of the proposed algorithm into two OpenCL kernels. One is in charge of computing the SADs and the other one is in charge of comparing these SADs to select the MV.

In the present approach, we transfer the needed frames of the test sequence at one time to the device's memory (global memory). We define the FS motion implementation CUDA kernel, which creates a thread for each result element for the SAD computation. Many threads are created in an attempt to hide the latency of the global memory by overlapping execution. Once all blocks from the search range are iterated and the best match found, the output vector is transferred to the global memory. The flowchart is shown in Fig. 2. Memory transfers are shown by wide arrows, the kernel running in GPU is called by the host.

In the FS CUDA kernel, we compute the SAD of every 16 px' column in a $16 \times 16$ block. The result is saved in a local variable SAD [*i*] [*j*], where *i* indicates the number of the currents $16 \times 16$ block and *j* indicates the number of the columns whose SAD is calculated. This calculation is done in a very classical way based on a for loop. Once the 16 SADs of all columns of a block are calculated, the SAD of the corresponding block is computed by applying PR, which is a tree-based approach exchanging data between threads within the same thread block [18].

Fig. 3 shows an example of the PR process with eight threads for the minimum among 16 data. The amount of data to be processed is halved after each iteration, resulting in the minimum after the last iteration. In simple terms, the vector is divided into two elements. The first element of the first half and the second one are summed, and the result is stored in the first box. Then, the second elements of the first and second halves are summed and the result is stored in the second box. The accumulation is iterated until the total sum is calculated. The value that is finally left is the SAD.

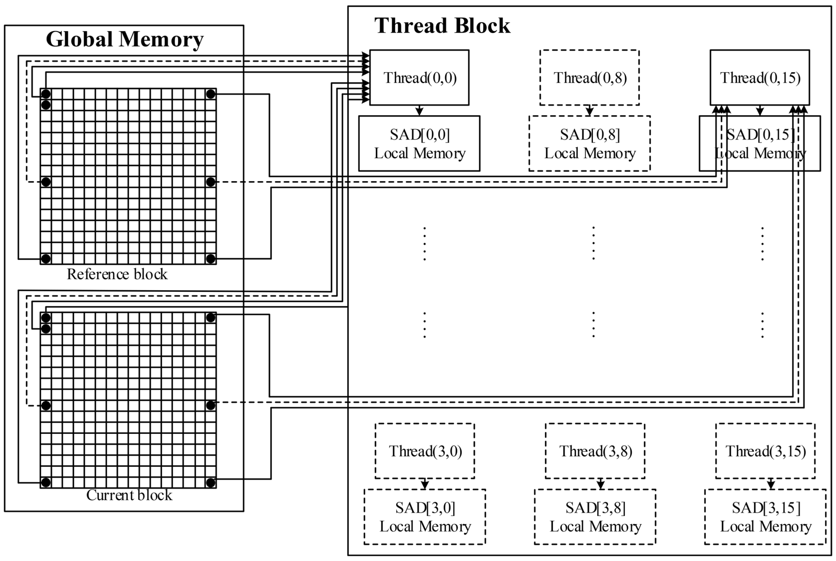


**Fig. 4** *Workload distribution*

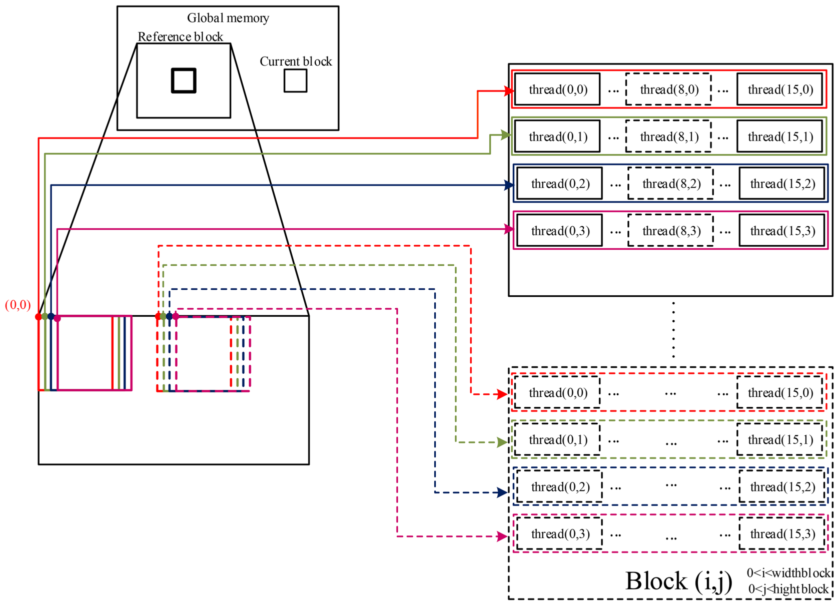


**Fig. 5** *First proposed architecture*

The advantage of this method comes from the parallelisation of memory accesses and arithmetic operations. In this method, memory accesses coalesce very well as all the threads in a warp read at the same time from adjacent memory locations.

Each warp can independently and concurrently work without any data hazard. Thus, no thread synchronisation is needed between iterations. Furthermore, in every iteration, all threads in the thread block are active, which means 100% thread utilisation.

Thus, minimising the amount of thread synchronisation and combining the per-thread results together in parallel reduce the parallel programme's running time and improve its performance.

In the proposed implementation, each thread works on its column in $16 \times 16$ block, reads $16{\times}2\ \text{px}^2$ values at the same position of the current block and the reference block, calculates SAD of one 16 px' column and the final value is written back to the local memory (Fig. 4) and not into register. In fact, for small arrays where all indices are known constants at compile time, the compiler places all consulted elements of the array into registers. When the compiler cannot settle array indices to constants, it must store arrays into GPU local memory. 'Local' here implies that it is local to each thread and not apparent to the others. Local memory actually exists in global GPU memory. Each thread has its own copy of any local array, and the compiler produces store and load operations for array reads and writes, respectively.

In the execution configuration parameter 'the number of blocks per grid or grid size', the primary concern is keeping the entire GPU busy. In fact, a device in which work is poorly balanced across the multiprocessors will deliver suboptimal performance.

To this end, in the present approach, the grid topology is mapped as follows: the grid has size $[W/16, (H+3)/4]$, that is, it has $W/16$ blocks in the $x$-direction and $(H+3)/4$ blocks in the $y$-direction, where $W$ and $H$ are, respectively, the width and the height of the image. Each block is of size (16, 4) with 16 threads along the $x$-direction, and four threads along the $y$-direction. The topology we are presenting here is not arbitrary or mere whims but is instead a response to numerous tests permitting to achieve a GPU highest occupancy and a minimum execution time.

As shown in Fig. 5, each CUDA block computes four search points in the search range allowing thus to reach maximum usage, unlike the majority of researches [13] which settle as much CUDA blocks as search points.

An example of MVs obtained by FS computation on the two frames is shown in Fig. 6.

There are different methods to estimate a CUDA kernel runtime [19]. The most portable option is to use CUDA's built-in timer functions, which will function across different operating systems. To measure runtime, we need to create and start a timer before calling the kernel. After calling the kernel, we need to make sure that the kernel has finished, then stop, and read the timer.

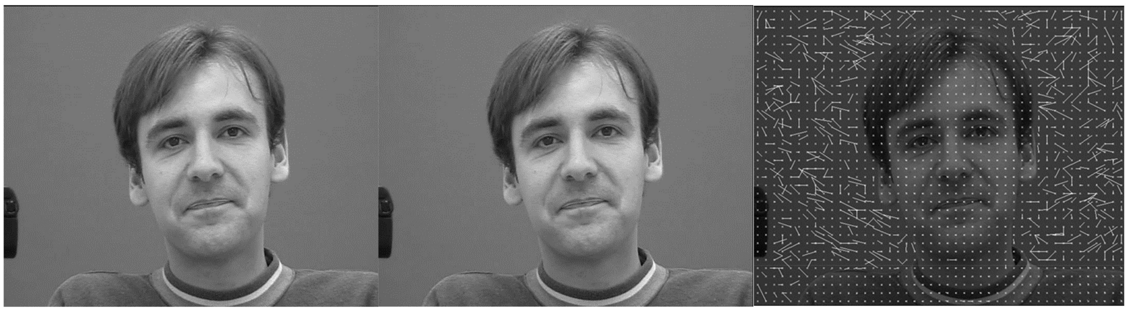


**Fig. 6** *Example of MVs obtained by FS computation on the two frames*
***(a)*** Reference frame, ***(b)*** Current frame, ***(c)*** Output with MVs

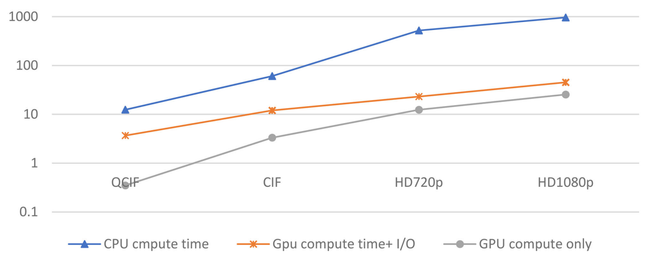


**Fig. 7** *Execution time for Intel Core I7 versus GTX480*

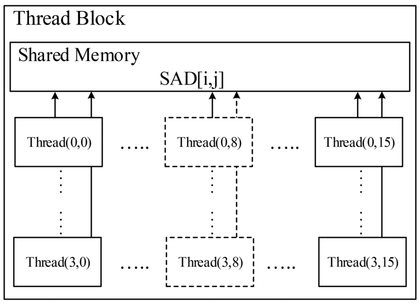


**Fig. 8** *Shared memory usage*

Here, cutStartTimer() and cutStopTimer() are used to place the start and stop events into the default stream, stream 0. The GPU will register a timestamp for the event when it attains that event in the stream. The cutGetTimerValue() function returns the time past between the recording of the start and stop events [20]. This value is formulated in milliseconds and has a resolution about half a microsecond.

We should note that the timings are evaluated on the GPU clock, so the timing resolution is operating-system-independent. We should mention also that the execution time computed for ME is done on two successive images of a grey-scale video sequence.

The experiments were performed on a personal computer with an Intel Core I7-3770 3.40 GHz CPU with 8 GB memory and a graphic card NVIDIA GeForce GTX480. This card is based on the GF100 Fermi chip, which is produced in 40 nm at Taiwan Semiconductor Manufacturing Company. The GTX480 uses only 480 of the 512 shaders of the GF100 silicon [21].

The software was coded in the programming language with CUDA toolkits version 5.0 and the developed software was created in Visual C++ 2010.

The timings plotted in logarithmic scale are given in Fig. 7. From these timings, it is clear that the implementation scaled well for larger image sizes on GPUs. In fact, for the HD1080p format we were able to achieve an execution time of 45.2 ms on the GTX480 against 962.7 ms on serial execution, indicating a significant speed up.

The proposed implementation promises improved utility in real-time ME, since the performance increase has multiplied and the execution time is lower than highly optimised libraries executing on a CPU.

Although the results of the GPU implementation are well above expectations, Fig. 7illustrates very clearly that most of the time is spent on nothing but input/output over the bus. Hence, it should be reduced to a negligible level with some programming effort and different GPU optimisation strategies developed by NVIDIA without degrading performance.

## 4 Balancing applications and optimisation interaction

In this section and in an effort to further improve performances, we attempt to apply different CUDA memory optimisation strategies. In fact, as said in [22], performance optimisation revolves around three basic strategies:

- Maximise parallel computing to reach maximum usage.
- Optimise memory utilisation to reach largest memory throughput.
- Optimise instruction usage to reach largest instruction throughput.

### *4.1 Local versus shared memory usage*

In the initial kernel version, we have used a two-dimensional (2D) variable to keep the values of SADs. This variable is placed into GPU local memory and not into register.

Regrettably, involving local memory is slower than storing array elements directly in registers; to partially cope with this problem, we can place the private array explicitly in the shared memory.

Shared memory has 32 banks arranged such that successive 32 bit words map to successive banks. In our case, we will allocate a __shared__array large enough to fulfil the private arrays of all threads of a thread block.

We will properly attribute elements of this new __shared__array to the threads of the thread block; thus, all elements of the new virtual private array for each thread are placed in its own shared memory bank as shown in Fig. 8.

To evaluate the benefits of such memory on different sequences, the performance of local versus shared memory are compared as shown in Table 1. The proposed scheme will be evaluated based on two criteria:

- *ΔPSNR*: It is the PSNR difference between the sequential implementation and the parallel one. It is defined by

$$\Delta\text{PSNR} = \text{PSNR(CPU)} - \text{PSNR(GPU)}$$

- SU: It is the speed up of the proposed FS algorithm per frame in the case of one reference frame. SU is the ratio of two execution times defined by

$$\text{SU} = \frac{T_{\text{CPU}}}{T_{\text{GPU}}}$$

- where $T_{\text{CPU}}$ and $T_{\text{GPU}}$ are the ME execution times of the sequential and parallel implementations, respectively. The data transfer time between CPU and GPU is included in the GPU-based ME execution time measurement.

**Table 1** Performances using local memory versus shared memory

| | Parallel computation | | | |
|---|---|---|---|---|
| | Local memory | | Shared memory | |
| | SU | ΔPSNR | SU | ΔPSNR |
| QCIF | 3.37 | −0.02 | 31.07 | −0.01 |
| CIF | 5.03 | −0.03 | 21.62 | −0.01 |
| HD720p | 22.54 | −0.01 | 48.04 | −0.01 |
| HD1080p | 21.29 | −0.03 | 48.23 | −0.02 |

Table 1 shows the performances of the two implemented schemes performed on different resolution sequences and using, respectively, local versus shared memories. As presented in this table, we can observe that the GPU implementation does not affect the visual quality of tested video sequences. It induces an insignificant PSNR loss compared with the CPU implementation.

However, Table 1 demonstrates that the approach using shared memory can reduce up to 50% of the FS computation time against only 20% when using the local memory.

To understand the reasons for this gap between the two parallel implementations, let us dig into the GPUs' activity with Fig. 9, which breaks the GPU time down into the transfer time, the compute and the idle time.

We note that for the implementation with the local memory, data transfers dominate the overall execution time, where it can reach up to 90% of the total time, against a maximum of 25% for the implementation with the shared memory.

In fact, because it is on-chip, shared memory is much faster than local and global memories thus justifying an increase in performances when using the shared memory.

For this reason, we will use this memory type in all the following paper.

## 4.2 Texture memory usage

In previous paragraphs, the global memory was used to store the current and reference image; however, the local or the shared memory was used to store the different SAD values. The obtained results show the efficiency of a suitable use of the shared memory.

CUDA architecture supplies another type of memory, called texture memory. Like constant memory, texture memory is another kind of read-only memory that can enhance results and reduce memory flow [22].

Although texture memory was originally designed for traditional graphics applications, it can also be used quite effectively in some GPU computing applications. Texture memory has some properties that make it extremely useful for computing.

It is cached on a chip, so in some situations it will provide higher effective bandwidth by reducing memory requests to off-chip dynamic random-access memory.

When accessing 2D arrays in global memory, using the Texture Cache has theoretically many advantages such as filtering and not having to worry as much for memory access patterns. In the course of this paper, we will verify the property and the limits downside of such memory.

**Listing 1 :**

```
texture<uint8_t, 2, cudaReadModeElementType> frameRef;
texture<uint8_t, 2, cudaReadModeElementType> frameCur;
```

**Fig. 10** *2D texture memory definition example*

The reference and current frames are stored in 2D cached texture memory as declared in Listing 1 (Fig. 10). All other variables are placed in fast register memory associated to the processor or in the shared memory. Only one write to the global memory is accomplished at the thread end in order to place the computed value of the MV.

Fig. 11 gives an example of 2D texture memory that provides access to current and reference frames.

The results are reported in Table 2 below.

The speed up shows that the more the amount of data to be analysed is important, the more the performance gain will be higher for both implementations. However, as it can be seen, using textures decreases slightly the performance in terms of speed up. In fact, in the worst case, where cache hit rates are very low, using textures is slower than normal memory access. Each thread has to first have a cache miss, then starts global memory fetch. The resulting total latency will be greater than a direct read from memory, whereas texture memory permits to attain best PSNR since texture version improves PSNR with over 8 dB compared with the global version, a feat that other work could not reach [23–25].

## 4.3 Pinned memory usage

In the foregoing, we have presented our results using the pageable memory; in this new section, we will discuss how to efficiently transfer data between the host and the device. In fact, when a data transfer from pageable host memory to device memory is produced, the GPU cannot reach data directly from pageable host memory. The CUDA driver must at first allocate 'pinned' or a temporary page locked, host array, transfers the host data to the pinned array, and then copy the data from the pinned array to the device memory, as illustrated in Fig. 12. The pinned memory is considered as a staging area for transfers from the device to the host [26].

As shown in Fig. 12 we can resume that it avoids the cost of the transfer between pageable and pinned host arrays by directly allocating the host arrays in pinned memory. Pinned memory is allocated using the cudaHostAlloc() or cudaMallocHost () functions in the runtime application program interface, and deallocate it with cudaFreeHost(). The sample shows how to use these functions (Fig. 13).

We have tried to check the performance of this memory by measuring the execution time in both cases as shown in Table 3.

In this paper, results obtained with pinned data transfer are appropriate, for all test sequences. Although in some cases, those obtained with high resolutions not perform as well due to the overuse of the pinned data since it is a scarce resource.

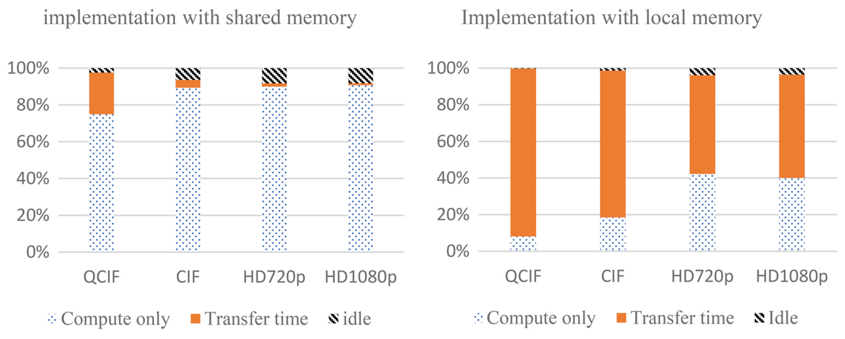


**Fig. 9** *GPU time breakdown*

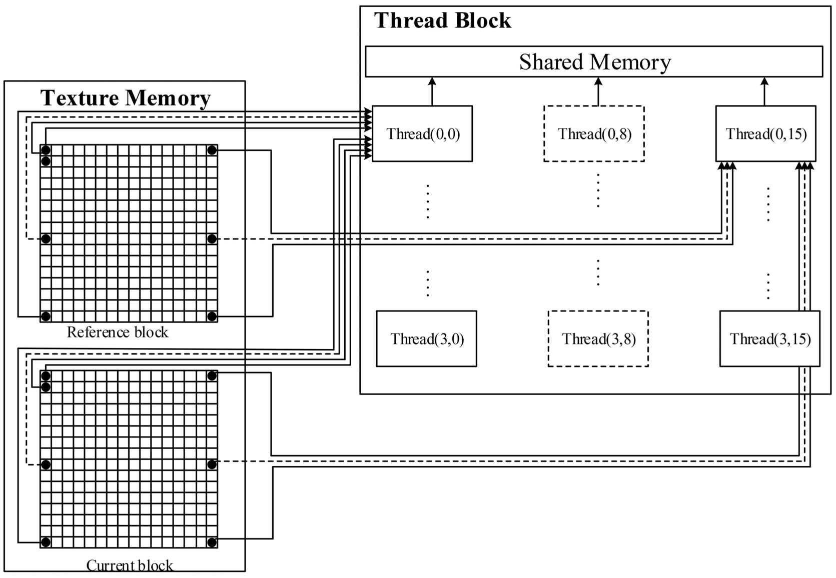


**Fig. 11** *Architecture based on texture and shared memory*

**Table 2** Speed up and ΔPSNR using global memory versus texture memory

| | Global memory | | Texture memory | |
|---|---|---|---|---|
| | SU | ΔPSNR | SU | ΔPSNR |
| QCIF | 31.07 | −0.01 | 3.14 | 8.18 |
| CIF | 21.62 | −0.03 | 8.54 | 5.9 |
| HD720p | 48.04 | −0.01 | 36.327 | 2.73 |
| HD1080p | 48.23 | −0.02 | 39.23 | 4.45 |

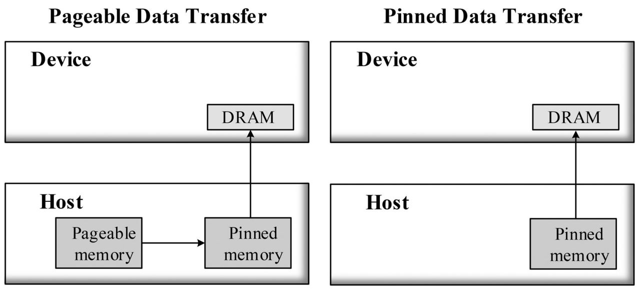


**Fig. 12** *Data transfer between the host and the device*

```
Listing 2:
CUDA_SAFE_CALL_NO_SYNC(cudaMallocHost(&h_frame1, frameSize));
CUDA_SAFE_CALL_NO_SYNC (cudaMallocHost(&h_frame2, frameSize));
CUDA_SAFE_CALL_NO_SYNC (cudaMallocHost(&visVectors, frameSize));
CUDA_SAFE_CALL_NO_SYNC (cudaMallocHost(&visResidualOrig, frameSize));
CUDA_SAFE_CALL_NO_SYNC (cudaMallocHost(&visResidualComp, frameSize));
CUDA_SAFE_CALL_NO_SYNC (cudaMallocHost(&visCompensated, frameSize));
CUDA_SAFE_CALL_NO_SYNC (cudaMallocHost(&h_mvf_CPU, numBlocks16x16 * sizeof(MV_t)));
CUDA_SAFE_CALL_NO_SYNC (cudaMallocHost(&h_mvf_GPU, numBlocks16x16 * sizeof(MV_t)));
                                        ...
                                        ...
                                        ...
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(h_frame1));
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(h_frame2));
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(visVectors));
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(visResidualOrig));
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(visResidualComp));
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(visCompensated));
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(h_mvf_CPU));
CUDA_SAFE_CALL_NO_SYNC (cudaFreeHost(h_mvf_GPU));
```

**Fig. 13** *Pinned memory allocation and deallocation*

## 5 Comparison to the state of the art

Aiming to accelerate the FS algorithm, researchers have proposed various fast ME methods. Although fast and practical on a single CPU, most of these algorithms cannot satisfy the high-performance systems to meet real time. Therefore, some research focuses on the implementation of ME algorithm on GPUs. These works proposed to H.264/AVC and HEVC were compared with ours in order to emphasis our contribution.

We can quote, for example, the work of Nien-Chen and Hsueh-Ming [27], which described a parallel ME algorithm via CUDA for the common intermediate format (CIF) and quart of common intermediate format (QCIF) sequence sizes. The authors have obtained a parallel programme 12 times faster than the serial one.

Zhang *et al.* in [14] present a parallel ME algorithm implemented using OpenCL as well as several optimisation strategies. In accordance with experiments, the motion estimator reaches 3× speed up compared with its execution by the CPU.

The research presented in [28] has mapped, also the classic sequential FS algorithm to CUDA parallel computing model by optimising memory usage. Experimental results show that the implementation on CUDA reveals significant amelioration up to 50 times than CPU counterpart available and can indeed speed up the FS for ME.

Reference [9] can be cited also as an example that proposed a GPU-based parallel ME algorithm to improve the performance of an HEVC encoder. The proposed encoder reduces total encoding time by 56.2% with 2.2% Blu-ray disc-bit rate increases against an HEVC test model (HM) encoder.

Another work has been published for the same function; it is the one of Kung *et al.* [29] who implemented the FS ME. The obtained speed up varies between 10.4 and 31.68 times for, respectively, the CIF and HD720p, whereas our GPU implementation has given a speed up of, respectively, 38.32 times and 59.93 times. Lee and Oh [30] propose a highly parallel variable block size FS ME algorithm with concurrent PR on (GPU). This approach minimises memory access latency by using the high-speed on-chip memory of GPU. Experimental results show that the proposed approach achieves substantial improvement up to 92 times than the CPU only counterpart.

All these literature searches mentioned above did not reveal the existence of any studies aimed at evaluating the PSNR.

Among the works that have treated the video quality, we can quote the one presented by Radicke *et al.* [31] who propose a fast inter-prediction algorithm on GeForce GTX 680 with CUDA toolkit version 5.0. Results show a speed up of 57.41% compared with the original HM10.0 with a small degradation of PSNR from −0.015 to −0.029 dB.

Another example is the research of Khemiri *et al.* [24], where the authors thought about optimising the computation by parallelising the algorithm on two successive GPU generations. The GPU performance for a block size of $64 \times 64$ px$^2$ is enhanced, compared with that with CPU, by a factor of almost 2 for Fermi GPU system and by a factor of 1.33 for Kepler GPU system. These results are obtained with a slight decline in video quality (ΔPSNR is around −0.1).

Reference [32] designed a high-quality HEVC ME implementation with the cooperation of CPU and GPU. By

**Table 3** Execution time using pageable memory versus pinned memory

| | GPU implementation | | | |
|---|---|---|---|---|
| | Pageable memory | | Pinned memory | |
| | Execution time, ms | SU | Execution time, ms | SU |
| QCIF | 0.4 | 31.07 | 0.29 | 42.8 |
| CIF | 2.8 | 21.62 | 1.58 | 38.32 |
| HD720p | 10.84 | 48.04 | 8.69 | 59.93 |
| HD1080p | 19.96 | 48.23 | 18.25 | 52.75 |

applying this mechanism, the encoding speed is promoted (speed up is 32.77 times) with a slight influence on quality (ΔPSNR is about −0.03).

Reference [25] proposes a highly parallel and scalable ME algorithm, named multilevel resolution ME (MLRME). The experiments showed that MLRME can achieve about 30–60× speed up on GPU compared with its serial method on a single core of the CPU. The results on several test sequences showed that the loss of PSNR of the picture coded by MLRME is about 1.5%.

The research published in [33] proposed an implementation of the ME on a low-cost NVIDIA GPU. On the basis of principally the SADs or the sum of square differences (SSDs), the authors proposed to accelerate these algorithms. The proposed parallel algorithm permits a significant reduction in the execution time that reaches up to 56.17 and 30.4%, compared with the CPU, for SAD and SSD algorithms, respectively. However, this improvement is accompanied by an average PSNR loss of 0.095 dB.

Most reviewed researches indicate that they bet on the acceleration without considering a quality improvement; however, our proposed work focuses on both. We successfully met this challenge through the efficient use of shared and texture memories.

We have revealed that the GPU can be specifically optimised for ME algorithm by using the texture caches, which are designed for graphics' applications, where memory access patterns exhibit a great deal of spatial locality. Hence, we not only improved the execution time by up to 40 times but we also improved the PSNR by about 8.

## 6 Conclusion

We have presented in this paper an original FS ME GPU implementation that can be applied on both H264 and HEVC encoders. This implementation is based on a single CUDA kernel, which is in charge of the SADs, the best SAD and the MV computations. We have started with the local memory, which is used only to hold automatic variables. The compiler makes use of such memory when it determines that there is not enough register space to hold variables. Automatic variables that are large structures or arrays are also typically placed in the local memory. Experimental results show that our method provides up to 21 times speed up compared with single CPU core implementation while preserving approximately the same PSNR.

To improve even more this speed up, we provided a number of optimisation strategies and derived profile information in order to help track down memory bottlenecks.

Obviously, using the local memory is slower than keeping array elements directly in registers; to partly mitigate this problem, we proposed to store the private array explicitly in the shared memory. In general, it is best to take advantage of the shared memory because threads within the same block utilising the shared memory can communicate. We have proved that combined with its excellent performance, this makes the shared memory a good 'all around' choice when used correctly. In fact, the proposed algorithm based on the shared memory allowed an increase in speed up reaching 48. In some cases, however, it may be better to make use of the other types of available memories.

Concerning the texture memory, we have proved that as it can be faster, the same speed, it can also be slower than 'naked' global memory access. There are no general rule facts for predicting performance using textures, as the speed up (or lack of speed up) is determined by data usage patterns within the code and the texture hardware being used. However, we can see that there is a real increase in the quality of the parallel algorithm since the PSNR increases and reaches roughly 8 dB while maintaining an acceptable speed up.

The last observation concerns the pinned data that should not be overused; otherwise, it reduces overall system performance. Using the pinned data, the amount of memory access is significantly reduced since our experimental results provide up to 50 times speed up.

## 7 References

[1] Chouchene, M., Sayadi, F.E., Bahri, H., *et al.*: 'Optimized parallel implementation of face detection based on GPU component', *Microprocess. Microsyst.*, 2015, **39**, (6), pp. 393–404

[2] Chouchene, M., Sayadi, F.E., Said, Y., *et al.*: 'Efficient implementation of Sobel edge detection algorithm on CPU, GPU and FPGA', *Int. J. Adv. Media Commun.*, 2014, **5**, pp. 105–117

[3] Bahri, H., Sayadi, F.E., Khemiri, , *et al.*: 'Image feature extraction algorithm based on CUDA architecture: case study GFD and GCFD', *IET Comput. Digit. Tech.*, 2017, **11**, (4), pp. 125–132

[4] Bahri, H., Sayadi, F.E., Chouchene, M., *et al.*: 'Accelerating Fourier descriptor for image recognition using GPU', *Appl. Math. Inf. Sci.*, 2016, **10**, (1), pp. 297–306

[5] www.nvidia.com/object/gcr-energy-efficiency.html, accessed 4 July 2017

[6] Bross, B., Marpe, D., Schwarz, H.: 'The New high-efficiency video coding standard', *SMPTE Motion Imaging J.*, 2013, **122**, pp. 25–35

[7] Radicke, S., Hahn, J., Grecos, C., *et al.*: 'A highly-parallel approach on motion estimation for high efficiency video coding (HEVC)'. Proc. IEEE Int. Conf. Consumer Electronics (ICCE), Berlin, Germany, July 2014, pp. 187–188

[8] Obukhov, A.: 'GPU-accelerated video encoding'. NVIDIA DevTech, 2010. Available at http://on-demand.gputechconf.com/gtc/2010/presentations/S12075-GPU-Accelerated-Video-Encoding.pdf, accessed 25 December 2016

[9] Lee, D., Sim, D., Cho, K., *et al.*: 'Fast motion estimation for HEVC on graphics processing unit (GPU)', *J. Real-Time Image Process.*, 2016, **12**, (2), pp. 549–562, doi: 10.1007/s11554-015-0522-6

[10] Carlos, G., Guillermo, B., Francisco, D.S., *et al.*: 'Fast-coding robust motion estimation model in a GPU'. Proc. SPIE 94000, Real-Time Image and Video Processing, San Francisco, CA, USA, February 2015

[11] Chen, D., Su, H., Mei, W., *et al.*: 'Scalable parallel motion estimation on multi-GPU system'. Proc. Second Int. Symp. Computer, Communication, Control and Automation (ISCCCA-13), Shijiazhuang, China, February 2013, pp. 628–632

[12] Cheung, N., Fan, X., Au, O.C., *et al.*: 'Video coding on multicore graphics processors', *IEEE Signal Process. Mag.*, 2010, **27**, (2), pp. 79–89

[13] Song, T., Koshino, M., Matsunohana, Y., *et al.*: 'Parallel implementation algorithm of motion estimation for GPU applications', '*Bulletin of Institute of Technology and Science*' (University of Tokushima, Tokushima, Japan, 2011), pp. 58–63

[14] Zhang, J., Nezan, J.F., Cousin, J.G.: 'Implementation of motion estimation based on heterogeneous parallel computing system with OpenCL'. Proc. IEEE 14th Int. Conf. High Performance Computing and Communication & IEEE Ninth Int. Conf. Embedded Software and Systems, Liverpool, UK, June 2012, pp. 41–45, doi: 10.1109/HPCC.2012.16

[15] NVIDIA corporation: 'How to optimize data transfers in CUDA C/C++'. Available at https://devblogs.nvidia.com/parallelforal, accessed 4 February 2017

[16] NVIDIA Developer Technology Group: 'GPU optimization fundamentals'. Available at https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf, accessed 14 January 2017

[17] NVIDIA Developer Technology: 'Tutorial CUDA'. Available at http://fr.slideshare.net/angelamm2012/nvidia-cuda-tutorial, accessed 21 January 2017

[18] NVIDIA Developer Technology: 'Optimizing parallel reduction in CUDA'. Available at http://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf, accessed 13 December 2016

[19] Farber, R.: '*CUDA application design and development*' (Morgan Kaufmann, Elsevier, Massachusetts, USA, 2011, 1st edn.)

[20] Couturier, R.: '*Designing scientific application on GPUs*' (CRC Press, Taylor Francis Group, Florida, USA, 2014)

[21] http://www.geforce.com/hardware, accessed 10 January 2017

[22] 'CUDA c programming guide', 2015. Available at http://docs.nvidia.com/cuda/cuda-c-programming-guide, accessed 20 January 2017

[23] Fan, W., Dajiang, Z., Satoshi, G.: 'OpenCL based high-quality HEVC motion estimation on GPU'. IEEE Int. Conf. Image Processing (ICIP), Paris, France, 27–30 October 2014

[24] Khemiri, R., Chouchene, M., Bahri, H., *et al.*: 'Fast SAD algorithm of HEVC video encoder on two successive generations', *Int. J. Imaging Robot.*, 2017, **17**, (2), pp. 95–105

[25] Xue, Y., Su, H., Ren, J., *et al.*: 'A highly parallel and scalable motion estimation algorithm with GPU for HEVC', *Hindawi Sci. Program.*, 2017, **1**, (15), pp. 1–15

[26] 'CUDA c best practices guide', 2015. Available at http://docs.nvidia.com/cuda/cuda-c-best-practices-guide, accessed 2 February 2017

[27] Wei-Nien, C., Hsueh-Ming, H.: 'H.264/AVC motion estimation implementation on compute unified device architecture (CUDA)'. Proc. IEEE Int. Conf. Multimedia and Expo, Hannover, Germany, June 2008, pp. 697–700

[28] Jing, Z., Liangbao, J., Xuehong, C.: 'Implementation of parallel full search algorithm for motion estimation on multi-core processors'. Proc. Second Int. Conf. Next Information Generation Technology (ICNIT), Gyeongju, South Korea, June 2011, pp. 31–35

[29] Kung, M., Au, O., Wong, P., *et al.*: 'Block based parallel motion estimation using programmable graphic hardware'. Proc. ICALIP, Shanghai, China, July 2008, pp. 599–603. DOI: 10.1109/ICALIP.2008.4590176

[30] Lee, D.K., Oh, S.J.: 'Variable block size motion estimation implementation on compute unified device architecture (CUDA)'. Proc. IEEE Int. Conf. Consumer Electronics, Las Vegas, NV, USA, 2013, pp. 633–634

[31] Radicke, S., Hahn, J., Grecos, C., *et al.*: 'A highly parallel approach on motion estimation for high efficiency video coding (HEVC)'. Proc. IEEE Int. Conf. Consumer Electron, Las Vegas, NV, USA, January 2014, pp. 187–188

[32] Fan, W., Dajiang, Z., Satoshi, G.: 'OpenCL based high-quality HEVC motion estimation on GPU'. Proc. IEEE Int. Conf. Image Processing (ICIP), Paris, France, October 2014, pp. 1263–1267

[33] Khemiri, R., Kibeya, H., Sayadi, F.E., *et al.*: 'Optimization of HEVC motion estimation exploiting SAD and SSD GPU based implementation', *IET Image Process.*, 2018, **12**, (2), pp. 243–253, doi: 10.1049/iet-ipr.2017.0474