Article

On Combining Wavefront and Tile Parallelism with a Novel GPU-Friendly Fast Search

Georgios I. Papaioannou [1], **Maria Koziri** [2], **Thanasis Loukopoulos** [1,*] **and Ioannis Anagnostopoulos** [1,*]

[1] Department of Computer Science and Biomedical Informatics, University of Thessaly, 35131 Lamia, Greece; geopapaioannou@uth.gr

[2] Department of Informatics and Telecommunications, University of Thessaly, 35131 Lamia, Greece; mkoziri@uth.gr

* Correspondence: luke@dib.uth.gr (T.L.); janag@dib.uth.gr (I.A.)

Abstract: As the necessity of supporting ever-increasing demands in video resolution leads to new video coding standards, the challenge of harnessing their computational overhead becomes important. Such overhead stems not only from the increased image data due to higher resolutions but also from the coding techniques per se that are introduced by each standard to improve compression. All modern standards in the field of video coding offer high compression efficiency, but this is achieved by increasing the computational complexity of the encoding part. Ultra-High-Definition (UHD) videos, bring new encoding implementation schemes that are being recommended for CPU and GPU parallelization. Therefore, several works are published to achieve better performance and reduce encoding complexity. Following this idea, we proposed and evaluated a hybrid encoding scheme that utilizes the constant growth of the CPU power with the massive GPU popularity in parallel. Taking advantage of the encoding schemes from the leading video coding standards, such as High-Efficiency Video Coding (HEVC) and Versatile Video Coding (VVC), which support parallel processing thru Wavefront or Tiling, in our work, we combined both of them at the same time as a whole, and in addition, we introduced a GPU-friendly fast search algorithm that is highly parallel and alternative to the default non-parallel TZ-Search. Through an experimental evaluation with common test sequences, the proposed GPU Fast Motion Estimation with our previous Wavefront per Tile Parallelism (WTP) was shown to provide valid trade-off between speedup and video coding efficiency, effectively combining the best of two worlds, i.e., WTP using CPUs and parallel Motion Estimation with GPUs.

Keywords: video coding; tile parallelism; wavefront parallelism; HEVC; VCC; WTP; motion estimation; GPU

check for updates

Citation: Papaioannou, G.I.; Koziri, M.; Loukopoulos, T.; Anagnostopoulos, I. On Combining Wavefront and Tile Parallelism with a Novel GPU-Friendly Fast Search. *Electronics* **2023**, *12*, 2223. https://doi.org/10.3390/electronics12102223

Academic Editor: Stefanos Kollias

Received: 13 April 2023
Revised: 10 May 2023
Accepted: 11 May 2023
Published: 13 May 2023

1. Introduction

Video encoding is a computationally demanding process that requires a well-designed and developed algorithm. An optimal solution requires a trade-off between quality and processing time. Today, most video standards focus on two goals: to improve compression performance and implement parallel architectures to minimize encoding time and at the same time, achieve better quality for higher video resolutions. Two of the most popular video standards, HEVC and the most recent VCC, are sharing three parallelization approaches. The first one included is Wavefront Parallel Processing (WPP). WPP allows for the creation of picture partitions that can be processed in parallel without incurring high coding losses. Rows of coding tree units (CTUs) are processed in parallel while preserving all coding dependencies. The processing pattern begins from the top left corner and progressively goes to the right bottom. In any case, the current CTU requires that top-left, left, top, and top-right CTUs are available to continue its encoding. For that reason, a shift of at least two CTUs is enforced between consecutive rows processed in parallel (Figure 1).

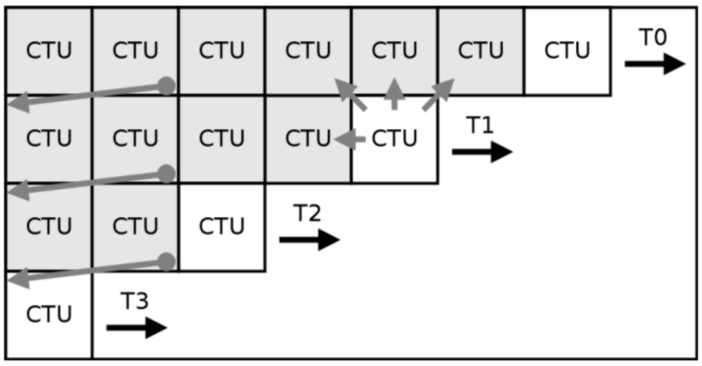


Figure 1. Principle of WPP and CTU dependencies.

Tiles on the other side divide a picture into independent, rectangular regions, enabling improved coding efficiency for parallel architectures. Tiles can be thought of as rectangular regions formed by the intersection of rows and columns (Figure 2). Tiles can have uniform spacing in a row and column boundary specification, or not. Moreover, Tiles share the same boundaries with CTUs and they yield a high pixel correlation. A similar partition scheme to parallelize the encoding or decoding process is Slices. Similar to Tiles, they are also independent pixel blocks so they can be coded separately (Figure 3).

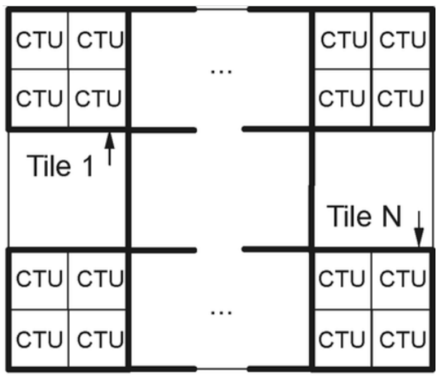


Figure 2. Principle of Tiles parallelism.

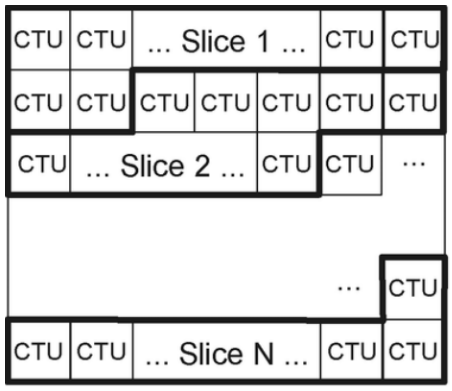


Figure 3. Principle of Slice parallelism.

Generally, tile partitioning is considered to be more flexible than slicing because Slices include headers that produce extra overhead that harms the coding efficiency. Therefore, in the case of multiple CPU threads that serve multiple slices (one thread corresponds to one slice), it may lead to non-negligible coding losses. On the other side, if we decrease the number of slices, we also decrease the performance. Furthermore, both video standards do not allow the co-existence of a Tiles and WPP parallelization scheme at the same time, by design [1]. We achieved a bypass of this restriction with our work [2] proposing a new multi-threaded encoding scheme that is based on Tile partitioning and per tile Wavefront parallelism (WTP) (Figure 4).

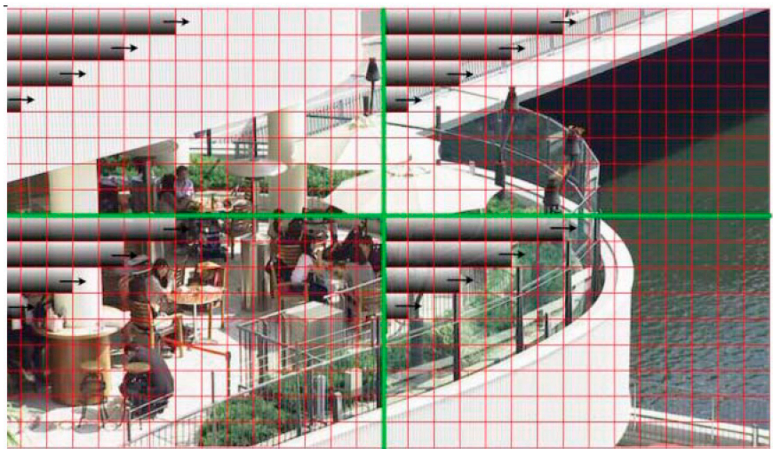


Figure 4. Example of Wavefront per Tile parallelism (WTP), 2 × 2 Tiles, 16 Threads.

In this case, the primary encoding scheme used is Tiles, but internally, our modified encoder enables a custom multi-threading Wavefront encoding each tile separately. To comply with the original output-encoded streaming, we had to propagate the original HEVC structures in a thread-safe environment. The frame is divided into independent Tiles that have the same size. Each tile has its own thread-safe WPP data structures (entropy coders, search & prediction, quantization & transform) (Figure 5), and they are treated as separate frames but with lower resolution in CTU metrics. Tiles essentially allow for a typical multithreaded WPP-encoding scheme to be used. For each one of those Tiles, an independent WPP is used for the encoding procedure (Figure 6) consisting of a new encoding parallelism scheme (WTP).

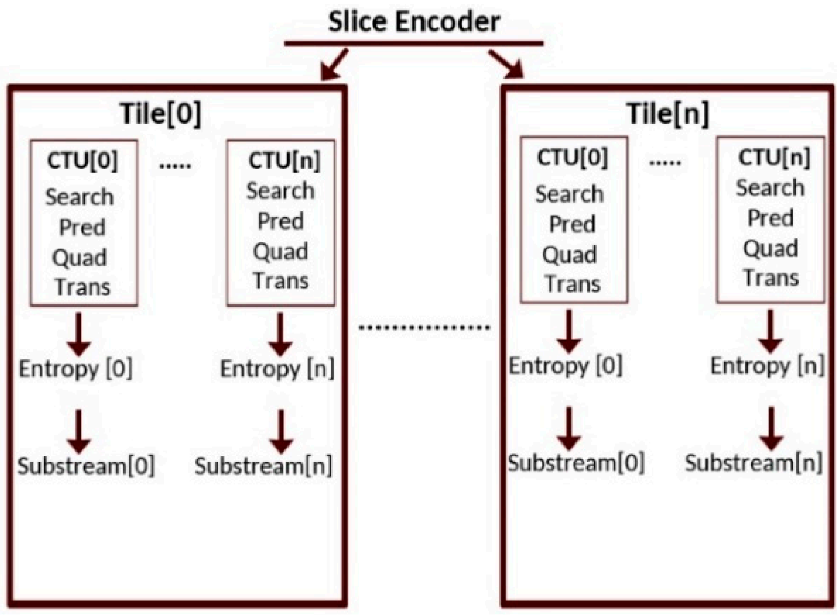


Figure 5. Wavefront thread-safe structures for each tile.

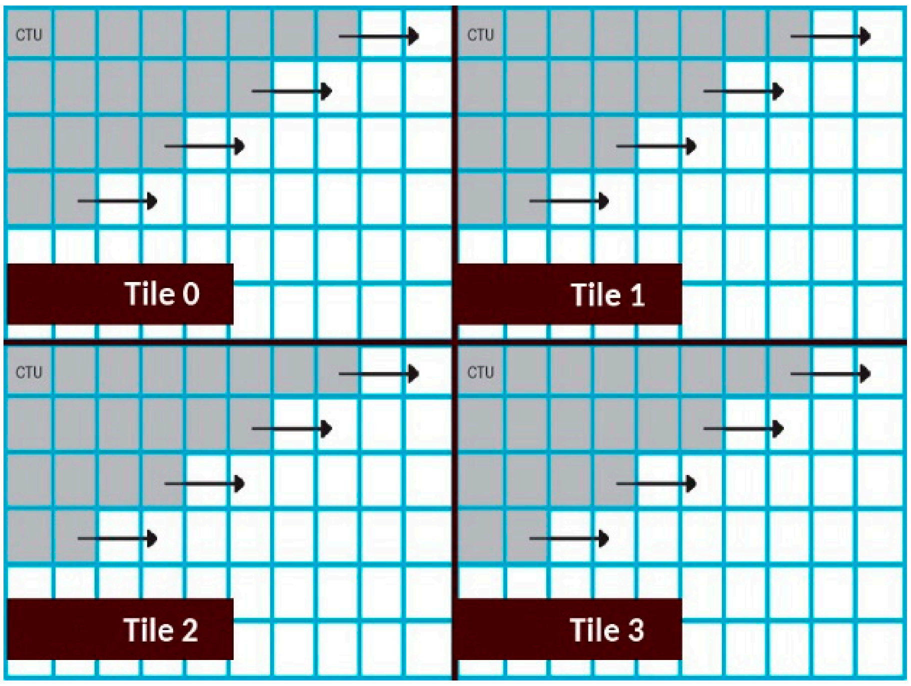


Figure 6. Wavefront encoding inside Tiles.

Furthermore, besides the multi-threading role contribution, in any modern video standard, coding efficiency has a huge impact that relies on the most critical part of the encoding process, which is the search for the best motion vectors and matching blocks. In this paper, we propose a novel parametrized GPU-friendly Fast Search algorithm that can further improve the performance in such an environment. We evaluated this algorithm with our proven WTP multi-threaded encoding scheme that is based on the HEVC video standard. It seems that its contribution to the encoding processing time is considerable and it can be maximized when the system has weaker CPU process power than the GPU itself. In addition, this fast search algorithm is parameterized for its leaping search steps, which means that we can feed more steps when more powerful GPUs are installed in the system, to achieve even more accurate results.

2. Related Work

Motion vector calculations are the most intensive part of any modern video encoder. Therefore, several works are published to minimize that cost, including software and hardware implementations.

Software implementations include CPU-only powered processors. Small diamond pattern search, large diamond pattern search, and horizontal diamond search [3,4] are some of them that are proposed to replace the default TZ search algorithm. Experiments show that 49% time saving could be achieved with a minor decrease in the PSNR ratio. Another implementation is the 6-point hexagon that reduces the motion estimation encoding time compared with the TZ search, by about 50% [5,6]. Xiong et al. [7] proposed a fast-inter-CU decision to reduce the encoding time, and Khemiri et al. [8] proposed a fast ME algorithm that is based on a diamond search pattern but introduced three mode decision levels. The time saving was about 57% with a minor PSNR decrease.

Hardware implementations include non-CPU integrated circuits, such as Field Programmable Gate Arrays (FPGAs), Application Specific Integrated Circuits (ASICs), and video cards. The most popular non-CPU hardware is video cards, which most of them nowadays include graphics processor units (GPUs) that are programmable for general-purpose calculations. Parallel SAD calculations using an FPGA Xilinx [9] are used to achieve $\times 3.9$ speedup in comparison to the baseline (single core or non-parallel) HEVC algorithm. Moreover, F.H. Shajin et al. [10] proposed a fast search quadrant-based algorithm that obtains better block matching, using an FPGA hardware platform. According to their experiments, they had 8.3562% better PSNR compared with the HEVC diamond search and 0.131% better performance for single-core executions. In addition, hybrid or heterogeneous algorithms were introduced that use both CPUs and the extra hardware (FPGAs, ASICs, or GPUs) [11–15], i.e., CPUs for coarse searching and GPUs for fine searching with analogous results. An average time saving for the latest is about 19% for single cores and $\times$10.75 when multi-cores are used.

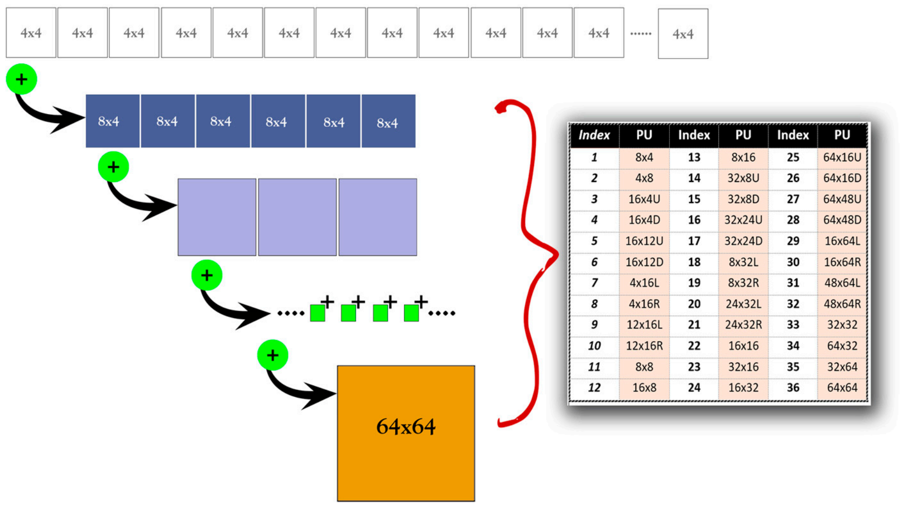
On the other hand, video cards are far more common hardware among users. A high parallel 5-step GPU ME algorithm presented by Chen and Hang [16] reduced the encoding time for the motion estimation by about $\times$12 compared with the baseline alternative, but it concerns the previous standard H.264/AVC, as well as the work by Rodriguez and Martinez [17], where they achieved a speedup of $\times$2 compared with the baseline without any significant PSNR loss. An interesting GPU-based fast motion implementation is the adaptive search range (ASR), proposed by Kim et al. [18] for the HEVC. The idea was to adaptively change the search area every time, to reduce the number of parallel calculations inside the GPU. The results showed a reduction in the encoding time of about 40.3% and an RD loss of 1.2% compared with the single core competitive. Jiang et al. [19] implemented the integer ME and fractional ME on the GPU using a full search algorithm and obtained a time reduction of up to 53% compared with non-parallel baseline competition. The new NVIDIA's Fermi GPU architecture was used by Klemiri et al. [20] to improve the SAD calculations, reducing the encoding time by about 56.17% without any significant loss. Gogoi et al. [21] applied a low-complexity IME algorithm using two different pattern structures (PS) with 38 search points, resulting in 8.725% and 9.072% time savings, respectively, in HM 16.8. A novel GPU full search multilevel resolution sample area (MLRME) was proposed by Xue et al. [22] that creates several down-sampling copies of the searching area to reduce the number of calculations. Experiments showed that the encoding quality declined by less than 1.5%, and a 30–60$\times$ speedup was achieved compared with on single-core CPU. A parallel motion estimation implementation was proposed by Zhang et al. [23], which includes pre-motion estimation, integer motion estimation, and fractional motion estimation. A rapid mapping table algorithm was introduced to improve the efficiency of data access, in addition to a quasi-integral-graph algorithm, which is designed to calculate SAD or SATD efficiently for blocks of different sizes thru GPU. A maximum of 12.7% was the encoding gain for this scheme compared with the single-core sequential code. Finally, a GPU-based low delay ME parallelization scheme was proposed by Luo et al. [24] that hierarchically uses a GPU parallelization by optimizing the ME process in a coding tree unit (CTU), prediction unit (PU), and motion vector (MV) layers. Experimental results demonstrate that the proposed scheme can achieve 41% encoding time savings with ME acceleration up to 12.7 times, and the incurred BD-BR loss is only 0.52% on average. Our work is different from the latter approach in many ways. (a) Best MVs have zero assumptions, so there are no dependencies and are all calculated at the runtime, using a z-order scheme using the full search range at the beginning (1st phase) and progressively reducing the search area, but leaving constant the searching points for all phases. (b) Our parallelization scheme optimizes only the IME part as a solid GPU block, giving better absolute speedup times and leaving space for further optimizations for the FME part. (c) Our core searching algorithm is versatile and easily parametrized to use any number of phases, search area boundaries, and searching points for each phase, according to the power of the GPU and accuracy we prefer. These parameters are passed from Host to Kernel at the runtime.

The main contribution of our work is to engage the existing hardware without any extra cost and to maximize the overall performance, especially when the system has weaker CPU process power than the GPU itself on specific topics. This is accomplished using an ultra-fast parametrized search algorithm in a parallel pattern, inside the video card hardware. Furthermore, by combining the GPU contribution to this work, with our previous parallelization scheme (WTP) running exclusively on CPUs, we propose an all-in-one encoding scheme with optimistic experiment results. Taking into account that exceptional video cards are nowadays very cost-effective and the fact that our fast search algorithm can be parametrized to delegate with more fine-grained computations in ultra-high definition resolutions, it is imperative to exploit the existing hardware to improve the computational complexity of the encoding part.

3. Implementation

For our WTP [2] encoding scheme, a custom thread pool is used to manage an arbitrary number of threads. Threads are typically set according to the number of available cores, but any other number can be used. The thread pool examines, in a loop, a reference table map of the available jobs for each virtual tile created, either sequentially or randomly according to an inline parameter. The first free thread is assigned to start the encoding process for the next available CTU for the tile selected. The way each thread is administrated from the application is also an important factor, and two main options are available to be selected: (a) to tie every single thread to each CTU row, even keeping them in a wait state cycle when they have to wait for dependency CTUs and release them back to the thread pool only when the row is fully encoded, or (b) to release them immediately when the current job is completed (for example it cannot encode any more CTUs for the current row because neighbor dependencies are not ready) and re-schedule them for the next available job. While in [2] the first option was selected, in this work, threads are released as soon as the current job is completed and re-scheduled to keep GPU cores busy as much as possible [25].

In the proposed algorithm, threads are selected to comply with physical CPU cores to achieve the maximum performance. If more CPUs are available, the speedup will automatically scale [26] with respect to the available video resolution and the GPU capabilities [27]. Extending the WTP encoding scheme, we added the power of the GPUs in the encoding process [28,29], exclusively for the motion estimation part, which is the most time-consuming task in a video encoder. In the HEVC case, motion estimation requires about 70% of the total computation time [20]. For this extension, we used the OpenCL framework [30,31] to set concurrent GPU kernels to compute the calculations required for the integer motion estimation part (IME). OpenCL was selected because it is a universal framework for GPGPU programming that supports most of the major GPU manufacturers and because it achieves better performance compared with other frameworks [32]. Our novel GPU fast search algorithm, named "4PhaseFS", is implemented as a high-parallel alternative to other non-parallel search methods. Despite the heavy parallelization, only one CPU thread can be used at a time. Mixing our high parallel WTP encoding scheme, which is CPU-based, with GPU for the IME part [33], we tried to balance the encoding load between CPUs and GPU. The GPU is charged to calculate the distortion ratio using a bottom-up accumulative scheme starting with 4×4 blocks (256 in total) as a base for all successive SAD (Sum of Absolute Difference) calculations that are progressively reaching the final 64×64 block. A parallel reduction method [34,35], as shown in Figure 7, is used to speed up this process using 16×16 work groups, which means 256 threads in parallel are executing the GPU program (device kernel). Based on the previous step, we calculate the SADs of all enumerated CU and PU possible modes (i.e., 4×8, 8×4, 16×4, 4×16, etc.) inside a CTU, as shown in Figure 8, building, this way, an index array of 593 SAD elements in total, as shown in Figure 9. The minimum distortion ratios from the previous processing step, which concerns the whole search area, are the motion vectors selected for the next step in the Host program (CPU). Memory optimization strategies are used to confine the data transfers between the host and the kernel [36].



Index	PU	Index	PU	Index	PU
1	8x4	13	8x16	25	64x16U
2	4x8	14	32x8U	26	64x16D
3	16x4U	15	32x8D	27	64x48U
4	16x4D	16	32x24U	28	64x48D
5	16x12U	17	32x24D	29	16x64L
6	16x12D	18	8x32L	30	16x64R
7	4x16L	19	8x32R	31	48x64L
8	4x16R	20	24x32L	32	48x64R
9	12x16L	21	24x32R	33	32x32
10	12x16R	22	16x16	34	64x32
11	8x8	23	32x16	35	32x64
12	16x8	24	16x32	36	64x64

Figure 7. A parallel reduction scheme is used to calculate all blocks from the bottom up. Thirty-six PU block reduction calculations are executed in total, excluding the initial 4 × 4.

The minimum estimated RD-ratio is defined from Equation (1), which is also computed inside the kernel (GPU program created with a modified C programming language).

$$J_{MV} = SAD_{MV} + \lambda R_{MV} \tag{1}$$

where J_{MV} is the total cost, SAD_{MV} is the SAD of the specific block inside the index array as described in Figure 8, λ is the Lagrangian multiplier, and RMV is the total number of bits required to encode the specific motion vector.

"4Phase" fast search is executed completely inside GPU cores, calculating the neighboring area of the current encoding CTU with a reduced step in every cycle (phases). That is true for both the searching area and the number of selected CTUs (leap step) for the calculations described above.

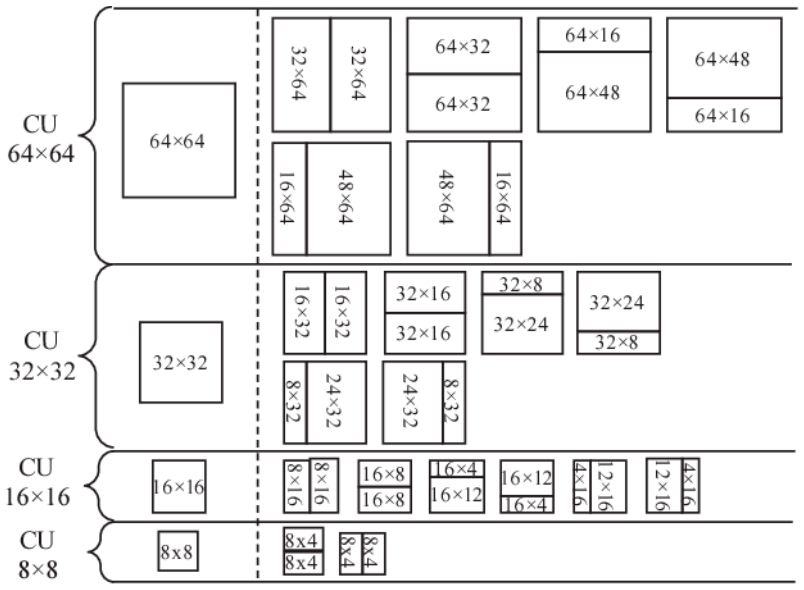


Figure 8. Enumeration of all CU and PU modes in a CTU.

The 1st phase is the initial coarse search, starting from the top left side of the initial 128 × 128 searching area, and uses 16-leap steps (meaning how many CTUs to skip before the next CTU can proceed) in a top-down z-order pattern. The CTU block that has the minimum SAD (sum of absolute differences) from each phase becomes the center for the next phase. From the 3rd phase and after, every new search area, as well as every leaping step, becomes half of the previous phase, meaning that a progressively fine-grained search is made. Following this pattern, starting the fine-grained search from the 2nd phase, it covers an area of 32 × 32 CTUs around the center found at phase 1, with 4-leap-steps, 16 × 16 for the 3rd phase with 2-leap-steps, and finally, an 8 × 8 area for the 4th phase with 1-leap-step (tests every CTU around the center), as indicated in Figure 9. The CTU was chosen from phase 4 of the GPU Kernel as the best candidate that the Host gets back to continue its process. The comparisons that are needed for the 4Phase fast search algorithm are always 256 in total (64 comparisons for each phase). Even though the 4Phase fast search always performs 256 comparisons (with the appropriate calculations) to obtain the motion vectors for the minimum SAD, this job is done ultra-fast by the GPU and it takes about 40% of the total time that the algorithm needs. The remaining 60% of the time is the data bus transfers overhead from the CPU to GPU and vice versa.

This means that an integrated GPU, or the next PCIe generations, can achieve better speedups, as well as better quality, since the 4PhaseFS algorithm can be adapted to be more accurate in its computations since there will be more headroom time to increase the searching area with detailer leap steps for each phase.

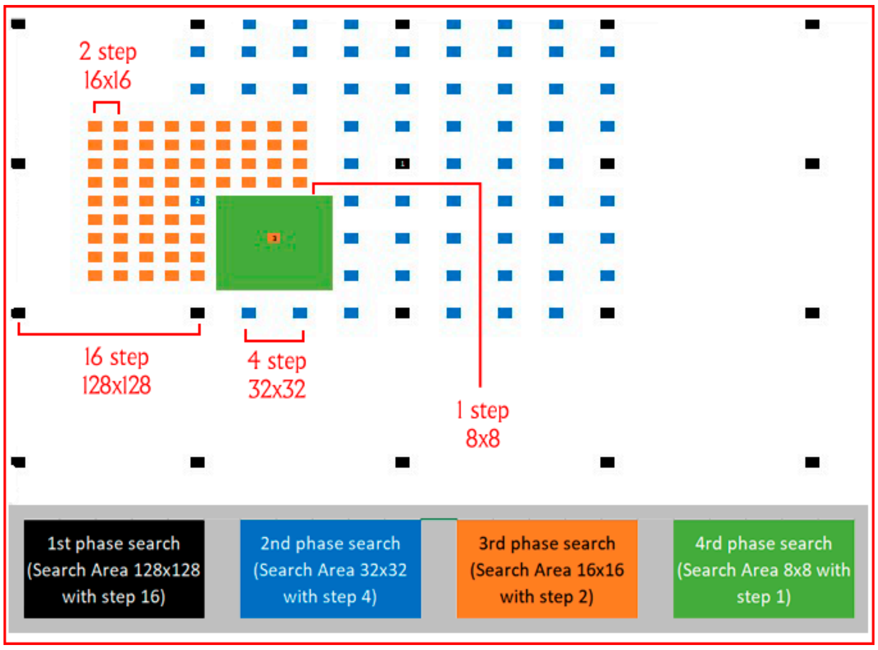
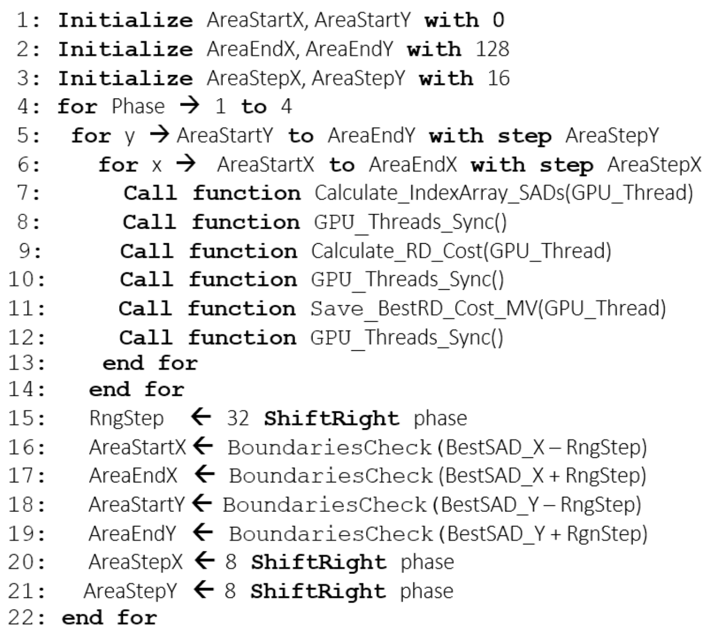


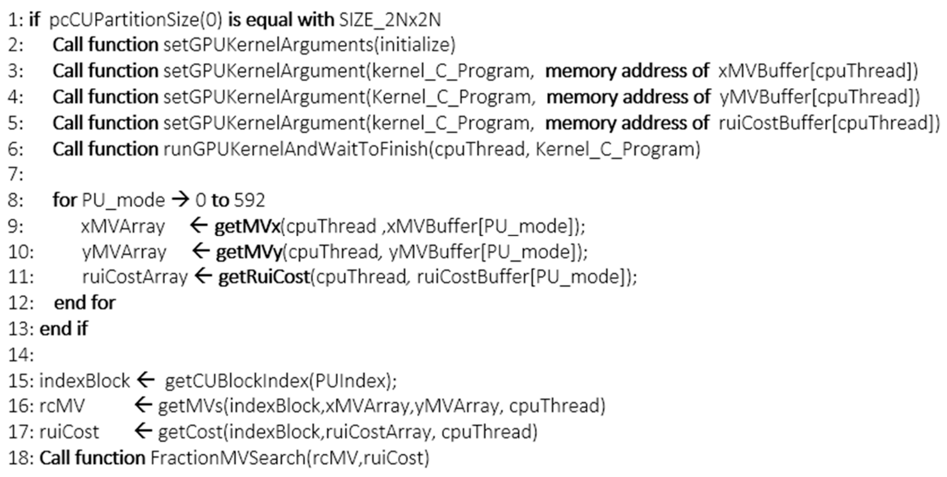
Figure 9. Visualization of the four phases. Steps leaps are expressed to video pixels either horizontally or vertically inside the searching area.

To summarize, the GPU program, called Kernel, executes two main functions for each position in the search area, which normally is double the CTU size. *Function A* calculates the SAD for each possible CU/PU block in the CTU to construct a new SAD Array and *Function B* calculates the rate–distortion ratio (RD-ratio) for each block and selects every time, the minimum cost, according to the SAD calculation history of Function A inside 593 data array elements. For the minimum cost calculated before, we also calculate and store in this function, the counterpart motion vector coordinates, which are 593 integer data arrays for X and Y parts accordingly. The resulting arrays of (a) ruiCost and (b) motion vectors are sent back to the Host (kernel caller) for further processing. The simplified algorithm of the kernel program (Figure 10a) and the way the host gets back the calculation results are shown below (Figure 10b).



```
1:  Initialize AreaStartX, AreaStartY with 0
2:  Initialize AreaEndX, AreaEndY with 128
3:  Initialize AreaStepX, AreaStepY with 16
4:  for Phase → 1 to 4
5:   for y → AreaStartY to AreaEndY with step AreaStepY
6:     for x → AreaStartX to AreaEndX with step AreaStepX
7:       Call function Calculate_IndexArray_SADs(GPU_Thread)
8:       Call function GPU_Threads_Sync()
9:       Call function Calculate_RD_Cost(GPU_Thread)
10:      Call function GPU_Threads_Sync()
11:      Call function Save_BestRD_Cost_MV(GPU_Thread)
12:      Call function GPU_Threads_Sync()
13:    end for
14:   end for
15:   RngStep    ← 32 ShiftRight phase
16:   AreaStartX ← BoundariesCheck (BestSAD_X – RngStep)
17:   AreaEndX   ← BoundariesCheck (BestSAD_X + RngStep)
18:   AreaStartY ← BoundariesCheck (BestSAD_Y – RngStep)
19:   AreaEndY   ← BoundariesCheck (BestSAD_Y + RgnStep)
20:   AreaStepX  ← 8 ShiftRight phase
21:   AreaStepY  ← 8 ShiftRight phase
22: end for
```

(**a**)



```
1: if pcCUPartitionSize(0) is equal with SIZE_2Nx2N
2:    Call function setGPUKernelArguments(initialize)
3:    Call function setGPUKernelArgument(kernel_C_Program, memory address of xMVBuffer[cpuThread])
4:    Call function setGPUKernelArgument(Kernel_C_Program, memory address of yMVBuffer[cpuThread])
5:    Call function setGPUKernelArgument(kernel_C_Program, memory address of ruiCostBuffer[cpuThread])
6:    Call function runGPUKernelAndWaitToFinish(cpuThread, Kernel_C_Program)
7:
8:    for PU_mode → 0 to 592
9:       xMVArray    ← getMVx(cpuThread ,xMVBuffer[PU_mode]);
10:      yMVArray    ← getMVy(cpuThread, yMVBuffer[PU_mode]);
11:      ruiCostArray ← getRuiCost(cpuThread, ruiCostBuffer[PU_mode]);
12:   end for
13: end if
14:
15: indexBlock ← getCUBlockIndex(PUIndex);
16: rcMV       ← getMVs(indexBlock,xMVArray,yMVArray, cpuThread)
17: ruiCost    ← getCost(indexBlock,ruiCostArray, cpuThread)
18: Call function FractionMVSearch(rcMV,ruiCost)
```

(**b**)

Figure 10. (**a**) The simplified GPU Kernel program with major steps executed. (**b**) The way the host calls the GPU kernel and gets back the MVs and ruiCost for further processing.

The first part of Figure 10, Lines 1–3, are the initialization of the variables that are used in the inner loops below. We need *AreaStartX*, *AreaStartY*, *AreaEndX*, and *AreaEndY* to avoid boundary errors (below zero or out of the 128 × 128 search area). Line 4 is the counter of the phases and lines 5–14 calculate the SAD and the Rate-Distortion Cost from the current CTU pointed from the x and y loop variables. The *Calculate_IndexArray_SADs* function fills the index array described in Table 1. This is performed in parallel from the available GPU cores and independently from each other. To calculate the RD costs for all 593 SAD elements, we must synchronize all GPU cores, which means that the Kernel should wait until all of them complete their calculations. At line 11, we save the best SAD Motion Vector for each one of the indexes, as well as the Cost. Lines 15–21 set the new search area, as well as the searching leap steps. The second part of Figure 10, Lines 1–15, are the GPU Kernel calling process from the Host side. This part is called only once, i.e., every new 64 × 64 CTU block is met. When the Kernel returns, we copy all calculated MVs and the ruiCost from the GPU memory back to Host data structures. From now on, for every PU mode pattern we receive from the encoder, we only search inside the Host's data structures to obtain the pre-calculated MV and ruiCost for further processing (lines 17–20).

Table 1. Index array of the SAD.

Index	PU	Index	PU	Index	PU
0–127	8 × 4	480–511	8 × 16	576	64 × 16U
128–255	4 × 8	512–515	32 × 8U	577	64 × 16D
256–271	16 × 4U	516–519	32 × 8D	578	64 × 48U
272–287	16 × 4D	520–523	32 × 24U	579	64 × 48D
288–303	16 × 12U	524–527	32 × 24D	580	16 × 64L
304–319	16 × 12D	528–531	8 × 32L	581	16 × 64R
320–335	4 × 16L	532–535	8 × 32R	582	48 × 64L
336–351	4 × 16R	536–539	24 × 32L	583	48 × 64R
352–367	12 × 16L	540–543	24 × 32R	584–587	32 × 32
368–383	12 × 16R	544–559	16 × 16	588–589	64 × 32
384–447	8 × 8	560–567	32 × 16	590–591	32 × 64
448–479	16 × 8	568–575	16 × 32	592	64 × 64

The costliest part for the GPU calculations is the data transfer from the Device (Video Card) to the Host (CPU) and vice versa because of the PCIe bus, which is much slower than CPU registers and the Host main memory data bus. For that reason, to minimize the transfer data costs, the kernel (GPU program) is implemented as a single program that executes both the A and B functions described above, at once, without having to return partial results to the host. When the 4-phase search cycle is completed, then and only then, the host receives back the final minimum RD cost array with respective MVs [36]. From this point so far, the CPU thread continues with the next step.

Each CPU thread is calling its Device kernel (GPU), which means that the Device executes multiple kernels concurrently, as shown in Figure 11. This is a new feature for the Video Cards manufactured in 2013 and after i.e., Nvidia's Fermi architecture (CUDA API version 3.0) [20]. The workhorse of the GPU is the streaming multiprocessor, or simply SM in the Nvidia world. In the OpenCL jargon, they are called Compute Units. The number of Compute Units may range from two to several dozen, and each Compute Unit contains the following: (1) execution units to perform 32-bit integer and single or double-precision floating-point arithmetic; (2) special function units (SFUs) to compute single-precision approximations of log/exp, sin/cos, and rcp/rsqrt; (3) a work-group scheduler to coordinate instruction dispatch to the execution units; (4) a constant cache to broadcast data to the Compute Units; (5) shared memory for data interchange between threads; and (6) dedicated hardware for texture mapping.

CUDA streams are used to manage concurrency between execution units with coarse granularity. This feature is implemented in the OpenCL framework with multiple queues. Therefore, each CPU thread uses a different queue to execute the same kernel but with different data. If the Compute Units are all busy, then the execution of the next kernel takes on a wait state until one becomes free. Thus, separate GPU streams are executing concurrently, and the operations requested in each queue are performed sequentially; in a sense, OpenCL queues are like CPU threads where operations are performed in order. Even when multiple queues are not used, keeping all operations asynchronous (i.e., out-of-order execution with only one queue) improves performance by enabling the CPU and GPU to run concurrently by letting the video card drivers automatically decide the best strategy for the load distribution. This method is working asynchronously, which means that a later encoding CPU thread may be serviced sooner than a previous one, so a synchronization engine should be used to keep track of the encoding process of the WTP. An arbitrary number of CPU threads request the same number of Video Card Compute Units to calculate the RD cost of the IME part. According to the capabilities of the Video Card being used, i.e., number of concurrent Compute Units, local memory size, L1 and L2 cache sizes, etc., the right balance must be chosen between the CPU's physical cores and the video card's Compute Units, to get the best results from each world combined. Increasing CPU threads to a number that exceeds the maximum Compute Units of the Video card does not offer any important gain to the encoding speedup since the CPU threads are obtaining a wait state. In addition, kernels with a very large number of threads or increased demands of local memory size (most GPUs today give about 64 KB of local memory for each Compute Unit) may need two or more Compute Units to execute the same code, which means that the concurrency level is dropping down. The Kernel for this extension is written carefully to avoid data waste and to be as compact as possible. Moreover, the code is optimized to avoid raw multiplications and division program statements and is replaced with left and right shifting statement commands, respectively, which are far faster.

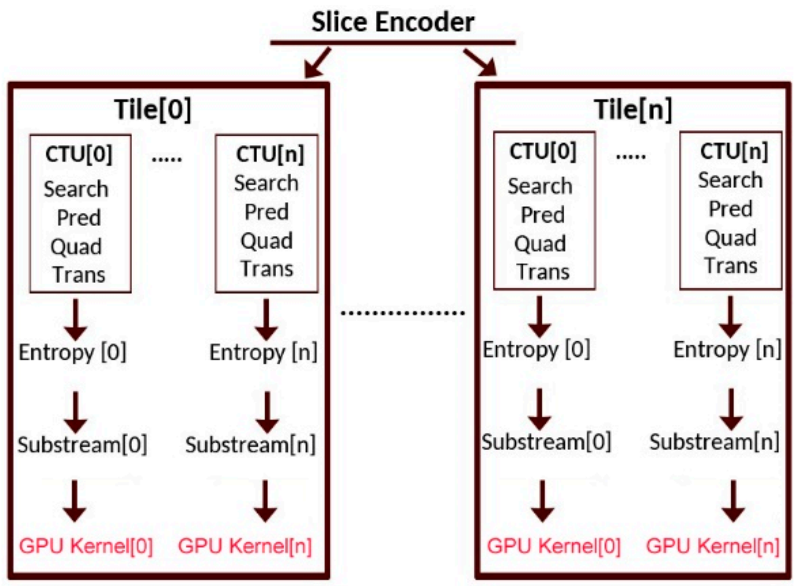


Figure 11. WTP encoding with GPU.

4. Experiments

4.1. Implementation Details

We implemented GPU Fast Search in HM 16.17 reference software using a custom C++ code for multi-threaded parallelization and with AMP (Asymmetric Motion Partitioning) enabled, a feature that was not used in our previous benchmark publication [2]. All algorithms were implemented using a thread pool with a configurable number of threads. In addition, in this implementation, we have changed the thread assignment strategy and routing. Instead of preserving the CPU thread until it finishes the whole encoding CTU line, obtaining a wait state cycle in case the neighbor dependencies are not ready—this is a WPP restriction; in this implementation, we free the CPU thread immediately as soon as it meets the above condition paying, of course, the thread scheduling initialization cost. We selected this approach to keep the GPU's core load as high as possible.

4.2. Setup

We conducted experiments on a Linux Server with two 12-core Intel Xeon E5-2650s running at 2.20 GHz as the base frequency and 2.90 GHz for turbo mode and 1.20 GHz for power save mode. The Graphics Card installed on our chain was an Nvidia Quadro P4000 with 8 GB of GPU Memory that combines 1792 CUDA Cores (drivers version 470.141.03). The Pascal architecture of the Video Card was introduced on April 2016 and uses the PCI

Express 3.0 × 16 system interface. To make experimentation time manageable we used, for our evaluation, the first 150 frames of Class A and B common test sequences [37] listed in Table 2. Results were obtained from Low Delay (LD) set up with an initial I frame followed by B frames and a GOP size of 4. Bit depth was 8, CTU size was 64 × 64, max partitioning depth was 4, and the search mode was TZ. As mentioned above, AMP (Asymmetric Motion Partitioning) was enabled. One slice was used. Experiments with WTP involved 24 threads (the number of physical cores available in our server), with tile partitions of 2 × 2, 3 × 3, and 6 × 4, respectively.

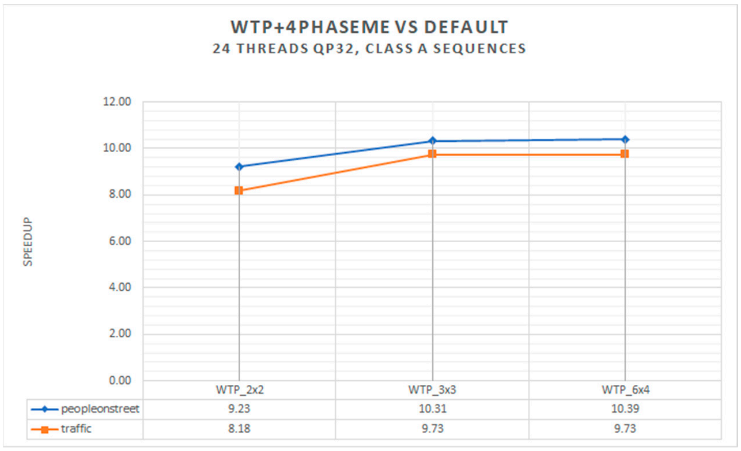
Table 2. Test video sequences.

Class	Sequence Name	Resolution
B	Basketballdrive	1920 × 1080 50 fps
B	Bqterrace	1920 × 1080 60 fps
B	Cactus	1920 × 1080 50 fps
B	Kimono	1920 × 1080 24 fps
B	Parkscene	1920 × 1080 24 fps
A	Peopleonstreet	2560 × 1600 30 fps
A	Traffic	2560 × 1600 30 fps

4.3. Coding Efficiency Results

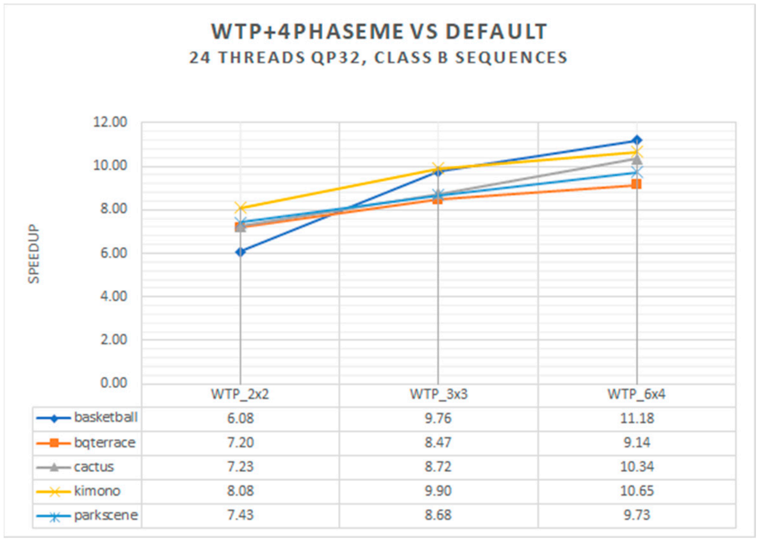
Next, we evaluated the compression efficiency and video quality of the GPU-accelerated WTP with the original WTP and the default HEVC TZSearch. We considered three tile partitions for WTPs, namely 2 × 2, 3 × 3, 6 × 4, and a 1 × 1 partition for the default. All algorithms run with 24 threads for QP = 22, 27, 32, and 37, except for the original TZSearch which runs with 1 thread (sequential).

Figures 12 and 13 plot the average speedup for Class A and B sequences with constant QP = 32 versus the default sequential TZSearch (1 × 1 Tile partition and 1 thread). In every case, we can observe speedups with an average rate of ×8.8 for Class B videos and ×9.6 for Class A videos. Figure 14 plots the average speedups, respectively, for Class A and B sequences with constant QP = 32 versus the original WTP (the one without GPU acceleration). In this case, the average speedup is ×1.18 or 18% of time-saving. Figure 15 plots the gain difference between the same setup but with different CPU frequency clocks, i.e., turbo mode @2.9 Ghz versus low-power mode @1.2 Ghz. Using a constant QP = 37, we can observe that for the relative average speedup between the original WTP and the GPU accelerated one, the gain is a little bit higher for the weaker CPUs than the stronger ones. These are relative comparison benchmark pairs, between the original WTP and the GPU-accelerated fast search WTP version, for each CPU frequency mode separately. Thus, while the absolute encoding time is significantly smaller for the high-frequency CPUs, the gain is smaller than the low-frequency competitive ones. Finally, in Figure 16, the BD-Rate increase is proportional to the tiling scheme used, with 2 × 2 partitioning, obtaining the lower increase from all video sequences. To capitalize on the usage of the available CPU cores, a 6 × 4 tile pattern scheme results in a 24-tile partition that leads to a relatively high bitrate increase (in Cactus it reaches 13%). This is expected since Tiles do not use the Coding Units from neighboring ones for prediction, and this results in an increase of the BD-Rate and simultaneously a PSNR decrease since this affects the CABAC (Context-Based Adaptive Binary Arithmetic Coding) learning process, as the context variables are reset at the beginning of every Tile. The BD-PSNR [38] decrease in Figure 17 is also proportional to the tiling scheme. In the worst case, we can observe a 0.24 quality drop for the kimono video and a 6 × 4 tiling scheme. BD-PSNR values represented in the plot are the average absolute PSNR experiment results for QP = 22, 27, 32, and 37 with 24 threads.



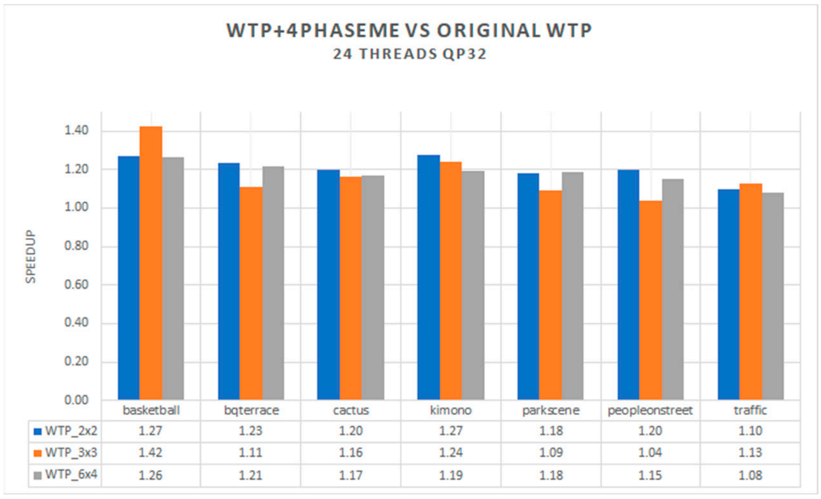
	WTP_2x2	WTP_3x3	WTP_6x4
peopleonstreet	9.23	10.31	10.39
traffic	8.18	9.73	9.73

Figure 12. WTP + 4PhaseME average speed up in Class A sequences for QP = 32 (Traffic, PeopleOnStreet) versus default TZSearch.



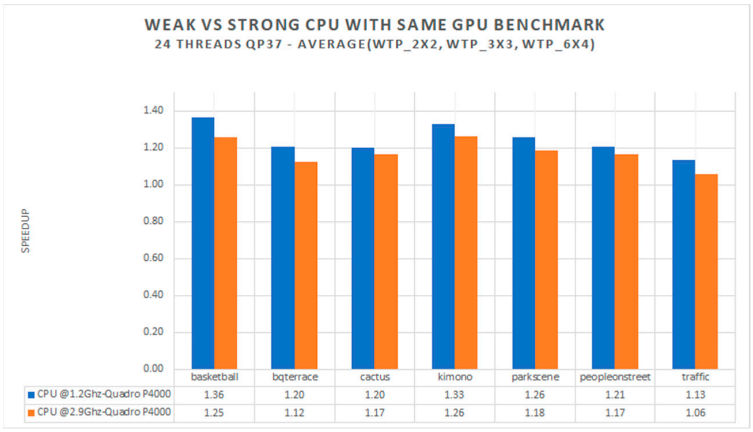
	WTP_2x2	WTP_3x3	WTP_6x4
basketball	6.08	9.76	11.18
bqterrace	7.20	8.47	9.14
cactus	7.23	8.72	10.34
kimono	8.08	9.90	10.65
parkscene	7.43	8.68	9.73

Figure 13. WTP + 4PhaseME average speed up in Class B sequences for QP = 32 (BasketballDrive, BQTerrace, Cactus, Kimono, ParkScene) versus default TZSearch.



	basketball	bqterrace	cactus	kimono	parkscene	peopleonstreet	traffic
WTP_2x2	1.27	1.23	1.20	1.27	1.18	1.20	1.10
WTP_3x3	1.42	1.11	1.16	1.24	1.09	1.04	1.13
WTP_6x4	1.26	1.21	1.17	1.19	1.18	1.15	1.08

Figure 14. Average speedup increase using WTP + 4PhaseME versus original WTP (QP = 32, AMP = true, Modified CPU threads scheduling).



	basketball	bqterrace	cactus	kimono	parkscene	peopleonstreet	traffic
■ CPU @1.2Ghz-Quadro P4000	1.36	1.20	1.20	1.33	1.26	1.21	1.13
■ CPU @2.9Ghz-Quadro P4000	1.25	1.12	1.17	1.26	1.18	1.17	1.06

Figure 15. WTP + 4PhaseME average speedup versus original WTP, using the same GPU but with different CPU frequencies for QP = 37 (average of WTP_2 × 2, WTP_3 × 3, and WTP_6 × 4, CPU-A = 1.2 Ghz and CPU-B = @2.9 Ghz).

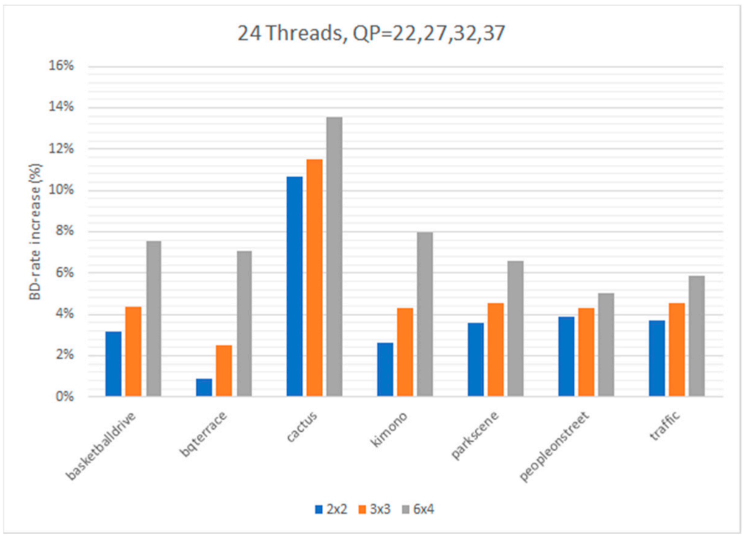


Figure 16. BD-rate percentage increase for 4PhaseMe (24 threads) versus default TZSearch (QP = 22, 27, 32, 37).

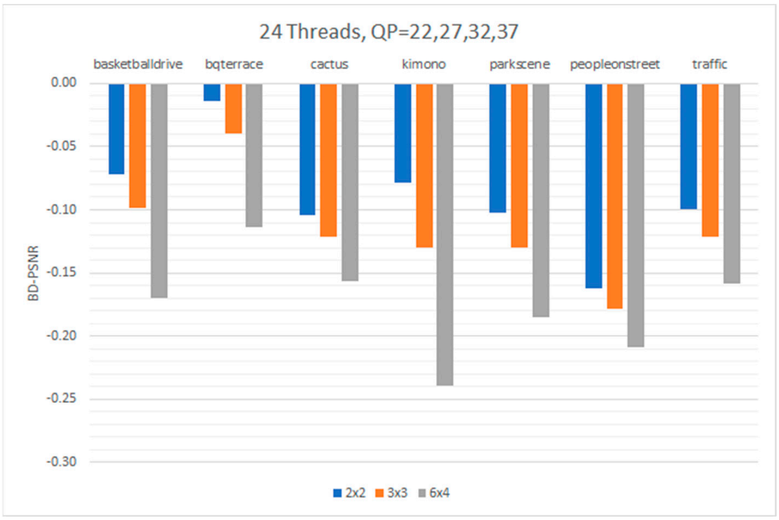


Figure 17. BD-PSNR difference for 4PhaseMe (24 threads) versus default TZSearch (QP = 22, 27, 32, 37, negative values show a quality drop).

5. Conclusions

In this paper, we present a novel GPU fast search motion estimation algorithm that works in parallel with a CPU-based multi-core encoding scheme WTP, which aims to minimize the encoding time. Results demonstrated that the 4Phase GPU fast search motion estimation algorithm can achieve an average speedup of $\times 8.84$ for Class B videos and $\times 9.60$ for Class A compared with the original sequential TZSearch. Moreover, the GPU contribution in the original WTP can increase the speed by an average of 18%. According to the results quoted above, undoubtedly, the GPU contribution is considered adequate to exploit any improvements. Using our already optimized and parallelized WTP encoding scheme, the GPU addition improved the encoding times by more than 25% in some cases, reaching 42% at its peak, when a WTP 3×3 pattern is used on a "basketball" video sequence. Class A video sequences seem to have a better average speedup than Class B sequences. Regarding the evolving GPU in the encoding process, many factors may have an impact on the final system performance, for example, PCI performance, video card resources, GPU performance when thermal throttles are met, operating system scheduling priorities, etc. It is worth mentioning from our experiments that different official video card driver versions can perform differently on identical setups.

According to the results demonstrated, it is worth highlighting that the average speedup gain is higher when CPU core frequencies are lower, enforcing the same setup but changing the scaling governor in our Linux server from performance to power saving. Scaling governors implement the algorithms to compute the desired CPU frequency, potentially based on the system's needs. Scaling drivers interact with the CPU directly, enacting the desired frequencies that the current governor are requesting. All experiments except the last mentioned in Figure 16 were executed with the default scaling governor (OnDemand), which scales the frequency dynamically according to the current load, i.e., jumps to the highest frequency and then possibly back off as the idle time increases. Despite the technical details and the way we implemented them in our work, the general idea of the searching algorithm can easily be applied to any platform, taking advantage of the newer video card innovations and add-ons, such as extensive and explicit memory (i.e., fewer data transfers from host to the kernel and vice versa) with higher bandwidth and speed clocks, more GPU cores and streaming processors (meaning more kernels are executed concurrently), and better support for concurrent kernel execution by the driver itself in newer architectures, etc. The negative effects of the BD-Rate increase and BD-PSNR decrease that are observed are more related to tiling since it affects the CABAC learning process. To reduce these effects, a balancing tile scheme is suggested, preferably a 2×2 pattern, to obtain the best of both worlds, combining speed and quality.

Cumulatively, the experimental results confirm the validity of our motivation, namely that we can benefit from the GPU computational power, especially when massive calculations take place in parallel for any video encoder that makes use of the motion estimation concept. As supreme video cards are becoming more and more cost-effective and our proposed fast search algorithm can easily be adapted and parametrized to delegate fine-grained computations in ultra-high definition resolutions, it is more than imperative to exploit the new possibilities and to improve the computational complexity. The gain is further increased when entry-level systems come into play operated by low-end CPUs. It is believed that the new-generation GPUs will boost the whole process and maximize the gain of the proposed GPU-friendly Fast Search without any extra costs.

Author Contributions: Conceptualization, G.I.P.; methodology, G.I.P. and T.L.; software, G.I.P.; validation, G.I.P., M.K., T.L. and I.A.; writing—original draft preparation, G.I.P. and T.L.; writing—review and editing, G.I.P., M.K., T.L. and I.A.; supervision, T.L.; project administration, M.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data available in a publicly accessible repository.

Acknowledgments: We acknowledge the support of this work by the project "ParICT_CENG: Enhancing ICT research infrastructure in Central Greece to enable processing of Big data from sensor stream, multimedia content, and complex mathematical modeling and simulations" (MIS 5047244), which is implemented under the Action "Reinforcement of the Research and Innovation Infrastructure", funded by the Operational Programme "Competitiveness, Entrepreneurship, and Innovation" (NSRF 2014–2020) and co-financed by Greece and the European Union (European Regional Development Fund).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Sullivan, G.J.; Ohm, J.-R.; Han, W.-J.; Wiegand, T. Overview of the High Efficiency Video Coding (HEVC) standard. *IEEE Trans. Circuits Syst. Video Technol.* **2012**, *22*, 1649–1668. [CrossRef]
2. Papaioannou, G.I.; Koziri, M.G.; Papadopoulos, P.K.; Loukopoulos, T.; Anagnostopoulos, I. Tile Based Wavefront Parallelism in HEVC. In Proceedings of the 15th International Workshop on Semantic and Social Media Adaptation and Personalization, Zakynthos, Greece, 29–30 October 2020.
3. Belghith, F.; Kibeya, H.; Loukil, H.; Ben Ayed, M.A.; Masmoudi, N. A new fast motion estimationalgorithm using fast mode decision for high-efficiency video codingstandard. *J. Real-Time Image Process.* **2014**, *11*, 675–691. [CrossRef]
4. Kibeya, H.; Belghith, F.; Ben Ayed, M.A.; Masmoudi, N. Fast coding unit selectionand motion estimation algorithm based on early detection of zero blockquantified transform coefficients for high-efficiency video coding standard. *IET Image Process.* **2016**, *10*, 371–380. [CrossRef]
5. Nalluri, P.; Alves, L.N.; Navarro, A. Improvements to TZ search motionestimation algorithm for multiview video coding. In Proceedings of the 2012 19th International Conference on Systems, Signals and Image Processing (IWSSIP), Vienna, Austria, 11–13 April 2012.
6. Nalluri, P.; Alves, L.N.; Navarro, A. Complexity reduction methods for fastmotion estimation in HEVC. *Signal Process. Image Commun.* **2015**, *39*, 280–292. [CrossRef]
7. Xiong, J.; Li, H.; Meng, F.; Wu, Q.; Ngan, K.N. Fast HEVC inter CU decision based onlatent SAD estimation. *IEEE Trans. Multimed.* **2015**, *17*, 2147–2159. [CrossRef]
8. Khemiri, R.; Bahri, N.; Belghith, F.; Sayadi, F.E.; Atri, M.; Masmoudi, N. Fast motion estimation for HEVCvideo coding. In Proceedings of the 2016 International Image Processing, Applications and Systems (IPAS), Hammamet, Tunisia, 5–7 November 2016.
9. Nalluri, P.; Alves, L.N.; Navarro, A. A novel SAD architecture for variable block size motion estimation in HEVC video coding. In Proceedings of the 2013 International Symposium on System on chip (SoC), Tampere, Finland, 23–24 October 2013.
10. Shajin, F.H.; Rajesh, P.; Raja, M.R. An Efficient VLSI Architecture for Fast Motion Estimation Exploiting Zero Motion Prejudgment Technique and a New Quadrant-Based Search Algorithm in HEVC. *Circuits Syst. Signal Process.* **2021**, *41*, 1751–1774. [CrossRef]
11. Qiu, Y.; Jiao, J.; Tang, Y.; Liu, Y.; Ren, J.; Zeng, X. A Heterogeneous HEVC Video Encoder System Based on Two-Level CPU-FPGA Computing Architecture. In Proceedings of the International Conference on ASIC (ASICON), Kunming, China, 26–29 October 2021.
12. Lee, D.; Sim, D.; Cho, K.; Oh, S. Fast motion estimation for HEVC on the graphics processing unit (GPU). *Real-Time Image Process.* **2015**, *12*, 549–562. [CrossRef]
13. Cebrian-Marquez, G.; Galiano, V.; Migallon, H.; Martinez, J.L.; Cuenca, P.; Granado, O.L. Heterogeneous CPU plus GPU approaches for HEVC. *J. Supercomput.* **2019**, *75*, 1215–1226. [CrossRef]
14. Zamanshoar, R.; Mansouri, A. Parallel Motion Estimation Based on GPU and Combined GPU-CPU. *Adv. Parallel Comput.* **2018**, *33*, 210–220.
15. Gogoi, S.; Peesapati, R. A hybrid hardware oriented motion estimation algorithm for HEVC/H.265. In Proceedings of the IEEE International Symposium on Smart Electronic Systems (iSES), Rourkela, India, 18–22 December 2019.
16. Chen, W.N.; Hang, H.M.H. 264/AVC motion estimation implementation on compute unified device architecture (CUDA). In Proceedings of the 2008 IEEE International Conference on Multimedia and Expo, Hannover, Germany, 23–26 April 2008.
17. Rodriguez, R.; Martinez, J. Reducing complexity in H.264/AVC motion estimation by using a GPU. In Proceedings of the 2011 IEEE 13th International Workshop on Multimedia Signal Processing (MMSP), Hangzhou, China, 17–19 October 2011.
18. Kim, S.; Lee, D.K.; Sohn, C.B.; Oh, S.J. Fast motion estimation for HEVC with adaptive search range decision for CPU and range. In Proceedings of the 2014 IEEE China Summit & International Conference on Signal and Information Processing (ChinaSIP), Xi'an, China, 9–13 July 2014.
19. Jiang, X.; Song, T.; Shimamoto, T.; Wang, L. High efficiency video coding (HEVC) motion estimation parallel algorithms on GPU. In Proceedings of the 2014 IEEE International Conference on Consumer Electronics-Taiwan, Taipei, Taiwan, 26–28 May 2014.
20. Khemiri, R.; Kibeya, H.; Sayadi, F.; Bahri, N. Optimisation of HEVC motion estimation exploiting SAD and SSD GPU-based implementation. *IET Image Process.* **2018**, *12*, 243–253. [CrossRef]
21. Gogoi, S.; Peesapati, R. Design and Implementation of an Efficient Multi-Pattern Motion Estimation Search Algorithm for HEVC/H.265. *IEEE Trans. Consum. Electron.* **2021**, *67*, 319–328. [CrossRef]

22. Xue, Y.; Zhang, C.; Su, H.; Ren, J.; Xiao, L. A Highly Parallel and Scalable Motion Estimation Algorithm with GPU for HEVC. *Sci. Program.* **2017**, *2017*, 1431574. [CrossRef]
23. Zhang, T.; An, X.; Zhao, X.; Gao, X. A Novel Parallel Motion Estimation Design and Implementation on GPU. *IEEE Access* **2019**, *7*, 11747–11753. [CrossRef]
24. Luo, F.; Wang, S.; Wang, S.; Zhang, X. GPU-Based Hierarchical Motion Estimation for High-Efficiency Video Coding. *IEEE Trans. Multimed.* **2019**, *21*, 851–862. [CrossRef]
25. NVIDIA. *CUDA C Programming Guide*; NVIDIA Corp: Santa Clara, CA, USA, 2014.
26. Yan, C.; Zhang, Y.; Xu, J.; Dai, F.; Zhang, J.; Dai, Q.; Wu, F. Efficient parallel framework for HEVC motion estimation on many-core processors. *IEEE Trans. Circuits Syst. Video* **2014**, *24*, 2077–2089. [CrossRef]
27. Obukhov, A. GPU-accelerated video encoding. In Proceedings of the NVIDIA GPU Technology Conference, San Jose, CA, USA, 20–23 September 2010.
28. Radicke, S.; Hahn, S.; Grecos, J.; Wang, Q. Highly-parallel HVEC motion estimation with CUDA. In Proceedings of the European Workshop on Visual Information Processing (EUVIP), Paris, France, 10–12 June 2013.
29. Ezzahra, F.; Chouchene, M.; Bahri, H.; Khemiri, R.; Atri, M. Parallel Full Search Algorithm For Motion Estimation On GPU. *Recent Adv. Electr. Electron. Eng.* **2019**, *12*, 317–323.
30. Munshi, A.; Gaster, B.; Mattson, T.; Ginsburg, D. *OpenCL Programming Guide*; Pearson Education: London, UK, 2011.
31. Wang, F.; Zhou, D.; Goto, S. OpenCL based high-quality HEVC motion estimation on GPU. In Proceedings of the 2014 IEEE International Conference, Paris, France, 27–30 October 2014; pp. 1263–1267.
32. Khemiri, R.; Bouaafia, S.; Bahba, A.; Nasr, M.; Sayadi, F.E. Performance Analysis of OpenCL and CUDA Programming Models for the High-Efficiency Video Coding. In *Digital Image Processing Applications*; IntechOpen: London, UK, 2021.
33. Gomez, A.; Perea, J.; Trujillo, M. Parallel Integer Motion Estimation for High Efficiency Video Coding (HEVC) Using OpenCL. In Proceedings of the Iberoamerican Congress on Pattern Recognition (CIARP), Lima, Peru, 8–11 November 2016.
34. Harris, M. Optimizing parallel reduction in CUDA. *NVIDIA Dev. Technol.* **2007**, *2*, 70.
35. Catanzaro, B. *OpenCL Optimization Case Study: Simple Reductions*; AMD Developer Center: Hyderabad, India, 2010.
36. Sayadi, F.E.; Chouchene, M. CUDA Memory Optimization Strategies for Motion Estimation. *IET Comput. Digit. Tech.* **2019**, *13*, 20–27. [CrossRef]
37. Bossen, F. Common Test Conditions and Software Reference Configurations. In Proceedings of the 9th Meeting of the JCT-VC, Geneva, Switzerland, 27 April–7 May 2012.
38. Bjøntegaard, G. Calculation of Average PSNR Differences between RD-Curves; Tech. Rep. VCEG-M33. In Proceedings of the ITU-T Video Coding Experts Group (VCEG) Meeting, Austin, TX, USA, 2–4 April 2001.