Hardware Deceleration of Kvazaar HEVC Encoder

Joose Sainio, Alexandre Mercat, and Jarno Vanne

Tampere University, Tampere, Finland
{joose.sainio, alexandre.mercat, jarno.vanne}@tuni.fi

Abstract. High Efficiency Video Coding (HEVC) doubles the coding efficiency of the prior Advanced Video Coding (AVC) standard but tackling its huge complexity calls for efficient HEVC codec implementations. The recent advances in Graphics Processing Units (GPUs) have made programmable general-purpose GPUs (GPGPUs) a popular option for accelerating various video coding tools. Massively parallel GPU architectures are particularly well suited for hardware-oriented full search (FS) algorithm in HEVC integer motion estimation (IME). This paper analyzes the feasibility of a GPU-accelerated FS implementation in the practical Kvazaar open-source HEVC encoder. According to our evaluations, implementing FS on AMD Radeon RX 480 GPU makes Kvazaar 12.5 times as fast as the respective anchor implemented entirely on an Intel 8-core i7 processor. However, the obtained speed gain is lost when fast IME algorithms are put into use in the anchor. For example, executing the anchor with hexagon-based search (HEXBS) algorithm is almost two times as fast as our GPU-accelerated proposal and the benefit of GPU offloading is reduced to a slight coding gain of 1.2%. Our results show that accelerating IME on a GPU speeds up non-practical encoders due to their enormous inherent complexity but the price paid with practical encoders tends to be too high. Conditional processing schemes of fast IME algorithms can be efficiently executed on processors without any substantial coding loss over that of FS. Nevertheless, we still believe there might be room for exploiting GPU on IME acceleration but GPU-parallelized fast algorithms are needed to get value for additional implementation cost and power budget.

Keywords: High Efficiency Video Coding (HEVC), HEVC encoder, Graphics Processing Unit (GPU), Integer motion estimation (IME), Full-search (FS)

1 Introduction

High Efficiency Video Coding (*HEVC/H.265*) [1] is currently the state-of-the-art video coding standard. It adopts the conventional hybrid video coding scheme (inter/intra prediction, transform coding, and entropy coding) [2] used in the prior video coding standards since H.261. Thanks to its new coding tools and extended block partitioning scheme, HEVC is able to increase coding efficiency by 40% over the preceding *Advanced Video Coding* (*AVC/H.264*) standard [3] for the same objective visual quality. However, these new features also introduce computational overhead of about 40% [2].

There are currently three noteworthy open-source HEVC encoders: *HEVC Test Model* (*HM*) [4], x265 [5], and Kvazaar [6]. HM supports all HEVC coding tools and is able to achieve high coding efficiency but it is far too slow for practical use. Kvazaar and x265 are practical solutions, of which academic Kvazaar was chosen for this work.

The complexity of HEVC is particularly leveraged in the inter prediction due to its numerous prediction modes and block partitions that have to be evaluated during the *rate-distortion* (*RD*) optimization. This paper focuses on *integer motion estimation* (*IME*) that is used in HEVC inter prediction to remove temporal redundancy of video scenes by searching the best matching block between the current and reference blocks. IME is only included in the encoder and it is one of the most complex coding tools in practical HEVC encoders [5], [6]. In software encoders, the complexity of IME can be primarily tackled by multithreading and *single instruction multiple data* (*SIMD*) optimizations. Further speedup can be pursued by offloading IME from a *central processing unit* (*CPU*) to external hardware such as *graphics processing units* (*GPUs*) which are considered in this paper.

GPUs have gained a lot of traction over the past decade. Especially, *general purpose GPU* (*GPGPU*) programming has made GPUs a popular option for accelerating various video coding tools. Particularly, several GPU implementations have been proposed for IME [7]-[13].The current high-end GPUs are massively parallel platforms with *single instruction multiple threads* (*SIMT*) model and their computational capabilities are much higher than those of CPUs.

The complexity of IME is strongly dependent on the used *block-matching algorithm* (*BMA*). The well-known *full search* (*FS*) is the simplest, but the most computation-intensive BMA, which exhaustively tests all candidate blocks in the search area. In addition, numerous fast BMAs such as *hexagon-based* (*HEXBS*) [14] and *test zone* (*TZ*) [15] algorithms have been developed over the years. They are much faster than FS but they also suffer from somewhat reduced search quality. In addition, their irregular execution flows and arbitrary memory accesses can be executed with CPUs but the nature of GPUs requires regular execution and data flow.

In this paper, we present a GPU implantation of the BMA of the practical Kvazaar HEVC. The FS is chosen as BMA due to the regular data flow and its low control overhead which make it a promising candidate for massively parallel GPU architectures.

The rest of this paper is structured as follows. Section 2 explains the previous work. An overview of the HEVC IME process is given in Section 2.1, the concepts behind GPU programming are described in Section 2.2, and prior art is surveyed in Section 2.3. Section 3 describes our GPU-accelerated FS implementation in Kvazaar. Section 4 presents the performance results of our solution, compares them with those of prior art, and discusses why the results were not as good as one might expect. Finally, Section 5 concludes the work.

2 Related Work

In HEVC encoding, each frame is partitioned into *coding tree units* (*CTUs*) of 64×64 samples. The coding tree of HEVC utilizes a quadtree structure to partition the CTUs into smaller *coding units* (*CUs*), called nodes in a quadtree, as illustrated in Fig. 1 (a).

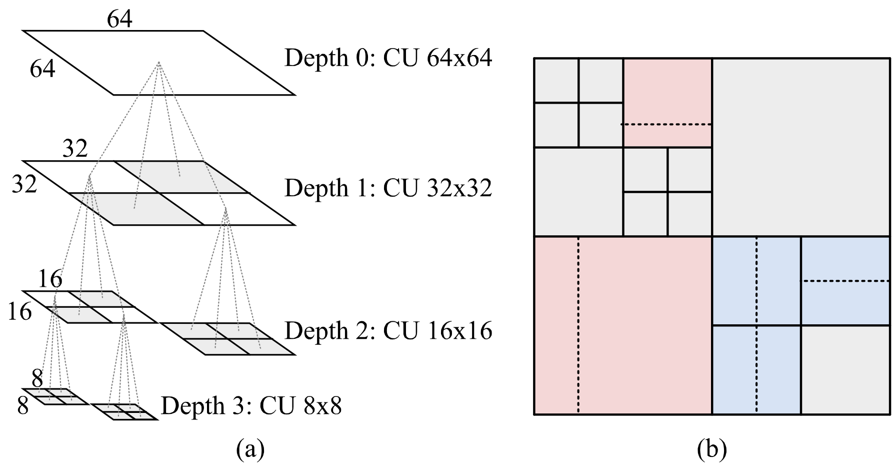

Fig. 1. (a) Partitioning of a CTU structure. (b) Example partitioning of a single CTU.

In HEVC, the size of CUs is equal to 2*N*×2*N*, where *N* ∈ {32, 16, 8, 4}. The HEVC encoder starts by predicting the blocks from their environment (in time and space).

2.1 Integer Motion Estimation in HEVC

The IME process determines integer *motion vectors* (*MVs*) in inter prediction. It is one of the most computation-intensive operation in HEVC encoding. In IME, CUs may be split into *prediction units* (*PUs*) of smaller size with the limitation of width and height being at least 4 pixels. A PU represents a picture region that shares the identical prediction information. The horizontal and vertical components of a MV describe the displacement of the PU in the reference frame. CUs can be divided into one, two, or four rectangular-shaped PUs called partition modes.

Fig. 2 shows all possible partition modes and associated luma PU shapes specified for the inter-coded CU of size 2*N*×2*N*. The supported mode set includes two Square modes (2*N*×2*N* and *N*×*N*), two *symmetric motion partition* (*SMP*) modes (2*N*×*N* and *N*×2*N*), and four *asymmetric motion partition* (*AMP*) modes (*nL*×2*N*, *nR*×2*N*, 2*N*×*nU*, and 2*N*×*nD*). Fig. 1 (b) exposes an example of the final partitioning of a CTU into PUs. The gray, blue, and red blocks represent the Square, SMP, and AMP partition mode, respectively.

The IME process tries to minimize the rate-distortion cost of the MV, which is composed of two variables: 1) a distortion between the reference and the encoded PU and 2) a bit cost of the MV. The most common metric for measuring the distortion is the *sum of absolute differences* (*SAD*) computation between the two blocks. The bit cost is yielded with the *advanced motion vector prediction* (*AMVP*) [2] where the MV of interest is compared with its *MV predictors* (*MVPs*) that are derived from the MVs of

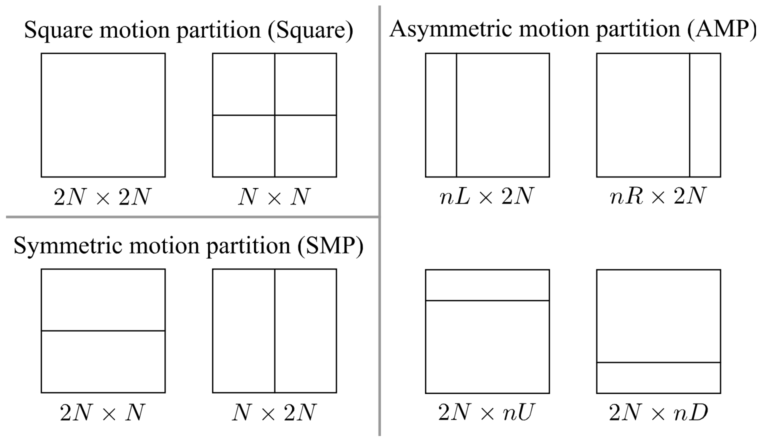

Fig. 2. PU modes in HEVC.

spatially and temporally neighboring PUs. After IME, the result is refined in a quarter pixel domain with *fractional motion estimation* (*FME*).

Three BMAs are considered in this work: *full search* (*FS*), *hexagon-based* (*HEXBS*) and *Test Zone* (*TZ*). FS algorithm, which is the simplest but the most computation-intensive BMA, exhaustively tests all candidate blocks in the search area. HEXBS [14] algorithm selects recursively the best corner of the hexagon until the best location is the center. TZ [15] algorithm combines diamond search and raster search.

2.2 GPU Architecture and Programming

There are two key differences between CPU and GPU programming: the SIMT nature of the GPU and the memory architectural differences. An overview of the GPU memory architecture is depicted in Fig. 3. Unlike CPUs that usually have a few dozen of cores at most, GPUs consist of thousands of less powerful cores. In GPUs, these cores are grouped up to units that are called *Multiprocessors (MP)* in this work. Each GPU vendor has a different number of cores in a single MP, ranging typically from 8 to 64. A single MP has only a single instruction decoder for all cores and all of them execute the same instruction at the same time. When branching takes place, all the cores of a MP execute all branches, including unnecessary branches. To achieve the best performance, GPU programs should avoid branching whenever possible.

CPUs and GPUs have a similar memory structure including a central memory, a couple levels of cache, and registers which are closest to the cores. However, the memory architecture of CPUs is optimized for low latency, whereas the memory architecture of GPUs is optimized for high throughput by extending the width of the global memory. When all the cores in a single MP request a memory access in a single clock cycle, the operation is coalesced into a single access as long as all the cores refer to a single continuous block of memory. Memory access coalescing is an important aspect when designing a GPU-friendly algorithm. Additionally, GPUs can directly access the central memory of CPUs through the PCIe-bus, but the transfer speeds are fairly low. The local memory, which doubles as L1 cache, is shared between the cores of a single

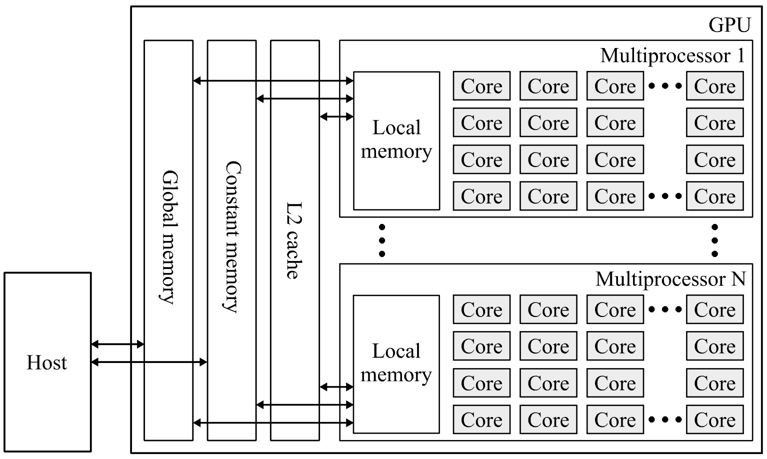

Fig. 3. The memory architecture of a GPU.

MP and can be used for fast data sharing. The programmer has direct control over memory allocation in the local memory, unlike the caches of a CPU. Nevertheless, the local memory size is typically limited to a few dozens of kilobytes per MP.

Work Items are the base elements of GPU programs. A single Work Item runs on a single core of a MP. Work Items are used to form a *Work Group*. A single Work Group is assigned to a single MP and each Work Group keeps this assignation until the end of its execution. A single MP can belong to multiple Work Groups simultaneously, as long as there are enough resources, registers, and local memory for each Work Group. Having multiple Work Groups at the same time allows hiding the latency of the high-throughput global memory. As context switch is a light operation on a GPU, once a Work Group has executed a memory operation, the MP can schedule another Work Group to run. This stresses the importance of reducing the amount of resources used per Work Group.

Currently, there are two notable options for GPGPU programming: CUDA and OpenCL. CUDA is a proprietary language maintained by NVIDIA and it only works for NVIDIA's GPUs. On the other hand, OpenCL is an open standard maintained by the Khronos Group. OpenCL is intended for programming many different platforms, including GPUs. All major desktop GPU vendors (NVIDIA, AMD, and Intel) provide an OpenCL driver for their GPUs. Due to these advantages, OpenCL was chosen for this work over CUDA. OpenCL works on event-based system, where the events can be used as dependencies between multiple tasks. The event system allows the programmer to schedule multiple tasks at the same time while making sure that they are executed in the correct order.

2.3 Previous Studies

For the time being, several efforts have shown successful hardware acceleration of IME in HM [7]-[9] and in practical HEVC encoder x265 [10]-[13].

Even though the work in [7] is based on the AVC standard, it is the first approach implementing hierarchical SAD calculation completely on a GPU. The search algorithm is FS and 4×4 blocks are used as a basis for the SAD calculation. The optimal SAD cost is searched using parallel reduction for each block size. The work in [8] is an extension of [7] for HEVC. The same hierarchical SAD calculation is used as a base. In addition, FME is performed on the GPU. Both implementations acquire about hundred times speedup when compared with respective CPU-only implementations. Conversely, the solution in [9] implements a nested diamond search that is similar to the TZ search. This solution is GPU friendly, but unlike [8], it cannot use hierarchical SAD calculation. The speedup is comparable to [8] with slightly higher quality.

In [10]-[12], a refinement search is performed on the CPU, after the GPU search part, to improve quality. The implementation of [10] is similar to that of [8] with the exception of the refinement search. Both IME and FME are refined on the CPU by using pre-calculated SADs transferred from the GPU to the CPU. Because the FME is done before the IME refinement, the FME has to be conducted on a larger area than usually would be necessary. The complexity of the refinement search is reduced in [11] by using a CPU to calculate only the best five search points found by a GPU. Instead of FS, [12] uses a regular search pattern that allows hierarchical SAD calculation. However, the refinement search cannot use the pre-calculated SADs for MVs that have not been calculated on the GPU. The approach of [13] is different since MV refinement is performed on the GPU instead of CPU. This is achieved by performing the IME in three steps where the first step selects an approximate MV for each block by using a subsampled picture. In the second step, the approximate MVs are used as the MVPs and updated accordingly. Finally, the updated approximate MVs are used in the FME level search as the MVPs.

3 Proposed GPU Acceleration Scheme for FS in Kvazaar

In this paper, we propose a GPU-accelerated FS IME that is based on [8] and [10]. The core principle of the algorithm is hierarchical SAD calculation with SAD reuse and the refinement search. Unlike in [8] and [10], we use 8×8 as a block size for SAD calculation since 4×4 blocks are not used in any presets of Kvazaar.

The MVPs in IME make PUs dependent on the previous PUs of the frame, so calculating all PUs at the same time reduces prediction accuracy by default. However, this is also the case with all previous works. Both [10] and our method utilize a pre-calculated table of bit costs and zero MV as the sole MVP. A refinement search is done on the CPU by using the SADs calculated on the GPU. Unlike [10], we use search range (the maximum displacement of the PU) of [-32, 32] instead of [-64, 64], since they do not have significant difference in quality but the latter is four times slower.

Algorithm 1 describes the proposed algorithm at high level. Choosing the granularity of the algorithm is important to gain the maximum benefit of the GPU. Choosing PU as the granularity would be an optimal choice for data dependencies according to AMVP algorithm. Nevertheless, this granularity would be too

Algorithm 1. Overview of the proposed method

```
1  Send the frame to be compressed to GPU
2  FOR EACH CTU Row index i:
3    Send CTU Row i + 1 of the reference to GPU
4    Expand reference
5    FOR EACH 8×8 block:
6      Assign block to MP
7      Load frame to local memory
8      Load reference from search area to local memory
9      FOR EACH MV in Search Area:
10       Calculate SAD
11       Save SAD to global memory
12
13 FOR EACH PU size:
14   FOR EACH PU in CTU row:
15     Assign each PU to MP
16     FOR EACH MV in Search Area:
17       Combine SAD cost from the SADs calculated
18       For the 8×8 blocks
19       Save SAD to local memory
20       Keep track of the smallest SAD cost
21     Parallel reduction to find the smallest SAD
22     From the search area assigned to the MP
23     Save 9×9 SADs around the best MV to global memory
24
25     Refinement search on CPU by calculating accurate
26     bit cost using the pre calculated SADs
```

fine grain and the GPU would not have enough work for each MP. Next logical granularity level would be CTU level since the CTUs from the reference frame have to be fully processed before starting the current CTU. However, it is still too fine grain for efficient GPU implementations. CTU row level is the first level that is able to fully saturate the GPU and is selected for this work. Going to a coarser granularity would increase the waiting time for the GPU and decrease the performances.

The process can be split into 1) the data transfer between the GPU and CPU and 2) the GPU computation on each MP, as described in Sections 3.1 and 3.2, respectively.

3.1 CPU Implementation and Data Synchronization

Following Algorithm 1, the current frame is first read by the encoder and fully sent to the global memory of the GPU (line 1). Conversely, the reference frames are moved at a CTU row at a time. Because the MVs may point outside of the CTU, when an encoding of a row starts, the row below the current row of the reference frame is sent to the global memory (lines 2-3). For the first row, both the current row and the row below are sent. For the same reason, the reference rows are then expanded by copying the

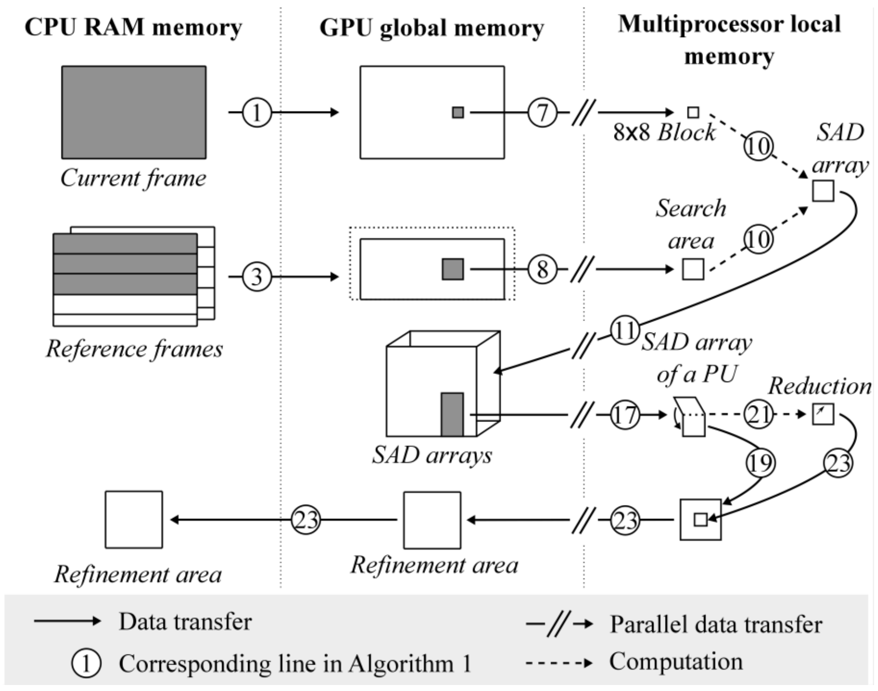

Fig. 4. Overview of the data transfers and computations.

edge pixels (line 4). A visual overview of the data transfers and computations is depicted in Fig. 4.

The rest of the execution is split into two halves: the calculation of SAD (lines 9-11) and the reuse part (lines 13-23). For each CTU row, the expansion task, a SAD calculation, and reduction tasks are launched at the same time. The SAD calculation depends on the expansion, whereas the reduction depends on the SAD calculation.

After the GPU has finished the reduction task (line 21), the resulting MVs are transferred back to the central memory of the CPU (line 23). Additionally, some SAD values around the MV are moved to the central memory for refinement search. The refinement search is finally performed like regular FS except that SAD values are read from the memory.

Fig. 5 depicts the *Bjøntegaard Delta Bit Rate* (*BD-rate*) of the video according to the refinement area size over the case that omits the refinement search entirely. The more the BD-rate is negative, the more higher coding efficiency. The results show that increasing the range of returned SADs improves the coding efficiency. The experimental set-up and metrics are fully detailed in Section 4. The improvement comes from the fact that on CPU, the MVPs can be used to accurately calculate the bit cost of the MV. On the other hand, Fig. 6 depicts the effect of the range of SADs returned to the relative encoding speed in *frames per second* (*fps*). The relative FPS is linearly dependent on the range while the quality changes roughly logarithmically. In order to perform the refinement search with the pre-calculated SADs, some overhead is introduced. The overhead can be seen from the trend line of Fig. 5 at the SAD range 0, which would be

1.00 without overhead. The results of Fig. 5 and 6 show that the quality improvement for larger ranges is low when compared to the increase in encoding time. According to this observation, the range is fixed to four in the rest of this work.

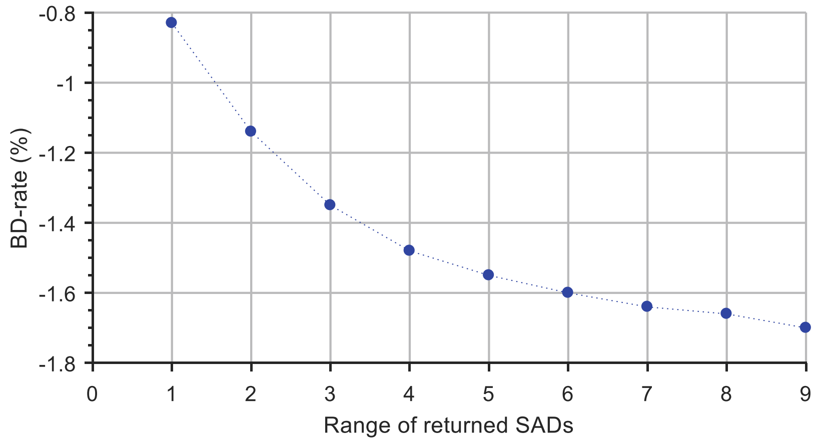

Fig. 5. Coding quality as a function of the search range of SADs.

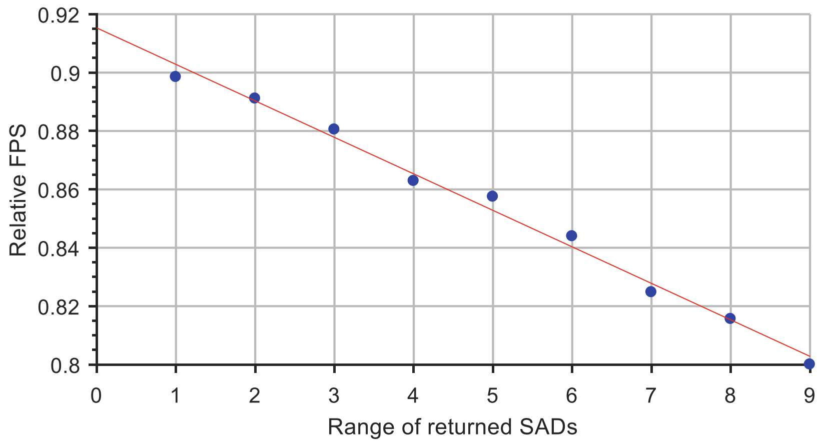

Fig. 6. Relative encoding speed as a function of the search range of SADs.

3.2 GPU Implementation and Computation

As explained in the previous section, the IME implementation on a GPU consist of two distinct steps: calculation of the SADs for each 8×8 block (lines 5-11 of Algorithm 1), and combining the SADs to calculate the optimal MV for each PU (lines 12-23). All the Work Groups are run parallel on the GPU. Each 8×8 block of the current frame is loaded to the local memory of the MP. The size of the reference area depends on the search range, which is fixed to [-32, 32] in this work, corresponding to a search area of 72×72 pixels. In order to reduce the local memory usage, only portion of the referenced area is saved to the local memory at a time. Each core of a MP calculates SADs in a stride of the width of the Work Group and each SAD value is saved into the global memory. The SAD calculation is finished once all SAD costs are saved to the global memory.

The reduction task to find the optimal MV is launched for each PU size. One Work Group is responsible for calculating the optimal MV for one PU. Similarly to the SAD calculation, the cores work in strides. Each core loads the values related to a single PU from the global SAD array and combine them to a single SAD value (lines 18-19). The estimated bit cost is added to the SAD to get the total RD cost of the MV. As each core calculates the combined SADs on its stride, they keep track of the lowest total cost. Additionally, the SAD values are saved to the local memory so that they can be sent back to the CPU for the refinement search. After all combined SADs have been calculated, each core has its own lowest total cost. A parallel reduction is performed to find the global minimum among the cores. At each step of the reduction, the amount of cores working is halved until there is only one core that compares the two last values to find the global minimum. Finally, the best MV and the SADs around the best MV are moved from the local memory to the global memory.

4 Performance Analysis

Our experiments are performed on an Intel i7-5960x 8-core processor with 16 GB of RAM and AMD Radeon RX 480 GPU. Different encoder configurations are benchmarked with the well-known *Bjøntegaard Delta Bit Rate* (*BD-rate*) metric [16] which represents average increment in bit rate (in percent) for the same *Peak Signal-to-Noise Ratio* (*PSNR*). The test set is composed of the 24 HEVC common test sequences [17] whose resolutions range from 2560×1600 to 416×240.

4.1 Coding Efficiency and Speed

Table 1 compares the coding efficiency and speed of the GPU-accelerated Kvazaar (version 1.0) *veryslow* preset over a corresponding Kvazaar anchor configuration implemented entirely on a CPU. The results are reported by executing the anchor with three different BMAs: FS, TZ, and HEXBS. The negative BD-rate results mean that our solution achieves better coding quality than the anchor (FS, TZ or HEXBS). In Table I, green and red results correspond to positive and negative results, respectively.

Implementing FS on a GPU makes Kvazaar up to 12.5 times as fast as the anchor with an average BD-rate degradation of 1.28%. The highest quality degradation comes from screen content videos of the test set. The GPU acceleration also performs relatively poorly with sequences having a lot of static content. In these cases, the anchor

Table 1. Proposed GPU-accelerated Kvazaar vs. original Kvazaar with different IME algorithms.

	Full search		**Test Zone**		**Hexagon based**	
Class	BD-Rate	Speed	BD-Rate	Speed	BD-Rate	Speed
HEVC-A	0.43 %	12.18×	-1.80 %	1.02×	-2.48 %	0.50×
HEVC-B	0.65 %	12.36×	-0.23 %	1.21×	-0.55 %	0.51×
HEVC-C	0.39 %	17.03×	-1.24 %	1.75×	-1.83 %	0.69×
HEVC-D	-0.04 %	16.78×	-1.00 %	1.38×	-1.18 %	0.64×
HEVC-E	0.20 %	6.48×	0.01 %	0.49×	0.00 %	0.36×
HEVC-F	6.04 %	10.61×	1.89 %	1.05×	-1.06 %	0.51×
Average	**1.28 %**	**12.57×**	**-0.39 %**	**1.15×**	**-1.18 %**	**0.53×**

FS only checks the zero vector most of the time and stops the search because the prediction is good enough whereas the GPU-accelerated FS always checks every possible candidate.

Furthermore, the obtained speed gain is lost when TZ and HEXBS are put into use in the anchor. With TZ, our GPU-accelerated approach is not more than 15% faster anymore. The good news is that our proposal achieves a slight BD-rate gain due to deployment of fast BMA in the anchor. With HEXBS, the gain increases to 1.18% but the anchor is almost two times as fast as our GPU-accelerated solution.

4.2 Comparison with Previous Work

The state-of-the-art techniques, presented in Section 2.3, can be split into two categories: techniques implemented on HM and x265. Because there are no previous attempts to accelerate Kvazaar on a GPU, our solution can only be compared with those made for HM and x265. However, comparing the speedup figures of our proposal with those of HM optimization techniques is challenging because HM is not intended for practical or real-time use. Therefore, only coding quality can be compared. Instead, comparing our results with those of x265 is relevant since both Kvazaar and x265 are practical encoders. Though, only the magnitude of speedup can be fairly compared in this case.

Among the existing solutions, the approach presented in [10] is closest to ours. It outperforms our proposal with quality but not with speedup. In [12], the FME was also implemented on a GPU which could explain the existing speedup gap. A quality difference between these two solutions is assumed to be caused by an inherent difference between the internals of x265 and Kvazaar.

The basic idea of [11] is also similar to ours but the speedup is slightly higher than that of [10], with similar effect to quality. However, neither mentioned the anchor version of x265, which makes accurate comparison of the speedups difficult.

The speedup obtained in [12] is really close to ours. Even though the speedup is higher than in [10], again, the anchor versions of x265 were not mentioned. The comparison of coding quality is also a bit difficult since they did not use the standard BD-rate, computed with four *Quantization Parameter* (*QP*) values, but they only reported the difference in bit rate and PSNR with a single QP value. Their solution had only a minimal effect on bit rate, which lets us assume that the coding efficiency of our proposal is at least as high as theirs.

The results of [13] are most interesting since they used a fast BMA as an anchor instead of FS. The main accomplishment of [13] was a real-time 4K 60 fps 10-bit encoder. However, the experiments were conducted using an old version of x265, whose performance is much lower than that of a newer version that would have been available. It is questionable whether the quality stays the same when the algorithm is applied to a more up-to-date version of x265. Moreover, the proposed solution of [13] is proven only on 10-bit video and thus it might not be suitable for general use.

4.3 Lessons Learned

Generally speaking, our results do not seem too poor when compared with typical optimizations introduced in the literature to an HEVC encoder. However, in our case, the overhead of the co-processor has to be counted in. The GPU in our solution has computational power of up to 5.8 TFLOPS and it consumes up to 150W of power. Considering this overhead, better results than 15% speedup and slightly improved quality can be expected.

In general, FS is not very efficient. As an example, FS checks 4225 different candidates for each PU with a search range of [-32, 32] whereas HEXBS checks a few dozen on average with the same search range. Although GPUs are more powerful than CPUs, the gap is not large enough to compensate for the inefficiency of the FS algorithm. Furthermore, even GPU-friendly fast BMAs might not make GPU acceleration of IME worthwhile. In Kvazaar, IME accounts for only about 15% of the total encoding time in both ultrafast and veryslow presets, so not more than 15-20% speedup is possible even in theory. Moreover, the dependency to MVP makes a perfect offloading difficult to achieve.

The loss in coding efficiency over the anchor FS implementation is as expected. However, it is still somewhat higher than those reported for x265 and HM, which can be explained by the differences in RD optimization processes of Kvazaar, x265, and HM. Nevertheless, our solution outperforms the fast anchor TZ and HEXBS algorithms in coding efficiency.

Overall, quality degradation is not a huge concern for fast IME algorithms compared with the speedup they are able to provide. The BD-rate difference between the anchor FS and HEXBS is about 2.5%, which would be acceptable even for about 1.5× speedup in most applications. In other words, a faster GPU-based IME implementation could have worse coding efficiency than HEXBS and it could still be considered successful, as long as it has a low overhead.

Though many prior works have included FME as a part of the GPU acceleration, to our best knowledge, nobody has evaluated acceleration of FME on a GPU separately. To conclude, it is in theory possible to accelerate Kvazaar veryslow preset by about 66% since Kvazaar uses about 40% of encoding time on IME and FME. Unfortunately, the feasibility of FME on a GPU is not explored to confirm this hypothesis.

5 Conclusions

In this paper, we presented an attempt to accelerate IME of the practical Kvazaar HEVC encoder on a GPU. The accelerated BMA was FS whose regular data flow and low control overhead make it a promising candidate for massively parallel GPU architectures. Even though our approach was able to accelerate Kvazaar by about 12.5 times at a cost of minor quality degradation, the gain was lost when our solution was compared with fast BMAs such as HEXBS and TZ algorithms. Furthermore, a closer look at previous studies revealed that our results are in line with them.

Despite the fact that our results are discouraging, we still believe HEVC encoder acceleration with GPUs could be a viable option but with more GPU-friendly fast BMAs. Another potential approach is to offload other coding tools such as FME together with IME to a GPU and this way reduce data dependencies and syncing between CPU and GPU. However, the data dependencies might still prove to be too great of a challenge to tackle with the GPU architecture.

Acknowledgements
This work was supported in part by the European Celtic-Plus project VIRTUOSE and the Academy of Finland (decision no. 301820). The authors would also like to thank all contributors of Kvazaar open-source project [6].

References

1. *High Efficiency Video Coding*, document ITU-T Rec. H.265 and ISO/IEC 23008-2 (HEVC), ITU-T and ISO/IEC, Apr. 2013.
2. G. J. Sullivan, J. R. Ohm, W. J. Han, and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, Dec. 2012, pp. 1649-1668.
3. *Advanced Video Coding for Generic Audiovisual Services*, document ITU-T Rec. H.264 and ISO/IEC 14496-10 (AVC), ITU-T and ISO/IEC, Mar. 2009.
4. *Joint Collaborative Team on Video Coding Reference Software, ver. HM 16.0* [Online]. Available: http://hevc.hhi.fraunhofer.de/
5. *x265* [Online]. Available: http://x265.org/
6. *Kvazaar HEVC encoder* [Online]. Available: https://github.com/ultravideo/kvazaar
7. D. Lee and S. Oh, "Variable block size motion estimation implementation on compute unified device architecture (CUDA)," *in Proc. IEEE Int. Conf. Consumer Electron.*, Las Vegas, NV, USA, Jan. 2013.
8. D. Lee, D. Sim, K. Cho, and S. J. Oh, "Fast motion estimation for HEVC on graphics processing unit (GPU)," *Journal of Real-Time Image Processing*, vol. 12, no. 2, Aug. 2016, pp. 549-562.
9. E. Hojati, J. Franche, S. Coulombe, and C. Vázquez, "Highly parallel HEVC motion estimation based on multiple temporal predictors and nested diamond search," *in Proc. IEEE Int. Conf. on Image Processing*, Beijing, China, Sep. 2017.
10. F. Wang, D. Zhou, and S. Goto, "OpenCL based high-quality HEVC motion estimation on GPU," *in Proc. IEEE Int. Conf. on Image Processing*, Paris, France, Oct. 2014.

11. X. Wang, L. Song, M. Chen, and J. Yang, “Paralleling variable block size motion estimation of HEVC on multi-core CPU plus GPU platform,” *in Proc. IEEE Int. Conf. on Image Processing* , Melbourne, Australia, Sep 2013.
12. H. Kao, I. Wang, C. Lee, C. Lo, and H. Kang, “Accelerating HEVC motion estimation using GPU,” *in Proc. IEEE Int. Conf. on Multimedia Big Data*, Taipei, Taiwan, Apr 2016.
13. F. Takano, H. Igarashi, and T. Moriyoshi, “4K-UHD real-time HEVC encoder with GPU accelerated motion estimation,” *in Proc. IEEE Int. Conf. on Image Processing*, Beijing, China, Sep 2017.
14. C. Zhu, X. Lin, and L.-P. Chau, “Hexagon-based search pattern for fast block motion estimation,” *IEEE Trans. Circuits Syst. Video Technol.,* vol. 12, no. 5, May 2002, pp. 349–355.
15. I. Werda, H. Chaouch, A. Samet, M. A. Ben Ayed, and N. Masmoudi, “Optimal DSP based integer motion estimation implementation for H.264/AVC baseline encoder,” *The Int. Arab J. of Inform. Technol.,* Vol. 7, No. 1, 2010, pp 96-107.
16. G. Bjøntegaard, “Calculation of average PSNR differences between RD-curves,” *document VCEG-M33*, Austin, Texas, USA, Apr. 2001.
17. F. Bossen, “Common HM test conditions and software reference configurations,” *document JCTVC-L1100*, Geneva, Switzerland, Jan. 2013.