Parallel H.264/AVC Motion Compensation for GPUs using OpenCL

Biao Wang, Mauricio Alvarez-Mesa, *Member, IEEE,* Chi Ching Chi, Ben Juurlink, *Senior Member, IEEE*

***Abstract*—Motion compensation is one of the most compute-intensive parts in H.264/AVC video decoding. It exposes massive parallelism which can reap the benefit from Graphics Processing Units (GPUs). Control and memory divergence, however, may lead to performance penalties on GPUs. In this paper, we propose two GPU motion compensation kernels, implemented with OpenCL, that mitigate the divergence effect. In addition, the motion compensation kernels have been integrated into a complete and optimized H.264/AVC decoder that supports H.264/AVC high profile. We evaluated our kernels on GPUs with different architectures from AMD, Intel, and Nvidia. Compared to the fastest CPU used in this paper, our kernel achieves 2.0 speedup on a discrete Nvidia GPU at kernel level. However, when the overheads of memory copy and OpenCL runtime are included, no speedup is gained at application level.**

***Index Terms*—Motion compensation, GPU, H.264/AVC, OpenCL.**

I. Introduction

Currently H.264/AVC is one of the most widely used video codecs in the world [1]. It achieves significant improvements in coding performance compared to previous video codecs at the cost of higher computational complexity. In addition, the pursuit for higher resolution content leads to more computational demand. Single-threaded performance, however, is no longer increasing at the same rate and now performance scalability is determined by the ability of applications to exploit thread-level parallelism on parallel architectures.

Among different parallel processors, Graphic Processor Units (GPUs) have emerged for general-purpose computing in recent years. The general purpose programming models such as CUDA [2] and OpenCL [3] further their popularity. GPUs consist of multiple SIMD (Single Instruction Multiple Data) engines that execute instructions in lock step. Applications with massive and regular parallelism can be executed efficiently on GPUs. This motivates the use of GPUs for accelerating video codecs.

The motion compensation stage in H.264/AVC takes a significant proportion of decoding time [4]. The computational complexity of motion compensation comes from the interpolation filters that generate fractional samples. The interpolation is independent for each sample inside each frame, which fits the GPU architecture well. In CPU optimized H.264/AVC decoders, however, the kernels are executed on a macroblock-by-macroblock basis in order to exploit data locality. GPUs, nevertheless, are not appropriate for this fine-grain computation approach because of memory copy and kernel launch overhead. As a consequence, an adaptation of CPU optimized implementation is required to offload motion compensation onto GPUs efficiently.

Although, motion compensation is parallel at the frame level, the performance on the GPU may suffer significantly when the threads operating in lock-step behave differently due to control or memory divergence. Divergence might appear due to the presence of multiple interpolation modes per macroblock partition and their respective memory access patterns. In order to mitigate the divergence effect we propose different strategies for implementing the data loading and computing phases of motion compensation.

The evaluation of the proposed motion compensation kernels has been conducted on three different GPU architectures in order to test and demonstrate the performance portability of the OpenCL implementations.

Although the proposed GPU implementation obtains speedup compared to an optimized CPU implementation at the kernel level, when considering the overhead of memory transfers and OpenCL runtime no performance benefits are obtained. This paper provides a quantitative analysis of the different sources of overhead (GPU architecture, memory, and runtime system) and their impact in the complete application performance. GPU video decoding can be effective only if those bottlenecks are removed.

In summary, the main contributions of this work are:

- We propose novel motion compensation methods for GPUs that mitigates the memory and control divergence effects.
- We compare and analyze the performance of the proposed algorithms on different GPU architectures.
- We integrate the GPU motion compensation kernel into an optimized H.264/AVC decoder and make a complete comparison against conventional CPU decoder.

The remainder of this paper is organized as follows. Section II describes the related work. Section III introduces the motion compensation in H.264/AVC. The proposed motion compensation kernel for GPUs is presented in Section IV. The experimental setup is presented in Section V and the experimental results are presented in Section VI. Finally, the conclusion is drawn in Section VII.

B. Wang, M. Alvarez-Mesa, C. C. Chi, and B. Juurlink are with the Embedded Systems Architecture group, TU Berlin, Einsteinufer 17, D-10587 Berlin, Germany. email: biaowang@mailbox.tu-berlin.de, {chi.c.chi,mauricio.alvarezmesa,b.juurlink}@tu-berlin.de

This work receives funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the LPGPU Project (www.lpgpu.org), grant agreement n° 288653, and China Scholarship Council (CSC) with grant number 2010607019.

II. Related Work

Video decoding can be offloaded to GPUs using either the dedicated hardware accelerators or the programmable cores. Although custom hardware accelerators are more energy efficient they lack flexibility since the function is hardwired for specific codecs. In addition, they require the use of proprietary APIs that leads to poor portability. To overcome these drawbacks, we choose a more flexible software-based solution with the portable OpenCL programming model.

Regarding the software-based motion compensation implementation on GPUs, Shen et al. [5] adopted a divide-and-conquer method in which blocks with the same interpolation mode are batched and executed with mode specific kernel for motion compensation of Microsoft proprietary codec WMV-8. Pieter et al. [6] employed a similar idea to offload the motion compensation in H.264/AVC onto GPUs using CUDA. However, this strategy requires extra mode-location mapping buffers to indicate the positions of the blocks with the same interpolation mode. As the position of the block is indicated at 4×4 block level, the size of these buffers in total will go up to 1/16 of the entire frame. Transferring these buffers from CPU to GPU will increase the memory copy overhead. Furthermore, multiple kernel launches are required per frame for the interpolation of one direction. For bi-directional predicted B frames, because the prediction modes of the two directions are not necessary the same, another round of kernel launches is required and the predicted result of one direction has to be stored in off-chip memory. This increases the kernel launch overhead to a great extent and introduces long latency memory access when merging the prediction results of B frames.

Comparatively, in our proposed solution, first, no extra mapping buffers are required, which reduce the memory copy between CPU and GPU. Second, only one kernel launch is required to process the entire frame, which minimizes the kernel launch overhead. Third, for B frames, we process bi-directional prediction in one complete kernel and store the intermediate results in on-chip local memory, which greatly reduces the cost for results merging. Finally, our motion compensation kernel has been integrated in a complete and optimized H.264/AVC decoder and evaluated in multiple GPU architectures.

III. Motion Compensation in H.264/AVC

Motion compensation is a block-based inter-prediction technique that predicts samples in current frame from previously decoded reference pictures. A tree structured partition scheme is adopted in H.264/AVC, varying from 16×16 down to 4×4 blocks. Each macroblock can have one of the following 4 partitions: 16×16, 16×8, 8×16, and 8×8. If 8×8 partition is chosen, it can have smaller partitions, namely, 8×4, 4×8, and 4×4 [1].

The samples within a partition are predicted from an area in a reference picture. The reference pictures are organized as reference picture lists and can be specified by a reference index. The location of the reference area is indicated by a motion vector. The resolution of luma motion vectors is quarter pel. Therefore, the luma motion vector may point to integer, half-pel, and quarter-pel positions. For integer position, the predicted partition is a copy of the referenced partition. For half-pel positions, the predicted pels are interpolated using a six-tap Finite Impulse Response (FIR) filter based on integer pels, either horizontally or vertically, as shown in Figure 1a and 1b, respectively. For the horizontal-vertical half pel as shown in 1c, the FIR filter needs to be applied twice. For quarter-pel positions, a bi-linear filter is applied based on pels at integer- and half-pel positions. The dependencies of quarter-pel to integer- and half-pel are shown in Figure 1d. In total, there are 16 different interpolation modes as indicated by the dashed rectangle.

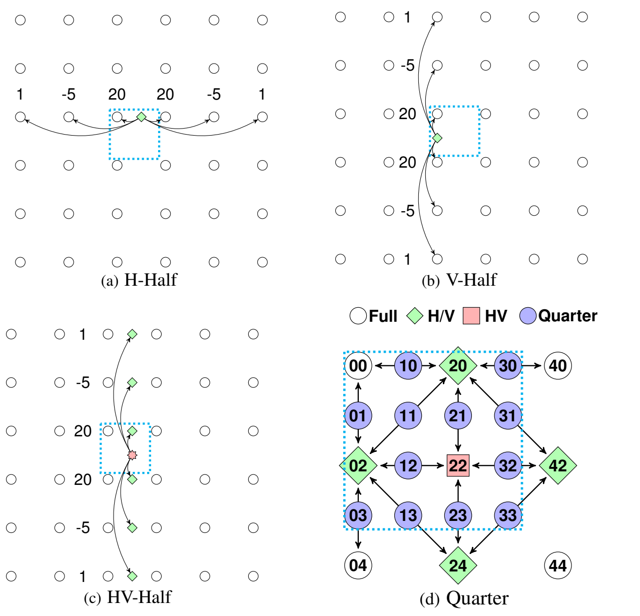

Fig. 1: FIR filter for half pel and dependencies for quarter pel

For the chroma components eight-sample resolution vectors are used. Interpolated samples are generated using linear interpolation, which is much less computational intensive than the luma interpolation [1].

IV. Offload Motion Compensation onto GPUs

This section starts with the code adaptation for the CPU-GPU integrated decoder. Afterwards, the motion compensation kernel's thread mapping and kernel design are presented.

A. A Hybrid CPU-GPU Decoder

Our baseline is an optimized CPU decoder [7] with entropy decoding and reconstruction decoupled on frame level. After entropy decoding, all the decoding kernels for reconstruction are applied in a single decoding loop on macroblock-by-macroblock basis. This approach results in high performance on CPUs due to the exploitation of data locality. GPUs, however, requires more data to benefit from its throughput oriented design. Therefore, motion compensation has been redesigned to be applied on frame level and the modified decoder consists of two parts, the motion compensation kernel processed on the GPU at frame level and the remaining kernels processed on the CPU at macroblock level.

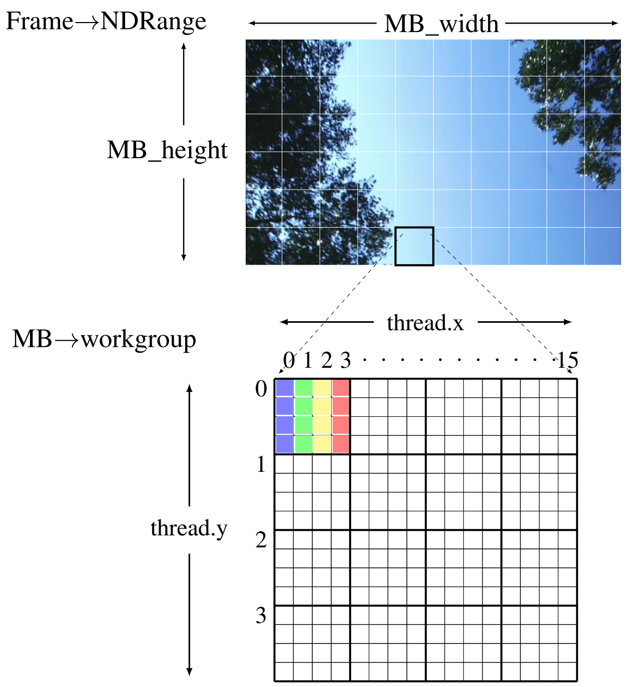

Fig. 2: Motion compensation thread mapping to the output frame and macroblock (MB) with NDRange and workgroup configurations

B. OpenCL Thread Mapping

OpenCL uses a hierarchical data-parallel programming model. At the top level, the global thread index space, termed as *NDRange*, is divided into *workgroups*. Each workgroup is further divided into *workitems*. Each workitem is an instance of a kernel execution, i.e a thread [3].

For motion compensation, a hierarchically mapping is adopted in which the entire frame is mapped to the NDRange, and each macroblock is mapped to each workgroup, as shown in Figure 2. A thread per column mapping is adopted in which each column of 4 samples in the output block are computed by one thread. Every 4 threads process an output block of 4×4 samples. Under this solution, a macroblock of 16×16 samples are processed by 64 threads, using a configuration of 16 thread in the horizontal direction and 4 threads in the vertical direction, as shown in Figure 2.

This mapping of 4 threads per 4×4 block is a tradeoff between parallelism and load balancing. On the one hand, a thread-per-sample mapping exposes the maximum parallelism. Because of the 6-tap filter, however, for an output block of 4×4 samples the input reference block can be up to 9×9 samples. This difference in input and output block size poses a challenge for a balanced thread mapping on GPUs. On the other hand, a thread-per-block mapping, in which one thread processes one block of 4×4 samples, can solve this load imbalance, but this not only leads to more divergence, but also exhibits less parallelism which is not optimal for GPUs.

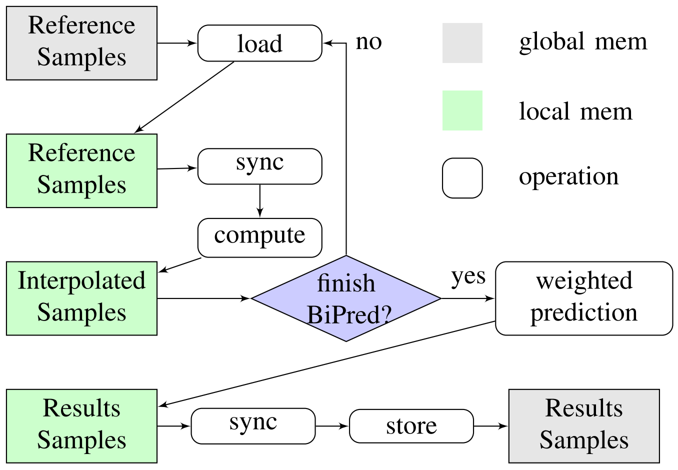

Fig. 3: High level overview of motion compensation workflow for single-stage and multi-stage kernels on GPUs

C. Motion Compensation Kernel Design

The motion compensation kernel consists of three phases: loading the reference data from memory, performing the appropriate interpolation filter(s), and storing the interpolated data back to memory. A straightforward approach (named as "baseline") to implement this kernel consists of having a separated control path for each of the 16 luma interpolation modes. In this implementation the load, interpolation and store phases can have control divergence when partitions within a macroblock smaller than 16×16 have different interpolation modes. Control divergence results in serialized execution in the GPU reducing the performance significantly.

As way to overcome the control divergence for the load phase consists of creating a "shared load" phase in which all the workgroup threads cooperate to load the data for all the partitions inside a macroblock. In order to perform the shared loading the kernel loads the reference samples from global memory to local memory first, and then applies the computation using a single-stage per interpolation mode, as in the baseline case. We refer to this approach as the single-stage mode. Although control divergence is reduced, memory divergence might appear. If the motion vectors of the macroblock partitions point to different reference areas, and the hardware cannot coalesce the memory accesses, memory diverge appears and causes stalls in the GPU cores reducing the performance.

In the shared load phase the input reference block can be up to 9×9 samples for each 4×4 output block. Horizontally, because 9 samples can't be evenly divided into 4 threads, we load 16 samples in total for better alignment of 32-bit memory access. Vertically, the horizontal process is repeated row-by-row. After the load phase a synchronization operation is required because the thread mappings of the load phase and the compute phase are different.

The single-stage approach reduces the control divergence of the load phase but can have control divergence in the computation phase for small macroblock partitions, if those have different interpolation modes. A way to mitigate this control divergence consists of reducing the number of control

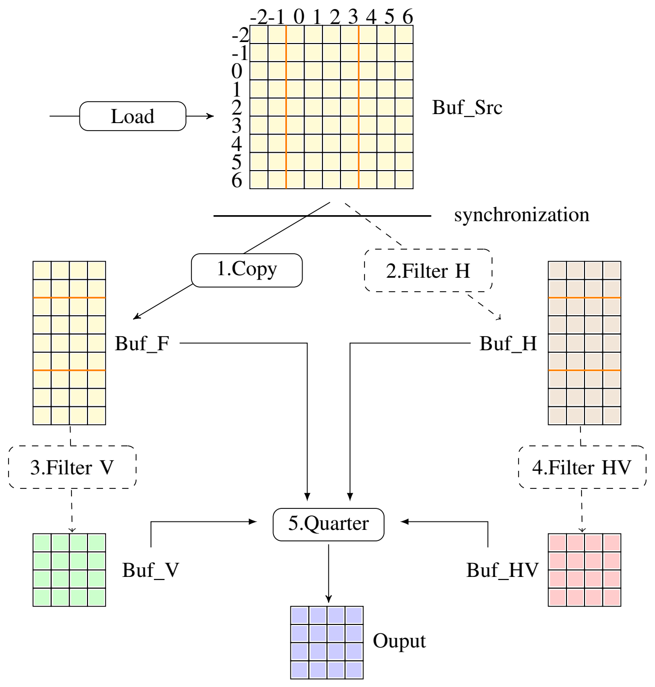

Fig. 4: Multi-stage design, each stage produces the required intermediate results of copy and H/V/HV filters except the final stage that produces output

paths used for the interpolation modes. Because some modes share the same computation it is possible to decompose each mode into finer stages and exploit the shared ones to reduce the number of control paths. We refer to this as the multi-stage mode. In this mode control divergence is minimized at the cost of adding some overhead, because, in some cases, the multi-stage performs more operations than necessary.

Figure 3 shows the high level overview of workflow for single- and multi-stage kernels. These two approaches only differ in the implementation of the computation phase. When the computation phase is finished, the interpolated samples are first stored to an output buffer for weighted prediction. If the mapped block is enabled for bi-directional prediction, the load and compute phases are repeated for the other direction, otherwise, the weighted prediction is applied to obtain the result samples. Finally, the result samples are stored back to global memory.

Figure 4 shows the complete 5 stages of multi-stage kernel for one output 4×4 block. For every stage, 4 threads process one row of consecutive 4 samples and repeats row by row to produce a block of samples. The *Copy* stage copies the original integer samples (from column 0 to 3) to another buffer *Buf_F*. Afterwards, according to the mode, stages of *Filter H, V, HV* perform the interpolation for horizontal, vertical and horizontal-vertical half pels, respectively. The results are stored accordingly to *Buf_H, Buf_V, and Buf_HV*. The dashed lines of these stages indicate that they are optional. The predicted values in one (for half and integer pel) or two buffers (for quarter pel) of *Buf_F, Buf_H, Buf_V, and Buf_HV* are then applied by the bi-linear filter to obtain the quarter pel in stage *Quarter*. Table I summarizes the stages required for each mode, where the "X" represents a stage hit.

TABLE I: Stage taken by each mode

Mode	Stage				
xy	Copy	Filter H	Filter V	Filter HV	Quarter
00	X				X
10	X	X			X
20	X	X			X
30	X	X			X
01	X		X		X
11	X	X	X		X
21	X	X		X	X
31	X	X	X		X
02	X		X		X
12	X	X	X	X	X
22	X	X		X	X
32	X	X	X	X	X
03	X		X		X
13	X	X	X		X
23	X	X		X	X
33	X	X	X		X

For the store phase, we found that in some architectures 32-bit memory accesses to global memory are much faster than 8-bit ones [8]. Therefore, all kernel implementations has a unified store phase. We change the thread mapping to store 4 consecutive samples to global memory, thus a synchronization is applied before the store phase.

V. Experimental Setup

We carry out our experiments on three different platforms. The GPUs, as well as their system configuration are listed in Table II. The *PE/CU* indicates the processing element number per compute unit. The Quadro 3000M is a traditional discrete GPU which is connected via PCI Express, while the HD4000 and HD7500G are integrated on the same die as the CPU. The CPUs with integrated GPUs had the their Turbo Boost (TB) frequencies enabled during the evaluation as they could not be easily disabled. The OpenCL decoder source code is available at [9].

For the evaluation we selected four (1920×)1080p videos (*blue_sky, park_joy, pedestrian_area and riverbed*) from Xiph.org [10] and two (3840×)2160p videos (*Lupo_candlelight and rain_fruits*) from EBU UHD-1 Public Test Set [11]. All videos are encoded using x264 [12] with the "Constant RateFactor" (referred to as CRF) encoding mode. To cover a wide range of bitrates, these videos are encoded using 9 CRF values ranging from 12 to 52. All partition sizes are enabled, as well as the weighted prediction and intra-prediction in P and B frames. The number of reference pictures is set up to 16. Figure 5 shows the bitrates of all the encoded sequences.

VI. Experimental Results

In this section, the divergence cost is investigated first, then proposed kernel performance on GPUs is evaluated. Finally, the performance of hybrid CPU-GPU decoder is presented.

A. *Effectiveness of Divergence Mitigation*

To evaluate the effectiveness of divergence mitigation techniques of the proposed kernels we conduct a series of experiments using a real video with synthetic interpolation

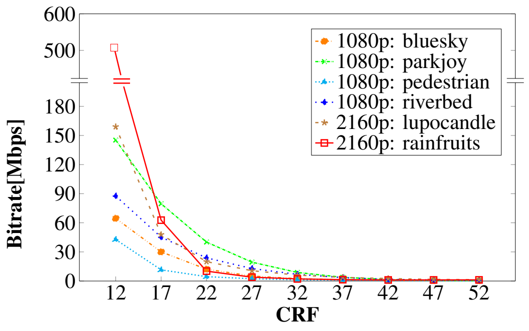

Fig. 5: Bitrate across CRFs for 1080p and 2160p videos

TABLE II: Multiple system configurations

GPU	Quadro 3000M	HD4000	HD7500G
Vendor	Nvidia	Intel	AMD
Compute Unit	5	16	4
PE/CU	48	8	64
Freq(MHz)	900	350(1050 TB)	327(424 TB)
Mem BW(GB/s)	80	25.6	21.3
Integrated GPUs?	No	Yes	Yes
CPU	i7-2760QM	i5-3317U	A6-4455M
Vendor	Intel	Intel	AMD
Cores	4	2	2
Freq(GHz)	2.4(TB off)	1.7(2.6 TB)	2.1(2.6 TB)
Mem BW(GB/s)	25.6	25.6	21.3
OS	ubuntu 12.10	windows 7	windows 7

modes. As base we use the pedestrian 1080p sequence with only 16x16 partitions. We arbitrarily modify the fractional and integer parts of the motion vector to artificially create 8×8 and 4×4 partitions. The performance of the proposed kernels on three GPUs across different branch configurations are presented in Figure 6. The notation of "F" and "I" indicate the modification to the integer and fraction part of the motion vector, respectively. In "I+F" both the integer and fractional parts are different.

The execution time of the baseline kernel increases with the amount of divergence on the Nvidia and AMD GPUs. The single-stage mitigates the control divergence for the load phase. An execution time reduction can be seen in both 4×4 and 8×8 configurations. The multi-stage decreases the execution time for the 4×4 configurations only. At 8×8F, the divergence mitigation benefit is counter balanced by the additional local memory data copies performed in the multi-stage approach.

For Intel GPU, the multi-stage approach has no advantage over single-stage with branch divergence configuration. This is attributed to the divergence optimization called "Basic Cycle Compression" for the Intel GPU [13]. Within this approach, if 4 contiguous workitems are not divergent within a warp (either 16 or 8 workitems), the inactivated cycles caused by the divergence within the warp can be compressed. This capability results from the difference between the warp size and the SIMD width of compute unit (4 workitems). The compute unit executes a maximum of 4 cycles for one warp. If inactivated cycles caused by the divergence are detected, they can be skipped by the compute unit. For the baseline kernel, the size of warp reported from the profiler is 8 workitems instead of 16, such as that in single stage and multi-stage. This increases

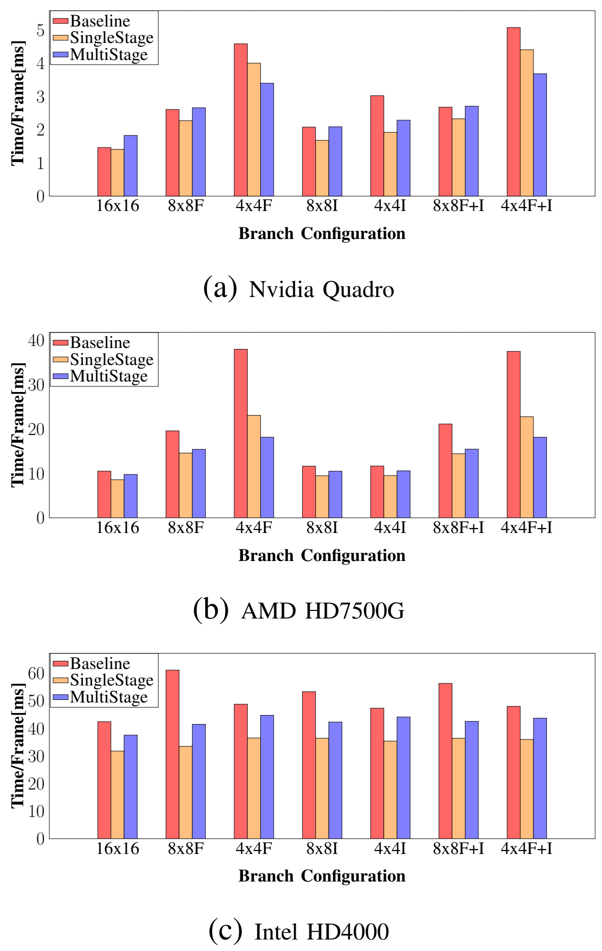

Fig. 6: Branch and memory divergence analysis for baseline, single stage, and multi-stage kernel using 1080p pedestrian video encoded with 16x16 partition only

the overhead of task-switching and may be one of the reason for its low performance.

B. Performance Using Real Videos

Figure 7 shows the results of kernel execution on GPUs and CPUs across all CRFs for two resolutions. The execution time includes both luma and chroma part. The chroma part takes up 36%, 29%, and 49% of the kernel execution time on GPUs from Nvidia, AMD and Intel, respectively. Because the chroma motion compensation is identical in both single- and multi-stage approaches, the execution time differences are only caused by the luma motion compensation.

Comparing the 1080p and 2160p results shows in general a similar behaviour for all configurations, only the execution time is 4 times higher for 2160p compared to 1080p due to 4× the number of pixels. With higher CRF, the execution time decreases because at lower bitrates more macroblocks are encoded into 16x16 blocks, which results a higher SIMD utilization. The performance change is more intensive with bitrate for the CPU compared to the GPU, as for smaller block sizes the CPU implementation processes them one-by-one sequentially with more call and loop overhead and a lower

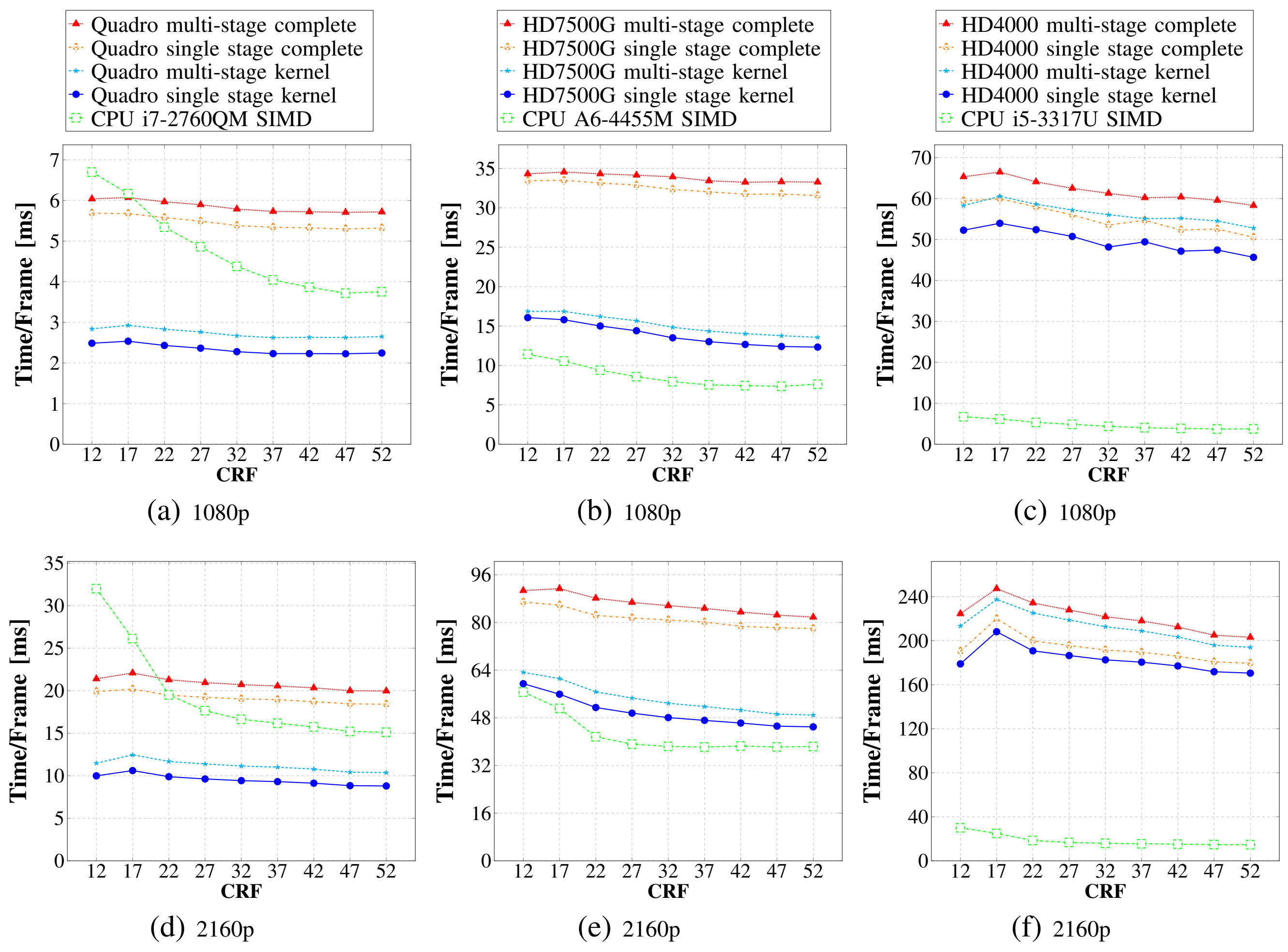

Fig. 7: Motion compensation time cost per frame across CRFs for 1080p and 2160p videos

SIMD vector utilization. The GPU execution time is more stable because the entire macroblock is executed in parallel independent of the block partitioning.

The speedups of the single stage over the multi-stage approach are presented in Table III for the evaluated CRF values. The single-stage kernel outperforms multi-stage in all cases because in real videos most of the macroblocks are 16×16 for 1080p and 2160p resolutions, as indicated by the average block size in samples in Table III. The maximum average block size is 256 samples in which all macroblock have a single 16×16 partition. For the speedup results only the luma execution time is considered, because the chroma execution time is identical.

Even for videos at CRF 12 in 1080p the average block size is 166.4 samples, which indicate that the block sizes are still fairly large. Because the multi-stage approach only performs better with 4x4 partitions and the maximum ratio of 4x4 block is only 8.93% found in any video, the single stage approach performs better for all tested videos. The multi-stage approach does have a lower worst-case complexity, and might be favored for its more consistent performance with outlier videos.

When comparing different GPUs the performance of motion compensation varies significantly. When considering the kernel only execution time the discrete Quadro GPU is the fastest platform, which is expected because of the higher computational capabilities. The single stage kernel achieves 2.34 ms per frame on average for 1080p, which is also 2.0× faster compared to the i5-3317U and i7-2760QM CPUs.

TABLE III: Speedup of single stage over multi-stage kernel across CRFs for 1080p and 2160p videos

CRF	1080p			2160p			1080p	2160p
	Nvidia	AMD	Intel	Nvidia	AMD	Intel	AVG block size	
12	1.21	1.06	1.27	1.22	1.08	1.35	166.4	190.0
17	1.23	1.09	1.28	1.27	1.13	1.28	191.6	233.2
22	1.25	1.11	1.28	1.29	1.15	1.37	213.3	249.0
27	1.26	1.12	1.29	1.29	1.15	1.36	226.3	253.2
32	1.27	1.13	1.33	1.29	1.15	1.35	238.4	254.6
37	1.27	1.14	1.27	1.30	1.15	1.35	247.9	255.1
42	1.28	1.15	1.35	1.30	1.14	1.34	252.1	255.5
47	1.29	1.16	1.33	1.30	1.14	1.33	254.3	255.9
52	1.29	1.15	1.34	1.30	1.14	1.32	254.2	255.8

TABLE IV: Percentage of mode 00 across CRFs for 1080p and 2160p videos

CRF	12	17	22	27	32	37	42	47	52
1080p	11.5	17.3	21.2	24.8	30.9	40.5	51.0	60.4	72.5
2160p	33.9	53.8	61.7	68.5	77.6	72.9	86.4	75.9	90.9

The execution time of the complete GPU motion compensation is significantly higher because it also includes the OpenCL buffer copy and command overhead. The OpenCL runtime overheads take 37% to 57% of motion compensation time on Quadro and HD7500G. On the HD4000 the overhead percentage is much lower, because the kernel execution time is much higher. Even for this platform the overheads are equivalent to the complete motion compensation time using the CPU solution. For the integrated GPUs the memory copy overhead is already optimized using zero copy [8]. For discrete GPUs, the memory copy overhead can only be reduced using asynchronous execution, which as we will show in the next section has moderate performance impact. Unless the OpenCL runtime overheads are significantly reduced GPUs cannot be efficiently used for video decoding applications in general.

Not only OpenCL runtime overheads need to improve for making a GPU offloaded video decoding solution feasible, though. For all 3 GPUs also the motion compensation kernel cannot efficiently use the available hardware resources. This is most prominent in the Intel HD4000 GPU as the execution time is 3 times higher than the comparable AMD 7500G GPU. We found that the main bottleneck is the local memory is very slow when using 8-bit operations. We tested the bandwidth of local memory using the sample benchmark “LDSBandwidth” [14]. With 32-bit memory access per thread, the read and write bandwidth of local memory is 86.3GB/s and 66.6GB/s, respectively. If we use 8-bit memory access, such as used in our kernel, only 7.9GB/s and 8.0GB/s are achieved for read and write, respectively. The performance on the HD4000 is further reduced due its lower performance for integer codes. In the HD4000, each compute unit has two 4-wide vector ALUs (Arithmetic Logic Units) and the second ALU is available only for floating point operands [15]. For the AMD HD7500G GPU the processing element is a four-way VLIW (Very Long Instruction Word) processor [8]. To attain high performance, instruction level parallelism (ILP) is required within each workitem so that the compiler can pack independent instructions to fill the VLIW slots. The proposed kernels process four samples per thread hence the compiler can unroll all loop statement to exploit ILP. Despite this a packing efficiency of only 50% for both single and multi-stage implementation is reported indicating that on average only two of the four VLIW slots are used. The packing efficiency is hindered by the condition statements for mode detection.

The Nvidia Quadro GPU using the Fermi core architecture seems to be the most suited architecture for motion compensation. Using the Nvidia profiler, however, we measured that around 57% of GPU issue cycles are utilized on average for the motion compensation kernel. Idle instruction issue cycles occur when their is not a free wavefront/warp available to schedule from. This mainly occurs when insufficient wavefront/warps are scheduled on a GPU compute unit, and also is the case for the Quadro GPU with only half of the possible warp slots filled on average. More warps cannot be kept in-flight, because only a maximum of 8 workgroups can be scheduled on a Fermi GPU and also the amount of local memory is insufficient for having more than 8 workgroups in-flight. This occupancy problems is also a concern for the AMD and Intel GPUs, but due to their other architectural limitations less noticeable in their overall performance.

C. Performance of hybrid CPU-GPU decoder

In order to evaluate the performance of the decoder with motion compensation offloaded onto the GPU, four versions of decoder are developed, all configured with SIMD acceleration for CPU. The first one (CPU-BASE) is the baseline optimized CPU decoder; the second one (CPU-MC) is the CPU decoder with frame-level decoupling, which is used to estimate the losses of data locality; the third one (GPU-MC-S) offloads motion compensation onto GPU and executes the luma and chroma kernel in synchronous mode; the last one (GPU-MC-A) executes luma and chroma kernel in asynchronous mode to overlap the kernel execution of luma and memory copy of chroma. The GPU kernel with the faster single stage implementation is selected for this experiment. The breakdown of the decoding time is presented in Figure 8. *Others* includes H.264/AVC parsing and OpenCL environment building time (GPU version only). *Entropy* refers to the entropy decoding time. *MC* not only include kernel execution, but also the offloading overhead (memory copy and OpenCL runtime), as shown in decomposed *MC-kernel* and *MC-overhead* in version GPU-MC-S. *REC* represents the time for reconstruction and motion compensation time is excluded for decoders with motion compensation decoupled.

Because of decreased data locality, the performance of the decoder with motion compensation decoupled on CPU is lower, with up to 9% for 2160p on i7-2760QM. In addition to the reduced data locality when offloading the motion compensation kernel, the overhead the OpenCL runtime is the main source of inefficiency. Asynchronous execution is an effective optimization for Quadro. It outperforms synchronous execution by 4% and 8% at decoder level for 1080p and 2160p, respectively. Overall no significant speedup is achieved and often a slowdown is observed on the overall application performance when offloading the motion compensation to the GPU.

VII. Conclusions

In this paper we evaluate the performance of offloading the H.264/AVC motion compensation onto GPUs. We presented several kernel designs to reduce the control divergence performance penalty. For real H.264/AVC videos, the kernel with more limited divergence mitigation actually outperforms the one with full divergence mitigation, because big blocks dominate in high resolution video.

Due to both architectural and OpenCL runtime limitations, however, the GPU solutions are still outperformed by the single-threaded highly optimized CPU SIMD decoder. Control divergence, in literature often described as the most limiting factor of GPUs, is not one of the limitations but can be mitigated with a proper motion compensation kernel design. Other obstacles towards accelerating parallel kernels that do not have highly computationally intensive floating calculations on the GPU, however still remain. The high OpenCL overhead

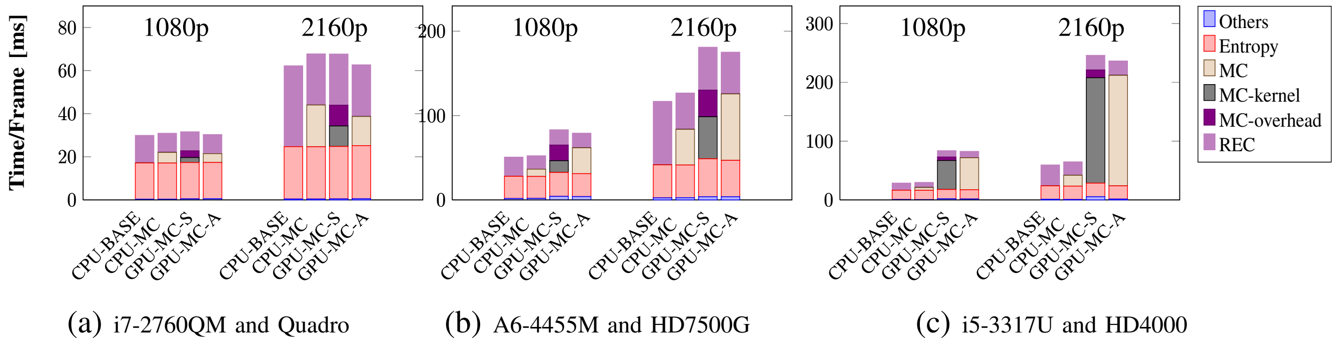

Fig. 8: Decoder time breakdown on three platforms

and low level GPU architectural limitations often are the main performance bottlenecks. GPU architecture, runtime, and programming models, however, are currently evolving fast and are expected to become more general purpose and more suited for video decoding applications.

References

[1] G.J. Sullivan and T. Wiegand, "Video Compression - From Concepts to the H.264/AVC Standard," *Proceedings of the IEEE*, vol. 93, no. 1, pp. 18–31, jan. 2005.

[2] NVIDIA, "NVIDIA CUDA C Programming Guide 4.2," http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[3] Khronos OpenCL Working Group, "The OpenCL Specification 1.1," http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf

[4] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "A Performance Characterization Of High Definition Digital Video Decoding Using H.264/AVC," in *Proceedings of the IEEE International on Workload Characterization Symposium, 2005.*, 2005, pp. 24–33.

[5] G. Shen, G. Gao, S. Li, H Shum, and Y. Zhang, "Accelerate Video Decoding with Generic GPU," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 15, no. 5, pp. 685–693, 2005.

[6] B. Pieters, D. Van Rijsselbergen, W. De Neve, and R. Van De Walle, "Performance Evaluation of H.264/AVC Decoding and Visualization Using the GPU," in *Applications of Digital Image Pprocessing XXX, PTS 1 AND 2*, 2007, vol. 6696, pp. 69606–1–69606–13.

[7] B. Juurlink, M. Alvarez-Mesa, C.C. Chi, A. Azevedo, C.H. Meenderinck, and A. Ramirez, *Scalable Parallel Programming Applied to H.264/AVC Decoding*, Springer, June 2012.

[8] AMD, "ATI Streaming SDK OpenCL Programming Guide 4.2," http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OCL_Programming_Guide-2013-06-21.pdf

[9] "h.264 OpenCL decoder," http://www.aes.tu-berlin.de/menue/research/research/highperformancevideocoding/

[10] "Xiph.org," http://media.xiph.org/video/derf/

[11] "European broadcasting union," http://tech.ebu.ch/testsequences/uhd-1

[12] "x264," http://www.videolan.org/developers/x264.html, A Free H.264/AVC Encoder.

[13] A. Vaidya, A. Shayesteh, D. Woo, R. Saharoy, and M. Azimi, "SIMD Divergence Optimization Through Intra-warp Compaction," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, New York, NY, USA, 2013, ISCA '13, pp. 368–379, ACM.

[14] AMD, "AMD SDK 2.9," http://developer.amd.com/developer-preview

[15] Intel, "Taking Advantage of Intel Graphics with OpenCL," http://software.intel.com/en-us/file/webinar001-taking-advantage-of-intel-graphics-with-openclpdf