Motion Estimation for H.264/AVC on Multiple GPUs Using NVIDIA CUDA

Bart Pieters[a], Charles F. Hollemeersch, Peter Lambert, and Rik Van de Walle
Department of Electronics and Information Systems – Multimedia Lab
Ghent University – IBBT
Gaston Crommenlaan 8 bus 201, B-9050 Ledeberg – Ghent, Belgium

ABSTRACT

To achieve the high coding efficiency the H.264/AVC standard offers, the encoding process quickly becomes computationally demanding. One of the most intensive encoding phases is motion estimation. Even modern CPUs struggle to process high-definition video sequences in real-time. While personal computers are typically equipped with powerful yet cost-effective Graphics Processing Units (GPUs) to accelerate graphics operations, these GPUs lie dormant when encoding a video sequence. Furthermore, recent developments show more and more computer configurations come with multiple GPUs. However, no existing GPU-enabled motion estimation architectures target multiple GPUs. In addition, these architectures provide no early-out behavior nor can they enforce a specific processing order. We developed a motion search architecture, capable of executing motion estimation and partitioning for an H.264/AVC sequence entirely on the GPU using the NVIDIA CUDA (Compute Unified Device Architecture) platform. This paper describes our architecture and presents a novel job scheduling system we designed, making it possible to control the GPU in a flexible way. This job scheduling system can enforce real-time demands of the video encoder by prioritizing calculations and providing an early-out mode. Furthermore, the job scheduling system allows the use of multiple GPUs in one computer system and efficient load balancing of the motion search over these GPUs. This paper focuses primarily on the execution speed of the novel job scheduling system on both single and multi-GPU systems. Initial results show that real-time full motion search of 720p high-definition content is possible with a 32 by 32 search window running on a system with four GPUs.

Keywords: GPU, graphics hardware, H.264/AVC, job scheduling, motion estimation, CUDA, video coding

1. INTRODUCTION

Today, the high-pace business of the videogame market keeps pushing graphics card production and performance levels higher and higher. With memory bus widths of up to 512 bit, high speed memory chips delivering up to 141 gigabytes per second, and high processor clock speeds, graphics cards deliver specifications never seen in consumer-level hardware before. Graphics cards these days hold a Graphics Processing Unit, called the GPU. This is a powerful processor predominantly used in videogames to render realistic 3-D environments. Because of the very high parallelism and floating-point computational capability, performance levels are impressive. Along with these GPUs, the graphics bus, now PCI-Express version 2, has evolved to a high-performance bus capable of delivering eight gigabytes of data per second in full duplex, a number that approaches the speed of the system memory bus.

While the first graphics cards provided a fixed function pipeline with no programmability at all, today's cards can be programmed either using a 3-D graphics API with so-called shaders or by using a vendor-specific API. Shaders are small programs running on the GPU, processing geometrical data or rasterizing 3-D scenes. OpenGL and Microsoft's Direct3D are examples of 3-D graphics APIs. These APIs are difficult and not transparent in use for general purpose applications. Therefore, the vendor-specific APIs came as an answer to these so called General-Purpose GPU (GPGPU) applications. Examples are the ATI CTM (Close To the Metal) and NVIDIA CUDA (Compute Unified Device Architecture) platforms. These general-purpose computing platforms allow users to directly address the GPU, without the need for extensively rewriting the algorithms for a graphics context.

[a] E-mail: bart.pieters@ugent.be, Telephone: +32 9 33 14983, Fax: +32 9 33 14896

H.264/AVC[1] is the latest international video coding standard developed by the Joint Video Team (JVT) and successor to MPEG-2 Video and MPEG-4 Visual. The standard comes with a set of new tools to enable high compression efficiency. These tools include tree-structured Motion Compensation (MC), multiple reference pictures, and quarter-sample motion compensation accuracy. To achieve a high compression efficiency using these tools, the encoding process becomes computationally-demanding, making it difficult on modern CPUs to encode in real-time[2].

A number of publications have targeted the powerful GPU architecture to accelerate motion estimation[3-9]. Most of these address the GPU using a 3-D graphics library such as OpenGL. Because of this, algorithms need to be translated to 3-D operations such as rendering a vertex grid[4]. While functional, the limited flexibility of the OpenGL library in the context of General Purpose GPU applications does not allow straightforward and flexible implementations.

Furthermore, except for Chen *et al.*[6], no system targets the H.264/AVC macroblock partitioning scheme. It is up to the CPU to determine the best partitioning. However, local best motion vectors for 4x4 blocks cannot simply be merged to determine the best motion vector for the parent macroblock or macroblock partitions. Therefore, a partitioning scheme has to be included in the actual search, and therefore, executed by the GPU. Also, all implementations target full search. No early-decision behavior is supported when a motion vector with low cost is quickly found. This causes coding complexity to be fixed. Another feature of the H.264/AVC standard, motion vector prediction, is also not considered in the current state-of-the-art.

Additionally, to the best of authors' knowledge, no GPU-enabled motion estimation algorithm inherently supports multiple GPUs. Latest developments show more and more systems having two GPUs: an on-board GPU integrated in the mainboard, and an additional discrete GPU. Furthermore, some state-of-the-art graphics cards are actually two GPUs on one board. Finally, because a computer system can be far more easily expanded with additional GPUs than CPUs, multi-GPU systems become an interesting computing platform.

With multiple GPUs, possibly each having its own processing speed, it becomes important to be able to steer these GPUs in a flexible way. While a discrete state-of-the-art GPU could be able to process two slices real-time, an on-board device could hardly be able to finish a single slice in time. Therefore, the platform must be able to steer the arithmetic intensity of the motion estimation process per device.

In this paper, we propose a GPU-based motion search architecture designed for encoding H.264/AVC video bitstreams. We essentially introduce a motion estimation algorithm using a GPU job scheduling system capable of performing motion search and macroblock partitioning for encoding an H.264/AVC bitstream on one or more GPUs. The architecture uses the recently introduced NVIDIA CUDA programming framework to steer the GPUs.

The remainder of this paper is organized as follows. Section 2 provides an introduction to the H.264/AVC standard with an emphasis on the encoding phases we plan to execute on the GPU. The NVIDIA CUDA platform is introduced in Section 3. Section 4 presents our GPU-based motion estimation platform based on our job scheduling system. Section 5 extends the previous section by discussing the addition of multi-GPU support to the platform. In Section 6, initial performance measurements are compared and discussed. Finally, Section 7 concludes this paper and addresses future work.

2. H.264/AVC MOTION ESTIMATION OVERVIEW

In this section, a short introduction to some of the coding tools available in the H.264/AVC standard is given, emphasizing the computational complexity of the respective encoder phases. For a detailed overview of the H.264/AVC standard, the reader is referred to Sullivan *et al.*[1].

2.1 H.264/AVC Overview

At the time of writing, H.264/AVC is the latest international video coding standard developed by the Joint Video Team (JVT), a collaboration of the Video Coding Experts Group (VCEG) of the ITU-T and the Moving Picture Experts Group (MPEG) of ISO/IEC. By using state-of-the-art coding techniques, the standard provides enhanced coding efficiency and flexibility of use for a wide range of applications. A trade-off between coding efficiency and implementation complexity was considered. The accomplished compression efficiency has seen an average improvement of at least a factor two compared to MPEG-2 Video[1]. However, the state-of-the-art coding techniques used in this standard require a significant amount of processing power[2,3].

2.2 Slice Prediction Modes

H.264/AVC divides pictures into slices, which are areas of the picture that can be decoded independently. Slices contain macroblocks with different macroblock subdivisions, determined using three iterative steps (tree-structured MC). First, slices are divided into macroblocks of 16x16 pixels. These macroblocks can be further divided into macroblock partitions of 8x16, 16x8, or 8x8 pixels. Finally, 8x8 macroblock partitions can in turn be divided in sub-macroblock partitions of 4x8, 8x4, or 4x4 pixels. It is clear that this tree-structured MC increases coding complexity over MC techniques of previous video coding standards. For instance, a high-definition video sequence of 1920 by 1080 pixels may consist of 129,600 sub-macroblock partitions, which all have to be predicted. Decoding is done on a macroblock-by-macroblock basis, where the decoding process of a macroblock is dependent on previously decoded macroblocks. This is reflected in the encoder, where typically each macroblock in a slice is handled serially.

To eliminate redundancy, macroblock sample values can be predicted either by intra or inter prediction. Intra prediction predicts sample values by using neighboring samples from previously-coded blocks in the current slice, located to the left and/or above the block that is to be predicted. Macroblocks predicted using intra prediction are called I macroblocks.

Inter prediction uses decoded pictures present in the Decoded Picture Buffer (DPB) to predict macroblocks. A first possibility is to predict the macroblock based on one or two of up to 16 previously decoded reference pictures. An index is used to choose the desired reference picture from the Reference Picture List (RPL), together with a motion vector to retrieve the correct prediction. Each macroblock or macroblock partition may have its own reference picture. Sub-macroblock partitions inherit the reference picture index from its parent macroblock partition. The macroblock predicted with this type of prediction is called a P macroblock. A slice containing only P and I macroblocks is called a P slice. A second possibility is to predict the macroblock based on the weighted average of two motion-compensated prediction values. Slices that contain macroblocks with this type of prediction are called B slices. MC can be performed at integer pixel level and sub-pixel level (e.g., half-pel and quarter-pel), hence different pixel interpolation strategies are needed. For instance, the calculation of half-pels uses a 6-tap interpolation filter.

To eliminate redundancy between the motion vectors of successive macroblocks, motion vectors in H.264/AVC are predicted using up to three previously decoded motion vectors from surrounding blocks, as shown in Figure 1. Because of this, macroblocks are interdependent and the encoding process needs to be completed in raster-scan order.

2.3 Motion Estimation

The sheer possibilities and number of inter prediction modes prove the flexibility of the H.264/AVC standard. However, this flexibility also introduces additional implementation complexity and extra requirements concerning processing power. To achieve an optimal encoding rate, each partitioning type has to be handled, all sub-sample motion paths have to be searched, and this process is to be repeated for all reference frames. Furthermore, inter macroblock dependencies prohibit efficient parallel encoding of macroblocks disabling the use of multiple threads per slice. With the increasing popularity of high-definition video sequences, computational complexity even further increases.

Various authors[2-5] conclude that motion estimation is the most demanding of all encoding tools. This is because of the large number of motion vectors that need to be searched within the search window. For each motion vector, the cost of that vector is determined using either the Sum of Absolute Differences (SAD) or another cost function. For a search window of 128 by 128 samples, this means that the cost function is used to compare each of these 16,384 positions to the current macroblock. Enabling 16 possible sub-sample positions per vector increases the number of comparisons by a factor of 16.

Advanced search algorithms are typically used to limit search positions, possibly combined with early-decision or early-out. In a typical video sequence, it can be seen that lots of motion vectors are predictable, either pointing to the current position plus a very small vector (e.g. {[-1,1],[-1,1]}), or a vector following the motion of the picture content according to previously found motion vectors. In order not to waste any processing power, motion search can best be started at these positions. When the cost of the found motion vector is lower than a certain threshold, the motion search can be aborted for that block.

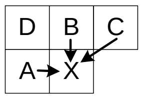


Figure 1: Motion vector prediction using motion vectors of surrounding blocks A, B, and C. If C is not available, D is used.

3. INTRODUCTION TO THE NVIDIA CUDA PLATFORM

Our GPU-driven motion estimation platform for H.264/AVC encoding relies on the NVIDIA CUDA computing architecture to steer the GPU. CUDA is a new computing architecture from graphics card manufacturer NVIDIA, targeted at using the GPU to accelerate the execution of computational problems. It enables the user to deliver a large amount of computations to the GPU, which is seen as a dedicated super-threaded co-processor in the CUDA programming model.

3.1 NVIDIA CUDA Programming Model

In the NVIDIA CUDA computing architecture[10], the GPU, which has its own DRAM, is viewed as a dedicated co-processor to the CPU or *host*. In this model, the GPU runs many jobs in parallel, called *threads*. Though the name suggests a similarity with traditional CPU threads, there are a number of differences, the most important one being the lightweight nature of the GPU threads. The GPU can manage a significant number of such threads simultaneously because GPU threads come with little overhead (i.e., little creation time). Consequently, context switching can be done fast. This enables the GPU to execute thousands of threads in parallel while the CPU typically processes only a dozen of threads, not necessarily in parallel. Consequently, to fully utilize the power of the GPU, the user has to provide a sufficient number of jobs to be processed in threads.

To use the GPU in calculations, data-parallel portions of an application are executed on the device as *kernels* which run many threads in parallel. NVIDIA has chosen to structure and group together threads in entities called *blocks*. A kernel is executed as a *grid* of thread blocks. Figure 2(a) shows this model. All threads in a thread block share data through low-latency shared memory and can cooperate by synchronizing execution with each other. Threads from two different blocks cannot communicate with each other.

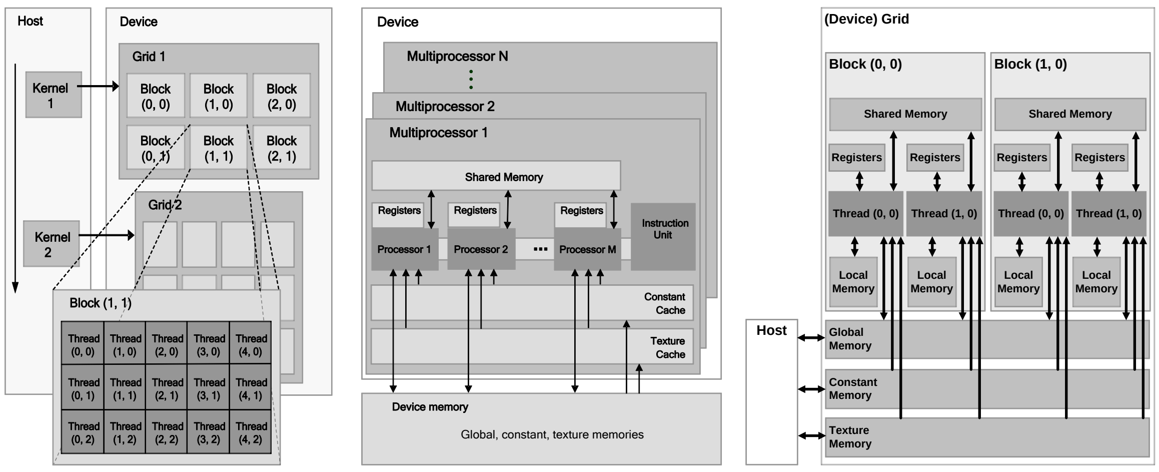


Figure 2: From left to right as stated in the CUDA Programming Guide[10]: (a) CUDA Programming Model; (b) CUDA Hardware Model; (c) CUDA Memory Model.

3.2 NVIDIA CUDA Hardware Model

Figure 2(b) shows the NVIDIA CUDA hardware model. The underlying hardware model differs from the programming model. Here, a number of multiprocessors are available, each having a fixed number of processors. Each multiprocessor is characterized by a Single Instruction, Multiple Data architecture (SIMD). Each processor of the multiprocessor executes the same instruction at a given clock cycle, but operates on different data. This concept is known as lock-step execution. Consequently, when different processors follow different execution paths, processors not following the same branch are suspended. Different execution paths are consecutively executed until the threads can converge back to the same execution path. This means control flow instructions, and therefore heterogeneous program execution paths, have to be avoided or an optimal computational throughput cannot be reached.

Each multiprocessor executes one or more blocks using time slicing. On a multiprocessor, each block is split into SIMD groups of threads called warps. At any given time, a single warp is executed over all processors in the multiprocessor. A thread scheduler periodically switches from one warp to another. This happens in a smart way, for example, when all threads from the warp execute a memory fetch and have to wait several clock cycles for the result. For the GeForce 280 GTX graphics card, the warp size is 32. This implies that, at any given time, 32 threads are executed simultaneously on a single multiprocessor. The GeForce 280 GTX has 30 multiprocessors. Consequently, at a given time, 960 threads are executed in parallel.

3.3 Memory Model

Figure 2(b) and Figure 2(c) show the CUDA memory model from respectively the hardware model and the programming model viewpoint. This memory model complicates programming CUDA because of its many layers but increases flexibility. On a given multiprocessor, each processor or thread has a number of registers available. Shared Memory, a very fast type of DRAM, is shared between processors in a multiprocessor. This memory is typically 16 KB in size. Global Memory is high-latency memory, available from all threads, but it is not cached. Constant Memory is a portion of the Global Memory to store constants accessible from threads. Texture Memory is a high-latency memory with some extra features specifically designed for texture filtering. Texture Memory offers for example bilinear filtering, as well as automatic texture address cropping. Both Constant Memory and Texture memory have a small cache enabling faster memory access for recurrent memory locations. It is up to the user to select the most appropriate kind of memory. CUDA supports atomic operations since architecture version 1.1, enabling inter-thread global memory access without introducing race conditions.

3.4 Addressing Multiple GPUs

Addressing multiple GPUs using NVIDIA CUDA can be done using dedicated run-time API calls. Multiple host threads are set up, with each thread launching its own CUDA kernel using its own CUDA context. Using the cudaSetDevice()-command, a GPU can then be selected and data can be communicated per host thread. Unfortunately, the current NVIDIA CUDA implementation (version 2.1) cannot copy GPU memory directly over different graphics cards.

4. SINGLE-GPU MOTION SEARCH USING CUDA

In this section, first, the general methodology of the architecture is discussed. Then, the proposed motion search architecture is described for use with a single GPU. A new job scheduling system is proposed for use with the NVIDIA CUDA platform, as well as a way to perform macroblock partitioning in the CUDA threads.

4.1 Methodology

Because they are independent of each other, slices can be processed in parallel in a GPU thread. However, because typically a low number of slices are used in a video picture, concurrency is low, and therefore, execution speed on CUDA hardware. In order to exploit the available GPU processing power, multiple macroblocks need to be processed in parallel per slice. However, because of inter-macroblock dependencies, drift can be introduced this way. Therefore, the proposed motion search architecture processes macroblocks serially in the CPU thread, but uses data calculated in advance in parallel on the GPU. Before starting the encoding process of a video picture, the input video sequence is analyzed and the results are passed to the encoder. By isolating bottleneck calculations and doing these calculations in advance in parallel, faster encoding times and/or better rate distortion can be achieved. Special care needs to be taken that the pre-calculations do not introduce drift at the macroblock encoder. For motion estimation, possible pre-calculations include motion search, macroblock partitioning, and motion vector cost calculations.

Figure 3 shows this method. Here, three CPU or host encoding threads encode three slices in parallel. Within each host thread, macroblocks are processed serially using the available motion information provided by the GPU motion estimation thread or device thread, running parallel to the host threads. The device thread will be implemented using NVIDIA CUDA kernel operations. Each CUDA thread will handle a certain amount of macroblocks of the video picture.

4.2 Scheduling System

Note that the motion information provided by the device controller thread in Figure 3 is only available when the device thread has finished, as a CUDA kernel finishes when all CUDA threads of that kernel are finished. Because of the hardware overhead of initializing the GPU, it would be inefficient to start a motion search for each macroblock. Then again, scheduling all macroblocks on the hardware could mean that the host thread is waiting while encoding the first macroblock, and precious CPU resources are not used. For this reason, it can be useful to enforce a certain order of calculations in the GPU thread. Furthermore, to ensure real-time processing, flexibility increases if the motion search can iterate from fast, but inaccurate, to slow but complete results, depending on the available time. For these reasons, we developed a scheduling system for motion estimation on top of the NVIDIA CUDA platform.

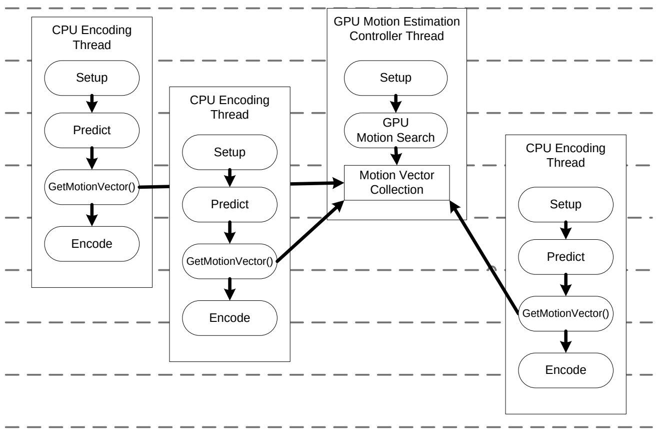


Figure 3: CPU and GPU encoding threads.

One important target of establishing a processing order is to use an efficient early-out strategy as explained in Section 2.3. The early-out behavior partially conflicts with the CUDA architecture. In CUDA, the hardware is occupied until all threads of the current kernel are finished. This means that the motion search for a set of macroblocks takes as long to process as the slowest search. So while all other threads are finished already with highly-predictable motion vectors, the hardware is still occupied for a few motion vectors. So in order to achieve an early-out behavior and make the results available for the host thread to process, the motion search is explicitly cut into several sub-parts.

To describe a sub-part, we introduce a description of a motion search task including its (intermediary) results. We call this a Motion Search Job or MSJ. This description can contain information about the macroblock position, the motion vectors already searched, the best motion vectors found so far, etc. Previous work in motion estimation on the GPU exploits the processing structure to describe the current work description for a certain pixel shader[3,7,8] or CUDA thread[6]. For example, the pixel position in the 3-D world defines the macroblock position. In our architecture, instead, we let each thread load the next MSJ, retrieve the process parameters from the MSJ, execute it, and update its status. This enables dynamic scheduling of jobs.

To enforce a processing order, for example to achieve an efficient early-out behavior, we schedule a number of MSJs on the hardware in a kernel, called an MSJ Batch or MSJB. Each MSJ in the MSJB will perform a part of the motion search. Next, the results of that MSJ batch can be downloaded to the CPU and used by the CPU encoder thread while the GPU in parallel is processing the next job batch.

Figure 4 shows this iterative process. In the figure, *n* job batches are scheduled. Each job iteration is processed and job descriptions are updated. Note that the same MSJ can be scheduled with different parameters multiple times in one batch, as well as over multiple batches. For example, Figure 5 shows how the task of searching the search window of one macroblock can be split over two MSJs. This process is repeated until all MSJs are done, i.e. found a good motion vector.

With this scheduling system in mind, Figure 3 can now be evolved to Figure 6. Here, each host encoding thread can use the motion vector collection currently presented by the GPU. For example, the third slice encoding thread can use the best motion vectors available. This is in contrast to the first and second encoding thread, which each use intermediary results.

4.2.1 Job loading and saving

Each thread can load its job in CUDA shared memory reserved for that thread. After the job is finished, the thread can either update its job status in the *busy list*, or write out the results in the *done list*. As the number of threads already in the busy list is unknown to each thread, special CUDA atomic operations are used to retrieve an MSJ busy list index. Even though atomic writes are slow compared to normal writes[10], the CUDA scheduler is still able to schedule additional ALU instructions in the mean time, hiding the large memory latency. To write results out in the done list, intermediate MSJ descriptions are dropped, and only relevant information is written out at a fixed position in global memory.

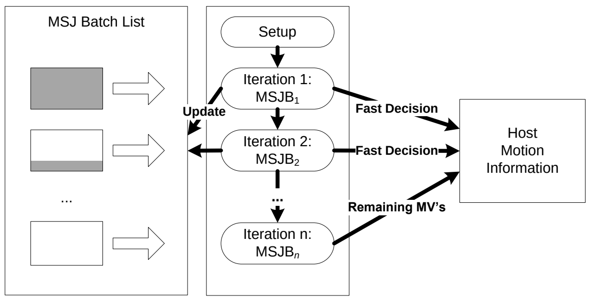


Figure 4: Iterative job scheduling process. A new MSJ batch is loaded and executed every iteration.

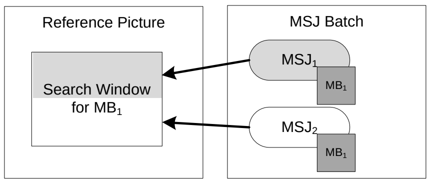


Figure 5: Two MSJs working together to process the search window of a single macroblock.

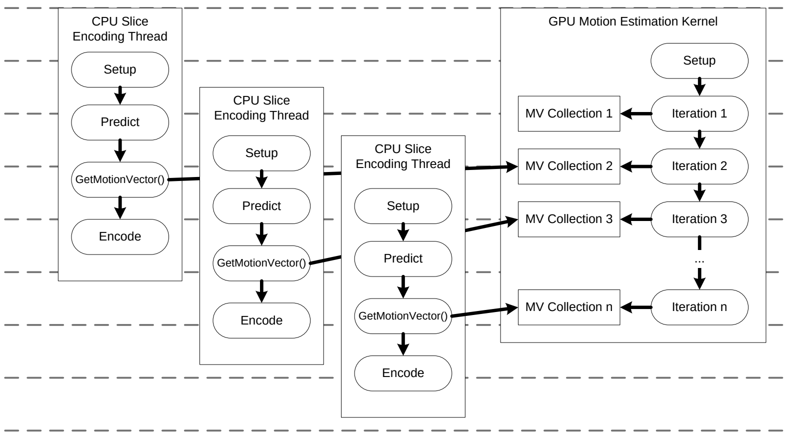


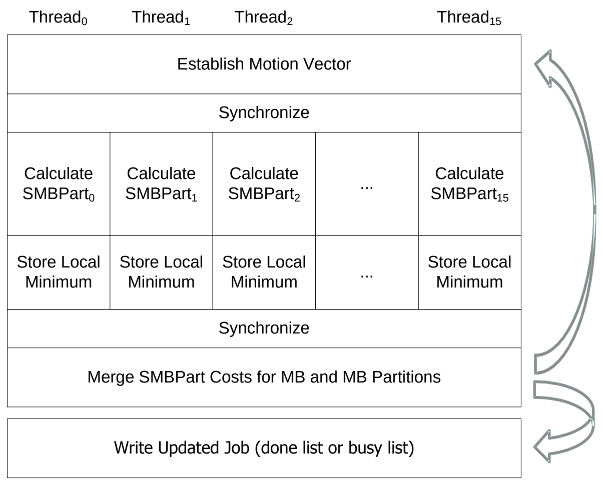
Figure 6: CPU encoding threads using each the latest available MSJ batch results from the GPU motion estimation kernel.

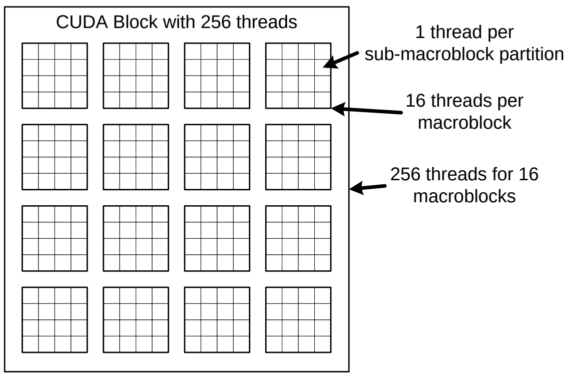
4.3 Macroblock Partitioning

As explained in Section 2, H.264/AVC uses tree-structured motion compensation where every macroblock can be divided into a number of macroblock partitions. The macroblock partitioning with the lowest cost is chosen by the encoder and coded in the output bitstream. For optimal results, all macroblock partitions need to be considered. Calculation of the partition costs by the CUDA threads is described in the following paragraph.

Each thread processes the smallest unit possible, a 4x4 sub-macroblock partition. This way, each macroblock is processed by 16 threads. In order to calculate the costs of the different partitions of the macroblock, the threads work together by sharing memory and synchronizing their execution. This is shown in Figure 7. Threads are synchronized after establishing the next potential motion vector. Each thread starts the motion search for its sub-macroblock partition and updates its local minimum cost and matching motion vectors. Next, threads are synchronized again and the different costs per sub-macroblock partition are merged to get the cost of the merged partitions, e.g. an 8x16 block. Note that though this merge action is required on a macroblock basis, calculations are spread over the 16 threads to achieve the most efficient arithmetic throughput. Next, the process can start over for the next motion vector. When a good motion vector was found, or the maximum number of iterations per thread was reached, the MSJ is updated and written to either the *done* or the *busy* list.

A motion search for one macroblock is described in one MSJ. This way, 16 threads work together to execute one MSJ. Once the entire part of the search window prescribed by the MSJ is done, the result is written to the busy or done list. Load and store operations to global memory are spread over the 16 threads. Because of this, CUDA's coalesced memory access feature is enabled. When CUDA threads each access a memory word on successive addresses, the memory words can be loaded from global memory using a single instruction. NVIDIA calls this memory coalescing and it allows up to a factor 4 more memory throughput, according to NVIDIA[10]. A CUDA block now contains 256 threads laid out in a pattern shown in Figure 8. This way, each CUDA block processes 16 MSJs of the current MSJ batch, representing 16 macroblocks laid out in a 4 by 4 pattern.





CUDA Block with 256 threads
1 thread per sub-macroblock partition
16 threads per macroblock
256 threads for 16 macroblocks

Figure 7: Sixteen CUDA threads work together to calculate the total cost of the macroblock and its partitions.

Figure 8: CUDA block mapping.

4.4 Multiple reference frames

By using a three-dimensional CUDA texture, a set of 2-D textures can be addressed using a reference picture index. This index is included in the MSJ description. This way, searching in multiple reference frames can be easily scheduled, either in one iteration, or, for example to respect real-time constraints, after the first few iterations on the first reference frame. This way, the more time available or the faster the encoding process is, the more reference frames can be tested.

4.5 Motion Information Cost Function and Motion Vector Prediction

The current implementation uses the Sum of Absolute Differences (SAD) as a distortion measure, due to its simplicity. Because of the modularity of the code, different cost functions can be implemented and inserted in the platform, using limited motion vector information from surrounding macroblocks.

In H.264/AVC, the motion vector for each block is predicted using three previously decoded motion vectors from surrounding blocks, as stated in Section 2.2 and shown in Figure 1. Therefore, the cost of a motion vector depends on the similarity of it with the motion vector predictor. Because each macroblock is processed in parallel, surrounding macroblocks *A*, *B*, and *C* in Figure 1 are not available. Therefore, when possible, a rough estimation of the motion vectors used in (macro)blocks *A*, *B*, and *C* is used.

At a given time, the best motion vectors since the last iteration of blocks *A*, *B*, and *C* are available. To access information over CUDA block boundaries, global memory write, read, and synchronization operations are necessary, making the rough estimation potentially slow. Because of this, only surrounding motion vectors within CUDA blocks are considered. As race conditions can occur when threads start reading shared memory of surrounding threads, synchronization operations are needed. However, because it is already a rough estimation, the impact of race conditions is believed to be minimal so synchronization operations can be omitted. In a worst case scenario, the prediction uses a mix of the previously best motion vector and the newly found best motion vector of macroblocks *A*, *B*, and *C*. Note that to avoid race conditions within motion vector components, the two vector components are written as a single 32-bit word.

4.6 Sub-sample Interpolation

Our current implementation only supports full sample motion search. A possible way to extend the motion search to sub-sample positions is described here. Upscaling the reference picture in advance of the motion search is a possible solution to introduce sub-sample precision. However, it is clear to see that because of increased search window size, calculations and memory requirements, this solution would perform poorly. Therefore, a sub-sample refinement pass seems more applicable to the search platform. This sub-sample refinement process starts with the current job list, and attempts to improve the found motion vectors by trying all sub-sample positions surrounding the motion vectors. In order to limit calculations, only sub-sample positions following the motion vector path are looked at.

4.7 Search and Scheduling Strategies

Multiple search patterns are possible, including raster-scan search, spiral search, diamond search, log search, etc. In the current implementation, spiral search[11] is used to achieve a full search with an efficient early-out behavior.

Because of the flexibility of our MSJ scheduler, multiple scheduling strategies can be used in order to achieve different encoder behavior. The early-out method can be used to enforce certain real-time constraints. By splitting the motion search in multiple MSJ batches, each host encoding thread in Figure 6 can access the currently available motion information results from those batches. While the first CPU encoding thread in Figure 6 only has collection 2 available at processing time, the second encoding thread can use the updated motion information from motion information collection 3. This way, the more processing time available, the better the next motion vectors can be.

An alternative scheduling can prioritize certain macroblock calculations as shown in Figure 9. Here, the search window of the top-left macroblocks of the slice are split over two MSJs each. This way, after the first iteration, these macroblocks will finish relatively quickly, while preserving a high hardware occupancy using the remaining MSJs in that batch. The next iteration, the remaining MSJs get full priority. Using this scheduling scheme, macroblocks located in the top-left of the slice, i.e. the macroblocks first encoded by the host encoder thread, have priority.

When using multiple reference pictures, another strategy can choose to search a new reference pictures with each MSJ batch. The more processing time available, the more batches executed, the more reference pictures are searched.

It is suggested by Zhao[3] that encoding macroblocks in a wavefront pattern instead of using raster-scan order preserves macroblock dependencies. This processing order can again be reflected in the MSJ batch, prioritizing certain waves in each MSJB.

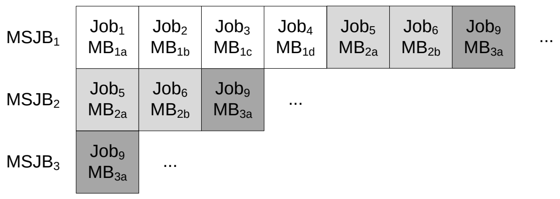


Figure 9: Using the MSJ scheduling system to enforce a calculation order.

5. MULTI-GPU MOTION SEARCH USING CUDA

The proposed job scheduling scheme presented in the previous section was designed with flexibility in mind, including addressing multiple GPUs to perform the motion search. Jobs can be scheduled on a specific GPU by extracting the required number of jobs from the host job list, uploading the jobs to the specific GPU, and executing the motion search. Each GPU on its own can then process the job list as described in the previous section. Figure 10 shows this for *n* GPUs. After each GPU has processed one MSJB from its part of the job list, the intermediary results can be downloaded from that GPU back to system memory. The more time available, the more MSJ batches can be processed on a single GPU (*m* in the figure). Naturally all GPUs require (part of) the current picture as well as (parts of) the reference pictures.

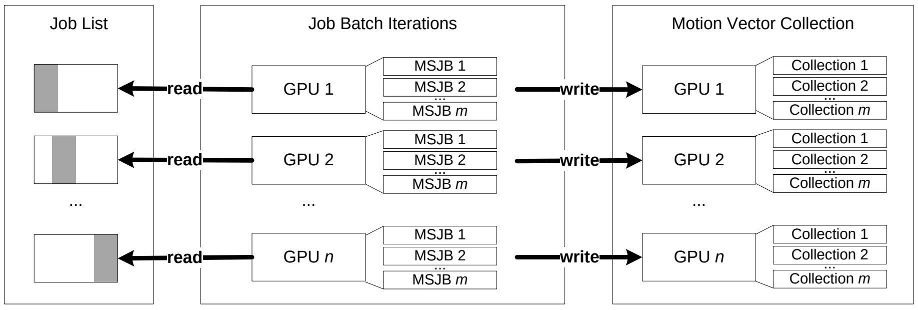


Figure 10: Scheduling the host job list on multiple GPUs.

5.1 Multi-GPU strategies

Multiple strategies are possible when addressing *n* GPUs, depending on the number of reference pictures, the real-time constraints, bus communication costs and GPU performance speed. The following subsections address two possible multi-GPU job scheduling strategies.

5.1.1 High Reference Picture Count

When the number of reference pictures is greater than or equal to the number of GPUs, it becomes interesting to use every GPU to process one or more reference pictures. The process duplicates the MSJs for each reference frame, sets up the proper reference picture index, and uploads the MSJ lists to the appropriate GPUs. Next, the motion search is started for all available GPUs. Each GPU will search its own reference picture.

Communication costs for all GPUs include the current picture, and a number of reference pictures equal to *referenceCount*/*n*. Once a picture is encoded by the host thread, that picture is uploaded to one GPU. Total communication costs for a single 1080p video picture searched with 4 GPUs is 10MByte for all GPUs.

5.1.2 Low Reference Picture Count

When the number of reference pictures to search is smaller than the number of GPUs, each GPU is used to process a part of the current picture or slice. The host MSJ list is divided into *n* parts and each part is uploaded to a GPU.

Communication costs for all GPUs include *1/n*th the current picture, and all reference pictures. For a 1080p video picture searched with 4 GPUs, this makes a total of 4MByte per frame for all GPUs.

In order to maintain the real-time constraints, jobs can be placed in an interlaced pattern in the device job list, shown in Figure 11. This way, the raster-scan encoding order used by the host thread is reflected as much as possible by the device processing order. This way, the first job iteration process can complete the macroblocks of the upper-left region of the video picture, while the remaining iterations gradually complete the job list.

	GPU_1	GPU_2	GPU_3	GPU_4
$MSJB_1$	Job_1 MB_1	Job_2 MB_2	Job_3 MB_3	Job_4 MB_4
$MSJB_2$	Job_5 MB_5	Job_6 MB_6	Job_7 MB_7	Job_8 MB_8
$MSJB_3$	Job_9 MB_9	Job_{10} MB_{10}	Job_{11} MB_{11}	Job_{12} MB_{12}

Figure 11: Job interlacing pattern example for processing 12 macroblocks on four GPUs.

6. RESULTS AND DISCUSSION

This section presents preliminary results of the motion search and partitioning platform.

6.1 Job scheduling system overhead

As can be expected, the job scheduling system introduces additional overhead as each MSJ has to be loaded from CUDA global memory into CUDA shared memory, and the updated MSJ needs to be written to CUDA global memory again. These writes put additional strain on the GPU memory bus. Also, the required shared memory is not available anymore for other threads, limiting the amount of CUDA blocks scheduled on a CUDA Multiprocessor at each time, therefore limiting the effects of the CUDA hardware scheduling.

The overhead can be lowered by minimizing the amount of MSJ batches and letting each job iteration search as great a portion of the search window as possible. For example, if a search window of 128 by 128 samples is considered and no early out, a total of 16384 bytes are read in the current picture per macroblock for the motion search, 429 bytes are read for the job description, and 429 bytes are written to update the job description. The extra memory overhead in this case is 5%. In case multiple reference pictures are searched in the same job description, the overhead percentage further drops to for example 1.25% for four reference frames.

If desired, additional macroblock partitioning schemes can be dropped, so the job description takes less bytes, and therefore, requires less memory bandwidth, reducing the scheduling overhead.

With each MSJB to process, the overhead further increases. Yet because results are available sooner, the CPU thread is not stalled. Therefore, the accomplished flexibility compensates the additional introduced overhead. Future work will focus on proving this once the motion search platform is integrated in an H.264/AVC encoder.

6.2 Single-GPU results over multiple iterations

Figure 12 shows the preliminary results for a motion search and partitioning process using three configurations, respectively using one MSJB with early-out disabled, two MSJBs with early-out disabled, and two MSJBs with early-out enabled. For the test, the job processing time in frames per second (fps) was measured including uploading to and downloading from the GPU. The search window was set to 32x32, and macroblock partitioning includes macroblock level or sub-macroblock partition level. Early-out behavior was enabled with a minimum SAD threshold of 30. Reference picture count was set to 1. A computer system with an AMD Phenom 4450 and a NVIDIA GeForce 280 GTX was used.

The chart shows that splitting the motion search over two MSJ batches, lowers the results from 101 fps to 96 fps. This means that while the GPU roughly needs the same time to process the results, the H.264/AVC encoder has the results from the first batch already at its disposal after the first batch finishes. Therefore, the CPU encoding thread can start processing immediately.

The results show the early-out mechanism to work, raising the processing speed from 101 to 119 fps when enabled. Further testing shows that the more batches are scheduled, the more impact the early-out mechanism has, therefore, making it useful for slow-performing hardware.

The results presented here are similar with the results presented by Chen *et al.*[6] for motion search and partitioning. The authors claim to reach a frame rate of 15.4 (4x4 SAD calculation plus variable block size generation) on an older GeForce 8800 GTX. Yet, because of the early-out behavior, our results are available sooner for the CPU encoding thread to process. The increased flexibility of the motion search platform also means that the CPU encoding thread can enforce its calculation order, which can result in better motion prediction and therefore better rate distortion. When early-out is disabled, the system offers true full search, where not only local best motion vectors for sub-macroblock partitions are merged, done by Chen *et al.*, but the SAD's of macroblock partitions are calculated for all possible motion vectors.

6.3 Multi-GPU results

Figure 13 shows the preliminary results for a motion search and partitioning process using the same computer configuration but with four NVIDIA GeForce 280 GTX graphics cards. Here, a single MSJ batch was used to search one reference frame. Again, the job processing time was measured including uploading to and downloading from the GPU. Macroblock partitioning includes macroblock level or sub-macroblock partition level. Early-out was disabled. Three configurations were tested with a search window of respectively 32 by 32, 64 by 64, and 128 by 128. Each macroblock is represented by four MSJs, one per GPU.

The results show the execution times for the multi-GPU configuration to almost scale linearly with the available GPUs. Using a search window of 32 by 32, real-time motion search for high-definition 720p video sequence is achieved at 43 frames per second.

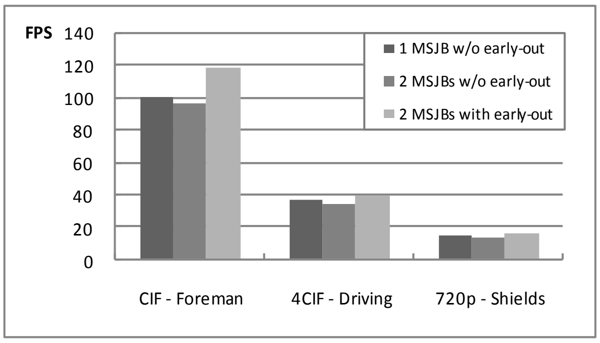


Figure 12: Results for a single GPU system using a GeForce 280 GTX card with a 32x32 search window.

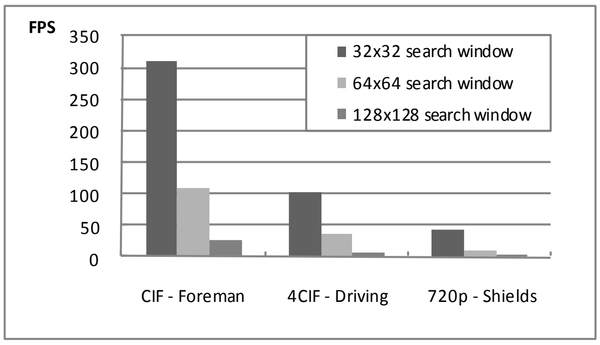


Figure 13: Results for a multi-GPU configuration using four GeForce 280 GTX cards.

7. CONCLUSION AND FUTURE WORK

A GPU-enabled motion search architecture was proposed capable of performing motion estimation and macroblock partitioning for H.264/AVC bitstreams. The architecture is built on the NVIDIA CUDA platform, and is capable of addressing multiple GPUs. To achieve this, a job scheduling system was introduced enabling early-out behavior on the SIMD architecture of the CUDA graphics cards. The job scheduling system shows increased overhead, yet, the available flexibility allows for multiple search strategies, including enforcing a calculation order, and using more than one graphics card. With this platform, real-time full motion search and macroblock partitioning for high-definition video (720p) on a PC with four NVIDIA GeForce 280 GTX graphics cards was accomplished using a search window of 32 by 32 (43 fps).

Future work will focus on optimizing the job scheduling system, adding sub-sample precision, and integrating the motion search platform in an H.264/AVC encoder. Next, the proposed scheduling strategies can be investigated and rate distortion ratios can be compared. Further work will also focus on investigating the feasibility of GPU-assisted encoding of scalable video content.

8. ACKNOWLEDGEMENTS

The research as described in this paper was funded by Ghent University, the Interdisciplinary Institute for Broadband Technology (IBBT), the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), the Fund for Scientific Research-Flanders (FWO-Flanders), the Belgian Federal Science Policy Office (BFSPO), and the European Union.

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, IEEE, 2005.
2. V. Lappalainen, A. Hallapuro, and T. D. Hämäläinen, “Complexity of Optimized H.26L Video Decoder Implementation”, IEEE Circuits and Systems for Video Technology, **vol. 13**, no. 7, pp. 717–725, 2003.
3. Z. Zhao and P. Liang, "Data partition for wavefront parallelization of H.264 video encoder", *International Symposium on Circuits and Systems (ISCAS 06)*, 2006.
4. C.-W. Ho, O. Au, S.-H. Chan, S.-K. Yip, and H.-M. Wong, "Motion Estimation for H.264/AVC using Programmable Graphics Hardware", *Proceedings of the IEEE International Conference on Multimedia Expo (ICME 06)*, pp. 2049-2052, IEEE, 2006.
5. F. Kelly and A. Kokaram, “Fast Image Interpolation for Motion Estimation using Graphics Hardware”, *Proceedings of SPIE Real Time Imaging VIII*, pp. 184-194, SPIE, 2004.
6. W.-N. Chen and H.M. Hang, “H.264/AVC motion estimation implementation on Compute Unified Device Architecture (CUDA)”, *International Conference on Multimedia and Expo (ICME 08)*, pp. 697-700, 2008.
7. C.-Y. Lee, Y.C. Lin, C.-L. Wu, C.-H. Chang, Y.M. Tsao, and S.-Y. Chien, “Multipass and Frame Parallel Algorithms of Motion Estimation in H.264-AVC for Generic GPU”, *International Conference on Multimedia and Expo (ICME 2007)*, pp. 1603-1606, 2007.
8. R. Strzodka and C. S. Garbe, “Real-Time Motion Estimation and Visualization on Graphics Cards”, *Proceedings IEEE Visualization 2004*, pp. 545-552, 2004.
9. Y.-C. Lin, P.-L. Li, C.-H. Chang, C.-L. Wu, Y.-M. Tsao, and S.-Y. Chien, “Multi-pass algorithm of motion estimation in video encoding for generic GPU”. *International Symposium on Circuits and Systems Conference*, pp. 21–26, no. 531, ACTA Press, 2006.
10. NVIDIA Corporation, “NVIDIA CUDA Compute Unified Device Architecture: Programming Guide”, version 0.8.1, April 2007.
11. C. Chok-Kwan and P. Lai-Man, “Normalized Partial Distortion Search Algorithm for Block Motion Estimation,” IEEE Trans. on Circuits and Systems for Video Technology, **vol. 10**, no. 3, pp. 417-422, 2000.