Collaborative Multi-Device Motion Estimation on Multi/Many-Core Platforms

Master Thesis

Author:
Alexandru Uta
Student Number: 2118475

Supervisors:
Frank Seinstra
Ben van Werkhoven
Anuj Ahuja (Hyves)

August 2012

Abstract

Social media platforms offer their users the possibility of sharing videos. After such videos are uploaded, the media platform has to encode them into a common format that can be played in a browser. Motion estimation has been identified as the most compute intensive process of the video encoding. In this thesis, we analyze and optimize a reference motion estimation method that is commonly used in the literature. This method is intended to run on many-core devices (GPUs). Moreover, we design our own Collaborative Method for motion estimation. We prove that the Collaborative Method is not only faster than the Reference Method, but it also allows the user to encode videos with better quality (expressed in PSNR values). This is achieved by eliminating the Reference Method's limitation in extending the search window size and allowing the user to choose arbitrary values for the search window size. Further, our Collaborative Method has been designed to make use of all compute devices available in the target system (CPUs, GPUs, accelerators). Multi-device scalability is also addressed and analyzed.

Contents

1 Introduction

In our days, the need for computation power that speeds up diverse processes is very high. High Performance Computing (HPC) techniques and algorithms are used nowadays in a broad range of domains of activity: speeding up complex scientific computations, in the academia, in software industry companies that have large distributed systems that need to be optimized etcetera.

One of the areas in which HPC can be employed is also video coding. This has proven to be a compute-intensive problem [1] as modern codecs imply repetitive time consuming operations to process all the frames in a video. The size of a raw video can reach the order of gigabytes, making it hard to stream over a network or even store multiple such files on a disk. Encoding achieves compression and can reduce the size of a raw, uncompressed video often by an order of magnitude. When analyzing such large amounts of data, HPC techniques could be applied to reduce the total processing time.

1.1 GPGPU Computing Motivation

In the last few years, general-purpose computing on graphics processing units (GPGPU computing) has shown that it can be a very useful tool in parallelizing certain applications by offering enourmous amounts of computing power, much higher than CPUs. The theoretical GFLOPs obtained using GPUs is often an order of magnitude higher than using CPUs. For example, Nvidia GTX 285 can leverage 1062.72 GFLOPs while the Intel Core i7 CPU can offer 109 GFLOPs. Obviously, not all applications are suitable for deployment on such architectures. Also, the main actor when it comes to developing HPC applications is the developer who should carefully analyze the problem that needs to be solved and choose an appropriate hardware for it. This is to be done taking into consideration that a theoretically faster hardware is not always better. What matters is the mapping between the problem to be solved and the underlying hardware architecture. Solving the problem on the specific hardware should allow obtaining near-maximal performance.

Since the development of Compute Unified Device Architecture (CUDA) [2] and OpenCL [3] much research has been done in this field and scientists tried to develop various applications for such many-core architectures. Moreover, optimizing these to achieve near-maximal efficiency for the targeted platforms still remains a very difficult task. Clearly understanding various features of the underlying architecture together with the performance bottlenecks determined by them is the key for producing an efficient application.

1.2 Project Motivation

A current trend amongst the social aspects of the Internet is that users upload, share, and watch movies on different platforms such as Hyves, Facebook and YouTube. Of course, for such media platform to be successful, it has to meet the immediate demands of the user: the media uploaded has to be readily available on the web as soon as

possible, thus offering low-latency, and at a similar quality to the one the user has recorded it. However, from the perspective of a provider of such services, this process is not as simple as just copying the file the user uploaded and serving it to the users. This is because the media file recorded by users can be encoded in multiple formats that are not supported for in-browser playback, i.e. they are not HTML5 compatible. Thus, the media provider will have to process the video uploaded by the user and transcode it to certain specific formats and resolutions that could be easily played in a browser. For example, such a provider may choose to encode all user uploaded video files to H.264 [1]. This is a relatively recent video compression standard that can achieve higher compression efficiency than previous video compression standards [1].

However, to achieve this compression efficiency, the encoding process has a very high complexity. Applying this is certainly not a trivial task, as it takes a lot of compute power to encode a video. This happens because, for example, in the motion estimation [1] phase of the encoding, a frame is split into NxN blocks and then, each block must be matched against each position of a reference search area (window) from another frame. It is worth mentioning that processing all these blocks can be done independently, in parallel. It would be very hard for a single modern CPU to get high performance while running such an algorithm because, due to its coarse-grained parallelism it would not be able to process a high amount of blocks in parallel. This problem, however, can be well-suited for employing GPGPU computing techniques to speed up computations [4]. Such platforms express massive fine-grained parallelism, making it possible to analyze a larger amount of blocks in parallel.

1.3 Research Questions

In this section we first present the research questions that are to be answered in this thesis. A discussion of these research questions is then presented for a better understanding of the general framework of the project.

1. How can one design a high performance, scalable, resource-friendly distributed media processing system?

2. How can one design a high performance video coding software solution that is able to split the compute kernel computation on all compute devices available on the underlying machine (CPUs, GPUs, accelerators)?

2.a How can one achieve high performance motion estimation by optimizing available solutions in the literature?

2.b Can the search window size parameter be modified without additional coding effort?

The extensive scope of this thesis is to create a system suitable for video processing (encoding, resizing) that is to be deployed in an industry environment. This system should help Hyves Netherlands [5] to meet the demands of their users as being a readily-available platform for sharing video content in an HTML5 format. This solution is needed because the company reports that in 2011, there was an average of 5600 videos uploaded per day, 233 videos per hour, while the average time for processing a single video was

20 seconds. Hence, for a better user experience, the average time for processing a video should be decreased as much as possible. Moreover, the average website load is reported to be much higher during holidays. We propose a scalable, maleable and resource-friendly high performance media processing system that could make use of varying amounts of computing power depending on the current load. Moreover, the software solution should be a general one, capable of running on a heterogeneous system and easily deployable on various platforms such as clusters, grids, clouds etcetera. When aiming at obtaining an HPC software that runs on various platforms, we consider OpenCL to be the most suited solution. This is because it has the capability of accelerating applications by making use of multiple hardware platforms: CPUs, GPUs, and other accelerators such as Cell B/E [6].

The core media encoding tool is enhanced using OpenCL capabilities to be able to run on various architectures (CPUs, GPUs). Moreover, if the system contains machines that are equipped with multiple OpenCL-capable platforms, then collaborative work must be supported. Hence, we will present our approach on creating an OpenCL-enabled framework that can accelerate the media encoding process by making use of all the compute resources available on a specific machine. Thus, CPU-GPU and multi-GPU collaborative work is investigated in this thesis. We provide a list of architectures that will be supported by our video encoding solution:

- single CPU
- single GPU (for these systems, a CUDA implementation is also presented)
- multi-CPU
- multi-GPU
- CPU-GPU systems.

The core work of this thesis is focused on achieving a high performance video encoding software solution. To define how an encoder/decoder algorithm works, we first present why video manipulation and processing is more difficult than image processing on which much research has been done and for which good high performance solutions exist [7]. One can see a video as a succession of frames (images). Subsequently, the first observation is that a video not only means more data than a single image, but moreover, a third dimension (time) is introduced, one of a different nature than the two spatial dimensions an image already expresses. Hence, one can deduce that frames exhibit temporal coherence - the majority of pixels in one frame are correlated to pixels in the previous frame(s). This means that a general video encoding algorithm may work by simply choosing some reference frames that are to be stored entirely while for the other frames the only data to be stored is the computed differences between them and the reference frames.

The above is, in fact, an informal definition of the motion estimation phase of an encoding algorithm [1]. According to [1, 8, 9, 10], motion estimation is the most time

consuming and compute intensive part of various encoding algorithms. Our work will focus on achieving a high performance motion estimation software module that could run on the list of architectures presented before. First, we will present our approach at applying various optimization techniques to improve a current solution available in the literature.

We noticed that the way in which this solution is designed and implemented for GPU platforms imposes a limitation in the "search window" size parameter. This limitation could lead to loss of quality for the encoded video. Increasing the value of this parameter is still possible using this method but will come with additional coding effort. Hence, we have created a "collaborative" method that does not limit the size of the "search window". Also, for our method, increasing the value of this parameter can be achieved without additional coding effort. Obviously, the drawback for increasing the search window size is the increase in the number of total computations needed. Hence, various platform specific optimizations are employed to reduce this overhead.

1.4 Contributions

In this thesis we present our approach at designing a multi-device collaborative motion estimation framework. The approach is capable of spreading the computation on multiple compute devices available on the underlying system. Moreover, we prove that our novel solution for motion estimation achieves better scalability and speed than existing solutions [8, 9, 10, 11]. In case of GPUs, for video resolutions higher or equal to 1280x720, the achieved speedup for our Collaborative Method is approximately 1.35. For CPU platforms the speedup is even greater, approximately 2.4, for the same frame sizes. Further, we show that our solution for motion estimation can easily extend the search window size. This parameter suffered a limitation in the existing solutions. By making use of this enhancement, we show that our collaborative method can achieve higher encoding quality which results as an important visual impact.

1.5 Thesis Outline

Section 3 presents necessary video coding concepts. In Section 2 we will present the overall system design. Section 4 presents an extensive description of modern GPU platforms and necessary notions for understanding implementation details. Section 5 presents the Related Work. Section 6 presents our approach to motion estimation between two frames, while Section 7 presents our approach to implementing a framework that supports collaborative multi-device motion estimation. In Section 8 we present our experiments and their results, while in Section 9 we draw the conclusions and present ideas for future work.

2 Video Coding Concepts

In this section we will present various notions regarding video coding and motion estimation. We start by formally defining the notion of digital video representation. According

to [1], this is a spatio-temporal sampling of a natural visual scene. The spatial sampling refers to representing a natural scene on a rectangular grid as a finite set of points (frame) that match as closely as possible the colour, brightness, shape and texture of the objects that appear in the scene. The temporal sampling refers to representing the natural visual scene as a set of frames sampled at regular time intervals. Playing back this set of frames produces the appearence of natural motion. Sampling at 25 or 30 frames per second is a standard for television broadcasting. However, higher frame rates improve the smoothness of perceived motion.

When representing a video frame as digital data two common systems are used: the RGB colour space and the YCbCr colour space. The RGB system involves representing the frame as three planes with different colour information: the red, green and blue planes. Hence, for each point (pixel) of the frame, three numbers have to be stored. The second colour space, YCbCr tries to adapt the digital representation of a frame to the human's visual system. This is less sensitive to colour that it is to brightness (luminance). Hence, this colour space uses four components for each pixel of an image. These are: the luminance component (Y), and the three chrominance (colour difference) components (Cb, Cr, and Cg). In order to convert from RGB to YCbCr, one can use the following set of equations:

$$\begin{aligned} Y &= k_r R + k_g G + k_b B \\ Cb &= B - Y \\ Cr &= R - Y \\ Cg &= G - Y \end{aligned}$$

Here, k_r, k_g, and k_b are constants that represent the weight of each color when luminance is computed. At a first glance, this system has no advantage over the RGB system because another plane is introduced in the frame representation and thus the more data needs to be stored. However, $Cb+Cr+Cg$ is a constant and thus only two of these numbers need to be stored or transmitted, while the other can easily be computed. In this way, the same amount of data is needed for both RGB and YCbCr systems.

Moreover, taking into account that the human eye is more sensitive to luminance rather than to colour, the amount of data needed for digitally representing a video frame can be further lowered. This is achievable by lowering the resolution for the Cb and Cr components by sampling them at lower resolutions than the Y component. There are several such sampling formats - 4:4:4, 4:2:2, 4:2:0. These numbers (except the last group) actually represent the ratio of Y:Cb:Cr samples. The 4:4:4 sampling format means that at each pixel position we keep a sample of each plane. The 4:2:2 sampling format uses less samples for the Cb and Cr components in the following way: for each four luminance components, there are stored/transmitted only two Cb and Cr components. The 4:2:0 format implies that for each four luminance samples, there is one Cb and one Cr sample. However, the numbers 4, 2, and 0 do not make sense and have been kept historically as a representation of the 4:1:1 sampling format.

Even using these lower sampling formats, storing a video or transmitting it over the network is still hard. For example, an HD video frame of resolution 1920x1080 with the 4:2:0 sampling is represented using 3110400 bytes (approximately 3 MBytes). Assuming a video of 1 hour at a frame rate of 25 frames per second, approximately 270 GBytes are needed to represent it digitally. Streaming such a video would necessitate a very high bandwidth, while storing many videos of this kind on a hard drive would be also difficult.

Data compression for video files is achieved using a video encoder. According to [1], a video encoder makes use of spatio-temporal redundancies of videos. Naturally, neighbouring frames of a video are correlated and exhibit multiple similarities. Thus, when encoding a frame, to reduce the data needed to represent it, only the differences between it and the previous frame(s) may be further stored. The spatial correlation intrinsic to a certain frame may be also used to reduce the actual size of a video file. This refers to the fact that neighbouring pixels often exhibit similarities, i.e. they have approximately the same value. For example, this could happen in a frame in which a large area represents the sky. Obviously, a large number of the pixels will have the same value.

According to [1], a video encoder consists of three modules: the temporal model, the spatial model and the entropy encoder. The input to the temporal model is an uncompressed video sequence (usually in YCbCr format). The temporal model reduces the temporal redundancies and constructs a prediction of the current frame. The output of the temporal model is a residual video frame which is created by subtracting the prediction from the actual frame and a set of motion vectors. This residual frame is used as the input to the spatial model which reduces spatial redundancy by making use of similarities between neighbouring pixels. The spatial model outputs a set of coefficients obtained by applying a transform (usually discrete cosine transform). These coefficients and the motion vectors are further compressed by the entropy encoder which basically removes statistical redundancy in the data and produces a bit stream. This final result is the actual encoded video.

Reportedly [1, 8, 9, 10, 11], the most compute intensive part of the temporal model is the motion estimation process. It accounts for 55%-65% of the total encoding time. This process tries to achieve temporal compression by predicting how certain objects from the current frame move in correlation to the reference frame. Achieving this at object level is not feasible because the encoder would first need to detect objects and then match them in the reference frame. Hence, motion estimation is done for blocks of pixels. The method is simple and straightforward: the current frame is split into MxN pixel blocks; each block is then matched against candidate blocks within a certain search window of the reference frame; after the best match is found, the motion vector (offset between current block and best match) is output.

The overall complexity of this block matching algorithm increases with the increase in the size of the search window. In videos which imply a lot of movement, large search windows are preferred because better matches for the current block may be found at higher distances. Finding a better match implies an increase in the quality of the com-

pressed video: when played back, the frame is reconstructed based on the motion vector for that match and the user could benefit from better visual experience. Moreover, the output size could be lower because the actual difference between the predicted block and the current block will be lower. Thus, the entropy encoder could encode it with shorter codewords.

Usually, to reduce the complexity of the encoding process, encoders choose to use smaller search window sizes. An alternative for reducing the complexity of motion estimation is to use algorithms that do not analyze the entire search space (search window). In this thesis, we reduce the computational complexity of the motion estimation process by parallelizing it on many-core platforms. This is achieved while not sacrificing the output quality by minimizing the search window size.

3 Overall System Design

One of the goals of this thesis is to create a system for the company Hyves Netherlands that would serve as a high performance media processing system. The video files uploaded by the users are to be processed by this system in the following way: each video file uploaded by a user is to be transcoded to a HTML5 compatible video format (i.e. H.264) and resized to a range of standard formats that are to be decided by the company. The formal requirements of such a system are:

- scalability with respect to the load
- maximize processing performance (responsible for this is the collaborative OpenCL-enabled framework presented in Section 7)
- maximize throughput
- minimize latency
- malleability
- resource friendliness - minimize power consumption

The current architecture that the company is using can be described in the following way: the company makes use of 16 media uploading servers; these servers respond to HTTP client requests to upload files; after a file is uploaded, the media uploader server transcodes this to a specific format and size and then stores it into a distributed filesystem.

In order to create a maleable, scalable and resource-friendly system that could make use of more/less computing power depending on the current load, we decided on using the follwing architecture: we keep using the 16 media uploader servers, but, instead of using their own CPUs for the compute intensive transcoding work, the media uploaders make requests to a queueing-server; the queueing system then stores the jobs; to this whole system we add "GPU-machines" that join/leave the system depending of the current

load and also make requests for getting jobs from the queue-server; after completing a job, the "GPU-machine" stores the results in a distributed filesystem. As we will further present in Section 6, we designed an encoding software capable on running on different platforms - multi-GPU, multi-CPU, etc. Thus, the work pool for the queueing system can be heterogeneous.

Below, in Figure 1, we depicted the chosen System Architecture. To make the system able to be power-efficient and also scalable, we designed this in an enhanced "Replicated-Worker" fashion. This means that there is not a fixed number of workers. Depending on the current load, processing "GPU-machines" can join and leave the system. This can be done by analyzing historic data about the user activity. Using this strategy works because it has been noticed [12] that for social-media websites, user activity tends to be clustered around certain moments of the day. Thus, one can pre-program these machines to join the system at predefined moments of the day. A benefit for designing such a system could be that it can easily function in an heterogeneous cloud environment. As a failsafe mechanism for having enough compute power at a certain moment of time, the following policies can be employed: if the job queue size is higher than a predefined value, the queueing-server requests that other GPU-machines join the work pool; if the job queue size is lower than a predefined value, the queueing-server requests that one or more GPU-machines leave the work pool to minimize the power consumption.

An alternative to this design could be employing a system where the job queueing-system is replaced by a "job-dispatcher" middleware. This would then not enqueue the jobs, but rather dispatch them to machines in the work pool. As a consequence, the machines in the work pool will have their own local queues to store the incoming jobs. Hence, the job dispatcher will act as a load balancer that uses a specific policy for scheduling the jobs to specific workers. For example, a simple round robin technique could be employed. Again, the job scheduler middleware will have the capability of using more/less machines in the work pool depeding on the system load. However, using such an approach with an heterogeneous work pool could imply the need of a more efficient scheduling - larger jobs should be sent to machines with more compute power (if available) to minimize latency. Of course, this issue still stands even for the other approach, where workers request jobs rather than waiting for them. But, this issue can be solved as follows: if a worker requests a job from the queueing-system and there are no jobs left, then it can query the other workers for uncompleted jobs. This "work-stealing" approach could offer the best performance results if a suitable way of splitting jobs could be found. [13, 28] show that "work-stealing" with job splitting achieves better performance than static load balancing algorithms that do not take into account fractions of jobs. Hence, if such a job sppliting technique is to be found, the first approach presented here is preferable.

For implementing the system depicted in Figure 1, one can think about multiple options. We consider that the Ibis framework [14] could be one of the most applicable solutions. Its design is aimed at simplicity of job execution, scaling on multiple nodes, even on multiple clusters/clouds/grids. Other options would be using an existing queueing system such as RabbitMQ that implements AMQP [15] or IBM WebSphere MQ [16].

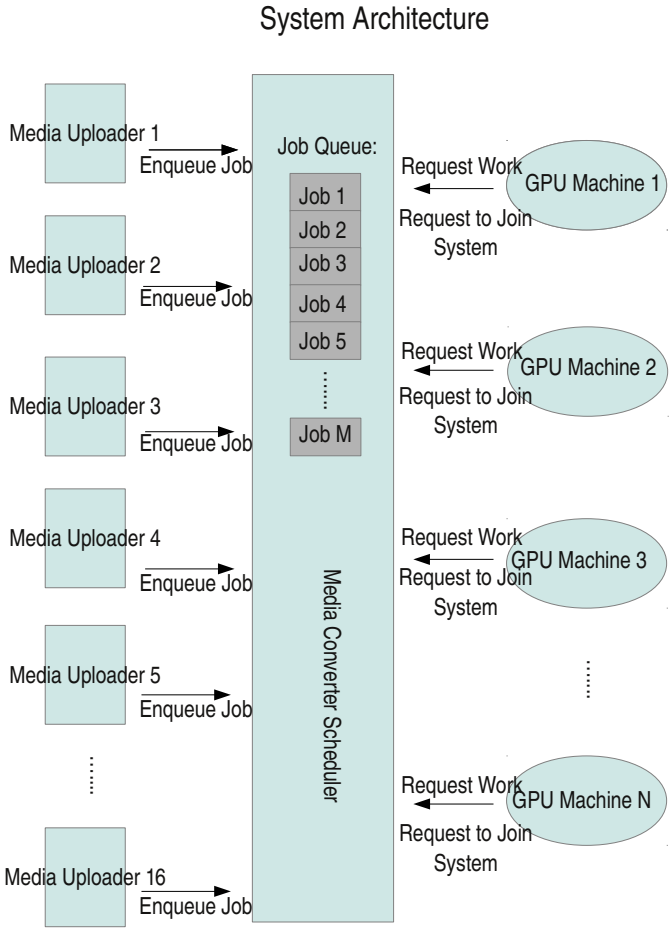


Figure 1: Overall System Architecture

However, making use of such industry queueing systems would make the developing process harder. This is because the aforementioned solutions only offer communication protocols between the available machines. Hence, another layer of software will be necessary to handle the joining/leaving the system by the "GPU-machines". With Ibis, this would not be necessary as the framework has the capability of keeping track of which machines joined/left the system and also offers easy to implement communication between nodes. In [29] such an approach is extensively presented and proven to perform well in systems where machines join/leave/fail. Nevertheless, in this thesis we will not focus on the development of such system due to time constraints. Our focus will be mostly on developing a scalable encoding framework that could work on various multi/many-core platforms.

4 Introduction to GPU Programming

The GPGPU programming model offered by CUDA is based on the SPMD (single process, multiple data) model, where a regular CPU acts as a host that offloads portions of the code to the GPU devices that act as coprocessors. The host can copy back and forth data to the device memory and call special operations specifically designed for the GPU (called kernels) such that the data transferred is processed by the device. This processing flow is visually depicted in Figure 2.

The CUDA toolkit offers the programmer a language that is an extension of ANSI C which permits writing a single program that encompasses both the host and the device code. An NVIDIA GPU consists of an architecture dependent number of streaming multiprocessors (SMs), each containing eight streaming processors (SPs). A continuous section of 32 threads residing on the same SM is called a warp. Threads in a warp execute the same instructions in parallel. As a hardware limitation, when threads in the same warp take different execution paths, multiple passes are necessary to complete the execution. As CUDA exposes a high-level programming language, it offers the programmer an abstract grouping of threads in a two-level hierarchy model. Thus, the user can define grids of thread blocks. Each block can run at most 512 or 1024 threads, depending on the specific device. Moreover, threads in the same thread block are executed by the same SM and are able to synchronize with each other.

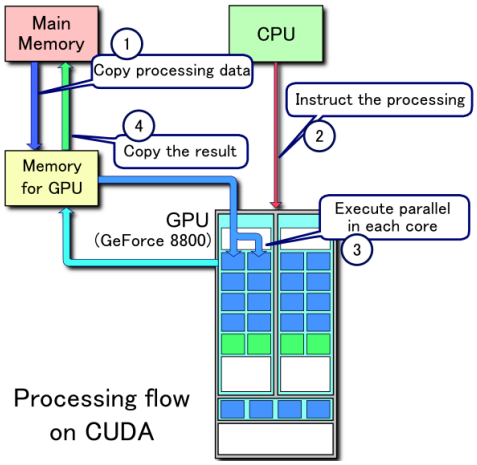


Figure 2: GPU Programming Model

CUDA also gives the user access to its different memory types and hierarchies. The global memory is a high-latency large storage typically used to store input and output data to which the host CPU can make memory transfers. Reads and writes to this global memory only achieve peak performance when they are coalesced (i.e. threads in the same warp access consecutive global memory locations). Also, each SM most often uses a number of 8192 registries, depending on the actual architecture. These registries are partitioned between the running threads. Another important type of memory is

the on-chip low-latency shared memory (up to 16KB per SM) that can be used in a highly parallel manner by the thread blocks. Its latency is considered equivalent to the latency of the registers, resulting in a very high speed of access. The constant memory is another on-chip read-only memory that offers low-latency and high bandwidth to concurrent accesses.

After presenting the memory hierarchy model of a GPU, it is crucial to point out that the number of threads running simultaneously on the SMs is dependent on the on-chip resource partitioning. The number of thread blocks in the same SM is limited by the shared memory and registry usage.

According to [17] there are four basic principles that should be taken into account when choosing to execute code on GPU platforms:

- The use of zero-overhead thread scheduling to hide memory latency. For example, on the GeForce 8800, there are 128 execution units available for use, requiring hundreds of threads to completely occupy them. In addition, threads can get starved of data due to the long latency to global memory. The way in which CUDA handles this is to generate and maintain thousands of threads in flight. This comes as a contrast to the CPU design where large caches try to hide memory latencies. Moreover, a high compute-to-memory-access ratio will be necessary to avoid saturation of memory channels.

- Optimization of the usage of on-chip memory to reduce bandwidth usage and redundant execution. For example, the shared memory can be used as a cache for the global memory.

- Grouping threads to avoid SIMD penalties and memory port/bank conflicts. For example, algorithms that create threads that require diverging control flow can suffer from performance degradation. Grouping threads to avoid this proves to be an efficient optimization. Also, appropriate thread grouping can avoid port and bank memory conflicts.

- The threads within a thread block can communicate via synchronization, but there is no built-in global communication mechanism for all threads. This helps to avoid the need for virtualization of the hardware resources and enables the same CUDA program to be able to run on different GPUs with a varying number of cores. The pitfall is that it limits the kinds of parallelism that can be utilized for a kernel call.

In this thesis, we apply these principles to optimize our GPU implementations. We also investigate how changes in the algorithm design, together with these optimization techniques affect the overall performance.

5 Related Work

In this section we will present various approaches present in the current literature regarding video coding on multi-core/many-core platforms. As we specified in the previous

section, motion estimation is the most compute intensive part of the video encoding process. Hence, this section will focus on presenting motion estimation implementations on many-core platforms.

The full search motion estimation algorithm (searching for the best match of a macroblock in the entire search window) analyzes the entire search space. This leads to a high complexity. In [1], various approaches of achieving faster motion estimation are presented. This is achieved by renouncing the full search in favour of methods that do not analyze the entire search space. Such methods are: cross search, spiral search, three step search, binary search, diamond search, hexagonal search. However, these kinds of algorithms that do not analyze the entire search space may not map very well on many-cores. This is because some of them imply many control flow instructions, branching instructions, that cannot achieve good performance on GPUs. Moreover, taking into consideration that GPUs can offer a much higher level of parallelism than modern CPUs, full search may be efficiently paralellized with the aid of these platforms. Additionally, with highly efficient implementations, the full search algorithm can be used in order to improve the quality of the motion estimation. Higher encoding quality is achievable because the entire search space is searched, and thus better matches could be found. Moreover, on many-core platforms, larger search windows could be used to further increase the output quality.

In this thesis, we present two full search motion estimation methods: the Reference method, which is depicted from the available literature and our collaborative method. The Reference Method is described in multiple papers [8, 9, 10, 11]. The way in which this Reference Method is designed is straightforward: each thread block (work group) searches for a best match of a macroblock from the reference frame; each thread from the block (group) analyzes a position within the search window by computing the sum of absolute differences (SAD) between the current macroblock and its candidate position from the search window; each thread then stores this SAD value in the device shared memory; a parallel reduction [18] is then employed to compute the least value and output the motion vector with the least cost (least SAD value) as the best match; based on this, the encoder creates the prediction of each macroblock of the current frame. As we will explain further in Section 6, this method cannot achieve coalescence for reading the current macroblock from the global memory. The same applies even for reading the search window pixels from the global memory.

In [8, 11] this approach is implemented in CUDA. These two papers implement motion estimation for the H.264 codec [1]. Moreover, the authors provide refinements of the motion estimation process at sub-pixel level. This is achieved using the same method, but with lower resolution frames (downsized by a factor of 2 or 4 in both directions) and with smaller block sizes (4 or 8). However, considering the basic principles for GPU optimization presented in Section 4, their implementations may not fully harness the GPU capabilities. When benchmarking this Reference Method, we designed our own optimizations to achieve good performance.

Another CUDA implementation of motion estimation is presented in [10]. In this paper the authors implement the unsymmetrical multi-hexagon search (hexagonal search).

This hexagonal search method evaluates candidates from the search window until a termination condition is satisfied. In this way, not all the search space is used, and the algorithm stops when a good enough solution is found. Even though this method does not search the entire search window for the best macroblock match, the authors show that for certain videos that do not exhibit a lot of motion the quality of the output (PSNR measured) is comparable to the full search method.

In [9], the Reference Method is implemented in OpenCL. In this paper, the authors acknowledge this method's bottleneck of covering a large search window. In order to overcome this, the search area is split onto multiple thread blocks (work groups). The method then works as before, each thread from a block computes a SAD between the current block and the candidate from the search window. However, using this method of splitting the work between multiple groups induces an overhead in computation as another kernel needs to be scheduled. This is because each thread block computes its best match, but as there are multiple thread blocks per search window, this second kernel is scheduled to compute the minimum between these best matches obtained for each thread block. Even if using this approach, this is probably not the most performant approach at increasing the search window size. We believe this because this approach of splitting the search window between different thread blocks still exhibits some bottlenecks: not achieving coalescence, same data is read multiple times from global memory, shared memory is not used for prefetching the search window. As a comment, we feel that a better approach at using larger search window sizes for this method would be to make use of a tiling [19, 30] technique, where a thread analyzes more than a single position from the search window. This would reduce the operational intensity of the compute kernel which could result in better overall performance. Investigating such approach for the Reference Method and comparing it to the solution to split the computation between multiple thread blocks remains for future work.

In the next section we will present our own approach at implementing full search motion estimation. Our collaborative method achieves coalescence when reading data from the global memory and can also make use of arbitrary search window sizes.

6 Motion Estimation Between Two Frames

In this section we will present two approaches for tackling the full search block matching motion estimation: a Reference Method inspired from the current literature [8, 9, 10, 11] and our own Collaborative method. The input for these algorithms consists of two frames: the reference frame (to which we encode against) and the current frame (that needs to be encoded). In order for the encoding to take place, both frames are first split into 16x16 blocks of pixels (macroblocks). Then, each macroblock from the current frame has to be matched against possible macroblocks from the reference frame. This is achieved by computing the "sum of absolute differences" (SAD) between the current macroblock and some of the reference macroblocks. The reference macroblock for which the minimum is achieved is considered the match and only their motion vector is saved for further encoding. Thus, the computational complexity of the serial algorithm that

will solve this problem is:

$$\mathrm{O}\big(\frac{N}{B} * \frac{M}{B} * B^2 * (W - B + 1)^2\big) = \mathrm{O}\big(N * M * (W - B + 1)^2\big). \quad (1)$$

Here, N is the width of the picture to which we apply the algorithm, M is the height, B is the block size and W is the window size, assuming that the windows and blocks are square.

6.1 Reference GPU Implementation (*RefGPU_naive*)

The method presented in this subsection is inspired by the available literature [8, 9, 10, 11] and adapted to achieve a full search block matching motion estimation. Adaptation for enhancing the search window size was also employed because newer GPU devices permitted this. We will explain further why the search window size is dependent on the underlying GPU platform. In what follows we will describe this method together with optimizations we designed for improving it.

First, the reference frame and the current frame are transferred from the host memory to the device memory. The dimension of the thread block gives the dimension of the search window. This is because each thread in the thread block will compute the SAD value between the current macroblock to be encoded and one position of the search window. For doing this, each thread loads 16x16 pixels from both images (one macroblock from each). For each pixel, a SAD operation is performed and the sum of all SADs is stored in the shared memory. After a synchronization, a parallel reduction is performed to compute the minimum between all evaluated positions of the search window. After this, the motion vector for the least value is written in the output buffer.

As a comment, implementing the block matching motion estimation in this way, the search window size is limited by the maximum number of threads in a thread block. For example, considering NVIDIA GPUs, depending on the specific compute capability (1.0, 1.1, 2.0, 2.1) this number can be 512, 768, 1024, 1536. The search window size value is given by the square root of these numbers. Indeed, the search window could be extended by expanding the amount of work done on each thread but this needs additional coding effort and would also need additional shared memory storage - the larger the search window is, the larger the shared memory should be. Moreover, this could come at another cost: using more shared memory can limit the number of thread blocks executed in parallel on a SM.

Taking into consideration this direct approach at parallelizing the algorithm in a SPMD fashion and the left side of the Equation (1), it is easy to deduce that we need to run $\frac{N}{B} * \frac{M}{B}$ thread blocks, each composed out of $(W - B + 1)^2$ threads. Each thread executes B^2 operations.

As presented in [18], the parallel reduction is done in $log(T)$ steps, where T is the size of the input/thread block. Thus, $T = (W - B + 1)^2$. This is achieved in the following way: in the first iteration, half of the threads are active checking if the value they computed is lower than the value computed by their correspondent from the second half of the threads; then, a quarter of the threads are active and do the same kind of work;

this happens until only one thread remains active and outputs the minimum value. In Figure 3 we present a pseudocode implementation for the "naive" block matching motion estimation kernel.

```
kernel ComputeLeastSAD(reference_frame, current_frame, output)
{
  int x = get_thread_global_x();
  int y = get_thread_global_y();
  // compute thread offset x for search window current position
  int offset_x = get_thread_local_x() - WINDOW_SIZE;
  // compute thread offset x for search window current position
  int offset_y = get_thread_local_y() - WINDOWS_SIZE;
  int macroblock_index = get_macroblock_index();
  // declare shared memory for the current thread block
  __shared__ int sad_values[THREAD_BLOCK_SIZE];
  int result = 0;
  for (int i = 0; i < BLOCK_SIZE; ++ i)
    for (int j = 0; j < BLOCK_SIZE; ++ j)
    {
      int px1 = get_pixel(x + j + offset_x, y + i + offset_y, reference_frame);
      int px2 = get_pixel(x + j, y + i, current_frame);
      result += compute_sad(px1, px2);
    }
  // save current value to shared memory
  sad_values[macroblock_index] = result;
  barrier(); // synchronize writing to shared memory
  int least_sad = parallel_reduction();
  output[macroblock_index] = least_sad;
}
```

Figure 3: Pseudocode for the "RefGPU_naive" GPU implementation

6.1.1 Optimizations Employed (*RefGPU_opt*)

As stated in Section 4, there are four basic principles to take into account when writing code that is to be executed on GPU architectures. For the *RefGPU_naive* implementation, discussed in Section 6.1, none of these have been adhered to besides the one related to shared memory bank conflicts. Thus, the kernel described before cannot make use of the full performance delivered by the GPU platforms.

The main observation from which we can start optimizing this method is that multiple threads from the same thread block access the same global memory areas when reading pixels from the current macroblock. This happens even for the pixels within the search window. Hence, multiple requests are done to the same regions of the global memory which results in multiple redundant stalling transactions. Moreover, coalescence is not achieved at all.

In order to improve performance by making use of coalesced memory transactions and by renouncing the redundant transactions, the following approach can be employed:

adjacent threads in the same thread block collaboratively read the current macroblock from source image; adjacent threads from the same thread block collaboratively read the entire search area from the reference image; then, the threads can perform their work as before.

In Figure 4 one can see the improvements added to the "naive" implementation. As a comment, this approach uses much more shared memory per thread block. This could lead to a limitation of the number of warps running in parallel on a SM. Moreover, increasing the window size may affect this behaviour even further. Also, memory bank conflicts will occur at runtime because threads will read overlapping regions from the shared memory. But, the stalling employed by this should be negligible compared to the improvement brought by the global memory throughput optimizations that we previously discussed. Another problem with this implementation could be that only one thread from the thread block writes the results to the output buffer. Hence, this implementation does not make use of coalesced memory transactions when writing to the global memory of the GPU.

```
kernel ComputeLeastSAD(reference_frame, current_frame, output)
{
  int x = get_thread_global_x();
  int y = get_thread_global_y();
  // compute thread offset x for search window current position
  int offset_x = get_thread_local_x() - WINDOW_SIZE;
  // compute thread offset x for search window current position
  int offset_y = get_thread_local_y() - WINDOWS_SIZE;
  int macroblock_index = get_macroblock_index();
  // declare shared memory for the current thread block
  __shared__ int sad_values[THREAD_BLOCK_SIZE];
  __shared__ char current_macroblock[BLOCK_SIZE * BLOCK_SIZE];
  __shared__ char search_window[WINDOW_SIZE * WINDOW_SIZE];
  // read macroblock collaboratively
  read_macroblock_pixels(current_frame, x, y, current_macroblock);
  read_search_window(reference_frame, x, y, search_window);
  barrier(); // synchronize reading the input data
  int result = 0;
  for (int i = 0; i < BLOCK_SIZE; ++ i)
    for (int j = 0; j < BLOCK_SIZE; ++ j)
    {
      int px1 = get_pixel(j + offset_x, i + offset_y, search_window);
      int px2 = get_pixel(j, y, current_macroblock);
      result += compute_sad(px1, px2);
    }
  // save current value to shared memory
  sad_values[macroblock_index] = result;
  barrier(); // synchronize writing to shared memory
  int least_sad = parallel_reduction();
  output[macroblock_index] = least_sad;
}
```

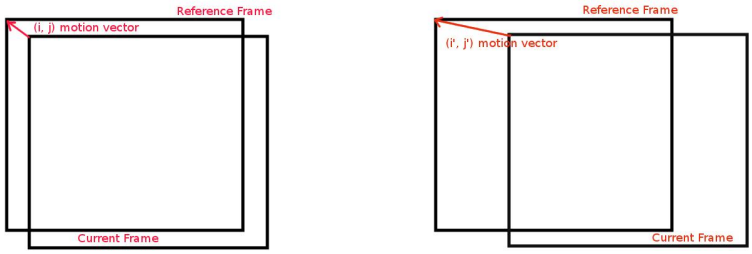
Figure 4: Pseudocode for the "RefGPU_opt" GPU implementation

6.2 Proposed Collaborative Method (*CollabGPU_naive*)

In this section we present our collaborative approach to computing motion estimation between two frames. We designed this algorithm in such a way that not only the search window could be extended without any coding effort or increase in shared memory usage, but it also adds another degree of parallelism.

To better explain this, we need to define first the notion of "job" for our algorithm. A "job" is a kernel call that computes for the two given frames (already in the device global memory) the SAD values computed for all the current blocks and their reference blocks at a certain overlapped (i, j) position given as input to the kernel call. Thus, with each job we compute the SAD value for a given motion-vector at each macroblock. Then, for each macroblock, the motion vector with the least SAD value is picked up as the final result. This technique holds, and gives correct results because moving the source picture with an offset (i, j) and overlapping it over the reference picture and computing the SAD values for all macroblocks is equivalent to computing the SAD values for each macroblock at an (i, j) offset in its search window.

This could be also seen as a "sliding-window" technique: the reference frame is overlapped at various positions over the current frame to compute the SAD values. As a comparison to the *RefGPU_naive* solution, we could state that the first method slides a macroblock around its search window, and our *CollabGPU_naive* slides the entire picture around the reference frame. As we will prove further, this method has some advantages over the other. In Figure 5 (a) and (b) one could see a visual explanation for our approach. As we will further explain, this approach does not use a very large amount of shared memory, thus not overly limiting the number of thread blocks that can be run in parallel on a SM. Treating this problem as a number of different jobs also helps in adding another degree of parallelism. Thus, if multiple compute devices (CPUs, GPUs etcetera) are present on a certain machine, these separate jobs could be split between them in order to reduce the total execution time.



(a) Motion Estimation Job for motion vector (i, j)

(b) Motion Estimation Job for motion vector (i', j')

Figure 5: Motion Estimation Jobs

6.2.1 Single-GPU Implementation and Optimizations

In this section we will describe our GPU implementation and its optimizations. The implementations were done in CUDA and OpenCL. The approach for implementing this Collaborative method is similar to the one for the *RefGPU_naive*. The difference is that for each scheduled job, a new kernel call is necessary, as each kernel call computes the SAD values for a given motion vector.

To explain what happens when each job is running, we first need to show that, in compliance with Equation (1) presented before, $\frac{N}{B} * \frac{M}{B}$ threads are scheduled. Each thread loads B^2 pixel values from both images and computes their SAD value. The value is then written to the output buffer. In total, $(W - B + 1)^2$ kernel calls are necessary. Threads can be grouped arbitrarily in thread blocks. In Figure 6, we present a pseudocode version of this method. We need to take into account that this is, again, a "naive" implementation. We will then further discuss the employed optimizations.

```
kernel ComputeLeastSAD(reference_frame, current_frame, output, offset_x, offset_y)
{
  int x = get_thread_global_x();
  int y = get_thread_global_y();
  int macroblock_index = get_macroblock_index();
  int result = 0;
  for (int i = 0; i < BLOCK_SIZE; ++ i)
    for (int j = 0; j < BLOCK_SIZE; ++ j)
    {
      int px1 = get_pixel(x + j + offset_x, y + i + offset_y, reference_frame);
      int px2 = get_pixel(x + j, y + i, current_frame);
      result += compute_sad(px1, px2);
    }
  output[macroblock_index] = result;
}
```

Figure 6: Pseudocode for the "naive" collaborative GPU method

As a comment, this first implementation of our Collaborative method does not comply with the recommendations presented in Section 4. It does not make use of shared memory so there couldn't be any bank conflicts, but otherwise, this implementation does not make use of the full computation power of GPU platforms. The only feature worth noticing is that this implementation makes use of coalesced memory transactions when writing to the global memory of the GPU, because adjacent threads write the results to adjacent memory locations.

At a first glance, this proposed Collaborative method is not easily optimizable and seems to lack the ability to fully utilize the underlying platform. However, due to the method's approach of splitting the work to be done in jobs, as described earlier, there are some optimizations that aim at fully using the device memory bandwidth.

First, we observed that the width of a macroblock is 16 (the H.264 block size).

Thus, each row of a macroblock occupies 16 bytes of memory. Making use of the device capability of reading 16-bytes wide vectorized data structures (int4 in CUDA and openCL terminology), we can read an entire macroblock line with just one read. It is now clear that adjacent threads read memory for adjacent macroblocks, and, if each thread reads an entire macroblock line, adjacent threads will have coalesced accesses to the device global memory. We have thus obtained full coalescence.

Even though this is a simple and efficient way of reading the current macroblock from the current image, this approach would not work for the reference image. This is because the macroblocks needed to be loaded from the current frame are 16 byte alligned. This approach would not work for the reference image due to the parameter "*offset_x*" from the kernel presented in Figure 6. this parameter would require us to load memory at misaligned positions from the global device memory which is not possible for current devices. This is because this parameter is not a multiple of 16, which is the macroblock size and also the width in bytes of an int4 data.

However, achieving coalesced reads and fully saturating the memory bus width is possible even in this approach. This is possible by making use of the device shared memory. In this way, each thread from the same thread block loads int4 data from the exact "underlying" macroblock of the reference image, not taking into account the aforementioned parameter "*offset_x*". This happens in the exact same way as for the current image's macroblocks, but taking into account the "*offset_y*" parameter which cannot affect the memory aligment. Then, each thread stores the current macroblock line into the shared memory of the thread block. After this, by making use of reading "char" data from the shared memory instead of "int4" data, each thread can then take into consideration the "*offset_x*" parameter. Also, using this approach, the number of threads in a thread block cannot be arbitrary any more. The best way of grouping threads is to create blocks of sizes equal to the number of macroblocks that fit in the width of the picture. This number is W/B, where W is the width of the picture and B the macroblock size. In this way, a thread block processes a row of macroblocks.

Hence, the proposed algorithm for taking into account the optimizations presented here is:

- (1) for each macroblock line do:
- (1.1) read macroblock line from current frame
- (1.2) read macroblock from reference frame and store to shared memory
- (1.3) synchronize threads
- (1.4) place pointer to reference macroblock making use of "*offset_x*" parameter (either to left or right, depending on the sign)
- (1.5) compute SAD sum for current macroblock line and reference macroblock line.
- (2) output result

Below, we present the pseudocode for this optimized kernel version.

```
function compute_sad_md_line(char *crnt_mb, char *mb_lines, int offset_x, int thread_index)
{
  // pointer arithmetics
  // set the pointer to correct reference macroblock index
  mb_lines = mb_lines + thread_index * BLOCK_SIZE + offset_x;

  // iterate through both macroblock lines
  // and compute the SAD value
  int result = 0;
  for (int i = 0; i < BLOCK_SIZE; ++ i)
    result += compute_sad(crnt_mb[i], mb_lines[i]);

  return result;
}

kernel ComputeLeastSAD(char *reference_frame, char *current_frame, int *output, int offset_x, int offset_y)
{
  int x = get_thread_global_x();
  int y = get_thread_global_y();
  int thread_local_index = get_thread_local_index();
  int macroblock_index = get_macroblock_index();
  //declare shared memory
  __shared__ int4 mb_lines[NO_OF_MBS];
  int result = 0;
  // for each macroblock line
  for (int i = 0; i < BLOCK_SIZE; ++ i)
  {
    // read current macroblock
    int4 crnt_mb = read_mb(x, y, current_frame);
    mb_lines[thread_local_index] = read_mb(x, y + offset_y, reference_frame, thread_local_index);

    barrier(); // synchronize threads

    result += compute_sad_mb_line(crnt_mb, mb_lines, offset_x, thread_local_index);
  }
  output[macroblock_index] = result;
}
```

Figure 7: Pseudocode for the optimized collaborative GPU method

In the listing presented above, there are several exception cases that are not implemented. This is done to preserve simplicity and in order to clearly present the idea behind the implementation. These corner cases are related to macroblocks that reside on the border (i.e. first/last macroblock of a row). The threads that handle these kinds of macroblocks should, in practice, should take care of possibly out of bound indexes when the "*offset_x*" parameter is negative/positive. When implementing, we solved this by adding a 16 byte border for each side of the image and we also made first/last thread of a macroblock read a 16 byte segment of memory to the left/right. We chose this approach because the other option of solving this (branching instructions) would have

implied multiple conditional instructions which would have led to divergent warps that would deteriorate the performance achieved.

It is also worth noting that in practice we employed two other optimizations: we unrolled both loops presented above. We chose not to present this into the pseudocode above for reasons of readability.

6.2.2 Details of CUDA Implementation

Until now, we only presented the idea behind our proposed Collaborative method and the optimizations that we used to make it faster. However, we have not presented any arguments or proofs that this solution could achieve the same performance as the one presented in Section 6.1. One could see that our implementation has some drawbacks. First, the method presented in Section 6.1 does all the computation needed within a single kernel call. For the method we proposed, there is a relatively large number of kernel calls to be made. Another weakness is that for each kernel call, the same data (current frame) is read again from the global memory into the shared memory/registers. We will focus on this aspect and the performance-penalty it implies in Section 8 where we present and analyze the experiments we made. Here we will only focus on the first drawback of our implementation - the multiple kernel calls.

To explain how we overcame this performance issue we must first remind the reader that GPU platforms are best at exploiting data parallelism. Task parallelism was not something achievable for GPU programming before NVIDIA released the Fermi architecture [20, 21]. However, the Fermi architecture, programmed in CUDA can offer the developer task parallelism. This is achievable by using multiple CUDA streams. In this way, multiple kernel calls issued on different streams have the ability to run in parallel. In this way, we have been able to launch our kernels concurrently and thus, a workaround for this drawback has been found. In Section 9 we will empirically show that using this technique, our Collaborative method, on a single GPU is as performant as the *RefGPU_opt* method.

6.2.3 Details of OpenCL Implementation

Since in Section 1 we mentioned our goal of designing a solution that will run on heterogeneous platforms, and also because the CUDA framework is only available on NVIDIA GPUs, we also implemented this solution in OpenCL. This language offers portable High Performance Computing code that runs on NVIDIA, AMD/ATI, Intel hardware. Moreover, the same code can run on both CPU and GPU and even other accellerators such as the Cell B.E. [6].

After developing the CUDA solution, porting it to OpenCL was an easy task. Most of the CUDA features have a one-to-one mapping in OpenCL. As we mentioned before for CUDA, our Collaborative method needs to make use of task parallelism in order to achieve the expected performance. OpenCL offers two ways of achieving task parallelism on a single device. The first is to use out-of-order queues (an OpenCL processing queue is equivalent to a CUDA stream - all commands issued on a queue/stream are issued in-

order, one command after the previous finished). An out-of-order queue has the ability to issue commands concurrently and does not guarantee the order in which they finish (i.e. it does not work in a FIFO fashion). The second option for task parallelism in OpenCL is using multiple command queues. This approach achieves task parallelism by concurrent scheduling of commands from different queues.

However, because OpenCL is a standard implemented differently by all hardware providers these solutions do not work, because, at the time our work is conducted, these features are not implemented. The out-of-order queues work only for CPU devices. Using multiple command queues does not meet the expectation of concurrent kernel execution on GPU devices. This is because even though Fermi GPUs offer task parallelism, NVIDIA only unlocked it in CUDA; for OpenCL this is not yet implemented. Also AMD GPUs simply do not offer this capability even in hardware. Moreover, documentation is scarce in this aspect and not much information can be found related to the subject.

Moreover, the only way in which OpenCL offers viable task parallelism for CPUs is the following: "the OpenCL task parallel programming model defines a model in which a single instance of a kernel is executed independent of any index space. It is logically equivalent to executing a kernel on a compute unit with a work-group containing a single work-item" [22]. Scheduling a single kernel call on only one work item would not be helpful for our *RefGPU_opt* method's approach even for CPU platforms. A redesign of the method would be needed in order to obtain correct results.

To overcome these issues and still be able to use our Collaborative method in OpenCL we need to propose a different strategy. We simulated the multiple kernel calls by scheduling more jobs into a single kernel call. In fact, we designed an approach which schedules all the needed jobs from a single kernel call. As we mentioned before, if we had $(W - B + 1)^2$ kernel calls, scheduling $\frac{N}{B} * \frac{M}{B}$ threads each, now we need to deploy, in one single kernel call $\frac{N}{B} * \frac{M}{B} * (W - B + 1)^2$ threads. They are grouped as before, one thread block being preffered to do the computation for an entire row of macroblocks. However, the parameters we used for an (i, j) job, namely *offset_x* and *offset_y* cannot be used any more. In order to overcome this, we have computed those numbers from the "global thread index" which is a number between 0 and $\frac{N}{B} * \frac{M}{B} * (W - B + 1)^2 - 1$. Using this strategy, we can compute these numbers in the following way:

- Compute the order number number of the current thread (i.e. order number of the (i, j) job that includes the thread) in the following way:
- $index = thread\_global\_index/(\frac{N}{B} * \frac{M}{B})$, where "/" is integer division. Then:
- $offset\_x = (index/(W - B + 1)) - B$ and
- $offset\_y = (index\%(W - B + 1)) - B$ (% is the modulo operation).

Hence, in this way, we have obtained the same algorithm without having to spawn multiple kernel calls.

6.2.4 CPU Implementation and Optimizations

In this section we will describe our CPU implementation and its optimizations. As we stated before, we aim at obtaining a software solution that can run on multiple devices and even when GPU platforms are not present in a machine. Hence, the implementation of the CPU version of our Collaborative method was done in OpenCL. Moreover, OpenCL offered the ability of simply executing on CPU the code already written for GPU devices.

The only modifications that had to be done are the in the host code. Here, we had to modify only the grouping of threads in blocks. For GPUs, a high number of threads in a block is normal and recommended for improving occupancy [4]. For CPUs, the approach is different because they offer a small number of cores that offer a coarse-grained parallelism. Usually the OpenCL compiler tries to issue the threads from a thread block to a single CPU core which will serialize their execution [23, 24]. Moreover, if any shared/local memory is used between the threads, it will be allocated in the CPU cache. If a large number of threads need to access a large amount of shared memory the performance will obviously degrade. This could get even worse if the threads need synchronization. Hence, for the CPU implementation we simply used smaller thread blocks. We will show in Section 9 how this parameter affects performance.

Another approach at implementing this for CPU devices has been taken. We tried to renounce using shared memory that could result in performance degradation. This is simple to achieve as only a small modification has to be made in our kernel. We stated before that shared memory had to be used when optimizing the Collaborative method because GPUs do not support misaligned reads from the global memory. CPUs have this ability and thus, reading an int4 at misaligned addresses solves our problem for using shared memory. Also, when using this, synchronization is not necessary between threads of the same work group. Hence, our for loop from Figure 7 can change to the one presented into Figure 8. We denote these two CPU methods as *CollabCPU_opt* and *CollabCPU_opt*.

```
// for each macroblock line
for (int i = 0; i < BLOCK_SIZE; ++ i)
{
  // read current macroblock
  int4 crnt_mb = read_mb(x, y, current_frame);
  int4 ref_mb = read_mb(x + offset_x, y + offset_y, reference_frame);

  result += compute_sad_mb_line(crnt_mb, ref_mb);
}
output[macroblock_index] = result;
```

Figure 8: Pseudocode for the optimized collaborative CPU method

A performance comparison for these two methods will be presented in Section 8.

Also, according to [23], to be able to benefit from a CPU SSE capability, vectorization has to be employed. However, our implementation already had this enabled from the GPU implementation optimizations. Thus, nothing had to be done in this aspect.

7 Multi-Device Collaborative Method

In this section we will describe the implementation of an OpenCL multi-device framework that can split the work between multiple and different compute devices available on a single machine. The work to be split is the Collaborative Method described in the previous section.

7.1 OpenCL Multi-Device Introduction

First, to familiarize the reader with the workings of OpenCL, we have to briefly describe how an OpenCL accelerated program is written. The same CUDA concepts apply - the program is executed on a *host* that offloads computation to a *device*. Due to the specific capability of OpenCL of running on a number of different platforms and devices, some auxiliary concepts have been introduced.

Thus, when one has to write an OpenCL program, first a *context* should be initialized. The context refers to which kind of platform does the program targets. For example, there exist AMD, NVIDIA, Intel platforms. Hence, supposing that multiple platforms are installed and available on a certain machine, an OpenCL program can make use of all of them by initializing multiple contexts. For example, if one wants to collaboratively use an Intel CPU and and NVIDIA GPU, contexts for the two platform have to be initialized.

Then, after the context has been initialized, the OpenCL API offers the programmer methods for querying the system about the available *devices*. Thus, for each initialized context a list of compute devices can be retrieved. This list can then be used to offload various kernels on the compute devices. Attention is needed, though, when using multiple OpenCL contexts. This is because, for example, the AMD OpenCL platform has the ability to launch kernels on CPUs and GPUs. Moreover, the AMD platform can run kernels even on Intel CPUs, and thus, if an Intel OpenCL platform is present, the two will indicate the same CPU device as available for computation offloading. Using the same physical device concurrently from the two different contexts could then lead to performance degradation. Probably it is worth mentioning that the NVIDIA OpenCL platform has only the ability of accelarating computations using NVIDIA hardware; the Intel platform has the ability of running only on Intel GPUs, but regarding CPUs it can use even AMD ones; the AMD OpenCL platform has the ability to use only AMD/ATI GPUs, but also Intel/AMD CPUs.

After initializing the necessary OpenCL *devices*, the programmer can simply follow the CUDA paradigm. Data can be transferred to each device (besides CPUs which use the host DRAM), kernels executed etc. The difference between OpenCL and CUDA is that the kernel code should be compiled at runtime targeting the specific platform on

which the kernel is to be executed. The alternative is to compile the kernel before the running of the program, save the binary code into a file, and then, when execution is needed, instead of compiling the kernel code, the OpenCL program can simply read the binary from the file and execute it. This might be an unwanted situation when the code is ported to other hosts that do not contain the same hardware because the compiled code is platform specific.

7.2 OpenCL Multi-Device Framework

From what we've presented in the previous subsection, the way in which one can implement a program which needs to make use of multiple OpenCL-capable devices is to make use of multiple *contexts*, if available, and use all the *devices* that correspond to those contexts.

Thus, for achieving this behaviour, we implemented a general C++ framework that is able to spread computation on multiple OpenCL devices. We created a general framework, rather than a problem-specific one, in order to offer the user the posibility of using this framework with generic applications. Hence, a user will only write the kernel code itself and the necessary logic behind the splitting of the work between multiple devices, synchronization etc. This framework is capable of making use of all the OpenCL platforms available and all the compute devices. The user can then load the kernel code and compile it for all the available platforms. Also, the user declares the memory objects, copies host data to device memory if necessary and the launches the kernel code, in parallel on multiple devices.

Our framework is composed of two C++ classes. The first, the *aggregator class*, initializes an available OpenCL context and loads the list of available OpenCL devices. For each such device, it instantiates objects of the *device class*. This second class is not directly available for the programmer, it is only used by the *aggregator class* as an interface for compiling the kernel code, copying memory back and forth to the device or run kernels. Instead, these actions are transparently done by the *aggregator class* for the programmer. For compiling, scheduling of kernels and memory transfers, the programmer only has to call these methods once for the *aggregator class* which will forward these to the *device class*. Also, this class has the ability of selecting the most appropriate kernel for the specific device. For example, if the current device is a GPU, the class will choose to compile and execute the GPU kernel. In case of a CPU, the CPU kernel will be chosen. Achieving this is possible using the *clGetDeviceInfo* function call with the *CL_DEVICE_TYPE* as a flag. The result of this function (*CL_DEVICE_TYPE_CPU, CL_DEVICE_TYPE_GPU, CL_DEVICE_TYPE_ACCELERATOR*) is then used to select the correct kernel. In Figure 9, we depicted the architecture of our framework. The host process/thread can create a number of aggregator classes that hold the available contexts. These objects will then create device objects that handle the kernel execution.

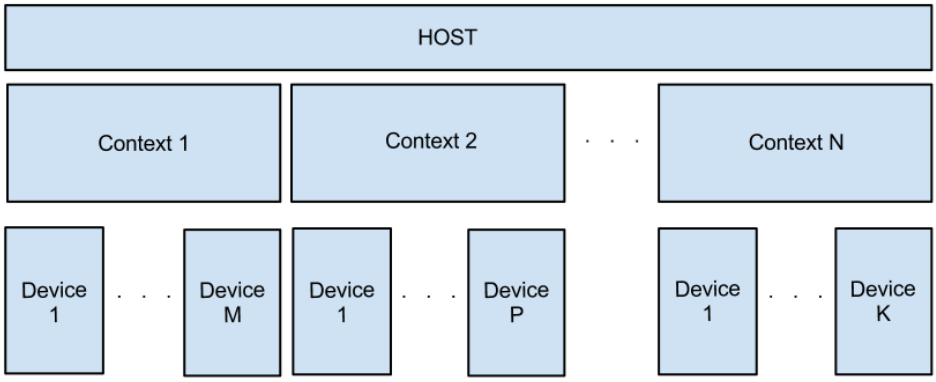


Figure 9: Multi-Device Framework Archtitecture

For memory copying between the host and the devices, we have considered multiple options. The first approach is to do this sequentially (*OpenCL-async*), in a non-blocking fashion, each device making use of its own OpenCL "in-order" *command queue*. In this way, the kernels can then be started for each device and will only execute after the copying has been done. Using this non-blocking approach, the host thread does not have to wait for each copy to take place and thus, copying data to multiple device can be done in parallel. For copying the data back from the devices the same approach can be used. However, this is not enough, we have to use the *event* objects used by OpenCL to synchronize the host thread with the ending of the copy process to use the data obtained. In Figure 10 we present the pseudocode for achieving this kind of behaviour.

The second approach for communication between the host and the devices is to create a host thread for each available device (*one-thread-per-device*). This could possibly reduce the overhead induced by the sequential execution. Also, synchronization will not be necessary because in this approach, in each thread the copying and kernel calls will be done synchronously. Another two aspects need to be discussed for this multithreaded approach. First, using threads could reduce the total execution time by better utilizing the host device. Thus, if the host is a multicore, in this way we assure that not only one core drives the computation, but more of them are used. Second, using this approach when also the host CPU is used as an OpenCL device could lead to performance loss. This is because the host CPU will have to run the kernel and also manipulate data in the same time.

Both approaches have been implemented, but, after benchmarking, we decided that the *one-thread-per-device* is faster because the event management of OpenCL seems to add significant overhead. Hence, in Section 8, the results we present are obtained using this technique.

```
...
// for each device
for (int i = 0; i < devices; ++ i)
{
  asyncCopyDataTo(device[i], data[i]);
}
asyncStartKernelOnAllDevices();
// for each device
for (int i = 0; i < devices; ++ i)
{
  asyncCopyDataFrom(device[i], output[i]);
}
synchronize();
```

Figure 10: Pseudocode for Data Synchronization

8 Experiments and Results

In this section we will present our approach to evaluating our high performance motion estimation solution. A comparison with the *RefGPU_opt* method is also presented. We will show that our *CollabGPU_opt* outperforms *RefGPU_opt* in speed and scalability. For single GPU systems a comparison of CUDA and OpenCL implementations is also presented.

The platforms on which our solution is evaluated is in accordance with the list presented in the introduction: single CPU, single GPU, multi-CPU, multi-GPU systems, and CPU-GPU systems. We will make a full comparison of the performance of the presented methods achieved on these platforms.

Finally, we will present qualitative measurements in PSNR (peak signal to noise ratio) values. We will show that these values are influenced by the search window size used for the motion estimation process - larger window sizes could bring an improvement in quality. This improvement should come from the fact that, in larger resolution videos which involve a lot of movement, motion vectors with lower costs could be found at distances that are greater than the search window size. Thus, we will show that our *CollabGPU_opt* method is not only faster, but, by being able to make use of larger search window sizes, it can also add qualitative benefits to the encoding process.

Further, we provide the list of frame sizes on which the motion estimation methods are evaluated. The image sizes we use are: 176x144, 256x192, 352x288, 352x480, 720x480, 1280x720, 1920x1080, 2048x1080, 2048x1080, 2560x1920, 4096x2048, 4096x3072, 7680x4320, 7680x7680. We decided to use this wide range of frame sizes to be able to present a complete evaluation which should include a scalability evaluation. We found this necessary as other papers [8, 9, 10, 11] that treat the subject of motion estimation only test their solutions on small resolution images. Also, we wanted to expand this further, in order to be able to thoroughly test the scalability of our implementations.

8.1 Testing Environment

In this section we will present the actual hardware and software platforms on which we ran our experiments. The HPC platforms on which our application has been run are embedded into the DAS4 [25] clusters located at the Vrije Universiteit Amsterdam and at Astron [31]. The operating system on which our software solution was run is CentOS Linux which runs a 2.6.32 Linux kernel. The compiler used is gcc, 4.4.6 version.

For developing our application we used the CUDA toolkit version 4.2 and OpenCL drivers offered by NVIDIA (version 4.2) and by AMD (version 2.6). The specific hardware on which our application was run is:

- Intel Xeon E5620 CPU
- AMD Magnycours multi-CPU
- NVIDIA GTX480 GPU
- AMD HD6970 GPU
- 8 x NVIDIA GTX580 GPUs

8.2 Scalability with Input Size on GPUs

In this section we will present a comparison between the *RefGPU_opt* and *CollabGPU_opt* methods. We show how these methods scale with the increase of the frame size.

Before presenting the results, a few theoretical aspects have to be presented. We take into consideration *Equation (1)* from Section 6, and also the fact that the underlying GPU platform limits the *RefGPU_opt* search window size. This happens in the following way: if the underlying GPU platform can schedule a maximum of P threads in a thread block, then the maximum search window size can be computed as follows:

$$(W - B + 1)^2 = P \Rightarrow W - B + 1 = \sqrt{P} \Rightarrow W = \sqrt{P} + B - 1,$$

where, same as before, W is the search window size (we assume a square window of $W \cdot W$ pixels) and B is the macroblock size (in our case 16).

We tested the scalability of the two methods for the maximum allowed search window size for the *RefGPU_opt* method. The GPUs used for these tests are the NVIDIA GTX480 and the AMD HD6970. The former can support a thread block of maximum size 1024, hence a search window of size 47, and the latter a thread block (work-group in OpenCL terminology) of size 256, hence a search window of size 31. For the measurements on both video cards we have used the OpenCL implementation. We also show the performance achieved by our CUDA solution on the NVIDIA GTX480.

Below, we present the achieved results for both platforms chosen. We first compare the two platforms' achieved performance for the search window size of 31 (maximum allowed by the AMD HD6970 *RefGPU_opt* implementation). This is done to point out which platform performs best on this kind of algorithms. Then, an evaluation with

window size set for 47 - maximum for GTX480 video card on *RefGPU_opt* method is presented. Here, we also compare CUDA and OpenCL implementations and also show that even though limited in thread-block (work-group) size for the *RefGPU_opt*, the AMD HD6970 can outperform the GTX480 for larger window sizes when our Collaborative method is used.

In Table 1 and Figure 11 we show the running time, in milliseconds, obtained by running both methods on NVIDIA GTX480 and AMD HD6970. The column headers *Ref* and *Collab* refer to running the *RefGPU_opt* and *CollabGPU_opt* methods. The results meet our expectations. Our collaborative method is faster than the Reference method with the increase in input size. This happens because the *CollabGPU_opt* method makes better use of the device memory bandwidth. All its memory transfers between the shared and global memory are coalesced and each thread reads a section of 16 bytes of memory. In the *RefGPU_opt*, each thread makes a transaction of only 1 byte, and not all of these transactions are coalesced because the search window size is larger than the number of threads in the thread-block. Also, using this method, there are more shared memory bank conflicts as different threads may read overlapping sections of the shared memory. Another reason for this better scalability of our method is that the thread block size is dependant on the problem size/frame size. The number of threads in a block for a certain frame size is equal to the number of macroblocks that fit on the width of the frame. Hence, this number increases with the increase in the image size. This leads to a better occupancy of the device when the problem size increases, making our method perform better for larger frame sizes.

Regarding the AMD HD6970, it is clear that it performs better than the other device, NVIDIA GTX480. This is just because it has better technical specifications. It can offer up to 2.7 TFLOPS, while the GTX480 only 1.344 TFLOPs. Taking this into consideration, one can only find curious why the *CollabGPU_opt*, when run on the HD6970 is slower than GTX 480 on small input sizes. We believe that this happens because what we stated earlier about the thread block size which is dependant on the input size. Moreover, it is worth mentioning that the HD6970 schedules threads in wavefronts of 64 threads at a time. The NVIDIA GTX480 schedules threads in warps of 32 threads. Taking all these facts into account, it is clear that for small input sizes, the HD6970 cannot perform well because its thread blocks will be even smaller than the wavefront size of 64. This leads to poor utilization of compute resources. Also, related to the AMD HD6970, the reader can see in Figure 1 that when it comes to higher input sizes (frame widths of 2048 pixels or higher) it performs much better than the GTX480. This happens because for these input sizes the AMD GPU achieves its highest occupancy as it can schedule only a maximum of 256 threads in a block. This number is achieved for frame sizes of 4096 or higher. On the other hand, the GTX480 can schedule a maximum of 1024 threads in a block, thus not reaching a high occupancy.

Table 1: Scalability with Input Size - Search Window 31

Frame Size	**#Macroblocks**	**GTX480**		**HD6970**	
		Ref (ms)	Collab (ms)	Ref (ms)	Collab (ms)
176x144	99	0	1	2	4
256x192	192	1	1	3	4
352x288	396	1	2	5	6
352x480	660	2	2	6	8
720x480	1350	4	4	10	9
1280x720	3600	11	9	23	24
1920x1080	8100	26	20	37	33
2048x1080	8640	28	21	44	26
2560x1920	19200	63	38	73	51
4096x2048	32768	106	64	120	67
4096x3072	49152	158	94	171	90
7680x4320	129600	415	246	423	231
7680x7680	230400	717	457	740	308

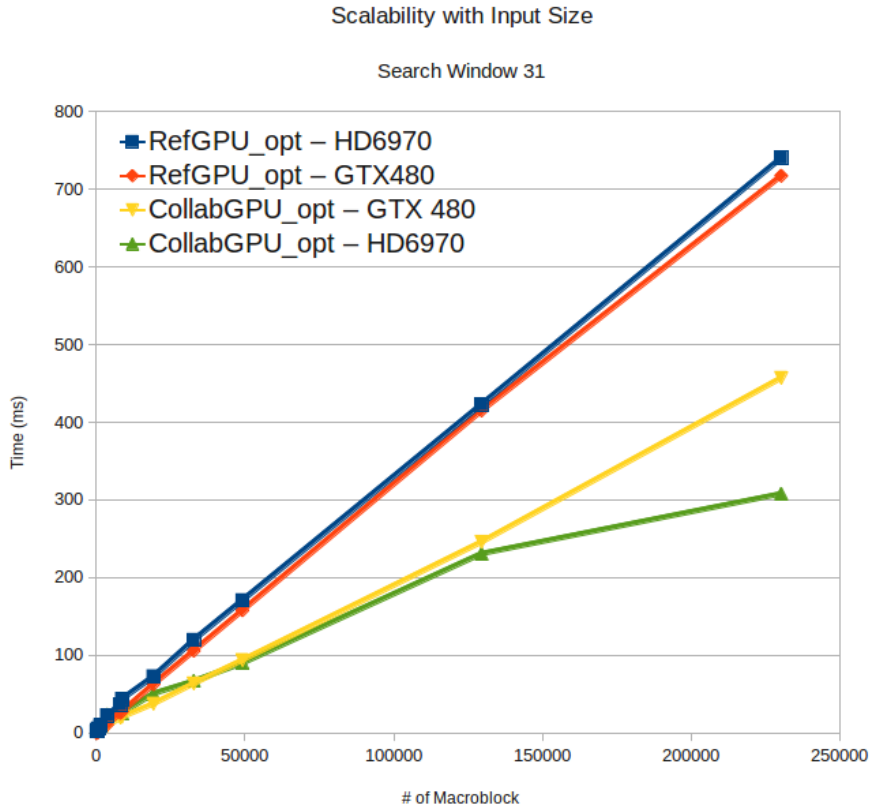


Figure 11: Methods Comparison and Scalability With Input Size - Window Size 31

In Table 2 and Figure 12 one could find the running time, in milliseconds, obtained

by running both methods on NVIDIA GTX480 and AMD HD6970. In the table, *RefGPU_GTX* refer to running the *RefGPU_opt* method on the NVIDIA GTX480 video card, while *CollabGPU_HD* refers to running the *CollabGPU_opt* on the AMD HD6970 video card. The running times presented are measured in milliseconds.

Table 2: Scalability with Input Size, Search Window 47, Platform Comparison

Frame Size	**#MBs**	**OpenCL**			**CUDA**	
		RefGPU_GTX	CollabGPU_GTX	CollabGPU_HD	RefGPU_GTX	CollabGPU_GTX
176x144	99	1	2	8	0	4
256x192	192	2	3	10	1	4
352x288	396	4	5	13	2	6
352x480	660	7	8	17	4	7
720x480	1350	14	15	19	9	12
1280x720	3600	37	29	43	24	24
1920x1080	8100	83	70	66	55	54
2048x1080	8640	88	64	64	59	54
2560x1920	19200	197	139	134	132	119
4096x2048	32768	336	238	204	225	204
4096x3072	49152	503	355	269	338	306
7680x4320	129600	1335	951	557	909	810
7680x7680	230400	2372	1787	961	1616	1439

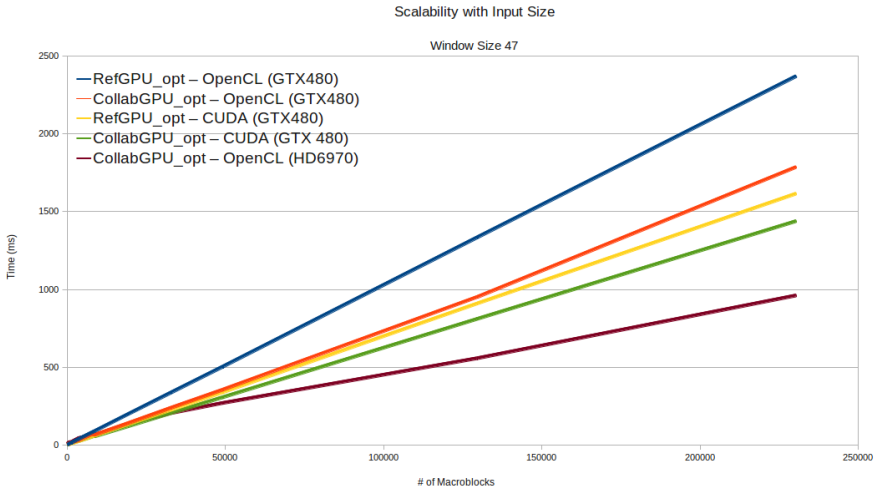


Figure 12: Methods Comparison and Scalability With Input Size

As expected and presented before, on both platforms, for lower image resolution, the *RefGPU_opt* performs slightly better, while our *CollabGPU_opt* performs better for very high resolution frames. Again, the HD6970 is the best performing platform for larger resolutions, while having its performance penalties on lower resolutions. The reasons for these results were presented before when presenting the results for the search window size of 31. The reader can also see that the CUDA implementation of our method

is faster than its corresponding OpenCL one. This happens even though the code of the two kernels is equivalent, as the OpenCL kernel code is simply a "translation" from the CUDA kernel code. As reported in [26], the results could be closer if more platform and OpenCL specific optimizations will be deployed in the OpenCL code. These optimizations refer to adopting as much as possible platform specific optimizations, use same kernel-level primitives and optimizations (loop unrolling, same memory access pattern, avoid branching), ensure that the kernel code is compiled to same ptx (assembly) code. Also, the authors report that the kernel launch time is larger for OpenCL than CUDA. Making use of these optimizations remains for future work. Moreover, one could see that for smaller input sizes, the CUDA implementation of *CollabGPU_opt* is slower than its OpenCL equivalent. This happens because, as presented in Sections 6 and 7, the CUDA version of this method makes use of multiple kernel calls while the OpenCL version does all the work with only one kernel call. Even though all these kernel calls are issued on multiple CUDA compute streams, they add a small overhead for scheduling the kernel call. For these small input sizes, the scheduling overhead cannot be hidden by stalling computations issued from other kernel calls. This behaviour is not a concern when the input size grows, as the kernel scheduling latency is hidden by the computations done on previous kernel calls. As another future improvement, we could implement the OpenCL scheduling strategy on CUDA in order to reduce these latencies.

As a conclusion to this section, the *CollabGPU_opt* method is more suited for encoding larger resolution videos. Another advantage that it employes is that it can work on larger search windows. These larger search windows should offer a better encoding quality - higher PSNR (peak signal to noise ratio) values. Moreover, we can also pick a best platform for deploying our video encoding solution. Thus, when it comes to lower resolution videos, one could choose any of the two GPUs benchmarked in this section, while for the three highest video resolutions, the AMD HD6970 is the clear winner.

8.3 Scalability with Variation of Search Window Size on GPUs

In this section, we provide the same type of measurements, but with a variation also in the search window size. We have chosen search window sizes of 25, 31, 36, 40, 47. As our results from Table 3, the measurements report the same behaviour as for the maximum search window size: the *RefGPU_opt* performs slightly better for smaller frame size, while *CollabGPU_opt* scales better with the increase in frame size.

Table 3 shows the results for the aforementioned search window sizes except W = 47, because this was presented in the previous set of benchmarks. The GPU platform on which the evaluation was made is NVIDIA GTX480. We chose this GPU platform because it permitted us to use a wider range of search window sizes for the *RefGPU_opt* method. Hence, on this platform we are able to better compare the two methods and their scalabililty with respect to the search window size.

In Figure 13 we plotted the results collected for running the motion estimation program written in OpenCL for various window and frame sizes. As expected, our *CollabGPU_opt* performs better than the *RefGPU_opt* method when the frame size increases. The reason for this behaviour was presented in the previous section. Also, our

Table 3: Scalability with Search Window Size

Frame Size	#MBs	W=25		W=31		W=36		W=40	
		Ref (ms)	Collab (ms)	Ref (ms)	Collab (ms)	Ref (ms)	Collab (ms)	Ref (ms)	Collab (ms)
176x144	99	0	1	0	1	0	1	1	1
256x192	192	0	1	1	1	1	1	1	2
352x288	396	1	1	1	2	2	2	3	3
352x480	660	1	2	2	2	3	4	5	5
720x480	1350	2	3	4	4	7	7	10	9
1280x720	3600	6	5	11	9	18	14	26	19
1920x1080	8100	16	11	26	20	42	35	59	47
2048x1080	8640	17	11	28	21	45	35	63	49
2560x1920	19200	38	22	63	38	101	65	141	90
4096x2048	32768	64	37	106	64	171	111	240	151
4096x3072	49152	96	54	158	94	256	165	360	225
7680x4320	129600	250	137	415	246	674	433	946	593
7680x7680	230400	423	264	717	457	1175	790	1659	1075

method performs better even when the search window size increases. This happens as a consequence of the fact that larger search windows also imply more data being read from the global memory on a single thread block for the *RefGPU_opt* method. This, combined with the fact that not all accesses are coalesced results in a higher number of uncoalesced reads. Moreover, with a larger search window, more shared memory bank conflicts appear. All these conclude to a lower scalability with both frame and search window size for the *RefGPU_opt* method.

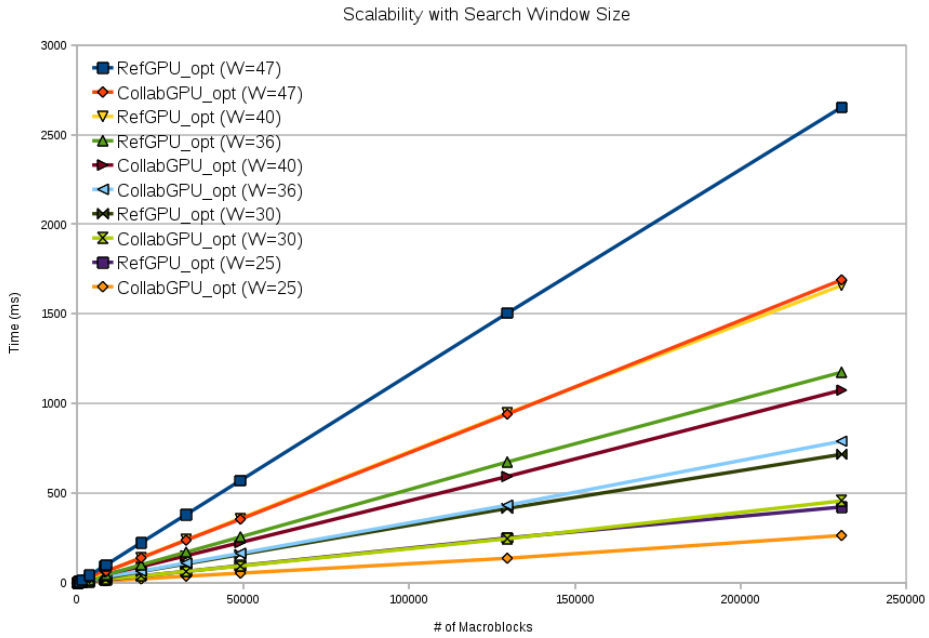


Figure 13: Scalability With Search Window Size - NVIDIA GTX480

As a comment, one can see in Figure 13 that our *CollabGPU_opt* achieves same performance as *RefGPU_opt* for larger window sizes. For example, the *CollabGPU_opt* for W = 47 is as fast as *RefGPU_opt* for W = 40. The same is applicable to other window sizes - visually one can observe that some of the curves of the graph are clustered two by two. In conclusion, if better quality is needed, for encoding larger video resolution. the most suited method is the *CollabGPU_opt*.

8.4 CPU Implementation Evaluation

In this section we will show how our Collaborative method performs on CPUs in comparison to the Reference method. We will also show how the modified version with unaligned reads behaves. Moreover, we will show how the size of the thread-block (work-group) affects the performance for the CPU version of these algorithms. We decided to run these methods on the same set of input sizes presented before and the search window size of 47 (largest possible for the *RefGPU_opt* method on the GTX480 video card) because we wanted to show how CPUs perform in comparison with GPUs on such algorithms. The platform on which we benchmark the solutions presented in this section is Intel Xeon 5620.

We begin by showing in Table 4 and Figure 14 how the variation of the threads per block parameter is an important optimization for OpenCL CPU implementations. As we mentioned before, in contrast with GPU platforms, where the preference is for a large number of threads per block, when aiming at high performance for CPU implementations, smaller block sizes are preferred. For the Reference methods, namely *RefGPU_opt* and *RefCPU_opt*, this number cannot be varied without a consequence - modification of the search window size. Ideally, a large search window is preferred. This implies a large number of threads per block. Hence, minimizing this number for the *RefCPU_opt* method could come with two costs: a smaller search window, or, a redesign of the algorithm - a thread computes more than one SAD computation. As we already mentioned, for our Collaborative method the thread block size is not connected to the search window size. Hence, optimizing the thread block size parameter is easily achievable. In Section 6 we presented two methods for implementing our Collaborative method on CPUs. The first is the same as the GPU implementation, while the second involves a further optimization: no shared memory and synchronization are needed because on CPU platforms, unaligned reads from memory are permitted. Naturally, this second version is expected to perform better because the removal of the overheads involved by the shared memory usage and the synchronization. In Table 4 and Figure 14 we present a comparison between these two methods while the number of threads per block varies. In the table and graph this is depicted with the parameter T=2,4,8. The running times presented are measured in milliseconds. We only present the results for T=2,4,8 because we noticed that a further increase in this parameter would not bring any performance improvements for the *CollabCPU_unaligned* method, while for the *CollabCPU_opt* method it would bring a decrease in performance because more shared memory is used.

Table 4: CPU Methods Comparison - Variation of Number of Threads per Block

Frame Size	**#MBs**	**CollabCPU_opt**			**CollabCPU_unaligned**		
		T=2	T=4	T=8	T=2	T=4	T=8
176x144	99	11	11	13	7	7	7
256x192	192	20	21	24	13	13	13
352x288	396	43	44	48	28	33	27
352x480	660	71	73	80	45	44	44
720x480	1350	148	151	164	95	92	91
1280x720	3600	404	404	440	260	249	246
1920x1080	8100	908	903	984	629	582	573
2048x1080	8640	966	965	1052	664	617	591
2560x1920	19200	2127	2133	2319	1497	1435	1394
4096x2048	32768	3732	3712	4061	3771	2786	2454
4096x3072	49152	6450	5976	6473	5737	4596	4026
7680x4320	129600	19729	16981	17163	15280	11843	11131
7680x7680	230400	34953	30121	30644	27069	21160	20352

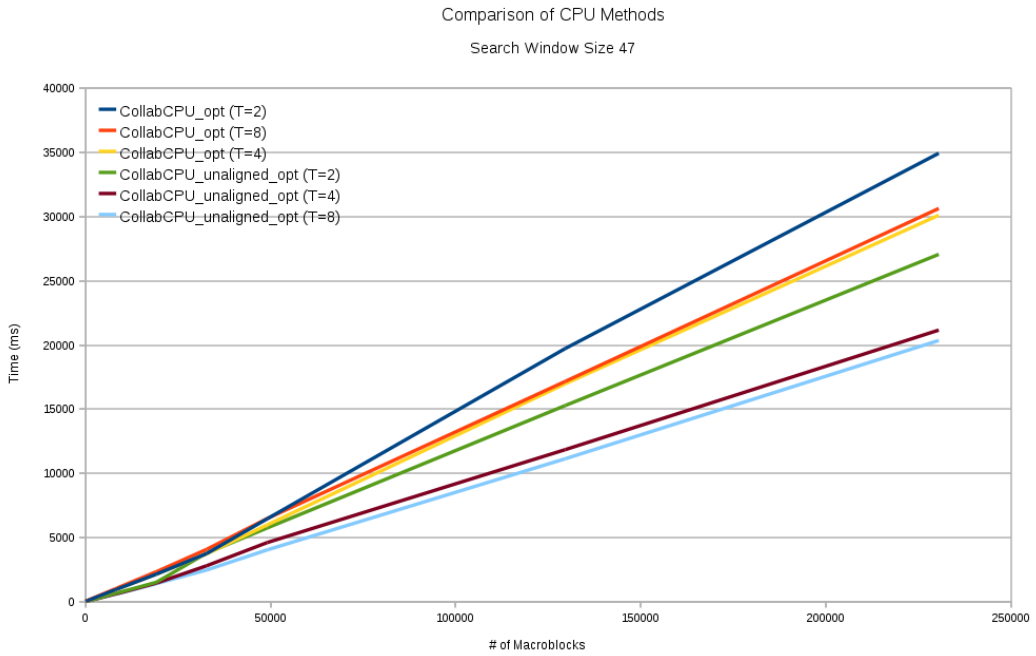


Figure 14: CPU Methods Comparison - Variation of Number of Threads per Block

As expected, the *CollabCPU_unaligned* method performs better than *CollabCPU_opt*. This is, of course, because the former does not make use of shared memory or synchronization. Also, for increasing values of T, the reader can see that, as expected, the performance increases for larger frame sizes. For *CollabCPU_opt* we notice a decrease in performance from T=4 to T=8. This is, as mentioned before, because of the increase in the shared memory usage. Hence, the best performing implementations here are *CollabCPU_opt* for T=4 and *CollabCPU_unaligned* for T=8. Next, we will compare these

two implementations with the *RefCPU_opt* method. This is presented in Table 5 and Figure 15. The measured running times are presented in milliseconds.

The results are the expected ones - both Collaborative methods perform faster than the *RefCPU_opt* method. This is because we have been able to optimize our *CollabCPU_opt* method adapting the thread block size parameter. The mean speedup of this implementation compared to the *RefCPU_opt* is 1.96. Optimizing further (i.e. making use of unaligned reads) permitted us to achieve a solution that does not use shared memory and synchronization between threads. This method, *CollabCPU_unaligned* achieves a mean speedup of 3.10 compared to the *RefCPU_opt* method.

Table 5: Comparison of CPU Methods

Frame Size	#Macroblocks	RefCPU_opt	CollabCPU_opt (T=4)	CollabCPU_unal (T=8)
176x144	99	30	11	7
256x192	192	48	21	13
352x288	396	92	44	27
352x480	660	149	73	44
720x480	1350	293	151	91
1280x720	3600	769	404	246
1920x1080	8100	1707	903	573
2048x1080	8640	1829	965	591
2560x1920	19200	4054	2133	1394
4096x2048	32768	6888	3712	2454
4096x3072	49152	10426	5976	4026
7680x4320	129600	27373	16981	11131
7680x7680	230400	48513	30121	20352

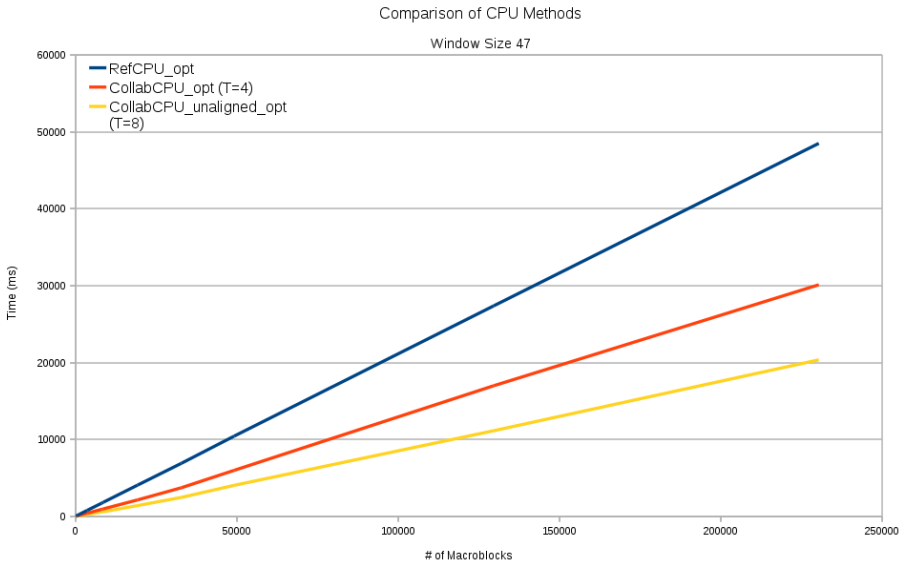


Figure 15: CPU Methods Comparison

In conclusion, the method that is best suited for deploying on a CPU platform is the *CollabCPU_unaligned* method with parameter T=8 threads per block.

8.5 Multi-Device Scalability

In this section we present the results obtained by running our application collaboratively on multiple devices. These platforms are, as stated in Section 1, the following: multi-CPU (AMD Magny-Cours Opteron), multi-GPU (multiple NVIDIA GTX-580), CPU-GPU systems (AMD HD6970 + Intel Xeon E5620). The evaluation was done using our OpenCL multi-device framework running *CollabGPU_opt* or *CollabCPU_opt* methods.

8.5.1 Multi-GPU Scalability

In this subsection we will present and analyze the results obtained by benchmarking our Collaborative method implemented in OpenCL on a multi-GPU machine. This machine is equipped with 8 NVIDIA GTX580 cards. To keep our results consistent with the other evaluations we made, we kept the same list of input frame sizes and the search window size of 47. We tested our solution using 1, 2, 4 and 8 GPUs in order to evaluate the scalability of our solution with respect to the number of compute devices. The results are presented in Table 6 and Figure 16.

Table 6: Multi-GPU Scalability, W=47

Frame Size	#Macroblocks	1 GPU (ms)	2 GPUs (ms)	4 GPUs (ms)	8 GPUs (ms)
176x144	99	2	2	4	10
256x192	192	3	2	5	13
352x288	396	5	4	6	14
352x480	660	8	6	9	16
720x480	1350	13	8	10	19
1280x720	3600	25	16	17	27
1920x1080	8100	59	35	28	36
2048x1080	8640	54	32	27	35
2560x1920	19200	118	69	50	62
4096x2048	32768	199	114	80	87
4096x3072	49152	300	166	117	130
7680x4320	129600	787	440	300	305
7680x7680	230400	1408	770	522	528

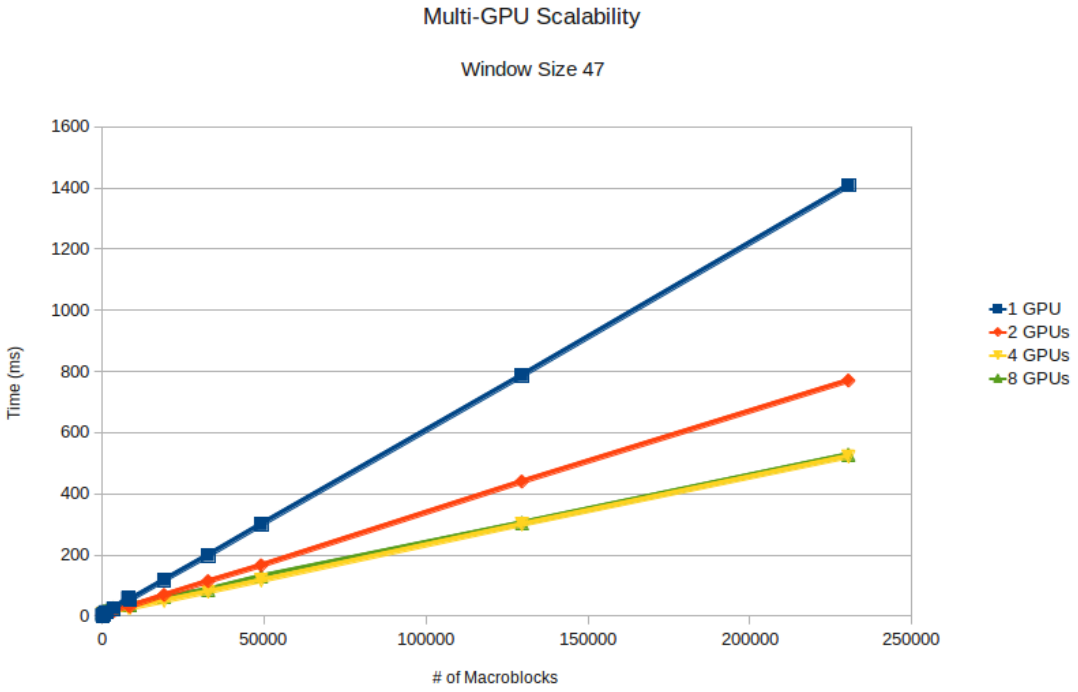


Figure 16: Multi-GPU Scalability

One can immediately notice from Table 6 and Figure 16 that the performance improvement of our solution is decreasing when the number of devices increase. In other words, the speedup achieved is not perfect when comparing results to the running times obtained when running the solution on only one GPU. Moreover, only when using 2 and 4 GPUs the performance increases. When using 8 GPUs, the performance is marginally lower than for using 4 GPUs. In order to explain why does this happen we present in Figure 17 the architecture of the 8 GPU machine. In this figure, one can see that each two GPUs share a PCIe switch and each two of these switches share an IO Hub. The performance bottleneck comes from the IO Hubs that can only serialize transfers that come from the same CPU. Thus, the communication phase of our Collaborative method (sending data to the device and copying it back) cannot be executed in parallel for more than two GPUs at a time. This is why the speedup is almost perfect for using two GPUs and, for 4 and 8 GPUs the speedup decreases. In conclusion, unless another architecture is used for connecting the CPUs and the GPUs, using 4 GPUs obtains the best performance for our solution. Moreover, for small frame sizes it is clear that the multi-GPU utilization does not bring the expected results. However, for larger frame sizes, the extra level of parallelism added (splitting computation on multiple GPUs) is worthwhile.

To present further proof that indeed the communication part of our solution limits

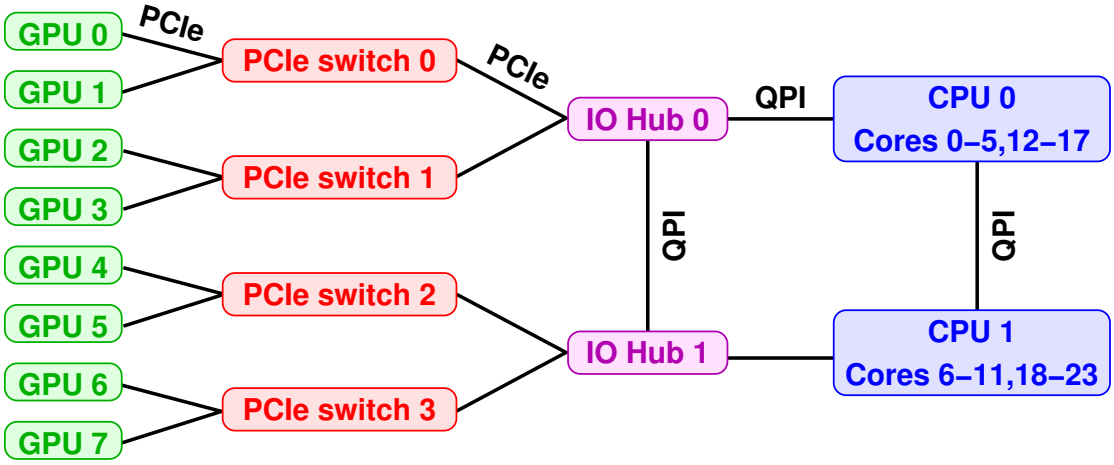


Figure 17: Multi-GPU Machine Architecture

the scalability with respect to the number of devices, we will further show that increasing the amount of work that is done on each GPU concludes in better scalability. This is achievable by increasing the search window size. This increase in the search window size only increases the amount of computation that is done on each device. The memory transferred between the host and the devices is independent of the search window size. It only depends on the frame size. For this experiment we used for benchmarking the highest eight frame sizes. Splitting computation on multiple devices for frames smaller than 1280x720 does not bring too much increase in performance. If one would need only to encode frame sizes of less than 1280x720, it would not make too much sense to invest in multi-GPU machines. We showed before that using only one GPU for small frame sizes achieves running times of less than 10 milliseconds for motion estimation.

In Table 7 and Figure 18 we present the results obtained by running our Collaborative method on 1, 2, 4 and 8 GPUs. The search window size was increased to 70 pixels.

Table 7: Multi-GPU Scalability, W=70

Frame Size	#Macroblocks	1 GPU (ms)	2 GPUs (ms)	4 GPUs (ms)	8 GPUs (ms)
1280x720	3600	68	44	30	34
1920x1080	8100	183	97	62	51
2048x1080	8640	182	96	59	51
2560x1920	19200	331	173	104	86
4096x2048	32768	563	295	171	137
4096x3072	49152	847	441	251	197
7680x4320	129600	2252	1168	708	510
7680x7680	230400	4002	2079	1162	851

Table 7 and Figure 18, in comparison with Table 6 and Figure 16 are relevant to show that the communication is the bottleneck for a multi-GPU architecture running our solution. However, if larger search windows are employed, thus increasing the work to be done on each device, better scalability is achieved. In conclusion, multi-GPU

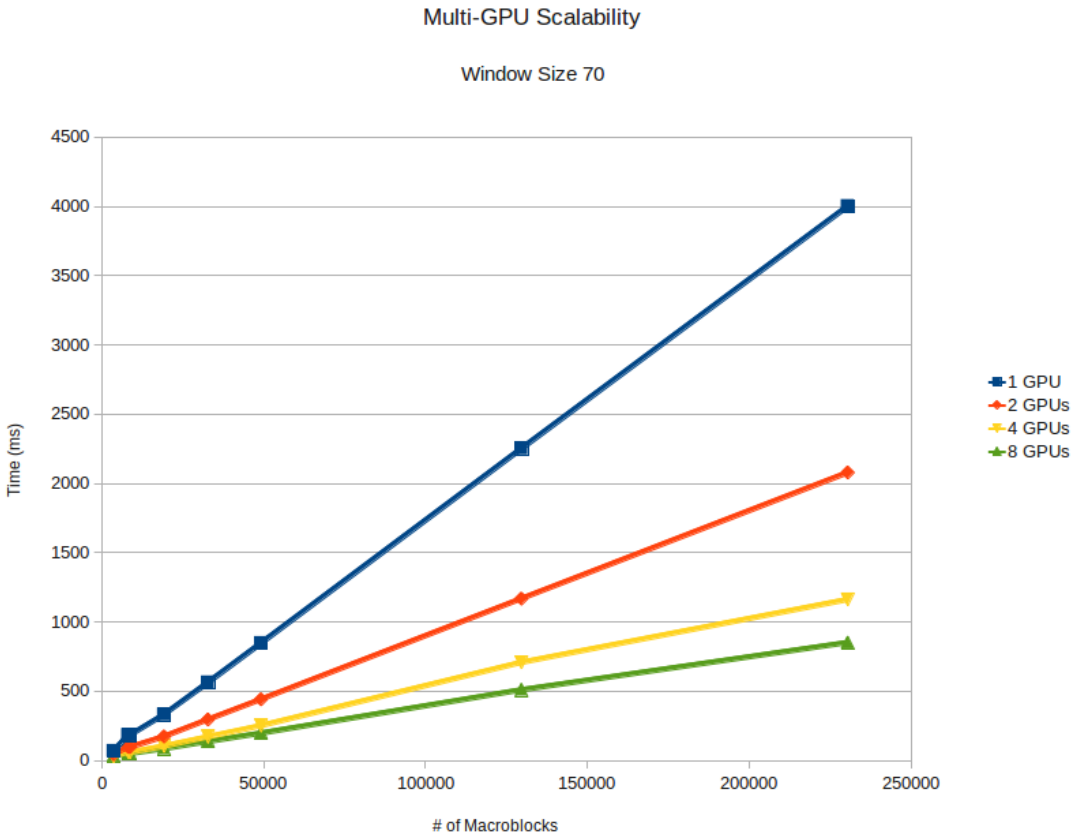


Figure 18: Multi-GPU Scalability

systems are efficient to use when one needs to encode a video with large frame sizes and for which large search windows are used. The usage of larger search windows could enhance the quality of the encoded video. Hence, when high encoding quality is needed for videos with frame sizes larger or equal to 1280x720 pixels, multi-GPU systems are suitable HPC platforms.

8.5.2 CPU-GPU Scalability

In this section we discuss our approach at achieving a motion estimation solution capable of running collaboratively on a CPU and a GPU that reside on the same machine. Usually, when developers write GPU applications, the CPU offloads computation to the GPU and, while the GPU is computing, the CPU stays idle waiting for the result. We want to achieve a motion estimation solution where through collaboration between these two devices, the overall performance of the system is increased.

Our first attempt at achieving this performance increase through CPU-GPU collab-

oration is similar to the multi-GPU approach presented before. Thus, each device would get to compute a part of the motion estimation jobs (as defined in Section 6). When deciding the amount of jobs to be computed by each device, we have to take into consideration the results previously presented. Our fastest CPU implementation is, on average, 7 times slower than the GPU implementation. Also, it is obvious that the difference in performance the two platforms exhibit increases when the frame size increases.

For very high resolution videos (larger than 1920x1080) there is a high difference between CPU performance compared to GPU performance. We concluded that the improvement obtained by this approach on such frame sizes would be less than 10%, thus we only ran our compute kernels collaboratively on CPU and GPU for lower resolution frames (less than 1920x1080).

The experiments that we ran are based on the following framework: for each frame size, we computed how much faster the GPU is compared to the CPU (denote this number x); then, the CPU would get only $\frac{1}{x}$ of the total number of jobs; after the computation, the results are merged. Unfortunately, our results are not the ones expected. Instead of decreasing with $\frac{1}{x}$, the running time (computation + memory transfers) slightly increases compared to the GPU-only running time.

After a performance evaluation breakdown of this implementation, we identified the main reason for this unexpected situation. This is the overhead added by the pthread library [32]. As we mentioned in Section 7, for the evaluation of the project, we use the *one-thread-per-device* approach, which means that when scheduling work for each OpenCL device, we use one CPU thread that manages the communication between the host and the device. The identified bottleneck is introduced, thus, by the "pthread_join()" call from the main processing thread. This call is issued in order to wait for the "device-threads" to finish copying back the memory from the devices.

We will illustrate how this happens with an example: in the case of 256x192 frame size, the AMD HD6970 GPU processes the input in 13 milliseconds; for the same input size, the Intel Xeon E5620 CPU processes the input in 13 milliseconds as well. Hence, splitting the computation between the two is straighforward: each device processes half of the input. In theory, the speedup should be approximately two. In practice, the overall running time of the collaborative solution is 14 milliseconds (1 millisecond more than the running time of the GPU-only computation). When we measured the actual running time of each "device thread", we got running times of 6 milliseconds for the CPU and 7 milliseconds for the GPU. Thus, the overall speedup would be indeed approximately 2. However, after measuring the time between the end of the copying of the memory from the devices until the exit from the "pthread_join()" call, the added overhead is of 7 milliseconds. Adding this overhead to the total compute time we achieve the 14 milliseconds overall running time.

In conclusion, this CPU-GPU collaborative approach is not feasible. When using small input sizes, the pthread library adds a performance penalty. For very large frame sizes, the amount of work that the CPU can handle is so small that the actual performance improvement is also negligible if we take into consideration the overhead induced by the pthread library.

8.5.3 Multi-CPU Scalability

In this section we will present the results obtained on the 48 core AMD Magny Cours and compare the two motion estimation methods. In Table 8 and Figure 19 the reader can see the actual results of our evaluation for the two methods on the multi-CPU platform.

Table 8: Multi-CPU Scalability, W=47

Frame Size	#Macroblocks	RefCPU_opt	CollabCPU_unaligned_opt (T=256)
146x144	99	14	3
256x192	192	19	5
352x288	396	31	10
352x480	660	45	16
720x480	1350	81	31
1280x720	3600	199	72
1920x1080	8100	436	149
2048x1080	8640	439	167
2560x1920	19200	1028	907
4096x2048	32768	1712	1370
4096x3072	49152	2560	2335
7680x4320	129600	6871	10268
7680x7680	230400	13573	19400

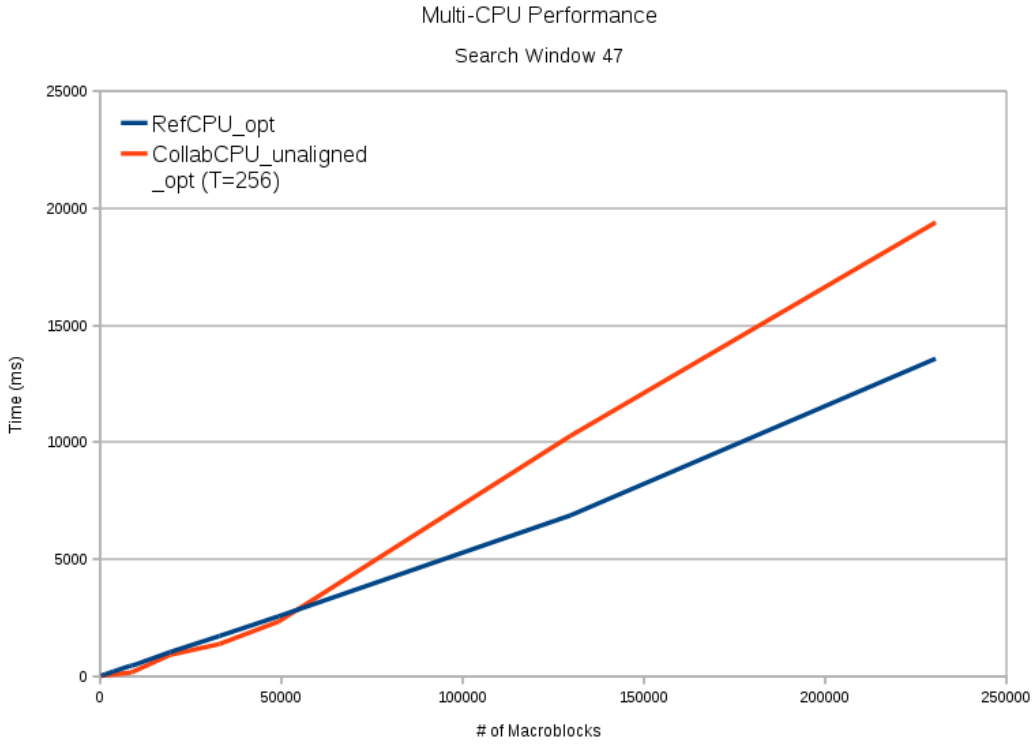


Figure 19: Multi-CPU Scalability

For this evaluation we used the the *CollabCPU_unaligned_opt* method which performs better than the *CollabCPU_opt* method for CPU platforms. Again, as for the single-CPU evaluation, we used multiple configurations of threads per block. The best results achieved are for 256 threads per block for this AMD Magny Cours multi-CPU.

Out of all the platforms on which we evaluated the two motion estimation methods, we found a platform on which the Reference Method performs better than the Collaborative Method. But, this happens only for the two largest frame sizes. Our explanation for this behaviour is that this AMD Magny Cours multi-CPU benefits from large caches that really speed up computation for the Reference Method, as it makes use of shared memory for loading the macroblocks and the entire search window. Hence, these large caches serve well as shared memory for the two largest frame sizes.

However, for the other frame sizes, the Collaborative Method performs better. This is an expected behaviour, as it adheres to the OpenCL CPU recommendations that we mentioned in Section 7. Also, the irregularities that can be seen in Figure 19 for the *CollabCPU_unaligned_opt* for the lower resolution frame sizes can be explained by the fact that we do not use a fixed thread block size. In our implementation of the Collaborative Method, for lower resolution videos, the thread block size is the minimum between the maximum number of macroblocks that fit on the width of the frame and 256 (the thread block size for which best performance is achieved). Hence, for lower resolution images, the number of threads per block is not 256, but a lower value. This approach breaks the multi-CPU cache hierarchies and the performance is discontinuous when the video resolution increases. After a certain input size, the thread block size stabilizes for the 256 size and the graph curve does not show irregularities any more. Using a fixed thread block size for the Collaborative Method implementation remains future work.

8.6 Qualitative Evaluation

In this section we present qualitative measurements and show the improvements that result from the ability of our *CollabGPU_opt* to arbitrarily increase the search window size. For the measurements we take into consideration the case of the AMD HD6970 GPU for which the *RefGPU_opt* method could not benefit from search windows larger than 31 pixels. For this experiment we only considered a reference frame and a current frame that is to be encoded. The reference frame is used to create a prediction for the current frame. We then measured the mean square error (MSE) and peak signal to noise ratio (PSNR) values between the current frame and its prediction. The search window sizes used are the following: 31, 35, 47, 55 and 79. Two datasets were used for these measurements. First, we used a still picture and simulated a camera movement by producing another picture using a geometric translation. This picture was taken by us, using a common digital camera. The geometric translation was peformed over a distance of 40 pixels in both dimensions. We chose this approach in order to show that our Collaborative method could easily catch longer motion vectors, while the Reference method could only test motion vectors bounded by the thread-block (work-group) size. For this generated dataset, the frame size used is 1280x720 pixels. In order to validate

the results obtained with the generated data, we chose a second dataset. The second dataset is composed by two consecutive movie frames. For this second dataset, the frame size used is 1920x1080 pixels. In Tables 9 and 10 we present the actual results. In Figure 20 (a) and (b) we show a cropped part of the reconstructed frame for the second dataset. The images presented here correspond to search window sizes of 31 and 79.

Table 9: Qualitative Measurements - First Dataset

Search Window Size	PSNR	MSE
31	15.02	2042.77
35	15.84	1691.45
47	47.64	1.11
55	47.64	1.11
79	47.64	1.11

Table 10: Qualitative Measurements - Second Dataset

Search Window Size	PSNR	MSE
31	18.10	1006.79
35	19.47	733.53
47	29.31	76.13
55	29.53	66.53
79	30.81	53.87

As expected, the results show that increasing the search window size improves the achieved encoding quality. Moreover, this does not only happen for generated data, but also for consecutive frames from real video scenes. Taking into consideration that the AMD HD6970 GPU could only achieve a search window of size 31 for the *RefGPU_opt* method, we can conclude that running our *CollabGPU_opt* method on this GPU can indeed increase the encoding quality because it can make use of larger search windows.

(a) Search Window Size 31 (b) Search Window Size 79

Figure 20: Qualitative Evaluation

8.7 Discussion

In the Experiments and Results section we presented the complete evaluation of our HPC motion estimation software module. We first showed that our *CollabGPU_opt* method performs faster (with a factor of approximately 1.35) and scales better than the *RefGPU_opt* method. Then, we showed that our Collaborative method scales well with respect to the search window size and does not imply a loss of performance when the search window size is increased. This is important because we also showed that higher search window sizes imply better encoding quality. The best performing platform for higher resolution videos is the AMD HD6970, while for lower resolution movies any of the two GPUs tested perform well.

Further, we showed that our Collaborative method is better suited than the Reference Method even on CPU platforms. It has proven to be faster with a factor of approximately 2.5 than the *RefCPU_opt* method. We discussed scalability also for multi-GPU machines. In this case, the achieved speedup depends on the underlying architecture and the actual workload (which is dependant on the search window size). For larger search window sizes, the speedup and the overall performance achieved is better than for smaller search window sizes. Hence, multi-GPU architectures are well suited for videos where better encoding quality is needed. However, taking into account the overall system design from Section 2, where multiple encoding machines are involved, it is natural to discuss the case of multi-GPU machines in this context. When analyzing this situation, the fundamental question to be answered is about the system needs: do we need faster encoding per video or better throughput (more videos processed in the same time)? Taking into consideration a social media company's needs (many small videos uploaded, not necessarily high encoding quality needed) of keeping latencies to a minimum while serving the needs of as much clients as possible concurrently, we can draw the conclusion that probably investing in multi-GPU machines is of a lower priority than investing in multiple single-GPU machines. Though, however, if the company seeks to build, for example, an online movie platform where higher encoding quality is necessary, these multi-GPU machines would bring extra encoding quality without sacrificing performance.

9 Conclusions and Future Work

In this thesis we presented our approach at achieving a high performance motion estimation solution designed to work on multiple multi/many-core devices. First, we implemented and optimized the *RefGPU* found in [8, 9, 10, 11]. Due to its design, this method implies two bottlenecks: it cannot make use of the underlying platform compute power and, more important, it limits the search window size parameter. The usage of small search window sizes leads to poor encoding performance.

Second, we designed our own *CollabGPU* method which solves motion estimation while eliminating the aforementioned bottlenecks. Due to the new design, this method splits the computation into different jobs for each motion vector of the search window.

Using this "sliding window" technique, we were able to make better use of the underlying platform compute capabilities and achieve better performance. We showed that our method is not only faster, but it also scales better than the previous method with respect to the frame size. Moreover, we showed that by making use of larger search window sizes, our Collaborative method can achieve greater encoding quality. This method was also deployed on CPU platforms for which further optimizations were designed. For both multi- and many-core platforms our Collaborative method is performing better then the Reference Method. For future work, we will analyze and compare the approach presented in [9] for extending the search window size with our Collaborative method. Also, enhancing the Reference Method with a tiling [19, 30] technique to increase the search window size might also be of interest.

Third, we designed an OpenCL framework that can make use of all compute resources of a system in order to achieve a high performance motion estimation. This framework splits the computation done by our Collaborative method on multiple devices. We showed that depending on the underlying system architecture and on the computation-to-communication ratio, our multi-device solution scales well. For future work, we will compare our Collaborative method with the Reference Method with regard to the scalability on multiple devices.

Regarding the CPU-GPU collaborative work, we showed in Section 8 that our approach at achieving this is not feasible. For future work, we would like to propose another approach at achieving CPU-GPU collaboration. This new method works as follows: while the GPU is computing motion estimation for the current frame, the CPU could achieve the same for the next frame, but with much smaller search window. In this way, the computation for the two will overlap. Using the CPU for estimating the motion for a future frame with smaller search window could help in the following way: if certain predefined PSNR value is obtained, then good quality is obtained because there is not much motion involved. In this way, the GPU would not have to process the next frame, and thus two frames would have been processed in the same time. However, if the predefined PSNR value is not met, the GPU will process this next frame making use of its larger search window and thus obtaining better encoding quality. Another way in which CPU-GPU collaboration could be achieved would be to use the same approach, but, instead of using a smaller search window for the CPU, one could use a downsized version of the frame. In this way, the CPU would act like a predictor for the GPU, helping it to adjust the size of the search window for the next frame: if the CPU detects high velocities are involved, then the GPU could use a larger search window; otherwise, the GPU could decrease the search window in order to spare valuable compute time.

We mentioned in Section 6, that we do not use a static thread-block (work-group) size for the motion estimation processing kernels. Our Collaborative method makes use of dynamic thread-block sizes (defined by the width of the picture divided by macroblock size). We took this approach because it offered a more natural and simplistic implementation of our Collaborative method. Usually, when implementing GPU compute kernels, programmers use fixed thread block sizes. Taking this approach enables the programmer to choose the thread-block size which achieves the best performance. Using this

kind of approach will be addressed in future work to further optimize our Collaborative method. Also, our compute kernels could further benefit from a tiling optimization as macroblocks (16x16 areas of pixels) can be naturally addressed as tiles.

Further, one of the most important future enhancements for the projects would be to create a high performance video encoder that could make use of our collaborative solution. Moreover, accelerating other encoding processes on multi/many-cores will be a challenge that we will have to address. For example, we could use a technique like in [27] in order to implement the variable length coding module [1]. Embedding such an encoder in the overall system design presented in Section 2 is also something to consider for future work.

To sum up, we achieved a high performance motion estimation module that is able to run on multiple devices in a collaborative fashion. This approach is not only faster than current approaches, but it also adds qualitative improvements to the encoded video file.

10 References

1. I. Richardson, H.264 and MPEG-4 Video Compression: Video Coding for Next-Generation Multimedia, John Wiley and Sons, 2004

2. http://www.nvidia.com/object/cuda_home_new.html, 2012

3. http://www.khronos.org/opencl, 2012

4. S. Ryoo, C. Rodrigues, J. Stratton, S. Ueng, S. Baghsorkhi, W. Hwu, Program Optimization Carving for GPU Computing, Journal of Parallel and Distributed Computing, pages 1389-1401, 2008

5. http://www.hyves.nl, 2012

6. T. Chen, Cell Broadband Engine Architecture and its First Implementation - A Performance View, IBM Journal of Research and Development, 2007

7. T. van Kessel, N. Drost, J. Maassen, H. E. Bal, F. J. Seinstra, User Transparent Data and Task Parallel Multimedia Computing with Pyxis-DT, CCGRID '12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012) Pages 17-24

8. W.Chen, H.264/AVC Motion Estimation Implementation on Compute Unified Device Architecture (CUDA), 2008 IEEE International Conference on Multimedia and Expo, pages 697-700, 2008

9. R. Gaetano, B. Pesquet-Popescu, OpenCL Implementation of Motion Estimation for Cloud Video Processing, 2011 IEEE 13th International Workshop on Multimedia Signal Processing, pages 1-6, 2011

10. N. Cheung, X. Fan, M. Kung, Video Coding on Multicore Graphics Processors, IEEE Signal Processing Magazine, pages 79-89, 2010

11. L. Chan, J. Lee, A. Rothberg, P. Weaver, Parallelizing H.264 Motion Estimation Algorithm using CUDA, IAP Massachusetts Institute of Technology, 2009

12. http://mashable.com/2010/10/28/facebook-activity-study/, 2012

13. J. Dinan, S. Krishnamoorthy, D. Larkins, J. Nieplocha, P. Sadayappan, Scalable Work Stealing, In Proceedings of 21st Intl. Conference on Supercomputing, 2009

14. R. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, H. E. Bal, Ibis: An Efficient Java-based Grid Programming Environment, In Proceedings of ACM-ISCOPE Conference on Java Grande, pages 18-27, 2002

15. J. Kramer, Advanced Message Queuing Protocol (AMQP), Linux Journal Archive, Volume 2009, Issue 187, 2009

16. K. Trivedi, D. Wang, D. J. Hunt, A. Rindos, W. E. Smith, B. Vashaw, Availability Modeling of SIP Protocol on IBM WebSphere, 14th IEEE Pacific Rim International Symposium on Dependable Computing, pages 323-330, 2008

17. S.Ryoo, C. Rodrigues, S. Stone, J. Stratton, S. Ueng, S. Baghorski, W. W. Hwu - Program optimization space pruning for a multithreaded GPU. In: Proceedings of the 6th annual IEEE/ACM International Symposium on Code generation and optimization, ACM (2008) 195204

18. M. Harris, Optimizing parallel reduction in CUDA, NVIDIA Developer Technology, 2007

19. B. van Werkhoven, J. Maassen, F. J. Seinstra - Optimizing Convolution Operations in CUDA with Adative Tiling. In: Proceedings of the 38th ACM IEEE International Symposium on Computer Architecture, 2011

20. D. Patterson, The Top 10 Innovations in the New NVIDIA Fermi Architecture, and the Top 3 Next Challenges, 2009

21. C. M. Wittenbrink, E. Kilgariff, A. Prabhu, Fermi GF100 GPU Architecture, NVIDIA, 2010

22. A. Munshi, The OpenCL Specification, Khronos OpenCL Working Group, 2012

23. Intel SDK for OpenCL Applications - Optimization Guide, Intel, 2012

24. P. O. Jaaskelainen, C. S. de La Lama, P. Huerta, J. H. Takala, OpenCL-based Design Methodology for Application-specific Processors, 2010 International Conference on Embedded Computer Systems, pages 223-230, 2010

25. http://www.cs.vu.nl/das4/

26. J. Fang, A. L. Varbanescu, H. Sips, A Comprehensive Performance Comparison of CUDA and OpenCL, 2011 International Conference on Parallel Processing, pages 216-225, 2011

27. A. Balevic, Parallel Variable-length Encoding on GPGPUs, In proceedings of the 2009 International Conference on Parallel Processing, pages 26-35, 2009

28. R. V. van Nieuwpoort, G. Wrzesinska, C. J. H. Jacobs, H. E. Bal, Satin: A High-level and Efficient Grid Programming Model, ACM Transactions on Programming Languages and Systems (TOPLAS), article no. 9, 2010

29. N. Drost, R. V. van Nieuwpoort, J. Maassen, F. J. Seinstra, H. E. Bal, JEL: Unified Resource Tracking for Parallel and Distributed Applications, Concurrency and Computation: Practice and Experience, Volume 23, Issue 1, pages 17-37, 2011

30. C. Xu, S. Kirk, S. Jenkins, Tiling for Performance Tuning on Different Models of GPUs. In: ISISE '09 Proceedings of the 2009 Second International Symposium on Information Science and Engineering, 2009

31. http://www.astron.nl, 2012
32. D. R. Butenhof, Programming with Posix Threads, Addison-Wesley Professional, 1997