BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY

DEPARTMENT OF INTELLIGENT SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

COMPARISON OF PARALLEL PROGRAMMING APIS

SROVNÁNÍ PARALELNÍHO PROGRAMOVÁNÍ API

BACHELOR’S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR AUTOR PRÁCE	SAMUEL ALFAGEME SAINZ
SUPERVISOR VEDOUCÍ PRÁCE	Ing. FILIP ORSÁG, Ph.D.

BRNO 2016

Abstract

Parallel computing is a relevant issue nowadays since virtually every CPU manufactured is built with multi-core features. The way to introduce parallel code in our applications is still a long road ahead, but some tools have already proved a high level of maturity. This thesis aims to be a limited and practical approach to them: it begins briefly describing the main application programming models in use for parallelism and laying the theoretical foundations for this topic. In order to illustrate the comparison, the implementation of a video stabilization algorithm based in plane matching is presented and ported to different modern APIs, such as OpenMP, CUDA and OpenCL, along with a methodology to compare efficiency and speedup. Also, we introduce some opinionated metrics to evaluate the developer experience while using these tools.

Abstrakt

Paralelní programování je v současné době moderní technologií, neboť téměř každý procesor bývá vybaven více jádry. Způsoby začlenění paralelního kódu do aplikací zatím nejsou příliš dokonalé, ale existuje již několik způsobů, které prokázaly svou vyspělost. Tato práce si klade za cíl představit některé z paralelních přístupů z praktického hlediska počínaje stručným popisem hlavních modelů paralelního programování a teoretického úvodu do problematiky. Jako demonstrační algoritmus ilustrující různé přístupy k paralelnímu programování je použit algoritmus stabilizace obrazu, který je naprogramován v moderních rozhraních, jako například OpenMP, CUDA a OpenCL. Dále je představena metodologie pro porovnání efektivnosti a zrychlení výsledného kódu spolu se subjektivním srovnáním uživatelské přívětivosti uvedených přístupů k paralelizaci.

Keywords

Parallel Computing, GPU, GP-GPU, OpenMP, CUDA, OpenCL, Video Stabilization, Gray-Code Bit Plane Matching

Klíčová slova

Paralelní Programování, GPU, GP-GPU, OpenMP, CUDA, OpenCL, Stabilizace Videa, Hledání Binárního Vzoru

Reference

ALFAGEME SAINZ, Samuel. *Comparison of Parallel Programming APIs.* Brno, 2016. Bachelor’s thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Orság Filip.

Comparison of Parallel Programming APIs

Declaration

I declare that I have created this thesis myself under the supervision of Ing. Filip Orság, PhD. I have cited all the bibliographic sources and publications used for the creation of this thesis.

. . . .
Samuel Alfageme Sainz
July 29, 2016

Acknowledgements

I need to thank everyone that has supported me in this adventure which is the Erasmus program: my awesome family, the friends who cared and asked me how all was coming along from the distance and the bunch of new friends who took great care of me in Brno. The experience has been wonderful and one-off a lifetime, has definitely changed me and eventually strengthened me in a troubled moment.

Special mention goes to the second-semester Spanish family: Coke, Nacho, Dani, Maider, Leire, Laura, Rober and Marc, you guys deserve a very bright future and I wish you all the best for the road ahead.

Siempre nos quedarán las risas.

Contents

Chapter 1

Introduction

In 1971 the fastest car in the world was the Ferrari Daytona, capable of 280kph. The world's tallest buildings were New York's twin towers, at 415 meters. In November that year Intel launched the first commercial microprocessor chip, the 4004, containing 2.300 transistors, each the size of a red blood cell. [...] Since then chips have improved in line with the prediction of Gordon Moore, Intel's co-founder: processing power doubles roughly every two years as smaller transistors are packed ever more tightly onto silicon wafers, boosting performance and reducing costs. [...] If cars and skyscrapers had improved at such rates since 1971, the fastest car would now be capable of a tenth of the speed of light; the tallest building would reach half way to the Moon.

The future of computing
The Economist — Mar. 12th 2016 issue

1.1 Motivation

Parallelism is a very important topic in computer science and industry today. For almost 20 years the performance of microprocessors increased, on average 50% per year relying in something called **frequency scaling**: as transistors evolved, they became smaller and faster, meaning that users and developers could just wait for the next generation to improve the speed of their applications. Since 2002, this average has slowed to about 20% because some physical limits are being met, resulting in a major change in microprocessor design: by 2005, the major manufacturers realized that the way to increase performance from then on laid in the direction of parallelism and started putting multiple complete processors on a single integrated circuit.

But not everything started with the end of frequency scaling. Parallelism has come a long way since the 1960s, when the first supercomputers were introduced. Therefore, it has been usually associated with high performance computing. Also, the evolution of technology has made possible hardware upgrades that got better each passing year. This change has

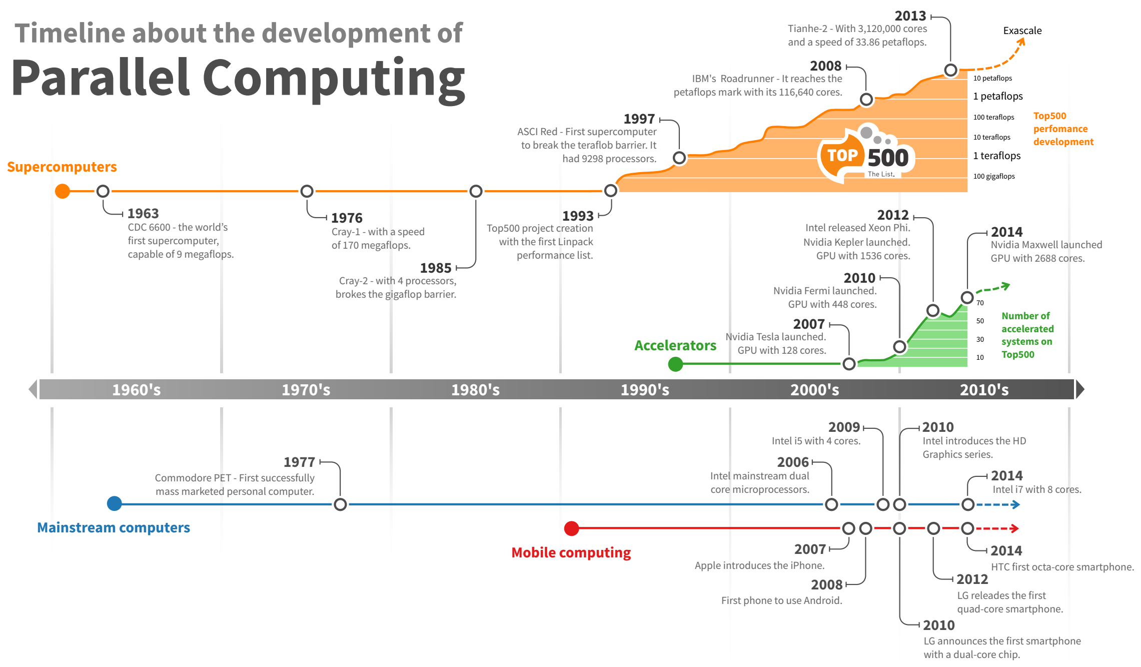

Timeline about the development of
Parallel Computing
Supercomputers
1963 CDC 6600 - the world's first supercomputer, capable of 9 megaflops.
1976 Cray-1 - with a speed of 170 megaflops.
1985 Cray-2 - with 4 processors, brokes the gigaflop barrier.
1993 Top500 project creation with the first Linpack performance list.
1997 ASCI Red - First supercomputer to break the teraflob barrier. It had 9298 processors.
2008 IBM's Roadrunner - It reaches the petaflops mark with its 116,640 cores.
2013 Tianhe-2 - With 3,120,000 cores and a speed of 33.86 petaflops.
Exascale
10 petaflops
1 petaflops
100 teraflops
10 teraflops
1 teraflops
100 gigaflops
Top500 perfomance development
TOP 500 The List.
Accelerators
2007 Nvidia Tesla launched. GPU with 128 cores.
2010 Nvidia Fermi launched. GPU with 448 cores.
2012 Intel released Xeon Phi. Nvidia Kepler launched. GPU with 1536 cores.
2014 Nvidia Maxwell launched GPU with 2688 cores.
70
50
30
10
Number of accelerated systems on Top500
1960's
1970's
1980's
1990's
2000's
2010's
Mainstream computers
1977 Commodore PET - First successfully mass marketed personal computer.
2006 Intel mainstream dual core microprocessors.
2009 Intel i5 with 4 cores.
2010 Intel introduces the HD Graphics series.
2014 Intel i7 with 8 cores.
Mobile computing
2007 Apple introduces the iPhone.
2008 First phone to use Android.
2010 LG announces the first smartphone with a dual-core chip.
2012 LG releades the first quad-core smartphone.
2014 HTC first octa-core smartphone.

Figure 1.1: Parallel computing timeline.[11]

been recorded by the Top500 project[21], which ranks the 500 most powerful computers in the world: since the the project started in 1993, the performance of the top system in the list has always shown a steadily grown and current supercomputers are massive parallel machines with millions of processing units. Some important events for parallelism can be seen in fig.1.1.

This project aims to be a practical approach to modern parallel programming frameworks and prove its value by implementing and improving the performance of a consumer algorithm such as a video-stabilization method.

The document is structured as it follows: this first chapter reviews the main parallel models and gives details on some popular implementations of them in use nowadays. Chapter 2 gives some insight into the theory that supports the parallelism itself and present some broad terms to help the reader understand the rest of the document. In Chapter 3 we describe the GC-BPM, the video stabilization algorithm selected to enlighten the frameworks comparison as well as the computer vision work environment required to start working on its implementation, which is later described in detail in Chapter 4 along with its equivalent in the different tools selected: OpenMP, CUDA and OpenCL. Chapter 5 would present the experiments that have to be carried out to test the improvement achieved by executing the algorithm in multiple processing elements and how this can be translated to numbers. And to conclude, the last chapter would summarize all the work made by the author in this Thesis and describe some feasible lines for future work.

1.2 Parallel Programming Models and APIs

Models exist as an abstraction above hardware and memory architectures to simplify its comprehension and offer a version of the underlying system, in this sense, we have to talk about programming models to refer the style of programming where execution is invoked by making what appears to be encapsulated library calls. What distinguishes a programming model from a normal library is that the behavior of the call cannot be understood in terms of the language the program is written in.

In parallel computing, the execution model usually exposes features of the hardware to achieve high performance. The large amount of variation in hardware causes a need for numerous parallel execution models and because it is not practical to make a new language for each it is common practice to invoke parallel execution model behaviors, like the creation of a thread or the launch of a kernel, via an API.

1.2.1 Shared Memory

This is the most rudimentary form of parallelism, and it's often used in combination with some others. Processes share a common address space, which they read and write asynchronously using mechanisms such as locks and semaphores to prevent data races and deadlocks.

Shared memory machines, OS, compilers and hardware support this form of programming. Some examples are:

- POSIX standard shared memory APIs.
- UNIX shared memory segments (`shmget`, `shmat`, `shmctl`, ...).

1.2.2 Threads

They are an extension of the shared memory techniques, introduced by E.W. Dijkstra in his paper *Cooperating sequential processes* where he proposed a model for concurrent operating systems based on "heavy" process with multiple lightweight concurrent execution paths, which after were renown as threads. Since then, many implementations have been created to exploit the multi-threading model:

- **Pthreads** (POSIX Threads) - consist in both an execution model and API, and are available as standard in most mainstream OS. It offers a low level interface to control everything regarding threads: from their management to synchronization, etc.
- **OpenMP** - High-level extension for C/C++. It allows many programs to be parallelized with relative ease and a few compiler directives. It appeared in 1997 as the joint effort of a consortium of manufacturers to simplify the parallelization of some constructions such as static loops (where the number of iterations is fixed) like the ones that can be found in numerical computations with matrices.

1.2.3 Message Passing

This model is oriented and best suited for distributed memory machines but it can be translated to any kind of machine as it is an abstraction for complex systems, at expense of the communication and synchronization steps falling into the hands of the programmer. Implementations are usually presented as a library of functions, directly called from the source code.

MPI - Message Passing Interface

Message passing libraries have been around since the 1980s, when their relevance increased to the point of creating a forum to standardize a implementation, resulting on the MPI specification, "de facto" standard for message passing nowadays.

Using MPI has many advantages. It can be used to solve a wider range of problems than thread libraries and, as said before, the programs can run on either shared- or distributed-memory architectures. On the other hand, creating a parallel application with MPI can be complex, because the programmer has to determine the data dependencies, divide the algorithm into tasks, and then implement the message passing and synchronization.

1.2.4 General-Purpose Computing on GPUs

The GPUs (Graphics Processing Unit) were designed initially just for graphics and rendering images by computer, but as manufacturing became cheaper and they proved to be deeply programmable and accurate with a very low costs (power consumption compared to regular purpose CPUs is only a fraction) they started to make appearance in supercomputers while beating in some aspects the performance of traditional CPUs as can be seen in figure 1.2.

Since NVIDIA launched its model Tesla GPU card in 2007, the use of such coprocessors along with the general-purpose processors has become a big trend. Many of the latest supercomputers are based on accelerators that have specialized hardware making them suitable for particular types of computation. Classified as hybrid manycore systems, they include lots specific-purpose processors packed with their own instruction set and multilevel hierarchy of memory.

In the framework of accelerating computational codes by parallel computing on graphics processing units (GPUs), the data to be processed must be transferred from system memory to the graphics card's memory, and the results retrieved from the graphics memory into system memory. In a computational code accelerated by general-purpose GPUs (GPGPUs), such transactions can occur many times and may affect the overall performance, so that the problem of carrying out those transfers in the fastest way arises. We can sum up what happens in a GPUs as it follows:

1. Setup inputs on the host (CPU memory)

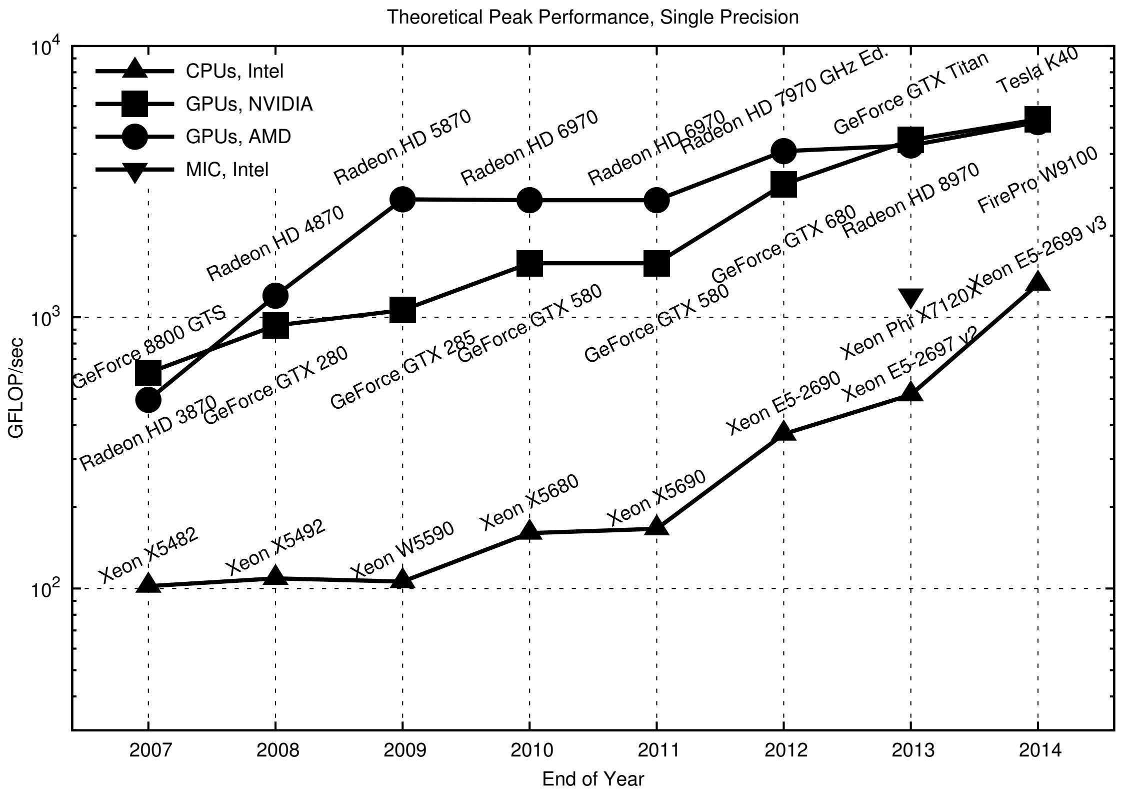


Figure 1.2: Comparison of annual theoretical peak GFLOP/sec in single precision. Higher is better.[18]

2. Memory allocations:
    (a) Inputs on the GPU
    (b) Outputs on the host
    (c) Outputs on the GPU
3. Copy inputs from host to GPU
4. Start GPU kernel
5. Transfer the output from GPU to the host

CUDA

The CUDA platform is just a software layer that gives the programmer direct access to the GPU's virtual instruction set and parallel computational elements. To put it in plain words, CUDA extends the GPU functionality to general purpose computing. It abstracts the parallelism details relying in the concept of **kernels**, which are just plain C functions that, when called, are executed in parallel by N different CUDA threads. [7]

1.2.5 Hybrid

They combine more than one of the previously described models to suit the combination of architectures they run in and they are currently the most popular model exploited on clusters of multicore machines. Some examples in use we can mention are made combining some of the already described frameworks such as **OpenMP + MPI** or **MPI + GPGPUs**. It requires to mix several tools with different programming models. Moreover, the appearance of accelerators has added an additional layer of complexity: A programmer must be proficient in MPI, OpenMP, and CUDA (or equivalent) to be able to take advantage of the current generation of parallel systems.[11]

OpenCL

As happened with OpenMP, when hybrid parallelism mixing CPUs and GPUs started to gain popularity, a consortium was formed by the main hardware and software manufacturers to develop a framework for supporting heterogeneous computing.

The main advantage of using OpenCL is that is the first open, royalty-free standard for **cross-platform** processors and GPUs allowing high level of portability while keeping the abstraction layer relatively close to the hardware, minimizing the overhead. As CUDA, it uses kernels as execution model, which in this case, can speak with many different devices. But all this advantages don't come without a price, and this one is that is not an easy framework to learn [19]: it comprises some unique data structures and functions and a particular way of doing as we will discuss later.

Chapter 2

Theoretical Foundations

2.1 General Aspects of Parallelism

When someone explains what is an algorithm to a broad public, usually starts by making a real-world analogy with a food recipe: a computation is just a problem broken into a discrete series of instructions, which traditionally were executed **sequentially**, that is in the order they were written, one by one by a single processor. This could fit pretty much as the definition for traditional serial computing.

2.1.1 Concurrent versus Parallel

At its simplest, parallel computing is the **simultaneous** use of multiple compute resources to solve a problem. When stepping into parallelism for the first time, one can be confused with concurrency concepts because the illusion of multiple tasks running at once. But a difference should be pointed: concurrent programming is just a form of computing in which several computations are executing during **overlapping** time periods, while parallel has some special characteristics:

- More than one **simultaneous processes** are running at the same time physically.
- There's a need for communication and/or synchronization systems to put together the results of these processes.

2.1.2 What Can Be Achieved With Parallelism?

Many computing problems are so huge that they cannot be solved sequentially in a reasonable time. In the natural world, many complex, interrelated events happen at the same time, yet within a temporal sequence. Compared to serial computing, parallel computing is much better suited for **modeling, simulating and understanding complex, real world phenomena** [5]. And something to take into account is that computer modeling is a never-ending run so every advance in performance just makes the models more detailed.

How is the improvement of the parallelism measured

Generally, we measure the performance of computer programs based on how much time do they need to run, this usually turns out to be directly related to the size of the input data set the program works with. If we translate this to the parallel world, and as squeezing more cores in an integrated circuit or rewriting an application to take advantage of parallelism is not cheap, we have some results to take into account before considering these options.

- **Speedup**: is used to measure the parallelism efficiency. For a given data set of size n, it is described as the ratio:

$$S = \frac{t(n)}{t(n,p)} \tag{2.1}$$

  - $t(n)$ the reference sequential algorithm run time.
  - $t(n,p)$ the parallel run time when using p processors.

- **Scalability**: is the property to keep the speedup growing constantly as the number of processors increase.

What cannot be achieved with parallelism: Amdahl's law

To some inexperienced eye, parallelism could look like the holly grail of computing: just add more processors and divide the time it takes to programs to run. Far from reality; Gene Amdahl noted[2] that the potential program speedup is defined by the fraction of code (t_p) that can be parallelized, which could not be as high as 100% as communications, synchronization or I/O must be carried out in a sequential fashion. We can realize from these notes that there are some limits to the scalability of parallelism.

If we decompose the run times in both members of the speedup fraction 2.1 based on how much of the program cannot be parallelized (t_s) we have the following equation.

$$\lim_{p\to\infty} S = \lim_{p\to\infty} \frac{t_s + t_p}{t_s + \frac{t_p}{p}} = 1 + \frac{t_p}{t_s} \tag{2.2}$$

As the number of processing elements tend to be infinite, the speedup reaches its asymptotic limit as can be seen in graphic 2.1. Meaning that, after a finite number of processors adding more just results in the same speedup, which can be translated in same run time for the same problem size.

In 1988 J.L. Gustafson [12] made some notes about Amdahl's law limitations, because it was only based in a fixed workload size, the sequential fraction of the program does

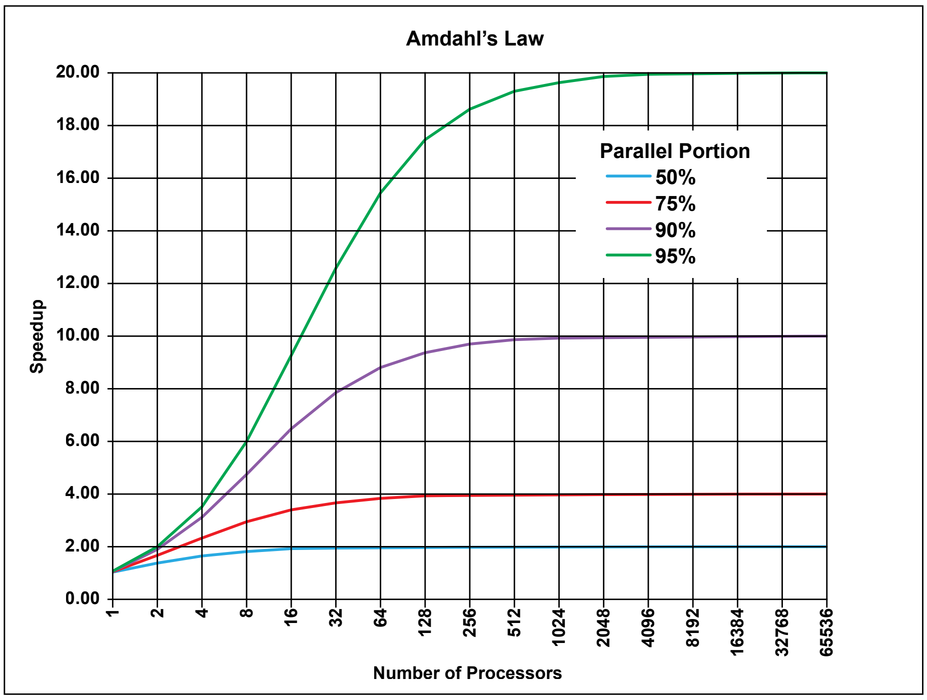


Figure 2.1: Speedup as processing units increase for different portions of parallelizable code. Source: https://commons.wikimedia.org/wiki/File:AmdahlsLaw.svg

not change while increasing the resources. Instead, Gustafson proposed to set the size of problems to fully exploit the computing power that becomes available as the resources improve. Under this vision, as equipment becomes faster, larger problems can be solved within the same time.

2.1.3 Parallel Architectures: Flynn's Taxonomy

In order to organize parallel machines, we can rely on Flynn's taxonomy[9], it distinguishes multi-processor computer architectures according on to how they can be classified along the two independent dimensions of **Instruction** and **Data** stream (sequences).

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Table 2.1: Matrix defining the 4 architectures in Flynn's taxonomy

- **SISD**: Represents traditional serial (non-parallel) computer.
- **SIMD**: Single instruction, meaning processing units execute the same instruction at any given clock cycle, but each one can operate on a different data element (because of this, it's well suited for *specialized problems* with a high degree of regularity, such as graphics processing). Two subgroups of this category can be distinguished:

  - **Processor Arrays**: e.g. Streaming Multiprocessors (SM) located in some modern GPUs.

  - **Vector Processors**: like the very first parallel computers, which introduced vector registers to allow operating with several data at once, still present in most of today's commercial CPUs.

- **MISD**: Separate instruction streams in contrast to a single data stream that feeds the processing units. Few examples ever existed, but some conceivable uses could be those requiring redundancy or multiple filters feed by the same data.

- **MIMD**: The most complex and common form of parallelism: every processor may be executing a different instruction stream with a different data stream than the others, so execution can be synchronous or asynchronous, deterministic or non-deterministic. Many MIMD architectures also include SIMD. Too broad to be useful on its own; is typically decomposed according to memory organization. We will review that in next section.

2.1.4 How Do We Write Parallel Programs

As it's neither trivial nor easy to write translators that convert sequential programs to parallel, we do need to actually rewrite our programs using a parallel approach and to do so we depend on the basic idea of partitioning and distributing the work to be done among the processing elements. When these can work independently, writing a parallel program could be much the same as the serial version, but problems appear when the data and the results are dependent on each other, like finding a matrix minimal or sorting a list of numbers.

There are two broad, but not closed, classes used when talking about parallel algorithm design:

- **Data parallelism** – where parallelism translates into performing the **same operation** to different data at the same time.

- **Task parallelism** – in which parallelism is the result of different computations at the same time. A fundamental form of task parallelism is pipelining, the technique exploited in the already mentioned vector processors.

As rule of thumb, we can say PCAM (also known as Foster's) Methodology [10] proposed by Ian Foster is a comprehensive summary on which steps should be taken to transform a sequential algorithm in its parallel variant. It comprises 4 stages, which initials give the methodology its name.

- **P**artition: split both the computation and the data into a large number of small tasks.

- **C**ommunication: identify the necessary communication between the tasks to perform the computation.

- **A**gglomeration: required to achieve data locality and good performance as fine-grained partitioning is usually not an efficient parallel design. They can be combined into tasks of larger size.

- **M**apping: the final step is just to distribute the tasks between the processors available, following some guidelines:
  - Tasks that can execute concurrently map to different processors.
  - Tasks that communicate frequently map to the same processor.

2.2 Parallel Memory Architectures

Computer memory architectures describe the way data is stored for the processor to access. As this is a fairly complex problem, again, we usually employ abstractions or models to simplify its understanding and isolate unnecessary details. Since the early era of computing, one model has remained as virtually all computers have followed, it is von Neumann architecture or "stored program computer". It comprises four main components: Input/Output, memory, a control unit and an arithmetic-logic unit. Both the program instructions and the data are stored in the memory, fetched by the control unit and then executed. This model is relevant for our purpose as parallel computers still follow this design, just multiplied in units.

As mentioned before, MIMD machines are too broad for most of the approaches as they comprise the majority of parallel implementations. There is a mainstream basic classification of them according to 2 criteria:

- How the data in memory is accessed.
- What kind of synchronization and communication mechanisms they use.

2.2.1 Shared Memory

Generally, these machines have in common the ability for all processors to access all the available memory as a global address space. According on how the memory access and the processors are, they're classified in:

- **Uniform** Memory Access (UMA) - represented by Symmetric Multiprocessor machines (SMPs, like the ones found in graphic cards), with many identical processors (SMs), resulting in equal access and access time to shared memory and cache coherence at hardware level, because of this there is no need to distribute data among processors. These do not scale well so they are limited to a small number of processors.
- **Non-Uniform** Memory Access (NUMA) - made by linking 2 o more SMPs, slowing down memory access across the links. To mitigate this effect, each processor has a cache storing the memory contents of some others, along with a protocol to keep the cache contents coherent.

2.2.2 Distributed Memory

As opposed as what happened in shared memory architectures, in these machines each processor has its own local memory and there's no such thing as global address space and to share the data among processors, they require a communication network. So when a processor needs access to data in another, it is the programmer responsibility to define the communications. Usually the inter-processor communication is achieved by message-passing. Also, as the compute nodes are not strictly connected, the need to synchronize appears and also becomes responsibility of the programmer.

Based on topology and technology used for interconnection, communication speed can range from almost as fast as shared memory to orders of magnitude slower. Two computer architectures fall in this category:

- Clusters (in Linux environments called Beowulf) which are the most popular and common form of distributed systems for parallelism. A cluster comprises a collection of independent nodes collaborating to achieve a common goal.
- Massive Parallel Processors (MPP) where processor and network infrastructure are tightly coupled physically.

2.2.3 Hybrid and Heterogeneous architectures

These machines use both shared and distributed memory architectures to increase the scalability at program complexity cost.

2.3 Benchmarking Techniques

As mentioned in section 2.1.2 a very important aspect when working developing parallel algorithms is to measure the time they need to execute and the speedup achieved by these techniques, first of all, to know if we are doing it right, and second, to explore the limits of what can be done with it.

Based on the specifics of the framework chosen, there are different approaches to measure the run time as they imply different models and behaviors. Here are some examples that have been taken from Gerassimos Barlas's book [4].

OpenMP

This case is pretty straightforward, as in every program using this API, there is an explicit point where the program launches the different threads, this marks when we can start measuring the time taken for the parallel fraction of the program. This can be achieved using some functions like `omp_get_wtime()`.

```
double timeStart, timeEnd;
timeStart = omp_get_wtime();
  // Portion of the program to be measured
timeEnd = omp_get_wtime();

printf("Total time spent: %f\n", timeEnd - timeStart);
```

Kernel-based models: CUDA and OpenCL

As there's many more details to take into account, these differ from the thread model, but as the platforms have evolved, the framework have as well, including some workarounds in the form of primitives and global variables. As example, in CUDA environments, the host can measure the timing of events (like kernel launches, memory transfers, etc.) on the GPU by inserting `cudaEvent_t` object in the streams.

```
// initialize the two events
cudaEventCreate(&startT);
cudaEventCreate(&endT);

// enclose kernel launch between startT and endT events
cudaEventRecord(startT, str);
doSomething<<<grid, block, 0, str>>>(...); // launch request has to be
     placed in a stream.
cudaEventRecord(endT, str);

// wait for endT event to take place
cudaEventSynchronize(endT);

// calculate elapsed time
cudaEventElapsedTime(&duration, startT, endT);
printf("Kernel executed for %f\n", duration);
```

Simplified Approach

After developing some of the different solutions, we realized that including the previous techniques was increasing hugely the complexity of the whole code, since they are more suited for a single-framework program rather than our three legged implementation. The final function used in the code was proposed and developed by Javier Fresno, a member of Trasgo Group of the University of Valladolid [22].

```
inline double cp_Wtime(){
  struct timeval tv;
  gettimeofday(&tv, (void *) 0);
  return tv.tv_sec + 1.0e-6 * tv.tv_usec;
}
```

Chapter 3

Algorithm

3.1 Overview

As video recording is mostly an error-prone process as it usually relies in the manipulation of some device by a human operator, the need to develop video post processing systems to avoid this kind of problems makes this an interesting field of study. Historically, many methods for image stabilization have been considered [20] and evolved hand in hand with hardware.

- Naive approach: One point track — locks the focus on a bright object in the background and keeps it in the exact same place, no matter how much the camera moves.
- Multiple point track: as we include more tracking points into account, we make possible to correct rotation or scaling in the original image, considering it only behaves well with static (meaning without motion) shots.
- Block Matching Algorithms: that use a post-process approach, where multiple video frames are used to determine the direction of the movement of the whole sequence.
- Advanced image post-processing: relies in filters that take into account many data and transformations. Some examples are the Deshaker algorithm included in VirtualDub or the Warp Stabilizer that comes with Adobe After Effects.
- Approaches based on metadata captured by extra elements integrated in modern video capture devices like gyroscopes, rolling shutters, etc.

To give some practical insight to this project we're going to focus in creating a parallel implementation of the **Gray-Coded Bit-Plane Matching** or GC-BPM algorithm proposed in[17]. This solution (which, as its name suggests, could fall in the Block Matching category) is an approach fairly straightforward with quite good results for non-rotating video sequences.

The main goal of this stabilization procedure is to calculate the "global" motion vector (GMV) of each video frame relative to the previous one. This vector tries to represent roughly the hand movement of the camera operator. With this information, we can determine whether the video sequence is displacing naturally or shaking unwittingly, and use it in a post process stage to generate a stabilized video sequence.

We can start by transforming each frame of the video into a grayscale image in which each pixel with coordinates (x, y) will be coded with 8 bits ($K = 8$) as it follows:

$$f^t(x, y) = a_{K-1}2^{K-1} + a_{K-2}2^{K-2} + \ldots + a_1 2^1 + a_0 2^0 \tag{3.1}$$

After obtaining these 8 bit planes, we can use them to compute their gray-code equivalent. The gray-coding step is quite relevant for this algorithm because it allows the next block to estimate the motion, even using a single bit-plane, by encoding most of the useful image information into a few of the planes. With the gray code, small changes in gray level yield a small, uniform change in the binary digits representing the intensity, meaning motion between frames. We could calculate this planes as:

$$g_{K-1} = a_{K-1} \tag{3.2}$$

$$g_k = a_k \oplus a_{k+1},\ 0 \leq k \leq K-2 \tag{3.3}$$

At each coordinate (m, n) within the search subimage, the matching method calculates C_j: a correlation matrix that represents the number of displaced pixels from the previous frame g_k^{t-1}, as:

$$C_j(m, n) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} g_k^t(x, y) \oplus g_k^{t-1}(x+m, y+n),\ -p \leq m, n \leq p \tag{3.4}$$

From there, we can obtain the 4 different local motion vectors V_j as the one pointing to the coordinate where this correlation is minimum for each subimage:

$$V_j = arg\ \min \{C_j(m, n), -p \leq m, n \leq p\}\ \forall j = \{1..4\} \tag{3.5}$$

At this point, we could carry on with the proposed on the paper [17] 3 Steps Search (3SS) to determine the motion vector using different bit-planes refining the search and reducing the computational complexity, but this doesn't contribute at the main purpose of our work, so we will use just the 6th bitplane as one of the tree mentioned with relevant information for our work. This been said, to estimate our final result we only need to apply a median filter; this is, to calculate the median for each component of the vectors involved, just to select the most representing vector as global. For this operation we would to take into account V_g^{t-1}, which is the global motion vector obtained by the algorithm in the immediately preceding frame, to feedback the filter:

$$V_g = \text{median}\{V_1^t, V_2^t, V_3^t, V_4^t, V_g^{t-1}\} \tag{3.6}$$

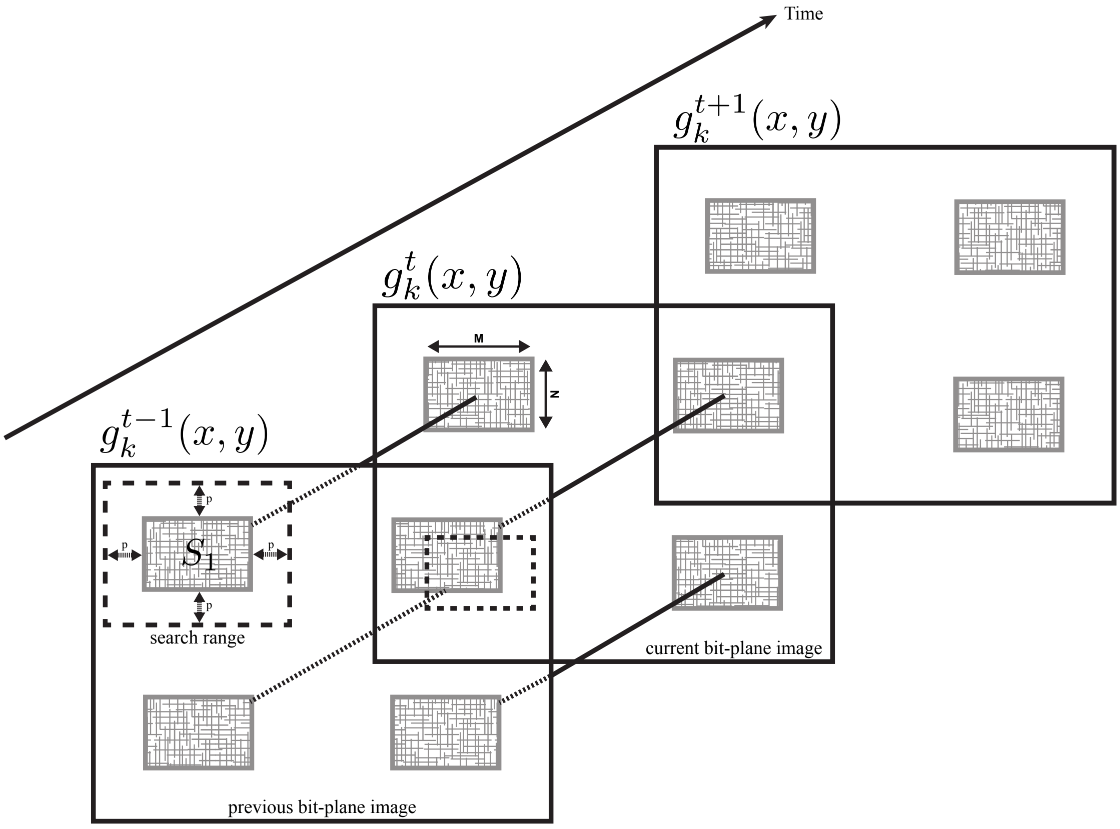


Figure 3.1: Estimation of local motion vectors from four subimages in a Gray-coded bit-plane

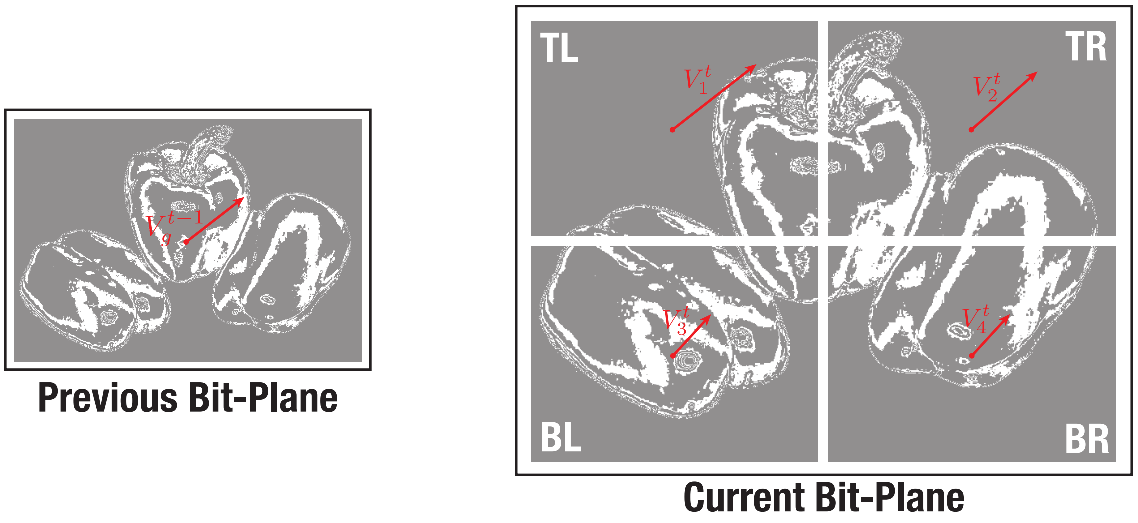


Figure 3.2: Obtaining the global motion vector from the previous one and the 4 local vectors for each frame

After gathering this information for all the single video frames, we could start the post-processing phase and use each GMV to precisely crop and center the original colored image sequence to generate a complete and fairly stabilized video, losing some resolution in the way, as described in the appendix D.

3.2 Work enviroment

To port all this math formulae and data transformations into code legible to a computer, two tasks appear to take into account:

First of all, in order to make the whole process "visible", the need to construct some kind of wrapper to **manage the different video transformation steps** arises: from the input sequence to the stabilized result. And also to be a bridge between the inputs, the different implementations of the algorithms and the outputs. As this is not a project about computer vision, we tried to avoid as much detail in this aspect as possible. We used OpenCV's C++ library for this purpose as it is a very mature framework with a quite soft learning curve, while providing some usable high level functions and data structures to work with computer vision that abstract some of the functionality to be carried out by this wrapper.

1. Video to single frames conversion, to load all the video frames in memory to make them manageable. They will be represented as OpenCV's `cv::Mat` data structures and loaded into a `std::vector` to represent the video. There's an important fact to take into account at this point: OpenCV has a well known bug (`http://code.opencv.org/issues/1287`) that causes the frame count to be incorrect when the video input has been modified with some tool without altering its metadata, this causes the program to fail at initial stages. The possibility to replace OpenCV in the video loading process keeps open to future work.

2. Transform single frames into bit planes: first of all losing all the color information to turn into grayscale images, decompose them into the different bit planes and use these to compute the gray-code planes as exposed in the equation 3.2. Then select the one that comes with relevant information (those identified as so by the paper [17] were the 4th, 5th and 6th), in our case it will be the 6th one. Finally store them in the original vector to feed the algorithm.

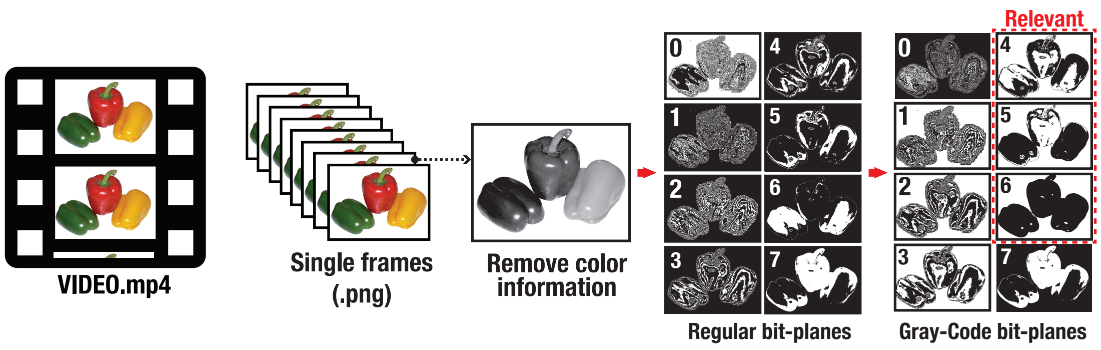


Figure 3.3: Different preprocessing steps made by the video wrapper.

The second problem to address is the translation of the GC-BPM procedure into code capable of being parallelized. We could take advantage of previous work implementing this algorithm in C/C++ [8] and Matlab [6] as reference for our work. The different algorithm implementation would be encapsulated in diverse functions that will take both the current and the previous bit planes as arguments as well as the x and y components of the previous frame's global motion vector and a couple of pointers to the current frame equivalent components that are set inside the function as they are calculated.

The inputs for all the solutions must specify the full path for the video file and, when talking about any of the parallel solutions, the number of processing units to be used to run the algorithm. And as to the outputs of the program concerns, every implementation must return two different pieces of information: the sequence of global motion vectors ordered according to the frame they belong and in first place and more important for the present case, the time measurement mark for the particular run as determined in-program by the function described in section 2.3.

Chapter 4

Implementation

4.1 Overview

Before going into specifics, we must discuss the implementation of the video stabilization strategy and some general aspects about the way to take advantage of parallelism in this particular problem.

We can broadly describe the algorithm in terms of pseudocode: first, we have a loop responsible of calling the stabilization routine passing the two frames required and the previous global motion vector. Then the method itself consisting in 3-nested-loops: one to consider the 4 subimages and 2 more to move the search window over the previous frame's subimage in X and Y directions. The different approaches, as it is usual when working with algorithms that involve operation with matrices goes trough correctly distributing the elements between the available processing elements.

The first way to exploit parallelism in which we can think of, and the most straightforward, is to parallelize the most external loop, the one that just repeats the second point of the process for each one of the subimages. But this will probably won suffice as the workload would be too big for a thread/work-item/processing-unit (stream, to be generic) and only could speed up this fraction of the program by 4, even if there's more processing elements available as happens with the GPUs.

The second approach to the problem involves the displacing search window in both dimensions for each of the 4 subimages that is translated in the two inner loops. This can be, relatively easy **as it can be reduced to a data-parallel problem**, in which each position of the so called correlation matrix will be filled by a different stream being possible to determine any of the elements of the matrix at any time as all the data is available from iteration one.

The problem is not that difficult to aboard in detail: on one hand, we have two shared and read-only memory portions which would be the current frame's central search window and the whole previous frame corresponding subimage. The last will be used inside the different streams to determine the appropriate search window to XOR with the first and carry on. Additionally, other shared memory portion that has to be modified by the streams would be the correlation matrix, each stream only modifies the right cell. The whole matrix contents only would be used when all of the streams have finished filling it, so there is no risk of data races or inconsistences. The only matter then is to distribute the indexes of the matrix between all the processing units. A graphic representation of this idea can be seen in the figure 4.1.

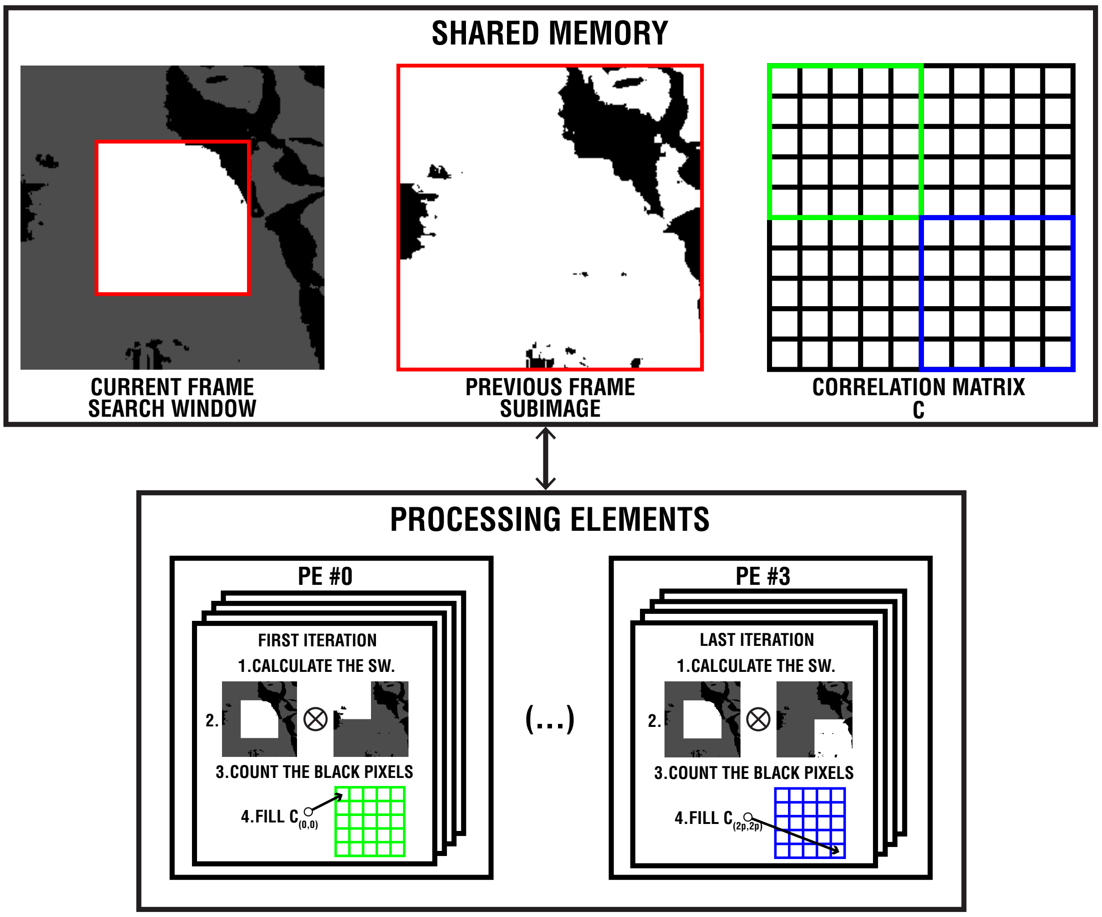


Figure 4.1: Parallelization of the 2 inner loops between 4 processors

Obviously, it would be interesting to combine this method together with the one mentioned in the first place as the only thing in which they differ is the subimage used. This could be done either by restructuring the code to include an implicit synchronization operation after all 4 correlation matrices are filled and the coordinates of the minimal calculated to get the GMV. Or, if the language supports it (like in the case of the two kernel-oriented execution models presented in this text), by using a three dimensional index space, where the Z-index would be the one determining the subimage (from 1 to 4).

The last way to board this issue would be to tackle down the data dependence problem between the iterations: as each global motion vector needs to take into account the previous one to be calculated and it might not be available yet. This would translate into parallelizing the (most outer) loop that calls the stabilization method, share the array that contains all the global motion vectors between all threads and synchronize them to calculate the result.

This method requires more implicit synchronization mechanisms and would imply more changes in the code to adapt the algorithm because it can be too many operations for a kernel. This solution differs from the ones presented until this point as can be classified as task-parallel instead.

As the reader can guess by the extension of the explanation given or as the method was qualified as "not that difficult", we finally choose the second one described for our purpose, it theoretically would offer enough speedup (up to a multiplicative constant of $1/(2 \cdot p)^2$ in the parallel fraction of the program if that many processing units are available) which will give enough representative results.

4.2 Solution

The software was designed in modules to separate all the responsibilities of the program as it involves a certain degree of complexity that must be managed to make it correct and legible at the same time. In this section, we would describe the different modules considered to build the final sequential solution, as it is the cornerstone of the project.

In first place, we needed an unified interface to run the program with all the possible inputs from an standard shell session. The file used for this purpose is `stabilize.cpp`, it contains the `main()` method, which handle the command-line arguments validation and also, what we called the "selector". It handles the execution of the different solutions based on the first argument passed to the application. As all the functions, sequential or parallel, require the same arguments (the two frames, the previous GMV and the empty pointers to the new), we could define a single entry point for the algorithm and dynamically select the version to be run based on the option selected.

An object-oriented approach would be to use a *strategy pattern* with a whole class hierarchy, but this will only make the solution more complicated compared, for example with the C-style way, which rely in modifying a function pointer in the switch statement. But as it is less obscure and antique, we could just take advantage of C++11 templates to highly simplify this just by switching between the target functions.

Then and also integrated in the previous file, as it is common to all the solutions, we have the "wrapper" we detailed in last chapter that receives and processes the video by dumping all the frames into memory and converting them to gray-code bit-planes by making use of OpenCV functions to ease the way they are later consumed by the algorithm.

A `commons.h` header file was included to improve the readability of the code and to factor out some common elements for all the versions, e.g. some constants relative to the proportions and internals of the algorithm or the declaration of the functions that handle the mathematical operative implicit to them that are later defined in the `helper.cpp` source file. Those functions are:

1. `argMin()` – used to find the coordinates of the minimal value on the squared correlation matrix (C) and return them as components of the local motion vector. This procedure should take into account a couple of important matters:

- As the matrix is represented by an array, it is indexed from 0 to $(2 \cdot p - 1)$. But it is originally intended to weight a displacement in a Cartesian plane, so to get the true local motion vector, we should normalize this indexes by subtracting p from the result.

- Also by definition of the procedure, in the case of a matrix in which all elements are the same (the result of 2 consecutive search windows containing no displaced pixels from one to another, meaning no movement) It should return the coordinates for the central element $(0, 0)$ instead of the first element $(-p, -p)$.

2. `median()` – this method was extracted from the code provided with [8] (originally called `median5fast()`) as a fast method to calculate the median of a 5 element vector. It is used twice, for coordinates x and y, as the last step of the algorithm to determine the global motion vector.

And last, the file `stabilization.cpp` contains the "core" of the algorithm, this is, where the GMV is determined by taking all the missing steps mentioned in chapter 3. The code listed here corresponds to the first version developed (`v1.0`).

```
    // Iterate over the 4 different search windows
    for(int i=0 ; i<4 ; i++){
        // Correlation matrix to store the results of the search
        int Cj[2*p][2*p];

         // Displacement loops: exhaustive search
        for(m_pos=0 ; m_pos < 2*p ; m_pos++){
            for(n_pos=0 ; n_pos < 2*p ; n_pos++){
                cv::Mat bxor;

                cv::bitwise_xor(
                    // Current bit-plane search window:
                    S.at(i)(cv::Rect(p, p, N, N)),
                    // Previous bit-plane search window:
                    S_prev.at(i)(cv::Rect(m_pos, n_pos, N, N)),
                    // Result of the XOR between the 2:
                    bxor
                );

                // Correct OpenCV XOR operation (see section 4.2.1)
                cv::bitwise_not(bxor,bxor);

                // Count the black pixels as (Total no. - Non-Black):
                Cj[m_pos][n_pos] = SW_SIZE - cv::countNonZero(bxor);
            }
        }
        // Determine position of the min Cj arguments
        argMin(2*p, (int *)Cj, &Vl_x[i], &Vl_y[i]);
    }
```

But this version had some serious limitations. One of the biggest concerns when working with parallel frameworks, specially in those oriented to GPGPUs, is that the operations we delegate them must be fine-grained enough as the SMs found in GPUs are really simple processors only capable of some basic arithmetic. Also, it is evident that any routine with an entry point in the `cv` namespace has only been likely compiled for the host CPU and

will not be usable in device code, making impossible to call any OpenCV function from a kernel.

However, OpenCV community saw the potential of the parallelism in 2010-2011 and started giving support both for CUDA and OpenCL (`http://opencv.org/platforms.html`). This constitutes further evidence that CUDA popularity is huge and increases over time. And also OpenCL's, as we will explain in section 4.3.3 as the computer vision framework relies part of its standard library in it to accelerate some of the operations when the hardware is available.

Something that we can point of, as it is directly related to this matter, is that this support offers the equivalent, highly optimized, parallel functions to some of the present in OpenCV's standard library. As an illustrative example, it can be transparent for the developer to parallelize the XOR function in our code to be executed by CUDA simply by calling `cv::cuda::bitwise_xor()` (`http://goo.gl/RKX68q`).

From these facts, two interesting lines of work were open:

The first one and more straightforward, would be to apply these parallel constructions to our program, making use of `cv::cuda` and `cv::gpu` libraries, like the one already mentioned. From an educated point of view, this solution has some pros and cons. It would not exploit the full parallel potential the algorithm has, as it reduces considerably the parallel fraction of the program.

Other disadvantage to study would be the potential overhead that those many calls and memory movements, if not handled properly by the compiler, could have in the overall performance. For example, if memory is not properly coalesced in the device many allocations should be carried in the GPU resulting in considerable delays. But however, it is a very interesting experiment since it relies the algorithm's main operations from already optimized functions to even more optimized parallel kernels.

The other way around the problem would be the natural approach. Since this sort of limitations are determinant to the completion of the project, they lead us to create a second version (`v2.0`) of the algorithm more generic and framework-independent. The main changes between these two versions can be seen in appendix B. Removing all the data structures and calls made to OpenCV's helper functions from the "algorithm's core": the `cv::Mat` variables and their manipulation, the XOR operation and the count of black pixels in the resulting image. This obviously involves implementing the equivalent helper functions to translate OpenCV data structures to plain-C which will be the ones passed to the threads/kernels to operate and fill the correlation matrix as explained before.

- `splitMat()` – will be the equivalent of calling OpenCV's `cv::Rect()` over a `Mat` object to crop it in 2 dimensions and get a submatrix of it. This is a vital step, since it has to be done for each iteration of the inner loop to determine the search window from a given boundaries and dimensions.

- `mat2arr()` – is the one that inverts the pixel values and packs all the raw data obtained from a `cv::Mat` to a generic `unsigned char` (byte) array. Thanks to OpenCV being mature enough, this translation could be easy if the data in the matrix is contiguous in memory as it is in our case, just by accessing the `data` member on any `Mat` object. Also, to take advantage of what we are trying to represent, we could compress every 8-pixels into a single `unsigned char`, this will improve the memory consumption and the overall performance, as the bitwise operations are carried taking advantage of vector processors, we can XOR 8 pixels at a time. (Or even more, if using even larger data types that also support this behavior).

- `xor2img()` – the responsible of applying the bitwise XOR operation iteratively to all the elements of both images to get the result.
- `count1s()` – this simple method calculates the number of bits set to one in an array of bytes, corresponding to the number of black pixels found in the image.

These functions had to be tested as they're crucial for the correct operation of the algorithm. So we developed and conducted some sanity tests to check this. Its explanation and results could be seen in the appendix C.

4.2.1 Problems Found

While developing and running the different versions of the algorithm, we found a few problems that deserve a special mention because they are closely related to the algorithm's results and performance:

- The time required by OpenCV's `VideoCapture` interface to dump all the frames in memory can be very high for computers equipped with integrated or low-end GPUs, since the work is carried in them. This problem should be addressed by loading the frames on demand, this is, saving just one frame and requesting the next to pass them as the stabilization routine arguments.
- While testing the helper functions and comparing results with `v.1.0`, we found that the way OpenCV's `bitwise_xor()` function works is not the same, by definition, than the common bitwise XOR operation, but rather its negation ($\neg(a \otimes b)$) as the figure 4.2 shows. This happens because the framework applies the operation at bit-level, but in an grayscale image black pixels are represented by $0s$ instead of $1s$ and the opposite for white ones (`0xFF`), resulting in the inverse image. This finding made us patch the original version of the algorithm just by adding a `cv::bitwise_not()` call after the initial XOR to get the right result.
- The different versions of the `mat2arr()` developed, while correctly packing the data into the byte arrays in linear time ($\mathcal{O}(n/8)$ where n is the number of pixels), were not as efficient in time as using OpenCV's native data structures and operations, since they rely in pointer arithmetic and other optimized techniques that reduce memory movements and allocations.

The last of the problems lead us to take the decision of limiting the number of frames that are taken into account to be stabilized for the sake of the time needed to run all the experiments. If we had more time, we could further deepen on the reasons behind these operations taking that long and optimize them, but as it is not crucial to the original

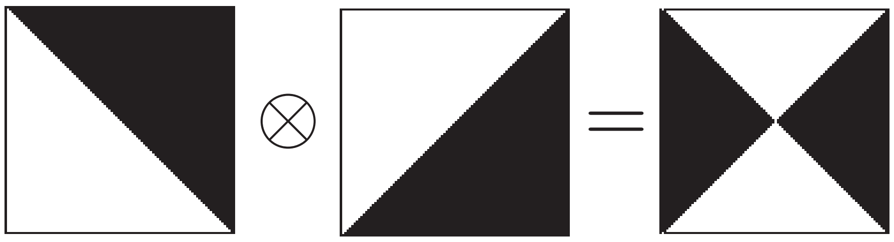


Figure 4.2: OpenCV's XOR operation

purpose of the project, we would let it be.

4.3 Parallel Implementations

In the following sections, we will present the specifics of the parallel solutions. Each follows a common structure discussing 3 main topics:

- How to compile and build the files required for each solution.
- Description and comments on the source code of the solution.
- Aspects related to the implementation that will be used in chapter 5 to lend color to the comparative.

4.3.1 OpenMP

Thanks to OpenMP being the result of the effort from a consortium and also from being around the parallel scene since 1997, the libraries required to compile programs that use this API are built-in in most mainstream, contemporary compilers. The only thing to take into account when compiling OpenMP-dependent code (that including `#include <omp.h>` header) is to use the `-fopenmp` gcc's option.

This was by far, the easiest implementation to construct since it could even have been done without removing the OpenCV function calls, as the code runs in the processor's different threads the routines had been compiled for their architecture. This sums up to the fact that there is no need to conduct many changes in the sequential code to parallelize in the way we exposed in this chapter's Overview. The code for this version can be found in the `stabilization_omp.cpp` source file.

By using OpenMP's pragma constructions, we annotate the code to declare which fractions of memory are shared and which ones are private, how to schedule the distribution of the thread blocks and also to tell the compiler that the iterations of the two `for` loops are the ones to be distributed among the threads set with `omp_set_num_threads()` function.

- `#pragma omp parallel for` – this *pragma* is a shortcut to specify a parallel construct that forms a team of threads based on the number of iteration of the loop(s) and starts their parallel execution.

- `private(m_pos, n_pos, Sp, Res)` – the local variables for the threads, meaning that each one would have a **different** copy of them. In order: the index to displace the search window on the X-axis, the equivalent for the Y-axis, the corresponding previous frame's search window and the result of applying the XOR operation.

- `shared(Sc, past, Cj, elements, p)` – these would be the shared memory portions between all the streams, all will hold the same copy: the current frame's search window, the past frame's whole subimage, the correlation matrix and the size in pixels of the subimage to crop it. (Also `p` have to be included as it is determined at runtime).

- `schedule(static)` – this clause tells the scheduler how to distribute the iterations to the threads, in our case, assigned in round-robin fashion.

4.3.2 CUDA

When it comes to compile CUDA code, the tools have evolved to nowadays' NVIDIA CUDA Compiler (`nvcc`) which eases the process that involves splitting, compiling, preprocessing, and merging intermediate files from CUDA kernels (`.cu`) to fully executable code. Still, to compile the code we have to worry about the GPU architecture details of the machine since different generations of cards offer different memory disposition and the capability to launch different number of threads, etc.

As we exposed in chapter 2, the way to work with CUDA code comes in the form of kernels: plain-C functions that are run by the SMs in what they call *thread blocks*, the key point when working with this paradigm is to correctly specify the geometry of this blocks since we are translating the loops that filled the correlation matrix in the sequential approach to a uniform distribution of their iterations between all the SMs available in the device. In our case, we choose blocks of 256 threads as a conservative approach to almost all CUDA architectures, and partitioning C_j in $n/256$ where n is its number of elements.

In our case, the file `kernel.cu` is the responsible of this task and contains almost the exact same code we had at the end of the loop. One important consideration to make is that to call the helper functions from both the host and the device code, as we do, is that we must annotate them with the labels `__host__` and `__device__` at the same time to tell the compiler to build them for both architectures.

Also, as it was described in the section 1.2.4 it is essential to correctly allocate and distribute the memory elements in both the host and the device. To do so and make our code more legible, we followed some community conventions as to label with the prefix `d_` the variables that are allocated and used in the device and with an `h_` their counterpart in the host.

These variables to copy in the device memory would be the same we pointed out as `shared` in the OpenMP version, but in this case, we have to specify the direction in which we would made the copy as we are talking about different address spaces. Obviously, the subimage and the search window need to be moved from host to the device and the opposite for the correlation matrix. All this operative is described in `stabilization_cuda.cpp`.

We have to point out that as calling this operations inside a loop make them the bottleneck of the solution, since the allocations and copies are one of the most time-consuming steps. Considering the use of three dimensional thread-blocks along with synchronization mechanisms could remove this overhead and speed up noticeably the whole program.

4.3.3 OpenCL

To build an OpenCL program, the procedures vary depending on the OpenCL implementation in use, as it is made for different environments. The absence of an proprietary compiler like the `nvcc`[1] together with the heterogeneous ideal of running in that many different platforms makes the whole process of compiling the code really hard for a basic user. Some difference between the 4 mainstream implementations (say: NVIDIA, AMD, Apple and Intel) is the location of the SDK path, which needs to be passed as an argument to the compiler, and also there are differences in this process based on whether if we are talking about C or C++ code. To handle all this complexity, we must rely on a `Makefile`. As example, we took one of the ones used for the code samples in the OpenCL Programming Book[23]. Nevertheless, a common denominator for all vendors is the use of the `-lOpenCL` flag in the compilation process.

The main difference found with CUDA comes as a result of the framework's stake to approach heterogeneous support in front of the very specific ecosystem controlled by NVIDIA itself. Everything must be done in a very *runtime-aware* way: it introduces the idea of context, which is just a runtime link to the device and platform, **the kernel has to be loaded and compiled at runtime** which is indeed a very obscure way to proceed, since too many intermediate steps have to be done to make the kernel available and launched, etc.

The whole summarized process described in detail in `stabilization_ocl.cpp` involves getting all the platform IDs (drivers), connect to compatible compute devices by creating a device group and assign it a compute context that allows the data transference between host and device and then initiate a command queue to execute the kernel in the devices selected. From that point, the program resembles very much the one we presented in the last section.

As a positive point, the only changes between the OpenCL and CUDA kernels are just the different terms used to annotate the functions and the shared memory portions passed as arguments (that in the case of OpenCL are marked with the `__global` identifier) and also the way to access the indexes to calculate the right search window. As final note, this kind of kernels are based on C99 which made us port the helper functions to a `.c` file compliant with this standard.

As we advanced earlier, OpenCV has the ability to be enabled when the hardware is

[1]There were some attempts made in this direction by the community, as it was the `clcc` (`http://clcc.sourceforge.net/`) but as the OpenCL standard advanced, the tools did not and stay outdated.

Index	**CUDA kernel**	**OpenCL kernel**
`m_pos`	`blockIdx.y * blockDim.y + threadIdx.y`	`get_global_id(1) * get_global_size(1) + get_local_id(1)`
`n_pos`	`blockIdx.x * blockDim.x + threadIdx.x`	`get_global_id(0) * get_global_size(0) + get_local_id(0)`

Table 4.1: Differences in the methods to access the indexes inside the kernels.

available with OpenCL, this way some computer vision operations could take advantage of the acceleration of the underlying platform for some of the built-in routines. Extending where we could take advantage of the parallelism in out program to the wrapper operations, the main sequential block of code of the program. This falls out of the scope of this project but could be an interesting topic for a possible complete parallel solution.

4.4 Conformance Testing

As mentioned before, this method has been around for more than 15 years now and it can be still considered relevant and sufficient for many consumer solutions. The overall performance of the GC-BPM as video stabilization method was evaluated on section IV: *Simulation Results* of [17] and it was one the reasons to choose it as frame topic for this project, it presents a highly reliable process at a very low price, as it works with very lightweight data structures, bitwise operators and also, as it has been discussed on this chapter could be parallelized in a pretty straightforward manner.

But as our solution is not totally complete, meaning it does not produce any video output but instead the indicators that determine the direction and strength in which the video is shaking, we should test the reliability of both the sequential and different parallel implementations. For this purpose we propose two easy reference tests over the same data sets to evaluate the numerical outputs.

1. **Tests with controlled image sequences**: if we decouple the frame extraction module of the wrapper and inject in memory both stabilized and shaking simple image sequences, while manually calculating the global motion vector by displacing the images in the axis of the canvas we can check the coherence of the result, as we do to present the interpretation of results in appendix D.

2. **Comparisons between parallel and sequential results**: this is straightforward and can be carried out after the general, sequential algorithm has been tested with the first procedure to ensure the results of all the programs are exactly the same no matter how many processing elements are involved for the calculations.

Chapter 5

Results

5.1 Methodology

One of the key points of this work is to offer some indicators relative to the performance of the different tools put into practice in building the parallel solutions. To achieve this, we have developed a slightly rigorous methodology to compare the contrasting results achieved by the algorithms.

To carry out with the experiments, we selected a video from Michael Messer (`https://vine.co/v/iVWbAjB1XmH`), a famous *viner*. Since the format of this vine videos is short enough for our purpose and usually shaky as they are recorded with smartphones. Also they have very convenient dimensions and frame rate for our experiments: Duration: 6s, 196 frames – Resolution: 480x480 pixels.

Different runs will be made with different number of processing units for the purpose of comparing the speedup introduced as these numbers increase. To avoid undesirable variability around the measurements and to refine the results we will launch the experiments 3 times and calculate the average of their run times. Also, we have to make some notes on the number of executions that will be made with each tool regarding the number of processing elements:

- Sequential: as we don't need to compare the different run times of the sequential version as we can't introduce more processing units by definition, we would only count with 3 runs with this configuration.
- OpenMP: these could be compared setting different threads, starting with the sequential equivalent (just one) and doubling this number up to the maximum available for the particular machine (determined at runtime by the function `omp_get_max_threads()`).
- CUDA: for this framework, we must determine the size of blocks, normal equivalence classes for the most common microarchitecture (NVIDIA Kepler) are 128, 256, 512 and 1024 threads. But some others have different inbuilt limits for execution.
- OpenCL: in this case, we execution model is mapped to the hardware, so the the

work group size can be at most `DEVICE_MAX_WORK_GROUP_SIZE` and depending on the platforms and devices available at runtime, can be executed in a CPU or in a GPU, having really different possibilities.

5.2 Experiments

There is really important to correctly characterize the environment in which all the runs are going to be made, in two different dimmensions: the CPU and the GPUs available and their microarchitecture, since it determines

The experiments by themselves can be performed on virtually any modern general purpose computer with the exception of those binaries that require of the CUDA platform to be run, which as mentioned in chapter 1 need some special hardware appliances (`https://developer.nvidia.com/cuda-gpus`) to be run. Both OpenMP and OpenCL implementations are open source and freely available for major manufacturer architectures. Because of this reason, **we have not been able to run the CUDA's implementation**, since to the date, we lack of a machine with these capabilities. However, the compilation phase was carried for all the implementations since `nvcc` is not related to the presence of a device.

The speedup comparison between the tree implementations stays on hold until some system available to run all four is available for this purpose. But nevertheless, we build all the tools and scripts to make this straightforward once the machine is ready. It will only take to compile and execute a couple of scripts to get the final results and graphics determining which one of the developed is the faster solution. We hope to have this results available for the defense of this project.

5.3 Results

To make the process of collecting the results of all the experiments simpler, we built a short script (`timecollector.py`) to dump the time indicators from the different program runs into a single `.csv` document to easily import those into an analysis tool, such as MS Excel or R lang, or just feed them to a `gnuplot` script to generate simple but meaningful graphics. This temporal data is collected by the script into a file with the following structure:

```
Run,API,Processing_Units,Time
0,0,1,XXXXXXXX
1,0,1,YYYYYYYY
2,0,1,ZZZZZZZZ
3,1,2,UUUUUUUU
...
```

It contains four differenced columns separated by commas:

- `Run`: The sequence number of the run, just to discriminate and later refer to a specific experiment.
- `API`: A numerical code associated to one of the APIs under study as it appears in `commons.h`: `1` for the sequential version of the algorithm, `2` for OpenMP, `3` for CUDA and `4` for the OpenCL version.
- `Processing_Units`: the number of streams (threads, threads per block or the size of the workgroup) involved in that particular run.
- `Time`: the really relevant variable in each row; reflects how long did it take for the program to determine all the global motion vectors.

5.3.1 Comparison of the APIs

We already offered the the really important points to compare these APIs from a developer's practical point of view thought sections 4.3.1, 4.3.2 and 4.3.3. Where the main differences in matter of implementation where exposed. In this section we will add some notes to this comparative about the user experience.

The main problem when trying to measure aspects like the ease of use and user friendly-ness is that those are both very relative aspects varying among developers. But we can still propose some opinionated metrics and show their values relative to this particular project:

1. Extra lines of code of the port as taken from the reference sequential algorithm.
2. Amount of official documentation and relative quality provided by the manufactur-er/consortium behind the API.
3. Approximate total time to make the solution compile.
4. Number of failed versions before getting the right results.

	OpenMP	**CUDA**	**OpenCL**
1.	6	21 + 16 (kernel) = 37	37 + 16 (kernel) = 53
2.[1]	Poor and unstructured	Complete and plenty	Limited and lacking examples
3.	2h	10h	11h
4.	1	?[2]	12

Table 5.1: Comparison of the different metrics considered for the 3 frameworks.

One aspect that is necessary to mention is the hard that has been to make all the APIs correctly work with a relatively modern setups, the exception would be OpenMP which is standardized in a really solid way that make it available in almost every environment and compiler. Special mention in this case goes to OpenCL, which have been particularly difficult to build and compile. Offering which we can call a *very obscure API*. We can get

[1]Some details and examples will be given later in this text.
[2]See section 5.2

a glimpse of this just by taking a look into AMD's official introductory tutorial to OpenCL [14], to see the humongous number of steps we have to take to make a simple Hello World style program with the framework.

Regarding the documentation and material offered by the responsible of maintaining the different APIs, we have some notes to make:

- OpenMP ARB does not offer any de facto documentation, but rather links [3] to community tutorials, books and examples.
- The NVIDIA developer zone [7] is a really great compilation of resources for the CUDA platform. It contains, from installation instructions, to the whole API reference and some guides to speed up the implementations, learn how to use their tools, and many more. Is the most updated (Sep. 2015) of the three analyzed here.
- Khronos Group, the association behind OpenCL standard, has a nice structured but really limited and outdated (2010) documentation [15], that also lacks of examples like the ones found in CUDA's and is constantly linking the OpenCL Specification, a 400 pages very technical document that does constitute a really good documentation repository.

5.3.2 General Comparison Between CUDA and OpenCL

Additionally, as CUDA and OpenCL frameworks rely on the kernel execution model, they are the most sensitive to be compared. In this section, we will mention some differences and facts with relevance when deciding between these two.

While OpenCL can natively talk to a large range of devices from different vendors, that doesn't imply that the code will run optimally on all of them without overhead and also there is no guarantee it will even run, given that different devices have very different instruction sets. If we stick to the OpenCL spec and avoid vendor-specific extensions, the code should be portable, if not tuned for speed. For now the programmer must face some efforts on the scale of a rewrite of the whole kernel code when switching devices for nontrivial programs. Fortunately though, the host-side stays the same across devices.

OpenCL does not support pinned host memory. This may cause a penalty of about a factor of two in host-device transfer rates. Pinned memory is memory allocated using functions like `cudaMallocHost`, which prevents the memory from being swapped out and provides better transfer speeds. Non-pinned memory is memory allocated using common `malloc` function. In this sense, pinned memory is much more expensive to allocate and deallocate but provides higher transfer throughput for large memory transfers. [16]

CUDA's synchronization features are not as flexible as those of OpenCL where any queued operation (memory transfer, kernel execution) can be told to wait for any other set of queued operations. CUDA's streams are more limited then, summed up to the fact that OpenCL supports synchronization across multiple devices. CUDA has more mature tools, including its own compiler, debugger and profiler while OpenCL does not have this kind of support. [16]

As we mentioned through the first chapters of this work, the world of High Performance Computing as it exists today consists of many GPU-accelerated systems and applications. As it was the pioneer in the field, NVIDIA's CUDA in some ways concentrated the monopoly as both programming language and infrastructure. The problem with this model is that software is almost always strictly controlled and its source code is most often propietary, and the hardware options are limited to one vendor.

Recently, some efforts have been made in order to make CUDA a platform more portable. In this sense, in 2015 major NVIDIA competitor, AMD presented the "Boltzmann Initiative" [13] that included something called Heterogeneous-compute Interface for Portability or HIP offering several benefits over other heterogeneous programming interfaces, like OpenCL:

- Developers can code in C++, and mix host and device C++ code in their source files.
- HIP API is less verbose than OpenCL.
- Because both CUDA and HIP are C++ languages, porting from CUDA to HIP is significantly easier than porting from CUDA to OpenCL.
- HIP offers an offline compilation model where the kernel binary is read in by the host code.

Chapter 6

Conclusions

This bachelor's project is the result of a one-semester temporal effort to review and compare from a practical point of view some mainstream but still specific parallel programming tools, some of its results are really relevant to the times considering the world of multicore processors is now a reality in everyday computers, even for embedded and mobile devices.

These consumer electronics and mobile applications could take advantage of this phenomenon, being more performant and offering better results, for instance, many modern smartphones could use a similar procedure as the one presented here to stabilize video files taken from the device in-built camera. Or one of the recent market booms: the very small-scale action cameras like GoPro includes an image processing unit that could benefit hugely from supporting multicore processing as their regular mode of operation involves this kind of problems.

We have implemented both sequential and parallel versions of the Gray-Coded Bit Plane Matching algorithm. The other core topic of this thesis was presenting a methodology to compare their performance, which stays on hold until they can be compared in the right environment, and a few representative metrics from the developer experience to offer a more complete comparison between these commercial parallel APIs.

6.1 Future Work

There are many different possibilities to extend this work at this point. The first path to consider would be the one leading to a true and complete stabilization solution, some examples on how to carry on where this project leaves would be:

- The direct application of the global motion vector determined by the algorithms to modify the original video sequence to correct all the involuntary displacements. This would be matter of employing different computer vision techniques to crop and rotate the original input to generate a lower-quality and size product with the important advantage of being stabilized. Some steps on how to build this result are broadly explained in appendix D.

- Despite the fact that GC-BPM, as formulated works well for translational (horizontal and vertical) motion, it includes no estimation or compensation capability for highly rotational nor zooming motion [6], considering these are both very common in practice, some different methods not suffering by this problem could be considered to be parallelized and compared to this particular one.

- Find the causes and solve the efficiency problems we reviewed in section 4.2.1 to get closer to the times obtained with the OpenCV standalone version.

- Also, some data structures could be reviewed when working with this kind of highly regular data patterns, if the price of going low level is not a problem, as logical matrices (containing only binary variables) are, some examples of such would be Quadtrees or even in more memory-critical conditions, bitmaps, in which even very lage images can be coded into a very small amount of data.

- Some other memory tweaks can be looked into, specially in the case of CUDA, which offers a complete memory hierarchy to the programmer, so large that can be used to optimize allocations and reduce the bottleneck of this solution. We are talking about the read-only memory portions for the SMs (constant and texture memories) that could be written by the host CPU with the whole frame sequence.

- Play with the idea of parallelizing the process of determining the GMV in the four search windows at the same time, this is, the outer loop of the algorithm to multiply the potential speedup of the application by a factor of 4. We already gave some instructions on how to construct such solution.

As video capture hardware is constantly evolving, the recording resolution keeps getting higher. Just a few years ago, 1080p or Full HD was the standard for high-quality video; today image sensors are capable of capturing 4K video, in each frame of which 4 complete FullHD frames could be allocated. The impact in algorithms like the one we have used for this thesis will be tremendous with a sequential approach: the search windows grows in size exponentially. As we have seen, parallelism comes very handy at solving this kind of issues. More recent approaches [1] to this algorithm make the whole process way more precise by decomposing each one of the frames into 9 different search windows instead of 4. This kind of methodological switch could have an enormous impact into performance as increasing this number introduces a multiplicative constant to the overall runtime of the programs, but if we take the benefits of calculating each local vector by a single thread and have the multicore infrastructure to support this model we could maintain a great performance as the resolutions keeps growing.

Second approach to deepen in the topics of this work would be to look in the way of exploring more parallelism options out there. As this is a one-semester project, its scope has to be limited to fit into time, that was the reason we only choose three popular frameworks from different programing models to compare, but yet several more APIs should be considered to have the full picture of the parallel landscape, some examples are: the maturity and reliability of Pthreads, the ubiquity of MPI, lots of hybrid and heterogeneous solutions like OpenHMPP, OpenACC or the newborn AMD's HIP... and the list goes on for a while, considering these are slowly proving they worth and becoming greatly adopted.

Nevertheless, it is already an arduous task to find some practical project to be developed and compared in such a diversified environment, this reflects the industry trend to use and develop more specific purpose APIs instead of pursuing tools that could work with different programming models abstracting some of the specifics.

Bibliography

[1] Sherin C Abraham, Maria Renu Thomas, Raisa Basheer, and P.R. Anurenjan. A novel approach for video stabilization. *2011 IEEE Recent Advances in Intelligent Computational Systems*, pages 134–137, 2011.

[2] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[3] OpenMP ARB. Openmp.org - resources. `http://openmp.org/wp/resources/`. [Online; accessed 21-July-2016].

[4] Gerassimos Barlas. *Multicore and GPU programming: An integrated Approach.* Elsevier, 2014.

[5] Blaise Barney. Introduction to parallel computing. `https://computing.llnl.gov/tutorials/parallel_comp/`. [Online; accessed 23-February-2016].

[6] A.C. Brooks. Real-time digital image stabilization. *Ee*, 420(March):10, 2003.

[7] Nvidia Corporation. Cuda parallel computing platform site. `http://www.nvidia.com/object/cuda_home_new.html`. [Online; accessed 27-March-2016].

[8] Martin Drahansky and Filip Orság. Real-time video stabilisation. *International Journal of Autonomic Computing*, 1(2):202–210, 2009.

[9] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.

[10] Ian Foster. Designing and building parallel programs, 1995.

[11] Javier Fresno Bausela. *Supporting general data structures and execution models in runtime environments.* PhD thesis, Universidad de Valladolid, 2015.

[12] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[13] AMD Inc. Amd launches 'boltzmann initiative' to dramatically reduce barriers to gpu

computing on amd firepro graphics. http://www.amd.com/en-us/press-releases/Pages/boltzmann-initiative-2015nov16.aspx. [Online; accessed 3-May-2016].

[14] AMD. inc. Intro opencl tutorial. http://developer.amd.com/tools-and-sdks/opencl-zone/opencl-resources/introductory-tutorial-to-opencl/. [Online; accessed 27-July-2016].

[15] Khronos Group Inc. Opencl 1.1 reference pages. https://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/. [Online; accessed 21-July-2016].

[16] Andreas Klöckner. Cuda vs opencl: Which should i use? https://wiki.tiker.net/CudaVsOpenCL. [Online; accessed 15-April-2016].

[17] Sung-Jea Ko, Sung-Hee Lee, Seung-Won Jeon, and Eui-Sung Kang. Fast digital image stabilizer based on gray-coded bit-plane matching. *IEEE Transactions on Consumer Electronics*, 45(3):598–603, Aug 1999.

[18] Karl Rupp. Cpu, gpu and mic hardware characteristics over time. https://www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/. [Online; accessed 27-March-2016].

[19] Matthew Scarpino. A gentle introduction to opencl. http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854. [Online; accessed 21-March-2016].

[20] Tom Scott. How youtube video stabilization works. https://youtu.be/BgAdeuxkUyY. [Online; accessed 13-July-2016].

[21] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. Top500 - list statistics. http://www.top500.org/statistics/list/. [Online; accessed 20-March-2016].

[22] University of Valladolid Trasgo Recognized Research Group. Trasgo group website. http://trasgo.infor.uva.es. [Online; accessed 23-July-2016].

[23] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, and Satoru Tagawa. The opencl programming book. *Fixstars Corporation*, 63, 2010.

Appendices

List of Appendices

Appendix A

CD Contents

- Source code for the solutions presented in chapter 4 in the `src` directory.
  - `v1` – Sequential version using OpenCV presented in 4.2.
  - `v2` – **The final sequential solution and their parallel equivalents.**
  - `test` – Simple source code and image examples used to test the helper functions in appendix C.
- LaTeX source code of this document in `latex` directory.
- The video file used as example to be stabilized, named `video.mp4`.
- The tools to get the speedup results: `timecolector.py` and `gnuplot` scripts to elaborate the graphics. In the `tools` folder.
- This document compiled in PDF format: `BP_FIT_Samuel_Alfageme_2016_v2.pdf`.

Appendix B

Sequential Implementation

B.1 First Approach – `v.1.0`: Use of OpenCV

```
#include <opencv2/opencv.hpp>

#include "commons.h"
#include "stabilization.h"

/* Gray Coded Approach based on:
 * [17] S. Ko, S. Lee, S. Jeon, and E. Kang. Fast digital image
 *      stabilizer based on gray-coded bit-plane matching.
 * ------------------------------------------------------------------
 */
void GC_BPM(cv::Mat Scurr, cv::Mat Sprev, int Vg_prev_x, int Vg_prev_y
        , int *Vg_x, int *Vg_y)
{
    int rows(Scurr.rows), cols(Scurr.cols);

    // Proportions of the search window and margins
    int N = 120;
    int p = (rows/2 - N)/2;

    // Size of the search window: squared (A. Brooks [6] approach)
    int SW_SIZE = N * N;

    // Local motion vectors (0-3)
    int Vl_x[5];
    int Vl_y[5];

    // We take into account the previous global motion vector (4)
    Vl_x[4] = Vg_prev_x;
    Vl_y[4] = Vg_prev_y;

    // Break both previous and current frames into 4 subimages:
    std::vector<cv::Mat> S(4);
    std::vector<cv::Mat> S_prev(4);

    S_prev.at(0) = Sprev(cv::Rect(0,0,cols/2,rows/2));
    S_prev.at(1) = Sprev(cv::Rect(cols/2,0,cols/2,rows/2));
    S_prev.at(2) = Sprev(cv::Rect(0,rows/2,cols/2,rows/2));
    S_prev.at(3) = Sprev(cv::Rect(cols/2,rows/2,cols/2,rows/2));

    S.at(0)      = Scurr(cv::Rect(0,0,cols/2,rows/2));
    S.at(1)      = Scurr(cv::Rect(cols/2,0,cols/2,rows/2));
    S.at(2)      = Scurr(cv::Rect(0,rows/2,cols/2,rows/2));
    S.at(3)      = Scurr(cv::Rect(cols/2,rows/2,cols/2,rows/2));

    int m_pos, n_pos;

    // Iterate over the 4 different search windows
    for(int i=0 ; i<4 ; i++){
       // Correlation matrix to store the results of the search
       int Cj[2*p][2*p];

        // Displacement loops: exhaustive search
       for(m_pos=0 ; m_pos < 2*p ; m_pos++){
           for(n_pos=0 ; n_pos < 2*p ; n_pos++){
               cv::Mat bxor;

               cv::bitwise_xor(
                   // Current bit-plane search window:
                   S.at(i)(cv::Rect(p, p, N, N)),
                   // Previous bit-plane search window:
                   S_prev.at(i)(cv::Rect(m_pos, n_pos, N, N)),
                   // Result of the XOR between the 2:
                   bxor
               );

               // Correct OpenCV XOR operation (see section 4.2.1)
               cv::bitwise_not(bxor,bxor);

               // Count the black pixels as (Total no. - Non-Black):
               Cj[m_pos][n_pos] = SW_SIZE - cv::countNonZero(bxor);
           }
       }
       // Determine position of the min Cj arguments
       argMin(2*p, (int *)Cj, &Vl_x[i], &Vl_y[i]);
    }
    *Vg_x = median(Vl_x);
    *Vg_y = median(Vl_y);
}
```

B.2 Second Approach – v.2.0: Independent Algorithm

As all the preprocessing work from v.1.0 is the same for this second version, we only present here the core differences.

```
    for(int i=0 ; i<4 ; i++){
       int Cj[2*p][2*p];

       int elements = S.at(i).cols;

       unsigned char* current = S.at(i).data;
       unsigned char* past    = S_prev.at(i).data;

```

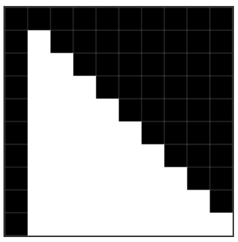
```
        unsigned char* c2 = new unsigned char[SW_SIZE]();

        splitMat(current, elements, p, p, N, c2);
        mat2arr(c2, SW_SIZE, Sc);

        for(m_pos=0 ; m_pos < 2*p ; m_pos++){
            for(n_pos=0 ; n_pos < 2*p ; n_pos++){

                unsigned char* p2 = new unsigned char[SW_SIZE]();
                splitMat(past, elements, m_pos, n_pos, N, p2);

                mat2arr(p2, SW_SIZE, Sp);
                xor2img(Sc, Sp, Res, dimension);
                Cj[m_pos][n_pos] = count1s(Res,dimension);
            }
        }
        argMin(2*p, (int *)Cj, &Vl_x[i], &Vl_y[i]);
    }
```

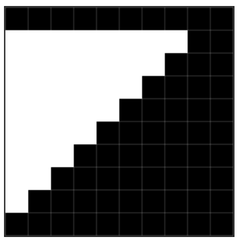
Appendix C

Conformance Tests carried on the Helper Functions

As for version 2 we no longer relied on functions of some trustworthy framework standard library, we were in need of performing some simple test to check the correctness of the helper methods developed.

For the sake of simplicity, we created the following 2 10x10px images: `img1.png` and `img2.png` to see if the process of packing the images into `unsigned char` (or bytes) data structures was being carried out properly.


(a) Enlargement of `img1.png`


(b) Enlargement of `img2.png`

```
1111111111          11111111 | 255
1011111111          11101111 | 239
1001111111          11111001 | 249
1000111111          11111110 | 254
1000011111          00111111 |  63
1000001111          10000111 | 135
1000000111          11100000 | 224
1000000011          11111000 | 248
1000000001          00011110 |  30
1000000000          00000011 |   3
                    10000000 | 128
                    01100000 |  96
                    0000     |   0
```

```
1111111111          11111111 | 255
0000000011          11000000 | 192
0000000111          00110000 |  48
0000001111          00011100 |  28
0000011111          00001111 |  15
0000111111          00000111 |   7
0001111111          11000011 | 195
0011111111          11110001 | 241
0111111111          11111100 | 252
1111111111          11111111 | 255
                    01111111 | 127
                    11111111 | 255
                    1111     | 240
```

In the previous listing, we can see, for both of the images, in the left an ASCII representation of each 10x10 matrix followed by the result in memory after executing the function `mat2arr()` in them as observed in a debugger program. As the reader can see, all the data is correctly packed and coalescent as it was explained in chapter 5.

The "core" function of the algorithm, the one that performs the exclusive-or operation between two images, was also tested to see if the operative was sound. To do so, we invoked the operation between the two presented matrices, which resulted in the following memory allocation and, when unpacked according to the original matrix dimensions, can be translated in the ASCII equivalent of the matrix presented by its side. We created the `PNG` version using OpenCV's functions to check their match.

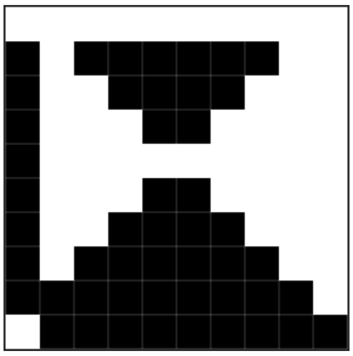


Figure C.2: Enlargement of `xor.png`

```
  0 | 00000000          0000000000
 47 | 00101111          1011111100
201 | 11001001          1001111000
226 | 11100010          1000110000
 48 | 00110000          1000000000
128 | 10000000          1000110000
 35 | 00100011          1001111000
  9 | 00001001          1011111100
226 | 11100010          1111111110
252 | 11111100          0111111111
255 | 11111111
159 | 10011111
240 | 11110000
```

To test the `count1s()` function, we calculated the number of black pixels for each one of the previous matrices, passing the arrays in which they were packed, resulting in 55 for `img1.png`, 64 for `img2.png` and 49 for the resulting `xor.png`, the same numbers that we can count in the presented images.

Finally, the `splitMat()` function was used with the arguments `(img1, 10, 3, 3, 4, result)` to simulate the selection of a centered 4x4 search window over `img1.png` as can be seen in figure C.3. This printed out the following results that correspond with the 16 ordered pixels of the submatrix: 0 for the black and 255 for the white:

```
255 0 0 0 255 255 0 0 255 255 255 0 255 255 255 255
```

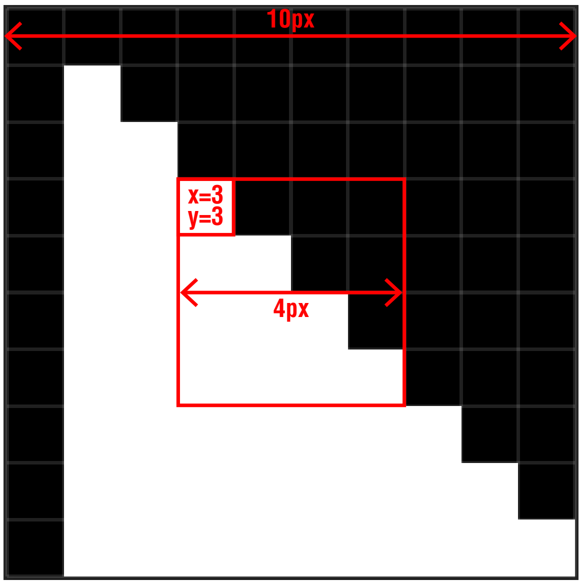


Figure C.3: Area of the matrix selected to be split

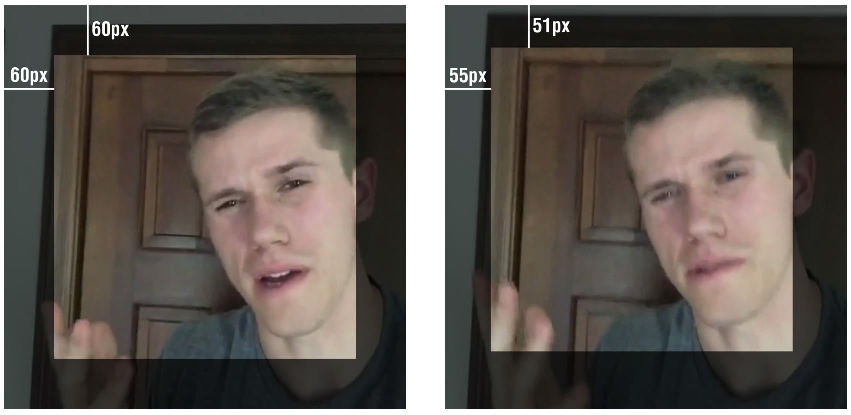
Appendix D

Example Results

The output of executing any of the programs is a generated `results.txt` file containing the sequence of Global Motion Vectors ordered by frame.

```
Time: 50.8518s
10,12
13,10
...
```

To both describe how to interpret this results and give some instructions on how to complete the computer vision application to generate the stabilized video we injected two close frames in the algorithm (just to be more significant, they don't have to be consecutive) to obtain their GMV. If we trace a "stabilization window" with the right size by leaving a margin of p in every side, the meaning of the vector is the displacement of this window from the reference frame to the other. For the particular case of the two frames selected on the figure D.1 this GMV was $(-5, -9)$ and the resulting stabilized video will be the consecutive cropped areas in the images.



(a) Reference frame (n) (b) Compared frame ($n+4$)

Figure D.1: Displacement of the stabilized window between the two frames