1 2 9 0

UNIVERSIDADE Đ
COIMBRA

João Carlos da Costa Barreiros

FAST SCALE-INVARIANT FEATURE TRANSFORM ON GPU

Mestrado Integrado em Engenharia Eletrotécnica e de
Computadores, ramo de especialização em Computadores orientada pelo Professor
Doutor Gabriel Falcão Paiva Fernandes e apresentada à Faculdade de Ciências e
Tecnologia da Universidade de Coimbra no Departamento de Engenharia
Eletrotécnica e de Computadores

Outubro de 2020

1 2 UNIVERSIDADE Đ COIMBRA 9 0

Fast Scale-Invariant Feature Transform on GPU

João Carlos da Costa Barreiros

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador: Doutor Gabriel Falcão Paiva Fernandes

Júri

Presidente:	Doutor João Pedro de Almeida Barreto
Orientador:	Doutor Gabriel Falcão Paiva Fernandes
Vogal:	Doutor Jorge Nuno de Almeida e Sousa Almada Lobo

Outubro de 2020

We can only see a short distance ahead, but we can see plenty there that needs to be done

- Alan Turing

Agradecimentos

Gostaria em primeiro lugar de agradecer ao professor Gabriel Falcão Paiva Fernandes pela sua orientação, dedicação, disponibilidade e paciência que teve para comigo durante o percurso desta dissertação. Destaco que o seu apoio contínuo foi fulcral para a conclusão da mesma.

Em segundo lugar gostaria de agradecer ao Instituto de Telecomunicações por disponibilizar todos os meios de trabalho necessários e também um ambiente amigavel e profissional.

Aos meus pais, por toda a ajuda prestada durante o meu percurso académico, sinto uma eterna gratidão. Agradeço todos os esforços que fizeram por mim para eu me tornar na pessoa que sou hoje.

Por fim, quero referir todos os meus colegas e amigos que me acompanharam também durante este percurso: Filipe, Óscar, Louro, Fraga, Martins, Costa, Veiga, Cavaleiro, Matos, Abegão, Janela, Bernardo e todos aqueles que conviveram comigo no dia-a-dia.

A todos,
Muito Obrigado

Abstract

Feature extraction of high-resolution images is a challenging procedure in both high and low-power signal processing applications. This thesis describes how to optimize and efficiently parallelize the scale-invariant feature transform (SIFT) feature detection algorithm and maximize the use of bandwidth on the GPU subsystem. Together with the minimization of data communications between host and device, the successful parallelization of all the main kernels used in SIFT allowed a global speedup in high-resolution images above 78x while being more than an order of magnitude energy efficient (FPS/W) than its serial counterpart. From the 3 GPUs tested, the low-power GPU has shown superior energy efficiency in almost every case. Achieving up to 6x less power consumption than the CPU for the same amount of work.

Keywords

Feature extraction, Scale-invariant feature transform (SIFT), GPGPU, CUDA, Parallel Programming.

Resumo

A extração de caracteristicas de imagens de alta resolução é um procedimento bastante desafiante tanto em aplicações de processamento de sinais de alta como de baixa potência. Esta dissertação descreve como otimizar e paralelizar de forma eficiente o algoritmo scale-invariant feature transform (SIFT) e maximizar o uso da largura de banda na GPU. Juntamente com a minimização das comunicações de dados entre o host e o device, a paralelização bem-sucedida de todos os kernels principais usados no SIFT permitiu um aumento de velocidade global de 78x em imagens de alta resolução. Ao mesmo tempo, a solução apresentada consegiu uma maior ordem de grandeza de eficiência energética (FPS/W) do que a sua contra-parte sequencial. Das 3 GPUs testadas, a GPU de baixo consumo mostrou maior eficiência energética em quase todos os casos. Alcançando até 6x menos consumo de energia do que a CPU para a mesma quantidade de trabalho.

Palavras Chave

Feature extraction, Scale-invariant feature transform (SIFT), GPGPU, CUDA, Parallel Programming.

Contents

List of Figures

List of Tables

List of Acronyms

GPU	Graphics Processing Unit
CPU	Central Processing Unit
SIFT	Scale-invariant feature transform
GPGPU	General-Purpose computing on Graphics Processing Units
CUDA	Compute Unified Device Architecture
ALU	Arithmetic Logic Unit
SM	Streaming Multiprocessor
SIMT	Single Instruction Multiple Thread
PCIe	Peripheral Component Interconnect Express
HPC	High Performance Computing
OpenCL	Open Computing Language
DRAM	Dynamic Random-Access Memory
SRAM	Static Random-Access Memory
OS	Operating System

1 Introduction

Contents

In the field of computer vision, feature extraction has always been an intriguing topic of research. Its purpose is to eliminate redundancies from datasets, thus allowing derived values (features) to be unique and informative. In image processing, feature extraction is used to detect and isolate desired portions of an image or a video stream.

In order to do so, developers often resort to algorithms that are invariant to changes in scale, rotation, illumination, and, to some degree, affine distortions. All these characteristics are essential to Scale-invariant feature transform (SIFT), an algorithm that has stood the test of time and that it is still popular in various computer vision applications, such as, image registration [12], object recognition [13], motion tracking [14], 3D modelling [15], among others.

In the past decade, the growth of storage capacity [16] and quality of digital cameras has indirectly contributed to a significant increase in the size and resolution of digital images, which in turn has made feature extraction algorithms computationally more demanding. Under these circumstances, linear solutions implemented using the central processing unit (CPU) are limited by computational power and are no longer capable of maintaining real-time execution for large scale images.

1.1 Motivation

However, in the field of parallel computing, a group of architectures, traditionally linked to image rendering (mostly in games), has gained prominence within this area. Also known as graphics processing units (GPUs), their functionality has extended beyond the visualization domain, to also accommodate general-purpose processing on GPUs or GPGPU.

Companies and organizations such as Nvidia and The Khronos Group respectively, have designed APIs and frameworks (e.g., CUDA and OpenCL), thus providing developers with the ability to exploit the full potential of GPUs and to build faster parallel solutions for more resource-intensive problems.

Even though CUDA is exclusive to Nvidia GPUs, it is still far better optimized than its OpenCL counterpart, with speedups up to 30% when executing the same task on the same hardware [17]. As a result, this makes CUDA a preferable choice for this work.

Additionally, as AI-related applications (where feature extraction is included) started making the shift from the cloud down to the power-constrained edge, the demand for low-power solutions rose likewise. In order to meet this criterion, the current work purposes to exploit the parallel computational power of modern low-power GPUs and simultaneously perform algorithmic optimizations capable of reducing computational complexity while incurring negligible accuracy losses.

Apart from that, while there are other feature extraction algorithms that are faster than SIFT, it cannot be overlooked that this algorithm is much more known and commonly used due to its high robustness and accuracy which other algorithms struggle to match [18]. It is therefore a suitable candidate to be parallelized on a GPU and to be the focal point of this thesis.

1.2 Objectives

This thesis proposes to explore the parallelism of different families of GPUs in order to speedup and improve the energy efficiency of the SIFT algorithm. The main objectives of this work are:

- Review the state-of-art of the SIFT algorithm;
- Develop a parallel solution of SIFT using CUDA;
- Apply CUDA optimizations iteratively to the proposed solution;
- Measure and compare execution times as well as energy consumption of the proposed implementation;
- Evaluate the obtained results and the feasibility of the GPU for future applications that involve feature extraction.

1.3 Dissertation Outline

This thesis is organized into six chapters. The first chapter introduces the topic of the thesis, the motivation behind it and the main goals this work intends to achieve. Chapter 2 focuses on the structure of the SIFT algorithm. Chapter 3 presents and explains the differences between the CPU and GPU architectures, furthermore, it also introduces the CUDA programming model. Chapter 4 describes the methods that were tested and implemented in order to accelerate and optimize SIFT. Chapter 5 provides results alongside with their analysis. And finally, Chapter 6 concludes this thesis and presents some suggestions for future work.

2
Background on SIFT

Contents

This chapter introduces the theoretical concepts regarding feature extraction and the techniques that SIFT employs in order to obtain keypoints and features from digital images and video streams. Lastly, it is presented the performance and energy efficiency of several parallel approaches that were found in the literature.

2.1 The SIFT Algorithm

SIFT can be decomposed into four major parts [1] [2]: Scale-Space creation (please see 2.1.1), extrema detection (see 2.1.2), attribution of orientations (2.1.3), and descriptor generation (2.1.4). Next, we address each step in detail.

2.1.1 Creation of the Scale-Space

In image processing, scale-space is a technique that allows the representation of a given image at different scales. This is useful in SIFT because it grants scale invariance to the algorithm. Besides that, it can also remove unwanted details from the input image through recursive use of Gaussian filters, which, subsequently, also avoids the formation of aliasing artifacts [19]. Gaussian filters can be defined as

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{2.1}$$

where x and y are the axis coordinates and σ the standard deviation.

The scale-space is divided into groups of images with equal size that are increasingly blurry, each group is called an octave, and each image of an octave is referred to as a layer. The first octave usually begins with an ($2\times$) upscaled version of the input image. Some implementations make this an optional step and prefer to use the original image instead, however the present work tries to stay true to the algorithm and therefore includes the upscaling by default. After each octave, the Gaussian image with an accumulated blur of 2σ of the initial layer is down-sampled by a factor of 2, creating a new octave. The cycle continues until the dimensions of the final octave reach a given threshold defined by the user.

Once the scale-space is generated, the algorithm proceeds to detect blobs in every layer. These blobs are regions that present similar properties, such as brightness or color. These are aspects are crucial later for keypoint extraction. A simple way to locate these regions would be by applying a Laplacian filter over each of the generated images. However, this approach is computationally demanding, and not practical for most applications. D.Lowe, the author of SIFT, suggests to calculate the difference of Gaussians (DoG) instead, whose result is approximate to the Laplacian of the Gaussian and is overall much faster [1] [20] [21].

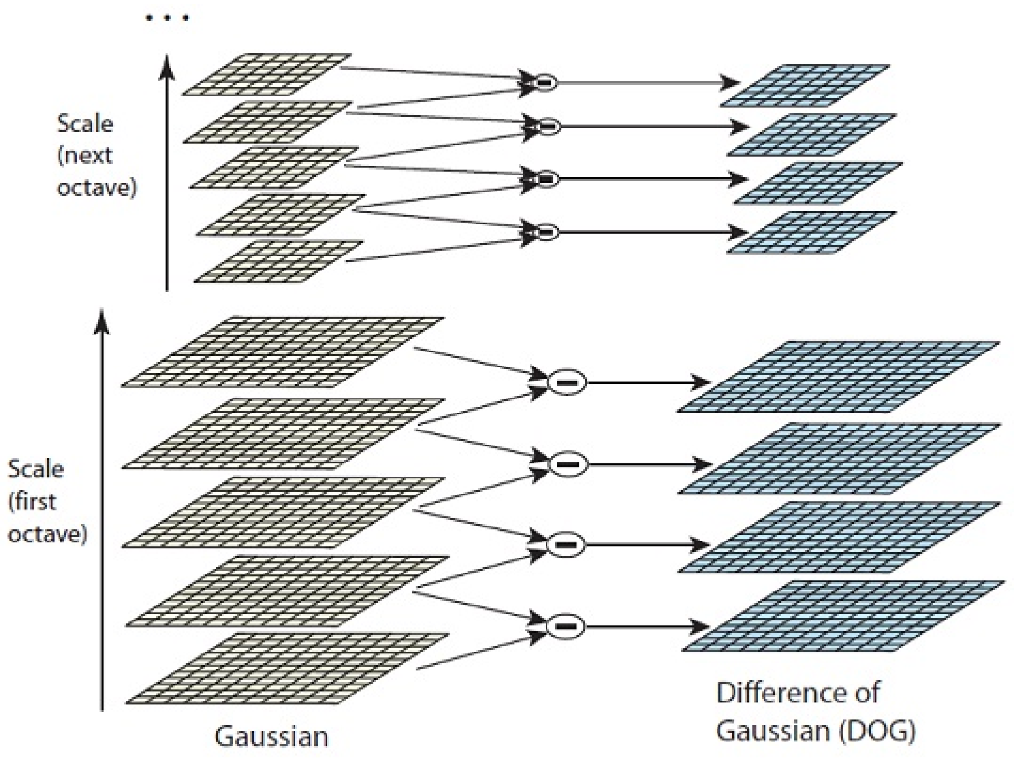


Figure 2.1: Difference of Gaussians, courtesy of [1].

Figure 2.1 shows that for each octave, a new group of layers is created from the subtraction of adjacent Gaussian layers.

2.1.2 Extrema Detection

The following step is to locate local extrema from the set of DoG images that were just generated. The algorithm transverses each point of each resulting non-bound layer and compares it to the neighboring pixels of that same layer and the adjacent ones (see Figure 2.2). This is equivalent to comparing each pixel with 26 neighbors in a $3 \times 3 \times 3$ volume. If the selected pixel is a maximum or minimum of these 26 points, then it is considered to be a point of interest or a keypoint.

A later approach to SIFT [1] goes a step further and determines the sub-pixel location of each keypoint to improve matching and stability. This method uses the quadratic Taylor expansion to interpolate the DoG function $D(x,y,\sigma)$ with the candidate keypoint as the origin. This Taylor expansion is given by:

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^T \frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x}, \tag{2.2}$$

where $\mathbf{x} = (x,y,\sigma)^T$ is a column vector that represents the offset from the point of interest. The location of the extremum, $\hat{\mathbf{x}}$, is determined by taking the derivative of

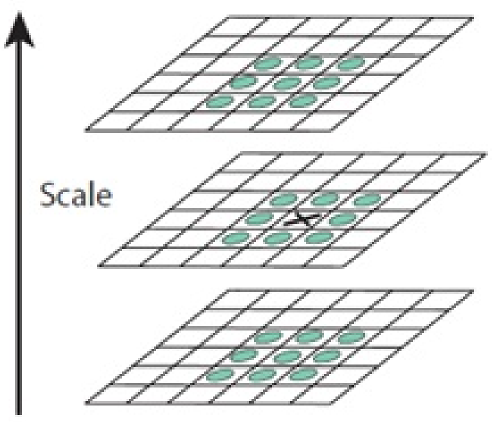


Figure 2.2: Extrema detection, extracted from [1].

this function with respect to **x** and setting it to zero, resulting in:

$$\hat{\mathbf{x}} = -\frac{\partial D^{-1}}{\partial \mathbf{x}^2}\frac{\partial D}{\partial \mathbf{x}}, \tag{2.3}$$

where:

$$\frac{\partial D}{\partial \mathbf{x}} = (\frac{\partial D}{\partial x}, \frac{\partial D}{\partial y}, \frac{\partial D}{\partial \sigma})^T \tag{2.4}$$

and:

$$\begin{aligned} \frac{\partial D}{\partial x} &= \frac{D(x+1,y,\sigma) - D(x-1,y,\sigma)}{2} \\ \frac{\partial D}{\partial y} &= \frac{D(x,y+1,\sigma) - D(x,y-1,\sigma)}{2}. \\ \frac{\partial D}{\partial \sigma} &= \frac{D(x,y,\sigma+1) - D(x,y,\sigma-1)}{2} \end{aligned} \tag{2.5}$$

One drawback this method usually has, is the detection of a vast number of extrema points, which may negatively impact performance, especially on larger scales.

Not all of these points provide useful information, in fact, some may have a very low contrast while others may be situated along edges. In either case, the best solution is to discard these points. For the former, we can analyze the intensity of each pixel by using the Taylor expansion in (2.2) and remove pixels whose value is less than a given threshold. For the latter, it is necessary to calculate the two gradients with orthogonal directions at the chosen point. From here we can attain three possible outcomes:

- A flat region, if both gradients are small;
- An edge, if one of the gradients is large and the other small;

- A corner, if both gradients are large.

Mathematically this can be achieved by analyzing the ratio between the eigenvalues of the second-order Hessian matrix, H:

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix}, \tag{2.6}$$

where D_{ij} are the derivatives obtained from the differences of the keypoint neighbors. Let α and β be the smallest and largest eigenvalues respectfully. Then the trace and determinant of H can be computed as:

$$\begin{aligned} Tr(H) &= D_{xx} + D_{yy} = \alpha + \beta \\ Det(H) &= D_{xx}D_{yy} - (D_{xy})^2 = \alpha\beta \end{aligned} \tag{2.7}$$

The SIFT algorithm discards keypoint candidates whose eigenvalue ratio $r = \frac{\alpha}{\beta}$ is higher than a predefined threshold. Since only this ratio is required, the computation of the eigenvalues can be skipped. The ratio of the Hessian matrix determinant and its trace are related to r by:

$$\frac{Tr(H)^2}{Det(H)} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r} \tag{2.8}$$

Therefore, to determine if a point is within the threshold, SIFT checks for:

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r} \tag{2.9}$$

2.1.3 Orientation assignment

The next stage consists of assigning orientations for each keypoint. Orientation is an essential aspect of the algorithm, as it grants invariance to rotation. This is achieved by analyzing gradient directions and magnitudes around each point of interest. In practice, SIFT uses a 36 bin circular histogram to store the essential data(orientation and magnitudes of the gradients). Afterward, it checks for the bin with the highest contribution and defines it as the predominant orientation of the current keypoint. Additionally, if any other bin is at least 80% of the maximum, then another keypoint is generated in the same location, but with that bin's orientation.

The magnitudes and orientations of the gradients can be obtained with equations (2.10) and (2.11), respectively:

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2}, \tag{2.10}$$

$$\theta(x,y) = tan^{-1}(\frac{(L(x,y+1) - L(x,y-1))}{(L(x+1,y) - L(x-1,y)})), \tag{2.11}$$

where m is the magnitude of the gradient, θ is the orientation, L is the smoothed image and x, y the coordinates of the point.

2.1.4 Keypoint Descriptor

Finally, SIFT generates the keypoint descriptors/features, which are unique fingerprints that help identify important and unique regions of the original image. In order to achieve this, it first calculates the gradient map around each keypoint using a 16×16 window, which is later divided into 16 cells 4×4 (Figure 2.3 shows a simplified version with a 8×8 window). Then in each cell the magnitude and orientation of the gradients are determined and their values are placed in an 8-bar histogram. In this histogram, the magnitude of the closest gradients to the keypoint weights more than the magnitude of more distant gradients. Once the histogram is completed, it produces a 128 element vector that, once normalized, corresponds to a feature.

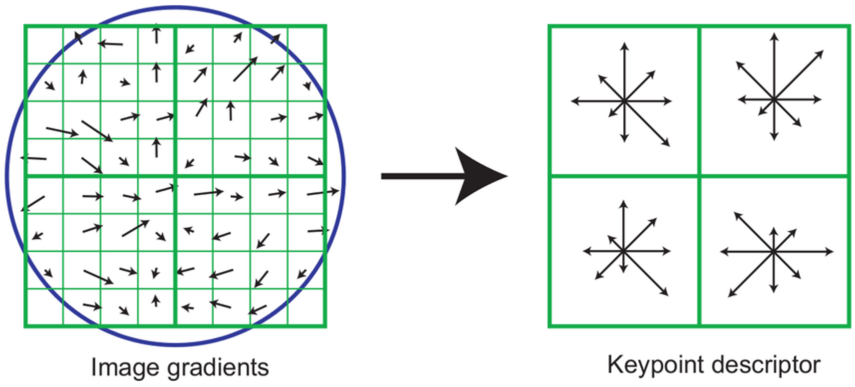


Figure 2.3: Descriptor generation, obtained from [1].

Some examples of applications using this algorithm can be seen in figures 2.4, 2.5 and 2.6, which illustrate image matching, image stitching and object detection, respectively.

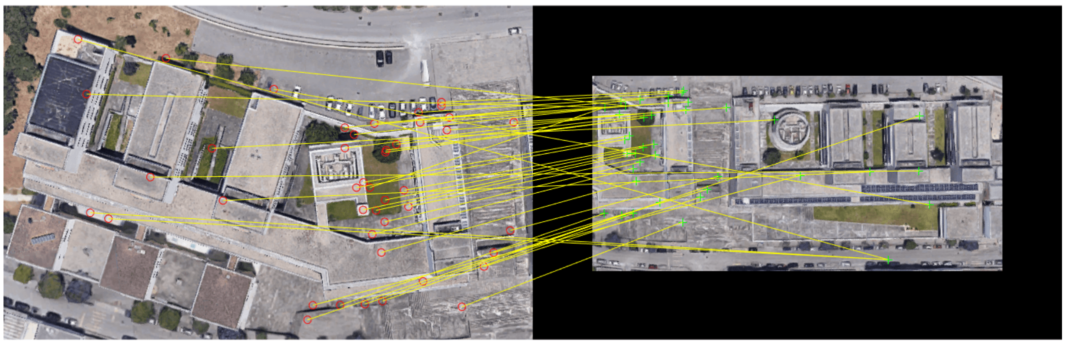


Figure 2.4: Example of image matching using SIFT.

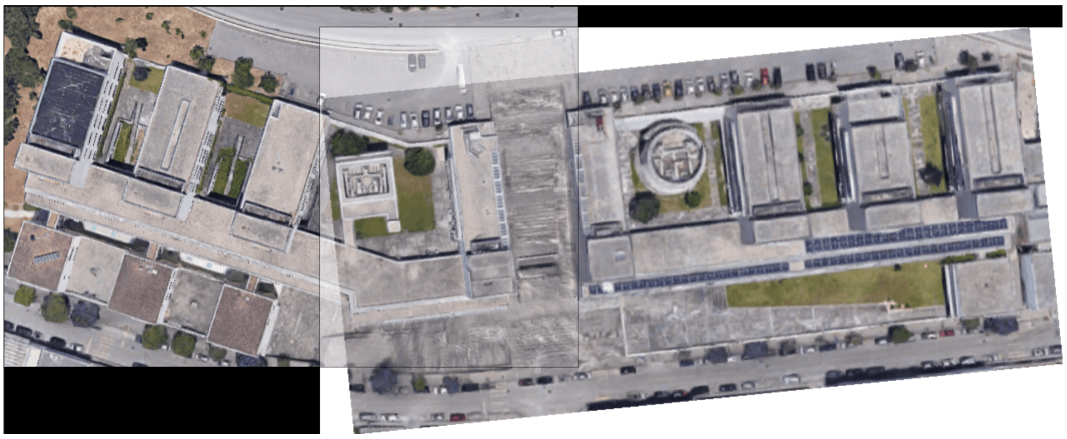


Figure 2.5: Example of image stitching using SIFT.

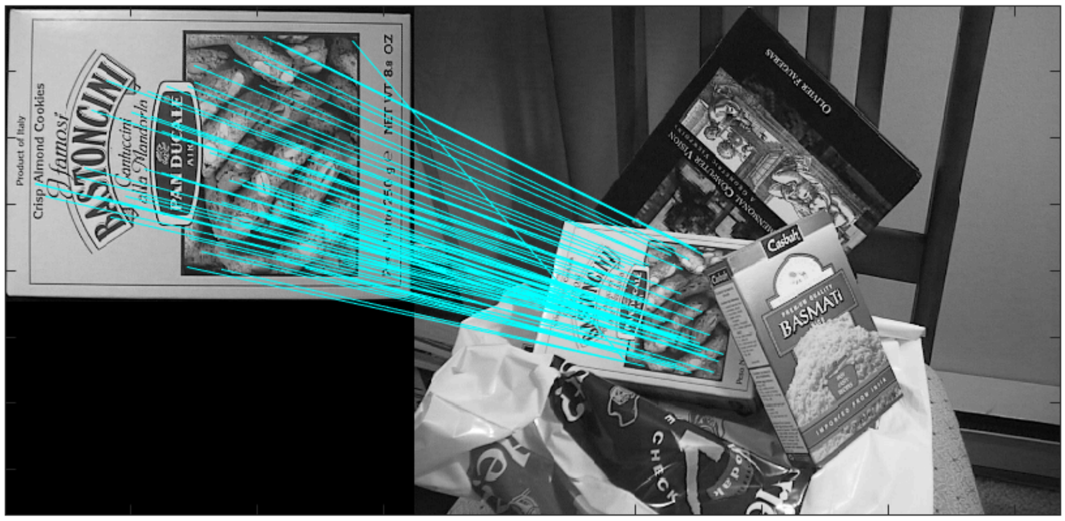


Figure 2.6: Example of object detection using SIFT, adapted from [2].

2.2 Related Work

In the following section, it is presented the results of several parallel implementations that were found in the literature. These are subsequently split into two different categories: High performance and low power. The former focuses on the parallelization and acceleration of the SIFT algorithm on mid to high-end GPUs, while the latter aims at a more energetic point of view, where efficiency is key.

2.2.1 High performance implementations

Table 2.1 shows the performance (in FPS) and efficiency (in FPS/W$\times$100) of several parallel SIFT implementations running on various platforms. Some of these achieve more than 30 FPS for large scale images, however it is worth to mention that the inner parameters of several steps can be changed to favor speed over accuracy. The implementation proposed in the current work uses the ideal parameters that were suggested by D.Lowe, the original author of SIFT [2]. This often translates into a greater number of detected features.

Table 2.1 Performance of various implementations of the SIFT algorithm on the GPU

Article	GPU	Image Dimensions	FPS	TDP	FPS/W $\times$ 100
[22]	GTX 480	1280 $\times$ 720	13	250 W	5.2
[23]	Tesla C2050	1280 $\times$ 960	20	238 W	8.4
[24]	GTX 960	1280 $\times$ 1024	29	120 W	24.1
[25]	GTX Titan Black	1280 $\times$ 960	48	250 W	19.2
[26]	GTX 9600	1200 $\times$ 800	11	95 W	11.6
[27]	Quadro FX 3400	1280 $\times$ 960	20	101 W	19.8

To give an interesting example, CUDASift [28], which claims to be the fastest implementation of this algorithm, omits some small steps in order to achieve greater performance. As a result, fewer keypoints are generated and accuracy drops when compared to the original algorithm.

2.2.2 Low-power implementations

The use of feature extraction algorithms in a low-power system is not something that is relatively new. In [29], the authors proposed an FPGA implementation of SIFT which achieved a 56 frames per second for a VGA resolution of (640×480). [30] proposed an ASIC solution of SIFT/SURF that managed 5.5 FPS in HD images while running at 23mW. [31] proposes another implementation of SIFT in a FPGA which runs 640×480 images at 32FPS. More recently [32] introduced an OpenCL-based SIFT accelerator for image features extraction on FPGA speeding up the algorithm up to 13.7$\times$ compared to software version and increasing the energy efficiency up to 1.38$\times$ more than the GPU accelerator on an high-end NVIDIA GPU. On the mobile side, [33] proposed a low power solution of SIFT using OpenCL on a smartphone, achieving an average of 5.89FPS on the [10] dataset while being 41% more power efficient than its CPU counterpart.

The proposed solution, however, is done in a relatively new environment, in this case, a dedicated low-power GPU. While some of the other solutions are faster or more efficient than ours, we can not ignore the complexity, development time needed and cost associated to the implementation of feature extraction algorithms on FPGAS and ASICS. The current work is also portable between various NVIDIA GPUs, and therefore its easier and faster to implement.

3

The GPU architecture and CUDA programming model

Contents

The main focus of this chapter is to analyze and compare the CPU and GPU architectures of modern computers in order to understand the strengths and weaknesses of each. Additionally, this section also explores how CUDA interacts with each.

3.1 CPU architecture

The central process unit (CPU) is essentially the brain of every computational system. It is responsible for basic arithmetic, logic, controlling, and input/output (I/O) operations. The first commercially available CPUs were designed with only one core in mind. As a result, they could only execute one instruction at a given time. This paradigm did not change for a while. In fact, it took 30 more years for the first dual-core processor to be released in the market [34]. Moreover, as this new architecture continued to evolve, the more obsolete its predecessor became. As manufacturers quickly realized that multi-core processors had the potential to provide better gains in the overall performance of programs that support multithreading, thus giving them a competitive edge.

Today, CPUs still possess a small number of cores, since its primary role has not changed much, continuing to act as a microprocessor optimized with significant amounts of local memory (i.e. cache) to perform sequential operations. However, due to the slowing down of Moore's Law, it is expected that the number of cores will scale upwards [35].

3.1.1 Memory Hierarchy

The CPU has a big emphasis on low latency, which is mainly possible due to the three memory components that exist alongside the processor (cache L1, L2 and L3)(see figure 3.1.). Since data is fetched in blocks, the existence of a cache can help the CPU predict the flow of execution by taking advantage of the principle of locality. In terms of speed, cache L1 is the fastest, followed by L2 and L3 (or LLC) [36]. The only drawback of this, is the relatively small size of each component since SRAM (the type of memory used by CPU caches) is rather expensive.

The CPU can still access other types of memory such as the DRAM or the Hard Drive, however this comes at a cost of higher latency and lower speeds overall. Hence the reason the CPU tries to avoid fetching data from these memories, unless it is forced to (in case of a cache miss).

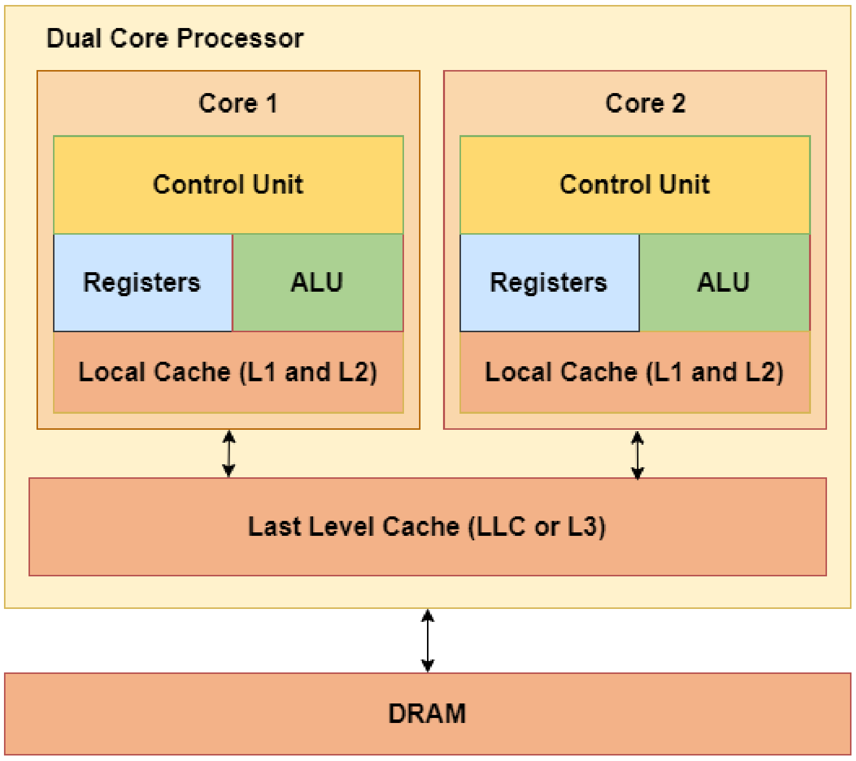


Figure 3.1: An example of a modern dual core CPU architecture.

3.2 GPU architecture

The graphics processing unit (GPU), on the other hand, was projected to have hundreds/thousands of small cores that can handle thousands of threads simultaneously (see figure 3.2.).

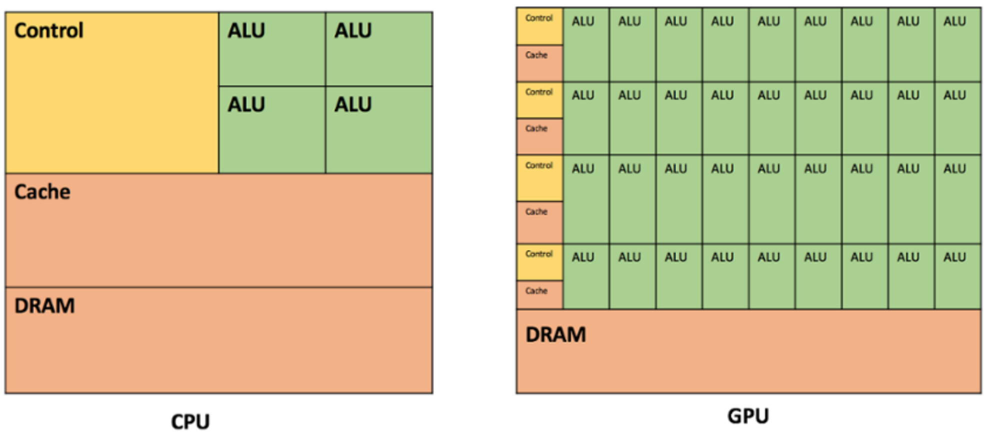


Figure 3.2: CPU vs GPU Architecture, adapted from [3].

Initially, these cores were built exclusively for rendering, however as times went on, new techniques emerged, and GPUs began to process more complex tasks such as realistic scenarios in two dimensions or even simple scenarios in three dimensions. Eventually, this technology started gaining popularity in the gaming and multimedia industry, which played a vital role in bringing GPUs to personal computers [37]. Taking into account the computational power of GPUs, it would be a matter of time before new applications began to surface besides graphic rendering (e.g. Blockchain, artificial intelligence, deep learning, among others).

Today, GPUs have become an essential component in modern computers. One of the main reasons why they are important and valued is because they can easily outperform the CPU at heterogeneous tasks. Their large core count and high bandwidth are reaching unprecedented numbers, thus opening doors for the future development of more GPU-accelerated applications [38].

3.3 CUDA Programming Model

CUDA (also known as Compute Unified Device Architecture) is the development platform for GPGPUs, which includes the combination of specific hardware and software for the development of solutions in heterogeneous systems. At first, this platform only supported the C language, but later, it expanded to C++, Java, and Python. This increase in accessibility enabled programmers to take better advantage of GPU resources that were previously inaccessible to them.

The CUDA API allows users to write kernels, which are functions that are executed on the device (GPU). These kernels, unlike normal CPU functions, are usually carried out by a large number of threads. To illustrate this idea, the following code performs the simple task of adding two vectors together (A and B) and storing the result in another vector (C).

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with 32 threads
    VecAdd<<<1, 32>>>(A, B, C);
    ...
}
```

In this sample, the VecAdd kernel is defined using the __global__ declaration specifier and the number of CUDA threads that are launched is configured inside the <<<...>>> syntax in the main function. In other words, the kernel in this example is executed with one block of 32 threads.

On a lower level, the CUDA processing flow would go as follow (illustrated in Figure 3.3):

1. Copy data from main memory to GPU memory;
2. CPU launches the GPU kernel;
3. GPU's CUDA cores execute the kernel in parallel;
4. Copy the resulting data from GPU memory back to main memory.

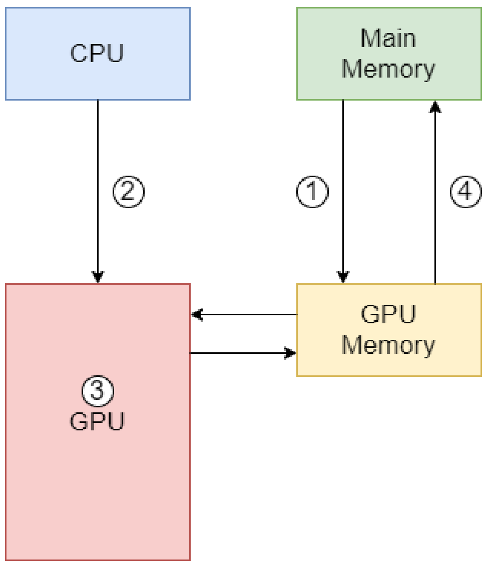


Figure 3.3: Processing Flow in CUDA.

Regarding complex tasks, they can still be executed in parallel on the GPU. In fact, the programming model of the CUDA platform is based on the notion that all problems can be factored into simpler sub-problems (see Figure 3.4.).

To better understand this, the programming model includes three fundamental abstractions [4]: a hierarchy of thread groups, shared memories, and barrier synchronization. These are exposed to the programmer as a minimal set of language extensions. Additionally, they can provide fine-grained data parallelism and thread parallelism, thus serving as a guide to the programmer on how to partition the problem into simpler sub-problems that can be solved independently and in parallel.

3.3.1 Memory Hierarchy

During runtime, threads can access data from various sources, as illustrated in Figure 3.5. Each thread has a private local memory. Each block of threads has a shared memory visible to all threads of that same block and all threads have access to the global memory.

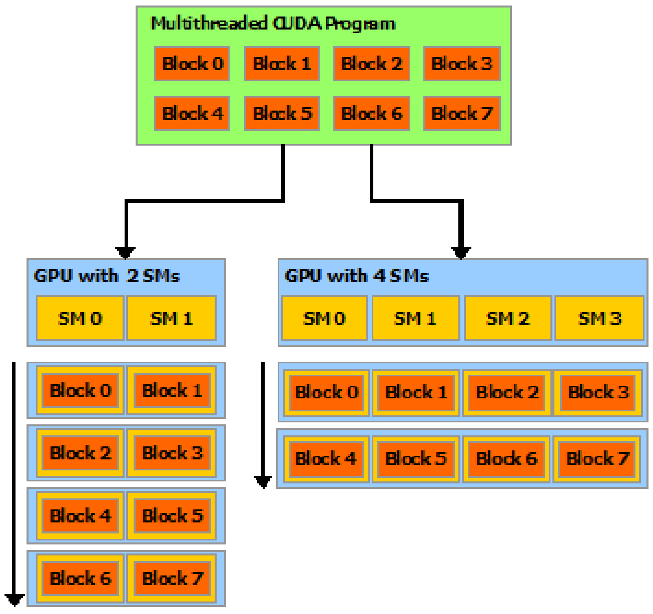


Figure 3.4: Scalability in CUDA, obtained from [4].

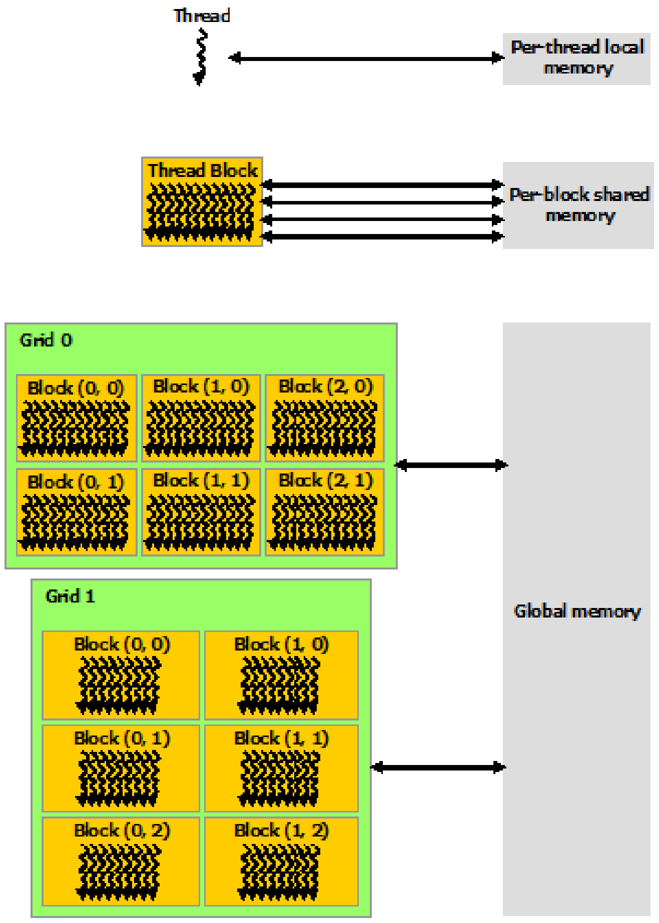


Figure 3.5: GPU Memory Hierarchy, acquired from [4].

Furthermore, blocks can be organized in a one-dimensional, two-dimensional or three-dimensional grid of thread blocks, as illustrated in Figure 3.6. The number of thread blocks in a grid is usually determined by the size of the data to be processed or the amount of processors in the system.

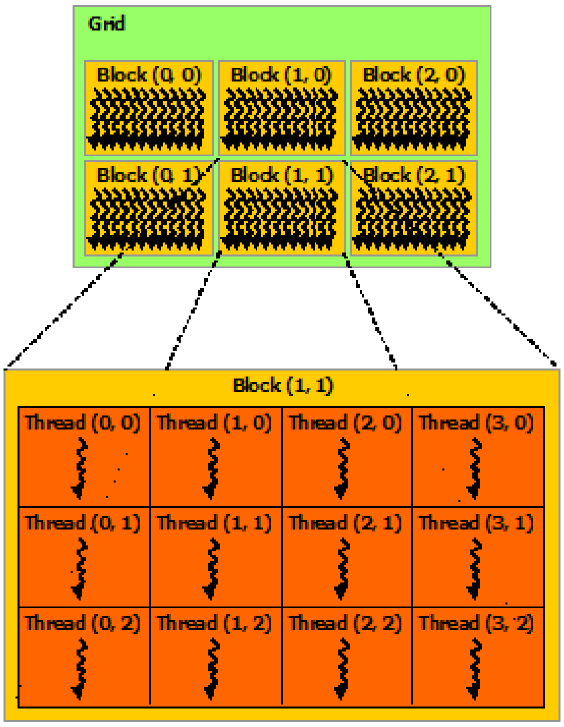


Figure 3.6: Grid blocks layout, extracted from [4].

There are also two additional read-only memory spaces that can be accessed by all threads: the constant and texture memory. These memory spaces are optimized for different memory usages [4].

3.3.2 CUDA optimization techniques

In order to take full advantage of the GPU resources, it is essential to have a good understanding of the many techniques that NVIDIA recommends [39]. Major optimizations can be separated into three main categories:

- Memory optimizations;
- Latency optimizations;
- Instruction optimizations.

Memory optimizations have the most significant impact in terms of performance. Their main goal is to maximize the use of bandwidth by using faster memories and mitigating wasted transactions. Some of these techniques are:

- **Minimizing data transfers between host and device**: Since the peak theoretical bandwidth between the main (host) memory and the device memory is slower than the theoretical peak bandwidth between GPU and device memory, programmers should strive to group and reduce the number of memory transfers between these components whenever it is possible, even if it means running kernels on the device that do not show performance gains.

- **The use of pinned memory**: Page-locked or pinned memory transfers are high bandwidth transfers between host and device. The usage of this type of memory can boost the speed of memory transfers, however, because this resource is scarce, it takes a longer time to allocate. This technique is, therefore, situational and depends on the application. Figure 3.7. illustrates both ways data can be sent to the device.

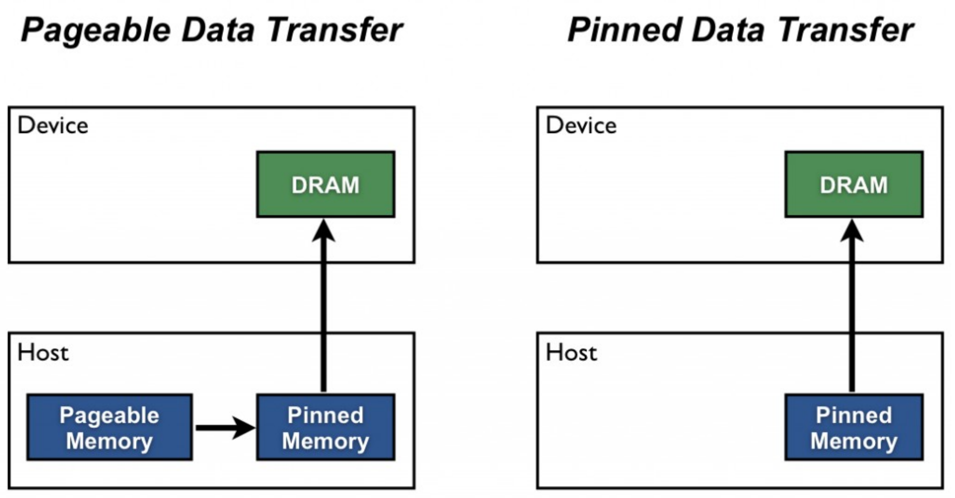


Figure 3.7: Difference between pageable and pinned data transfers, obtained from [5]

- **Asynchronous transfers**: Data transfers between the host and the device can either be blocking or non-blocking. In blocking transfers, control is returned to the host thread only after the data transfer is complete. On the other hand, in non-blocking transfers, control is returned immediately to the host thread. This means that non-blocking transfers are asynchronous and can also be overlapped. Consequently, this allows the programmer to perform a concurrent copy and execute, which, when implemented correctly, is faster than the sequential method (see Figure 3.8).

- **Coalesced access to global memory**: Memory coalescing is a technique that allows an optimal usage of global memory bandwidth. That is, when par-

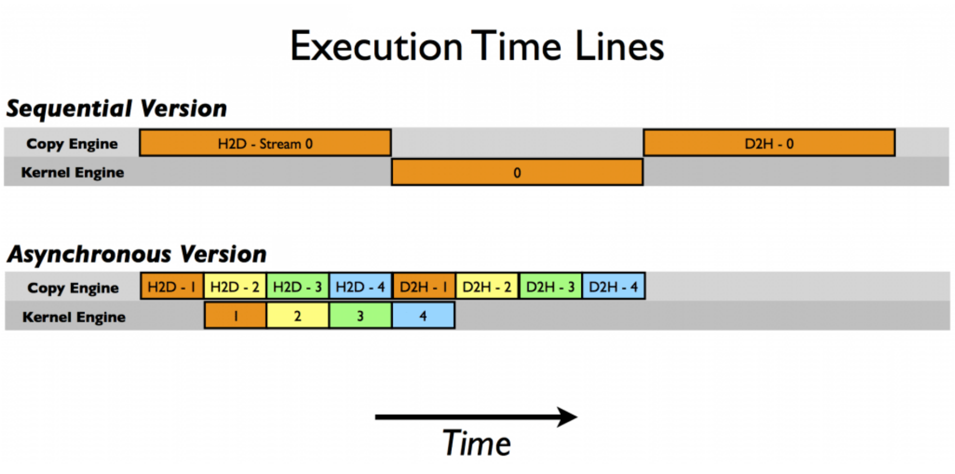
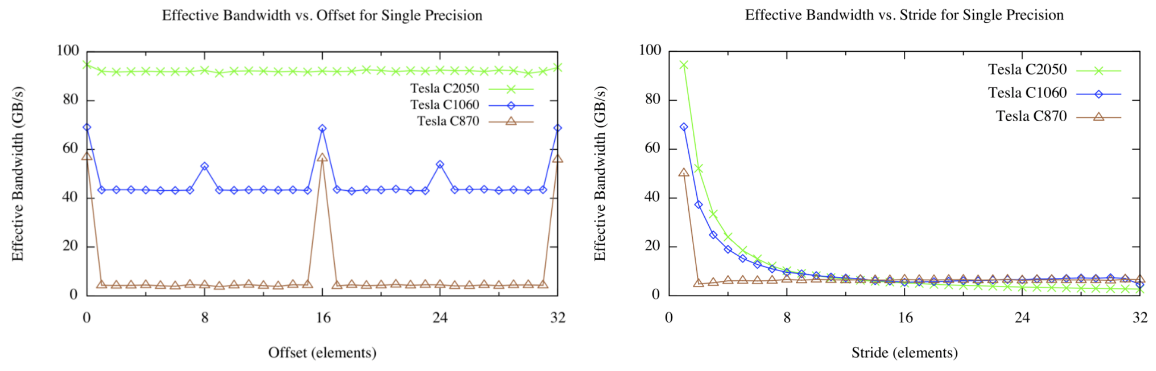


Figure 3.8: Asynchronous vs synchronous transfers, adapted from [6].

allel threads running the same instruction access consecutive locations in the global memory, the most favorable access pattern is achieved [40]. In the case of misaligned or strided memory accesses the same can not be said. In fact, both suffer performance penalties as seen in Figure 3.9a and 3.9b.



(a) Misaligned memory access

(b) Strided memory access

Figure 3.9: The performance impact of different memory access patterns, extracted from [7].

On a side note, the increasingly larger cache sizes have diminished the performance impact of misaligned memory accesses on newer devices.

- **The use of shared memory**: Because it lies on-chip, shared memory has higher bandwidth and lower latency than global memory. Its primary use includes inter-block communication, reducing redundant global memory transfers, and avoiding non-coalesced accesses. While shared memory can provide a significant boost in performance, when multiple threads try to read or write to the same memory bank, a bank conflict occurs and those accesses are serialized, degrading performance (see Figure 3.10). Strategies that are used to avoid those bank conflicts usually include padding or changing the address patterns.

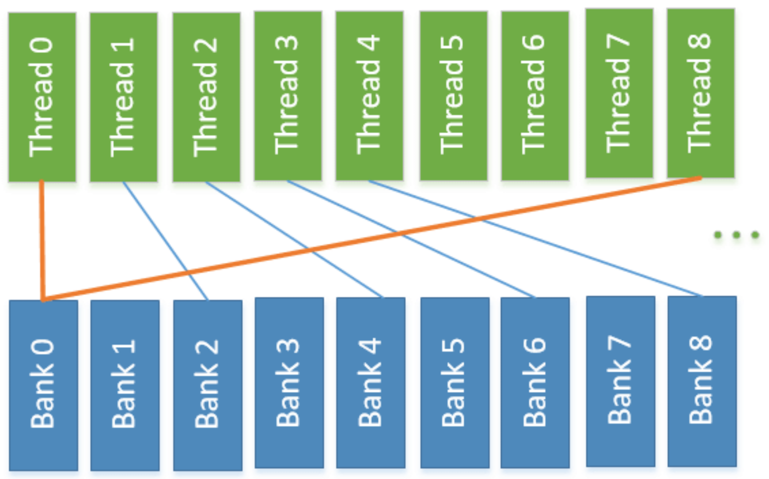


Figure 3.10: Example of a 2-way bank conflict, obtained from [8].

- **The use of texture memory**: The read-only texture memory space is cached. Therefore a texture fetch costs one read from texture cache unless a cache miss takes place. In that case it costs the same as reading from device memory. Texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve the best performance.

One way to achieve better performance is to keep the multiprocessors fully occupied. A program that has too many idle threads is considered to be poorly balanced and suboptimal in terms of performance. Hence, the importance of maximizing occupancy, that measures the ratio between active warps per multiprocessor and total possible active warps. While a higher occupancy does not necessarily imply a higher performance, it is still an overall good practice. Furthermore, it helps to hide latency arising from register dependencies.

Another point that should also be taken into consideration is the fact that streaming multiprocessors (SMs) launch threads in warps, or groups of 32 threads. This is important because the number of threads per block should be a multiple of that number for optimal efficiency.

Finally, instruction optimizations are a part of a lower-level category that includes integrated functions and fast math libraries. The main goal of these optimizations is reduce the number of instructions that are needed to get the same amount of work done.

4

Parallelizing SIFT on the GPU

Contents

This chapter discusses several methods that were developed during the course of this work in order to obtain performance gains by parallelizing SIFT on the GPU.

4.1 The SIFT Datablock

One of the biggest hurdles in parallel computing is the proper management of memory transfers between host and device. As mentioned in the previous chapter, consecutive transfers between the two can hinder the overall performance of the solution. SIFT is no exception to this rule. Hence it is better to parallelize a significant part of the algorithm and then run it on the GPU rather than sending data back and forth. Figure 4.1 shows how the proposed implementation is laid out.

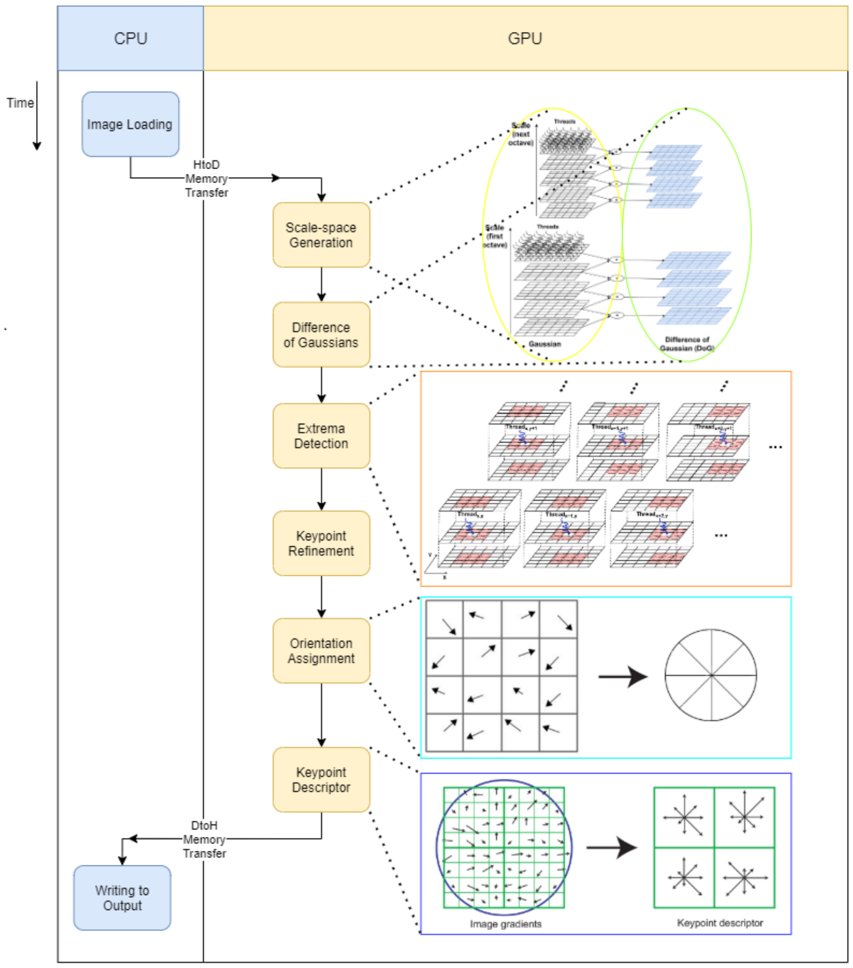


Figure 4.1: The proposed execution flow of the SIFT algorithm on a GPU.

4.2 2D Gaussian Convolution

2D convolution is a complex task that demands a considerable amount of computational power. For this reason, the current work presents two different approaches that were tested and evaluated in order to develop an optimal solution.

4.2.1 Separable convolution

In image processing, the Gaussian blur is the result of the convolution between a given image and a Gaussian function. This type of blur, not only it is used to reduce noise and detail from the image, but it can also enhance image structures at different scales without introducing aliasing artifacts, which is essential for scale-space representation.

The main issue is that 2D Gaussian convolutions are very taxing both in terms of time and resources. Even with the implementation of the optimizations mentioned in the previous chapter, this part would still be a bottleneck in SIFT.

Fortunately, the Gaussian blur contains a property known as filter separability, that if used correctly, can mitigate this bottleneck and grant significant performance gains. In other words, the effect of applying a 2D Gaussian filter to an image can also be achieved by applying two consecutive 1D Gaussian filters to the same image, one in the horizontal direction and another in the vertical direction (see Figure 4.2).

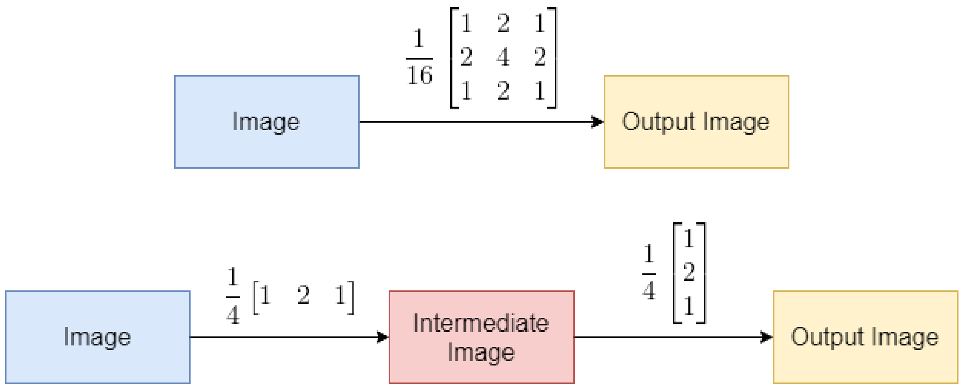


Image
$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$
Output Image
Image
$\frac{1}{4}\begin{bmatrix}1&2&1\end{bmatrix}$
Intermediate Image
$\frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix}$
Output Image

Figure 4.2: Example of a spatially separable convolution using a Gaussian filter.

In the example of Figure 4.2, the 2D matrix:

$$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix} \tag{4.1}$$

can also be written as the product of these two one dimensional filters:

$$\frac{1}{4}\begin{bmatrix}1\\2\\1\end{bmatrix}\cdot\frac{1}{4}\begin{bmatrix}1&2&1\end{bmatrix}=\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix} \tag{4.2}$$

This property has the advantage of offering more flexibility to the implementation, not to mention it also reduces the overall arithmetic complexity and bandwidth usage. Assuming the 2D filter has size $L \times L$ and the input image $M \times N$ then this technique saves a total of $(L^2 - 2L)MN$ operations, as long as $L > 2$ (see figure 4.3).

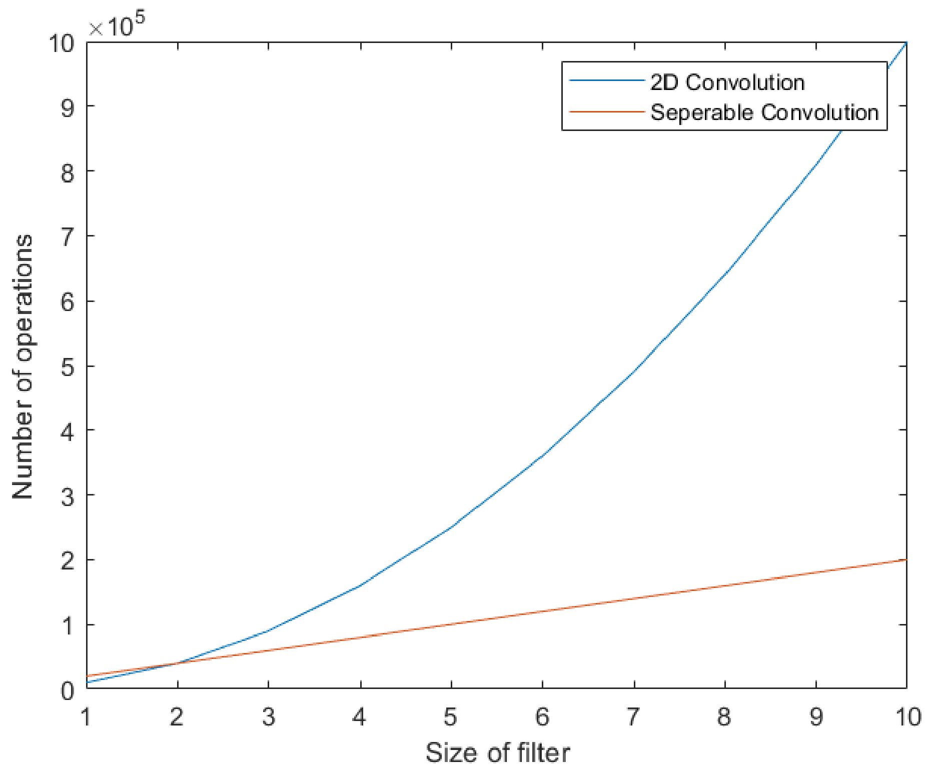


Figure 4.3: 2D convolution versus separable convolution in a 100x100 image

On the GPU, the present work takes advantage of the various resources that are available in order to build a fast and efficient solution. For instance, operations with high arithmetic intensity (i.e convolution) are performed on shared memory instead of global memory, due to its high bandwidth and low latency. Gaussian filters are stored in constant memory since they do not change. Other optimizations involve padding thread blocks to achieve the alignment required for coalesced loads as well as loading multiple pixels per threads to reduce the number of idle threads. Finally, it is also possible to process rows and columns in parallel.

4.2.2 Box Filters approximation

Another way to approach convolution is with box filters. A box filter or mean filter is a spatial domain linear filter that has a box-shaped impulse response. It is a low-pass filter that applies an averaging blur to the input image (4.3 shows an example of a typical box filter).

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix} \tag{4.3}$$

Unlike Gaussian filters, box filters always have equal weights, which can simplify the complexity of convolutions. However, these filters alone are not suitable for scale-space representation.

Fortunately, they also contain special properties that can prove to be useful for the current work. According to the central limit theorem, box filters can converge to a Gaussian blur when applied multiple times [41]. Figure 4.4. demonstrates that as the number of passes increases, the more accurate the shape becomes when compared to a bell curve.

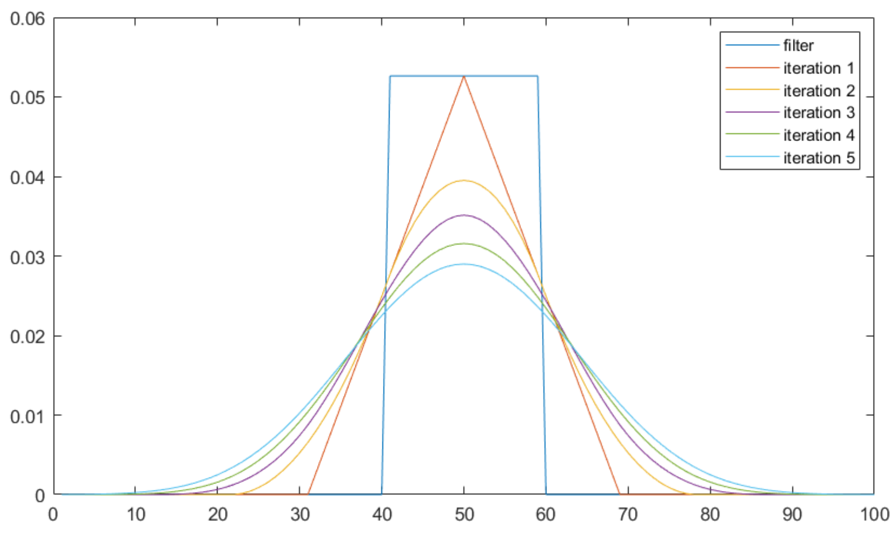


Figure 4.4: Gaussian approximation using a box filter.

On the GPU, the proposed implementation uses the sliding window method. That is, as the kernel moves from left to right, it adds in the contribution of the new sample on the right, and subtracts the value of the exiting sample on the left. This only requires two additions and a multiplication per output value, which makes convolution independent of the filter size (see Figure 4.5).

In addition to the above, box filters are separable, which means two one dimension filters can achieve the same as a 2D filter while using fewer operations. In CUDA, this can also be exploited to process rows and columns in parallel.

4.3 Difference of Gaussians

The DoG is a relatively simple task when compared to the rest of the algorithm. Mathematically, it performs pixel-wise subtraction over every pair of layers from the scale-space. Since there are no data dependencies associated with this operation, it can be done entirely in parallel, as shown in figure 4.6.

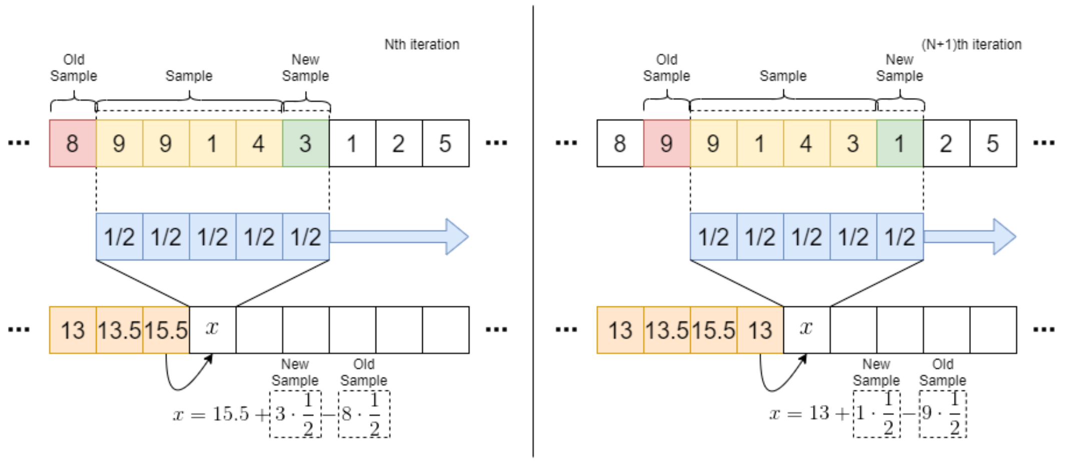


Figure 4.5: Example of a 1D convolution with box filters using the sliding window method.

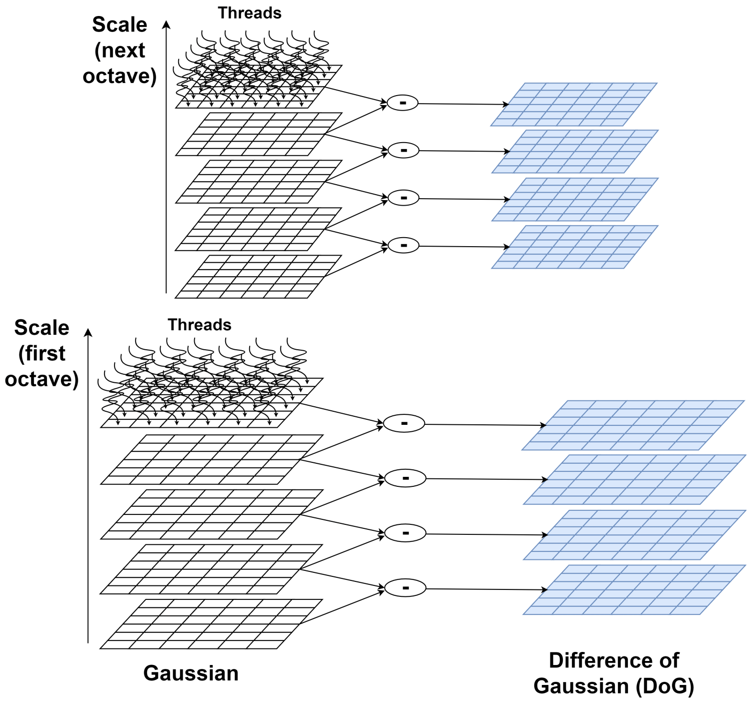


Figure 4.6: Difference of Gaussians on the GPU.

Nevertheless, some optimizations can still be made in order to avoid unnecessary memory accesses. A naive approach would load a pair of images each time it needs to calculate a DoG layer. This is suboptimal because some images are used more than once. The current implementation increases the workload of each thread. Instead of doing a single subtraction, each thread performs a total of $N-1$ subtractions, where N is the number of layers per octave. This allows for every needed image to be loaded only once, saving a total of $\frac{N-2}{2N-2}$ of all memory accesses.

4.4 Extrema Detection

Finding the position of the local minima/maxima requires 26 comparisons per pixel. It is, therefore, a time-consuming task in which the control flow tends to diverge. Consequently, load balancing becomes harder, which may negatively impact performance.

The proposed implementation tries to minimize the effects of branching by creating a bitmask that is capable of evaluating if a given pixel is a minimum or a maximum. This bitmask can be very useful for the current case, as it converts most branches into bitwise operation, which reduces the number of data dependencies and consequently accelerates parallelism.

In CUDA, the ideal approach would be to take advantage of shared memory, however in this case, neighborhood regions overlap each other (as seen in figure 4.7), which tends to generate several bank conflicts. To avoid this scenario, our implementation opts for texture memory instead, as it is cached and also designed to be efficient for spatially-localized accesses in 2D arrays.

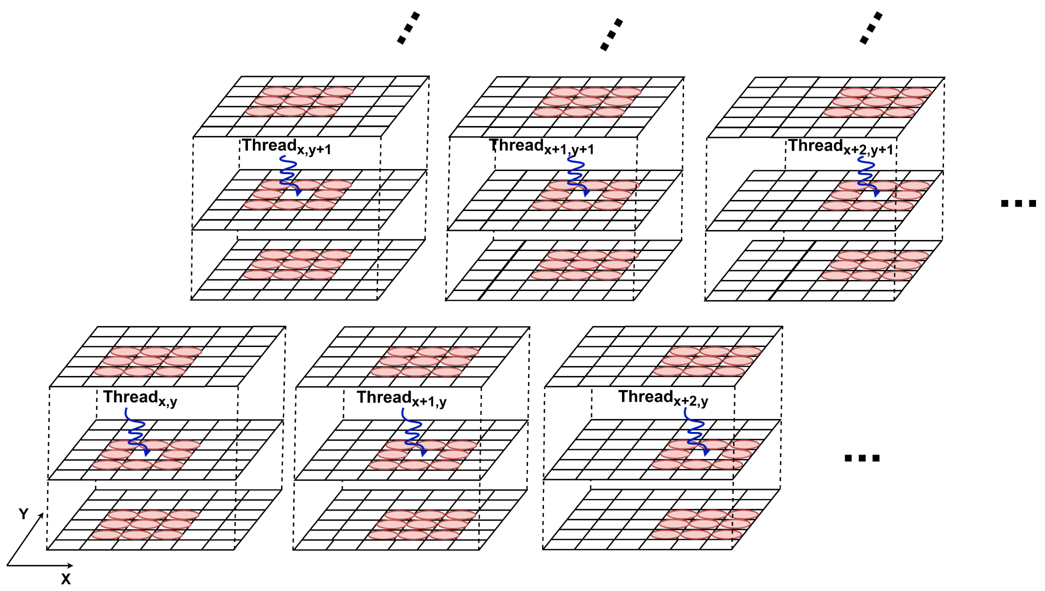


Figure 4.7: Extrema Detection on the GPU.

Regarding subpixel refinement, it can be achieved by using the Gaussian elimination, however there is a faster way that bypasses most data dependencies. That is, through a closed circuit based on the Laplace expansion.

4.5 Orientation Histogram

A naive implementation of the 36 bin histogram on the GPU would make each allocated thread responsible for computing the predominant orientation of each feature point (or keypoint). However, this approach suffers from load imbalance and non-coalesced accesses. The current work proposes to use a warp (32 threads) per keypoint instead, as this does not present the disadvantages mentioned above [42] [43].

The radius of the neighbor region around a given point is calculated with $r = 3 \times \lambda_{ori} \times \sigma_{oct}$, where r is the radius, $\lambda_{ori} = 1.5$ by default [44] and σ_{oct} is the scale of the octave in which the keypoint was located.

During run-time, each warp loads the gradient from those regions in a coalesced pattern, thereby increasing the number of simultaneous accesses to the same bin. However, since the radius is considered relatively small, atomic operations (operations that are guaranteed to be performed without interference from other threads) can be applied without significant performance losses.

4.6 Feature Descriptor

In order to generate SIFT descriptors, this work makes 512 threads cooperate in 16 warps (groups of 32 threads), where each group calculates 8 values of a descriptor, representing a histogram for one of the 16 squares surrounding the keypoint. Together they define the grid that is aligned with the image that contains their assigned square ($16 \times 8 = 128$ elements for each feature vector). For every pixel that is inside this region, weighted gradient information is extracted and added to the 8-bin histogram that makes up the descriptor.

After obtaining the descriptors of each keypoint, the final step is to normalize its values in order to reduce the effects of illumination changes, which can be done using an L2 normalization.

4.7 Pinned memory optimizations

The usage of pinned memory in SIFT is advantageous in a case where allocation is only done once, preferably at the beginning. A video stream, for example, can reuse the same memory space as long as its dimensions do not change(see figure 4.8), which allows the algorithm to bypass the overhead of re-allocations and improve the overall performance.

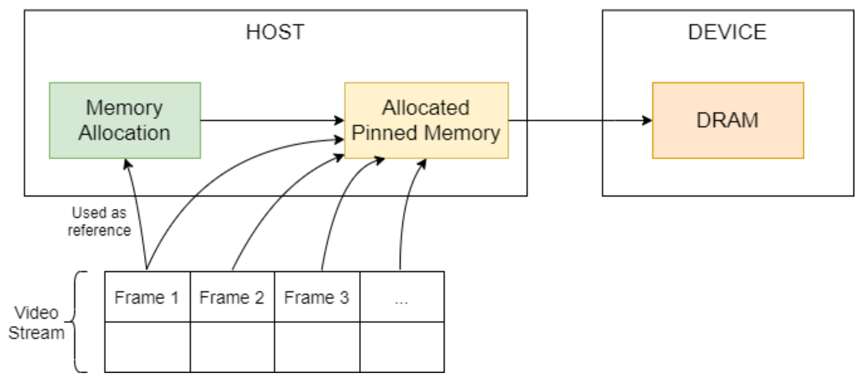


Figure 4.8: Example of how a video stream can take advantage of pinned memory transfers. Due to memory constraints, each frame must be processed sequentially, however the allocated memory space of the first frame is reused by every other frame, hence allocation is only needed once.

5 Experimental Results

Contents

This chapter presents and analyzes the results that were obtained from the parallelization of the SIFT algorithm. Each experiment was repeated multiple times on three different test benches (see table 5.1) to ensure consistency within the results.

Table 5.1 Test System Configurations

Workstations	GPU	CPU	RAM
a)	GTX TITAN X	Intel Core i7-4790K @ 4.00 GHz	32 GB
b)	GTX 1060	Intel Core i7-8750H @ 2.20GHz	16 GB
c)	Jetson TX2's GPU	ARM - Cortex A57 @ 2GHz	32 GB

5.1 System Setup

All of the GPUs present in this environment are part of the Pascal micro-architecture developed by NVIDIA. Consecutively, these devices only differ in their parameters, with some having more or fewer capabilities/constraints than others. For instance, the GTX TITAN X is a high-end GPU mostly intended for HPC. Hence it requires more resources than a typical GPU. The GTX 1060, despite the lower performance, is still known to be cost-efficient and versatile for various tasks. And finally, the Jetson TX2 has a custom-built GPU with the purpose of maximizing energy efficiency (over performance) on low power applications.

A more detailed overview of each can be seen in the following table:

Table 5.2 GPU specifications

	GTX TITAN X	GTX 1060	Jetson TX2
Memory Size	12 GB	6 GB	8 GB
Memory Type	GDDR5	GDDR5	LP-DDR4
Memory Bus	384 bit	192 bit	128 bit
Bandwidth	336.6 GB/s	192.2 GB/s	59.7 GB/s
Base Clock	1000 MHz	1506 MHz	854 MHz
Boost Clock	1089 MHz	1709 MHz	1300 MHz
Memory Clock	1753 MHz	2002 MHz	1866 MHz
TDP	250 W	120 W	7.5 W - 15 W
SM Count	24	10	2
L1 Cache (per SM)	48 KB	48 KB	48 KB
L2 Cache	3 MB	1536 KB	512 KB
FP32 (float) performance	6.691 TFLOPS	4.375 TFLOPS	750.1GFLOPS
FP64 (double) performance	209.1 GFLOPS	136.7 GFLOPS	23.44 GFLOPS
Cuda Cores	3072	1280	256

The different characteristics of each device allow reduction of any bias regarding the use of GPGPU in feature extraction algorithms, both in terms of performance and energy efficiency.

In this thesis, experiments were made in a controlled environment in both Windows 10 and Ubuntu 17.04 OSes with temperatures ranging from 30 to 60°C to prevent thermal throttling. The current work is divided into two different parts: A parallel solution made in CUDA and a serial version made in C. The latter is a heavily modified version of [44], which tries to follow Lowe's algorithm as close as possible. The GPU solution was developed iteratively from the serial version and, likewise, does not deviate from the premises of the algorithm.

The compilers that were used were the GNU Compiler Collection (GCC) for the sequential version and the NVIDIA CUDA Compiler (NVCC) for the parallel version. In addition to the above, various flags were explored to increase performance. These include:

- Level 2 optimizations (-O2) for both serial and parallel versions;
- Enabling PTX assembler optimizations for the GPU code (-Xptxas O2);
- Defining the maximum register count for each GPU device in order to achieve higher occupancy (-maxrregcount=N[1]);
- Matching the SM architecture accordingly (-arch compute_6x[2])
- Compiling for the 64 bit platform (–machine 64) to permit the usage of more than 4GB of RAM memory.

For testing purposes, the serial version is only limited to one CPU (Intel Core i7-8750H@2.20GHz) core set to high priority in order to mitigate any possible interference from the OS scheduling system.

The datasets that are used to evaluate the performance of implemented work were obtained from [11] and [10]. They consist of a collection of .pgm images of various sizes and dimensions.

Finally, execution times are measured using functions that have very high accuracy and precision. In Linux, the program resorts to clock_gettime() from the time.h C library. In the case of Windows, since there is no direct equivalent, QueryPerformanceCounter() and QueryPerformanceFrequency() are used.

5.1.1 GPU Memory transfers

Before taking a throughout inspection of the current work, it is analyzed how memory transfers between Host and Device can impact the performance of the algorithm.

[1]N is a number which can be obtained using the NVIDIA occupancy calculator for that device.
[2]"compute_61" for both the Titan X and GTX 1060 , "compute_62" for the Jetson TX2.

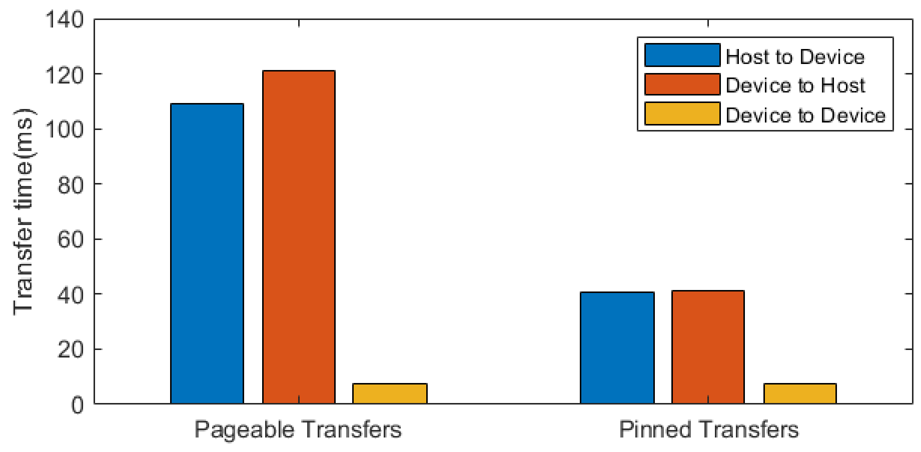


Figure 5.1: HtoD, DtoH and DtoD transfer times for 5MB of data on a GTX 1060

Figure 5.1 shows the times for both pageable and pinned memory transfers from host to device (HtoD), device to host (DtoH) and device to device (DtoD). The usage of pinned memory can cut the transfer times by more than a half in both DtoH and HtoD (DtoD is not affected by pinned memory), however that leads to longer allocation times as a consequence. On the other hand, as previously mentioned in chapter 4, a video feed can bypass consecutive allocations as long as every frame shares the same memory space. Thus, in that case, the usage of pinned memory can be beneficial for SIFT like applications.

5.1.2 CUDA shared memory

Shared memory is often considered to be faster than global memory due to its lower latency and higher bandwidth. However that might not be always the best answer. Besides having a more complex implementation, shared memory can also be subject to overheads that are absent from other versions (e.g. bank conflicts). Another point take into consideration is that sometimes it is better to favor a simpler implementation and better scalability over small gains. It is therefore crucial for developers to evaluate and decide what is the optimal solution for each segment.

Figure 5.2 for example, shows that convolution using shared memory is capable of achieving twice the performance of convolution using texture memory (which is an abstraction of global memory).

On the other side, there are also cases where the usage of shared memory is infeasible. For example, when comparing pixels in various vicinities in order to find a local minimum or a maximum (local extremum detector).

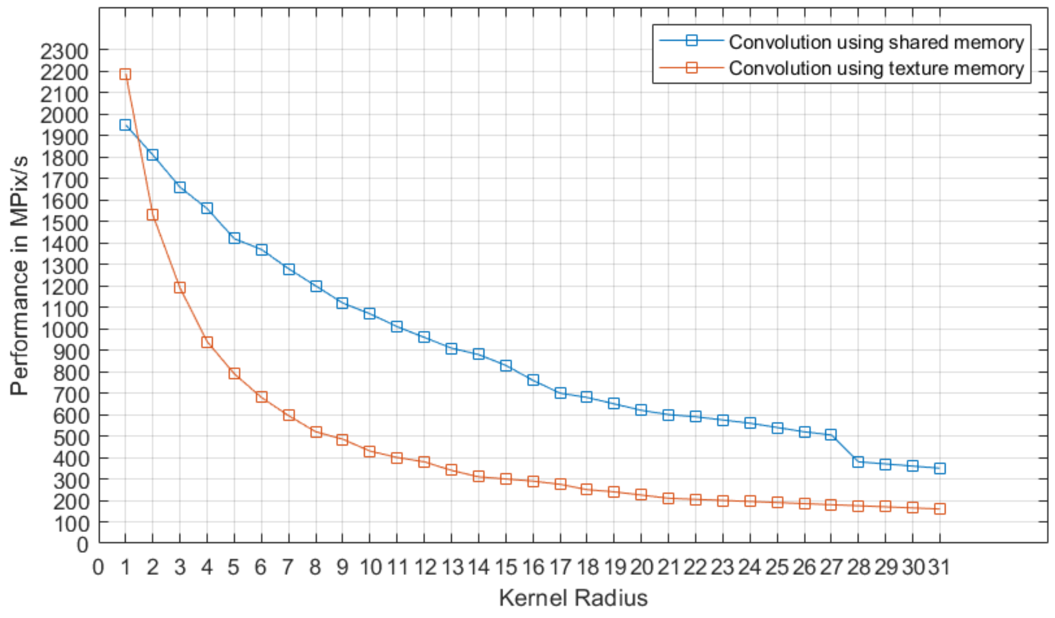


Figure 5.2: Convolution using shared memory vs convolution using texture memory, adapted from [9]

.

5.1.3 Gaussian Convolution vs Box Filters

In the previous chapter, two different approaches to convolution were introduced. In this part, these methods are compared using various images from the datasets in order to determine the performance and accuracy of both implementations.

As shown in graph 5.3, the original 2D Gaussian convolution is by far the slowest implementation in all cases. As explained previously, this result is mostly due to the sub-optimal implementation which generates more instructions than other solutions.

The Gaussian separable convolution is the fastest method present in this part. This is to be expected as this version explores more parallel techniques which were previously impossible, like for example: The execution of rows and columns in parallel.

Box filters are an alternative solution to Gaussian filters. However they require the minimum of 3 passes in order to resemble a bell curve (see figure 4.4). In terms of performance, they are close to the 1D gaussian filters mostly due to the fact that each pass of this filter only requires only 1 multiplication and 2 additions per pixel regardless of the filter size. (This can prove beneficial when attempting to simulate a Gaussian filter with a high standard deviation).

In general, convolutions are operations that can be very well optimized to run

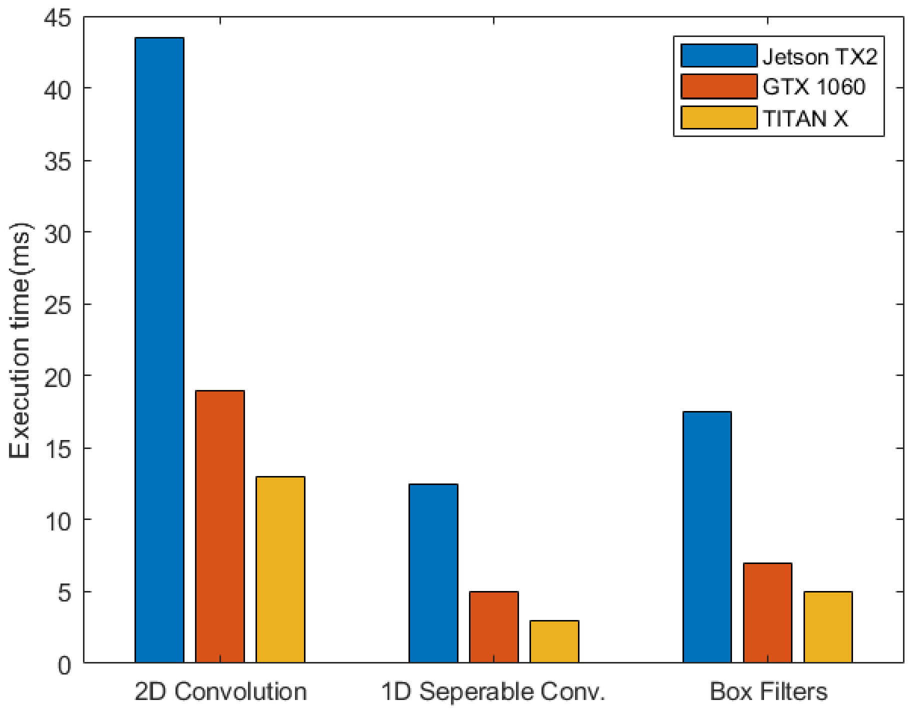


Figure 5.3: The performance of the three convolution methods that were analyzed in a 2560×1920 image using $\sigma = 10$.

on GPUs. For this reason, it should come to no surprise to see speedups up to 10x or more relative to sequential methods.

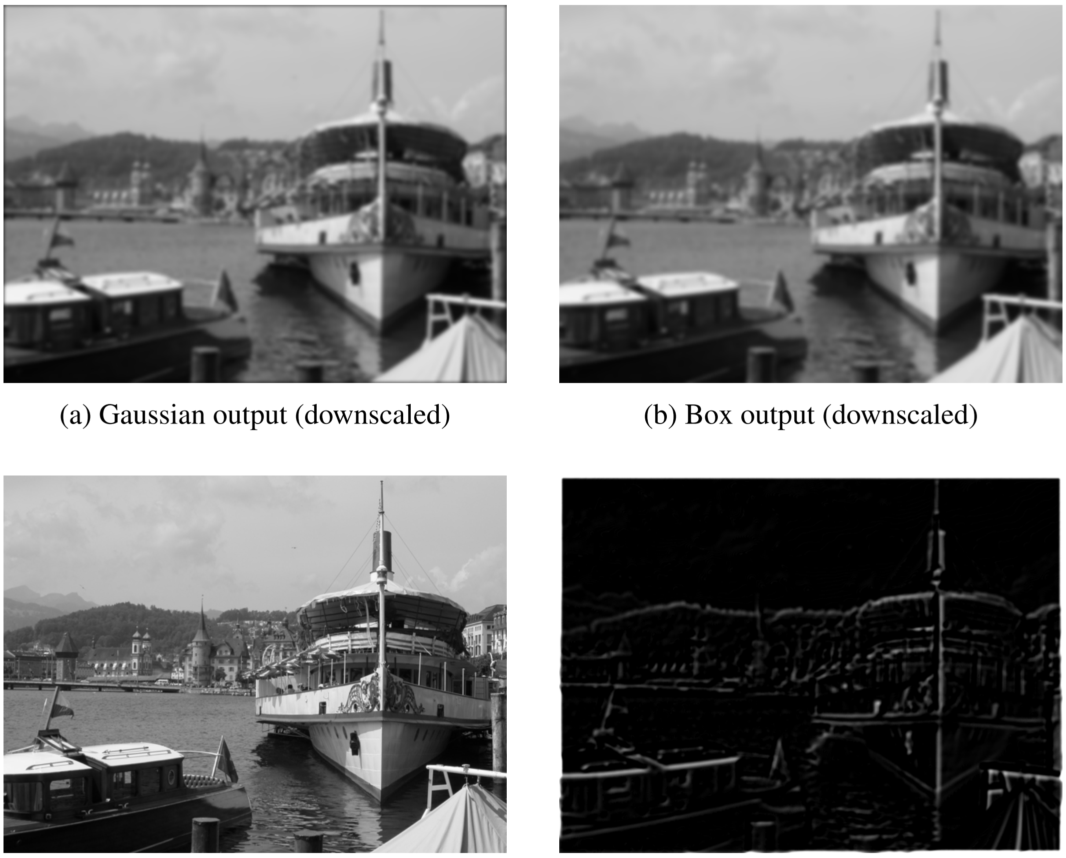
In terms of accuracy, Figure 5.4. shows the differences between each method, which in this case are very small.

5.2 Scale-space generation

In order to generate the Gaussian scale-space it is required at least one of the convolution methods mentioned above, an upscaling kernel, and a downscaling kernel. In this work, all of these functions are parallelized in order to maximize performance. The upscaler uses bilinear interpolation, while the downscaler resamples images from every other pixel.

Seeing that each kernel can be called multiple times, it makes sense to analyze the accumulated times that each takes on a given image (see Table 5.3). As a matter of fact, the convolution kernel is called once for each existing layer, the downscaling kernel is called once for each octave and the upscaling kernel is only executed once at the very beginning.

[3]Gaussian separable convolution


(a) Gaussian output (downscaled)

(b) Box output (downscaled)

(c) Original image (downscaled)

(d) Differential between outputs (amplified 10x to allow visual inspection, since the error between the two proposed methods is extremely small).

Figure 5.4: Comparison between the Gaussian convolution and Box convolution using $\sigma = 10$.

Table 5.3 Accumulated times (in milliseconds) for the scale-space generation of a 2560×1920 image

Kernel	GTX 1060	Titan X	Jetson TX2	CPU
Convolution[3]	19.02	14.2	401.55	1131.54
Downscale	0.34	0.27	1.23	2.05
Upscale	1.7	1.54	8.54	22.95
Total	21.06	16.01	411.32	1156.54

These values show that, across every device, the convolution kernel has the biggest impact on the scale-space. Both the downscaler and the upscaler by themselves do not present a significant bottleneck. However, its worth mentioning that by introducing the upscaling kernel, every layer of the first octave becomes twice the size of the original image, which consequently adds more time to the other kernels. It is for this reason that some implementations decide to skip it entirely.

In terms of performance between machines, the fastest time recorded was from the TITAN X, achieving a speedup of 72.26× relative to the serial version. The GTX 1060 follows next with a speedup of 54.92×, and finally the Jetson TX2 with a speedup of 2.81×, but with a much lower energy cost.

To give a better perspective, these speedups may vary according to the size of the input image as seen in figure 5.5.

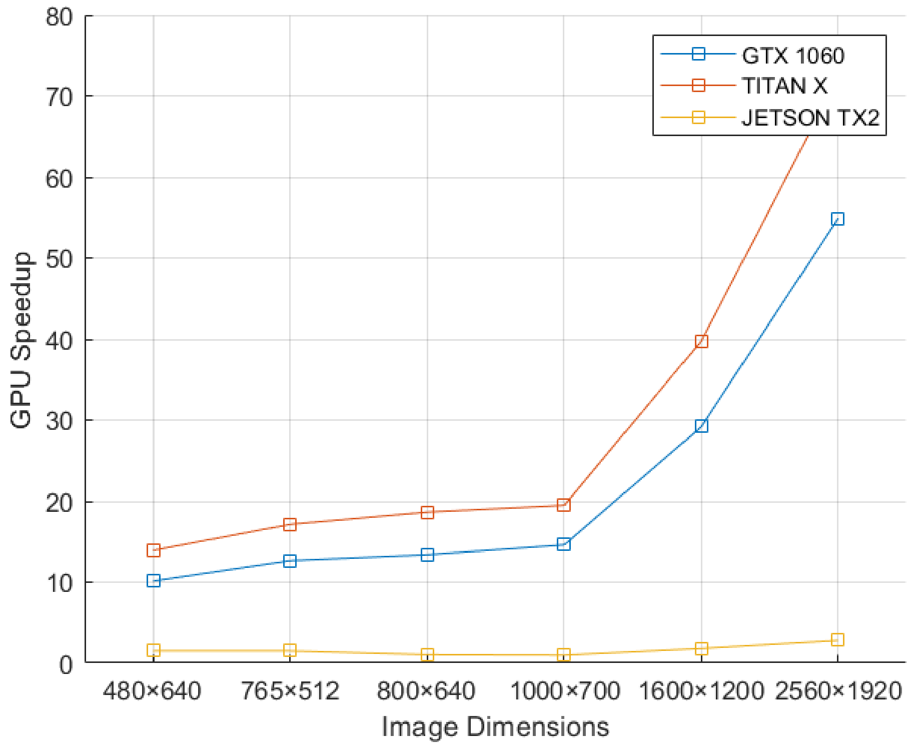


Figure 5.5: Speedup of each device relative to the CPU while generating the scale-space.

5.3 Difference of Gaussians

The difference of Gaussians is an interesting segment to run on a GPU, since the only requirement of this step is to perform a maximum of 2 subtractions per pixel for each octave, it can be done almost instantaneously as there are no data dependencies associated with this operation (see Table 5.4).

In this part, it is possible to see a big difference in performance between the CPU and GPU. While the CPU needs to iterate through every pixel of every generated layer, the GPU can just use its grid of thread blocks to simplify the problem, bringing huge gains for every device.

Table 5.4 Difference of Gaussians on CPU and GPU in milliseconds

Image Size	GTX 1060	Titan X	Jetson TX2	CPU
480×640	< 1	< 1	1.56	12.1
765×512	< 1	< 1	2.1	16.28
800×640	< 1	< 1	3.46	19.14
1000×700	< 1	< 1	4.6	24.86
1600×1200	< 1	< 1	7.141	66.44
2560×1920	1.17	< 1	11.43	165.22

5.4 Extrema Detection

The next step, on the other hand, requires a more detailed analysis since its implementation is more complex by design. At first, it was attempted to build a naive version in global memory that used branches, this approach was quickly discarded as the yielded results were about the same if not worse than the sequential equivalent. On the second attempt, global memory was replaced with shared memory. This version not only was more difficult to implement but also brought up several issues, mostly regarding bank conflicts. Due to these circumstances, it was also scrapped. The third and final attempt explored texture memory in order to take advantage of its caching mechanisms, and branchless operations to prevent divergence in the algorithm.

The results in figure 5.6 show that the parallel implementations achieve higher speedups relative to the serial version as the size of data increases.

Since texture memory is cached it should also be optimal for this type of problem since each pixel interacts with every other pixel in its vicinity.

5.5 Orientation Histogram

A basic serial histogram is relatively simple to implement. For each keypoint it tries to find the corresponding bin for each orientation and increments its value. Parallel histograms, on the other hand, are more difficult to implement due to possible collisions. The main challenge here is that the output location of each element is not known prior to its reading. Therefore it is impossible to create a parallel histogram that completely avoids collisions. Nowadays, the improved atomics performance from newer architectures has allowed this problem to become less of an issue. Hence this work proposes to use a combination of shared memory and atomics in order to build the orientation histogram of each keypoint.

Figure 5.7 shows that the GPUs achieve various gains, however the overall impact of this step is not so significant, mostly due to the fact that the radius around each keypoint is relatively small and therefore only a small patch of data is pro-

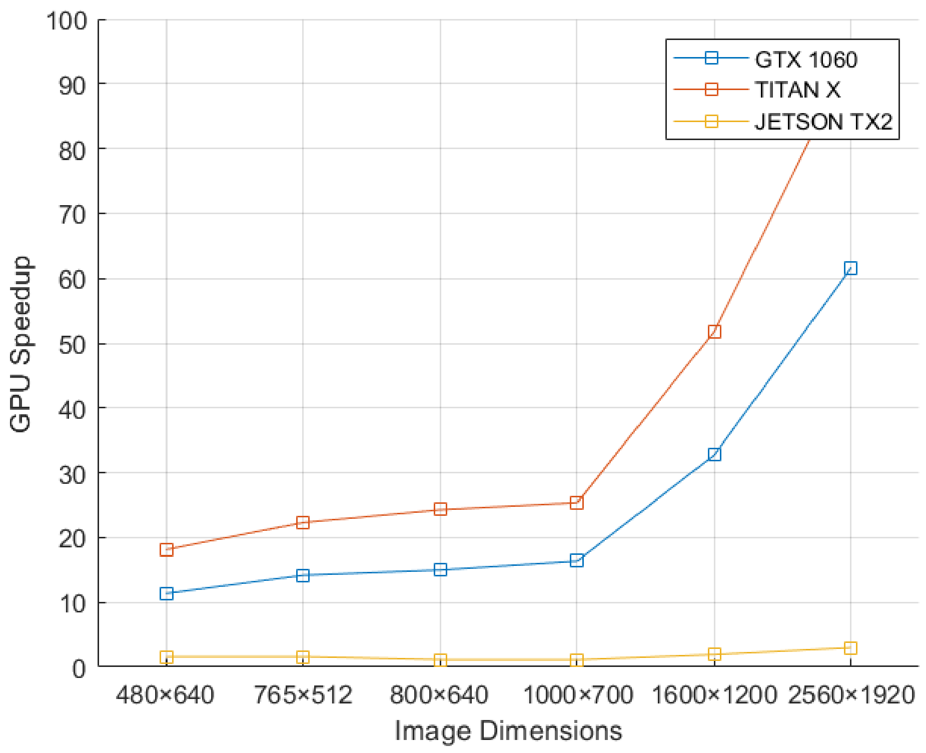


Figure 5.6: Speedup of each device relative to the CPU while locating extrema points.

cessed to create each histogram.

5.6 Feature Descriptor

The feature descriptor has a similar implementation to the previous method, however in this case, the amount of data that is handled is much larger. Which in turn can result in lower relative gains between each machine as stress increases due to a greater number of collisions in each cell (see figure 5.8).

In this graph the TITAN X presents the best performance with a relative speedup up to 69.97×, the GTX 1060 comes next with a relative speedup up to 53.26× and finally the Jetson TX2 with a speedup of 2.8×.

5.7 Global SIFT Behavior and Results

In general, results show that for small scale images, the serial implementation has no problem keeping up with real time execution. However, as image dimensions start to expand, the number of detected keypoints increases and the overall performance diminishes. To sum up, the exponential growth of the complexity of the SIFT algorithm and the lack of resources from the CPU to handle it are the main

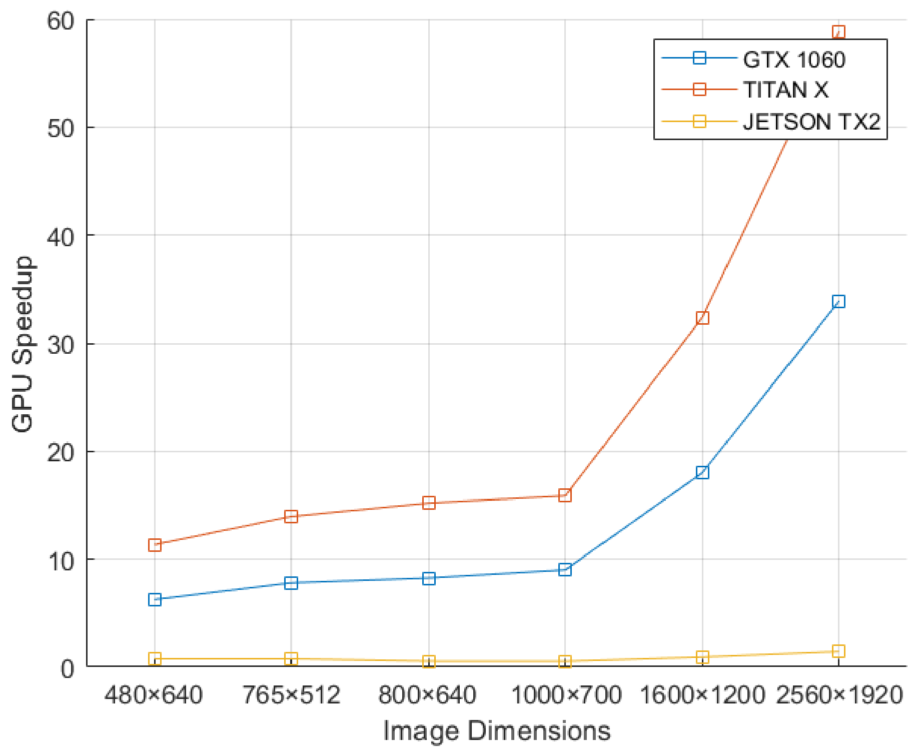


Figure 5.7: Speedup of each device relative to the CPU while finding the orientation of each keypoint.

factors for performance degradation in a serial solution.

5.7.1 Performance analysis

This increase in demand does not reflect the same level of impact on a mid to high end GPU. Mostly due to the fact that parallel implementations split the problem into more simpler pieces which can then run concurrently. In reality, the limiting factor of the GPU ends up being its grid size and processing power, which explains the differences that were obtained between the GTX 1060 and the TITAN X (see Table 5.5).

It is important to reiterate that the CPU is only running one thread and that the main purpose of table 5.5 is to highlight the importance of parallelism in SIFT like applications.

5.7.2 Energy Efficiency Analysis

As seen in the results from the previous parts, the Jetson TX2 can not compete with the other GPUs in terms of speed. This is mostly because it was built to be a low power GPU that maximizes energy efficiency over performance. In order to

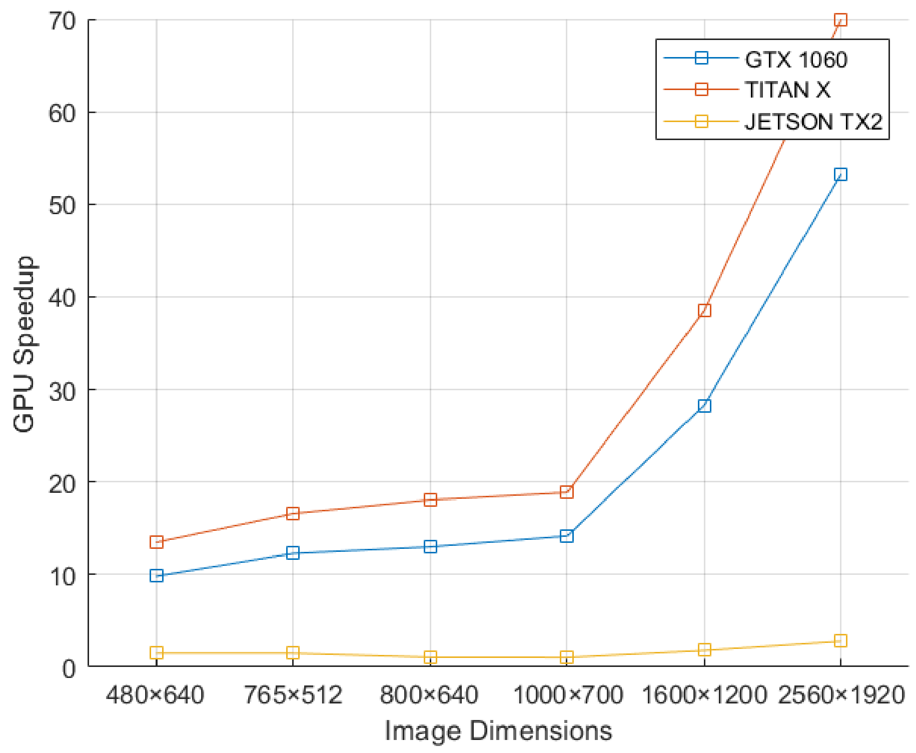


Figure 5.8: Speedup of each device relative to the CPU while generating feature descriptors.

Table 5.5 Summary of times(measured in milliseconds) and speedups of each segment in order to extract features from a 2560x1920 image.

	GTX 1060		TITAN X		JETSON TX2		CPU
Segment	Time	Speedup	Time	Speedup	Time	Speedup	Time
Scale-Space	21.06	54.92	16.01	72.26	411.32	2.81	1156.54
Difference of Gaussians	1.17	141.21	0.42	392.26	11.43	14.46	165.22
Extrema detection	12.87	61.62	8.42	94.14	262.79	3.02	793.06
Orientation histogram	2.93	33.89	1.7	58.84	68.55	1.45	99.13
Feature descriptor	20.48	53.26	15.58	69.97	388.47	2.81	1090.45
Total	58.5	56.5	42.12	78.45	1142.56	2.89	3304.4

make a fairer comparison, table 5.6 demonstrates the performance per watt of each implementation.

While the other GPUs can outperform the low-power GPU version in most cases. An analysis to the energy consumption of each platform shows that the Jetson TX2 is far more power efficient for the same amount of work. It is only surpassed by the GTX 1060 at the last two larger images due to its limitations and constraints.

On lower scale images the JETSON can spend up to 6× less power than its CPU counterpart running @2.20 GHz while achieving the same level of accuracy.

Table 5.6 Performance per watt of each device (FPS/W × 100)

Image Resolution	GTX 1060	TITAN X	JETSON TX2	CPU
2560×1920	**14.25**	9.5	11.67	0.67
1600×1200	**18.81**	12.99	18.67	1.67
1000×700	23.08	15.61	**26.14**	4.47
800×640	29.98	21.16	**33.94**	5.81
765×512	33.33	22.86	**37.61**	6.82
480×640	35.92	24.99	**44.44**	9.18

5.7.3 Matching the robustness of the SIFT parallel algorithm

In this subsection, it is presented various examples from the two datasets in order to show the location of the extracted features obtained from the proposed implementation (see figures 5.9, 5.10, 5.11, and 5.12). As well as examples of image matching in order to demonstrate the robustness of the algorithm to scale changes, rotation changes and illumination changes (see figures 5.13, 5.14, 5.15, 5.16, 5.17, and 5.18). In the end, a deeper analysis of these claims is made.

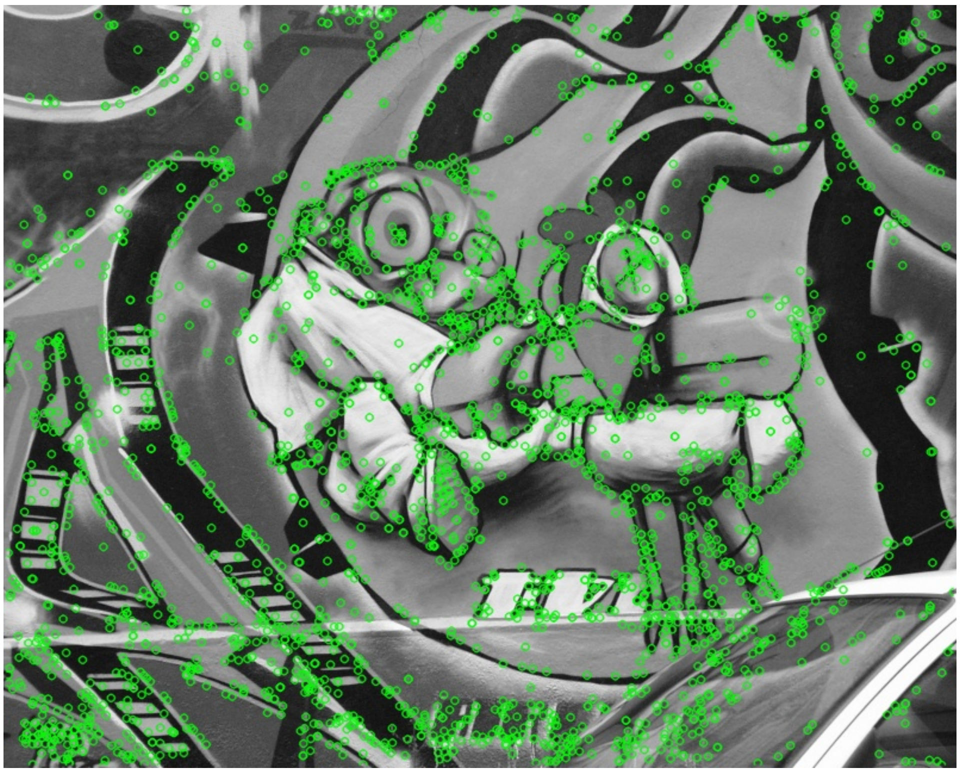


Figure 5.9: 800x640 image from [10]: 2832 keypoints detected.

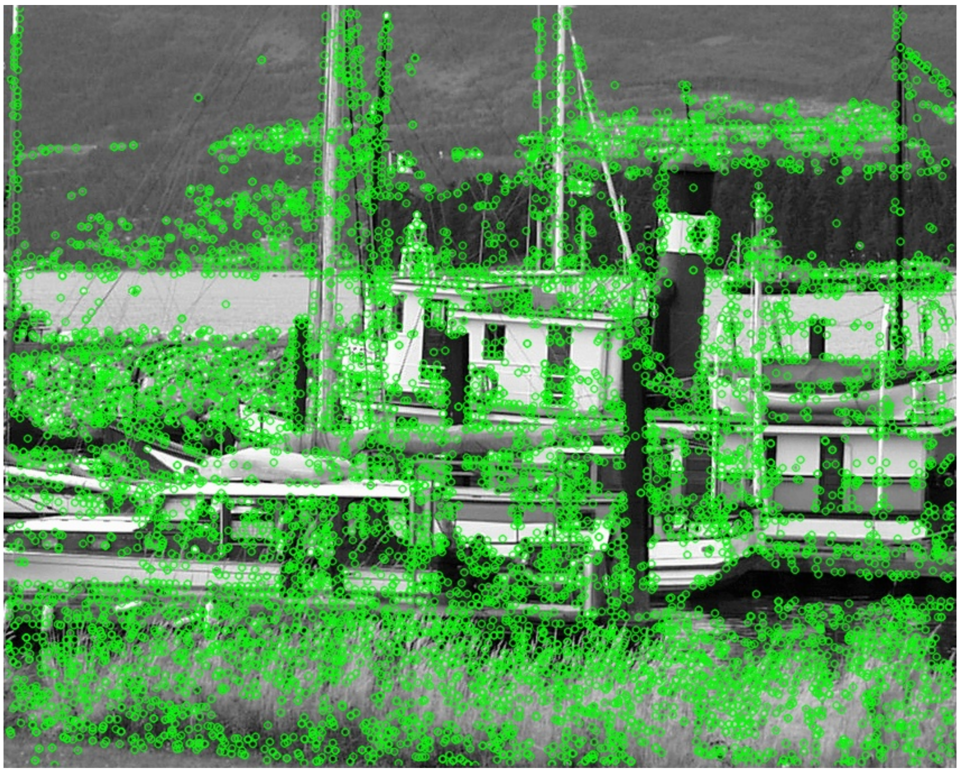


Figure 5.10: 850x680 image from [10]: 8950 keypoints detected.

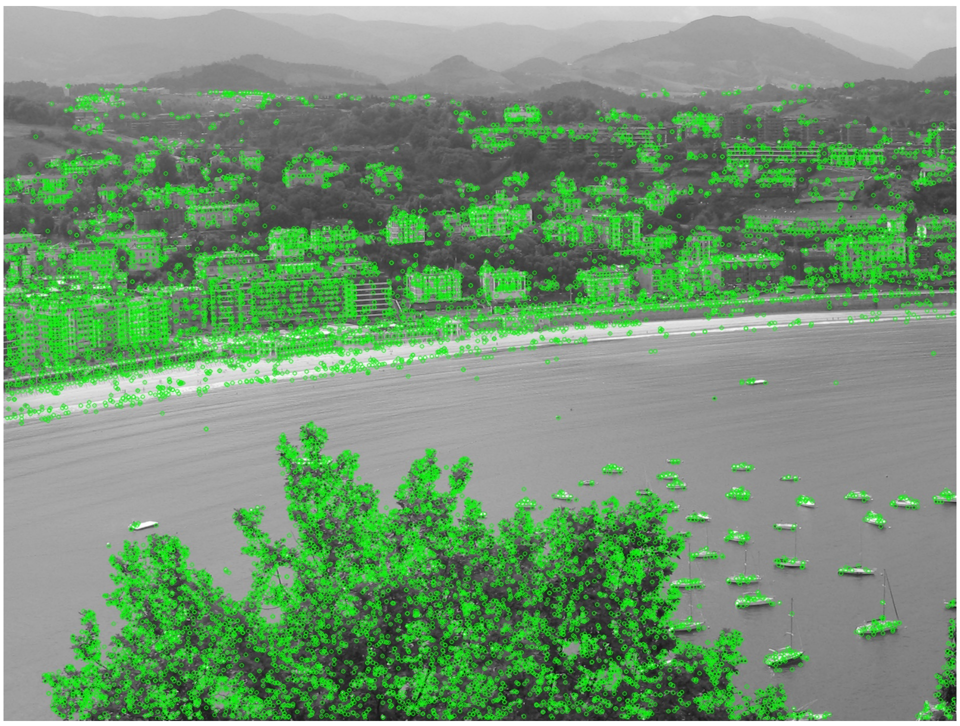


Figure 5.11: 1600x1200 image from [11]: 18021 keypoints detected.

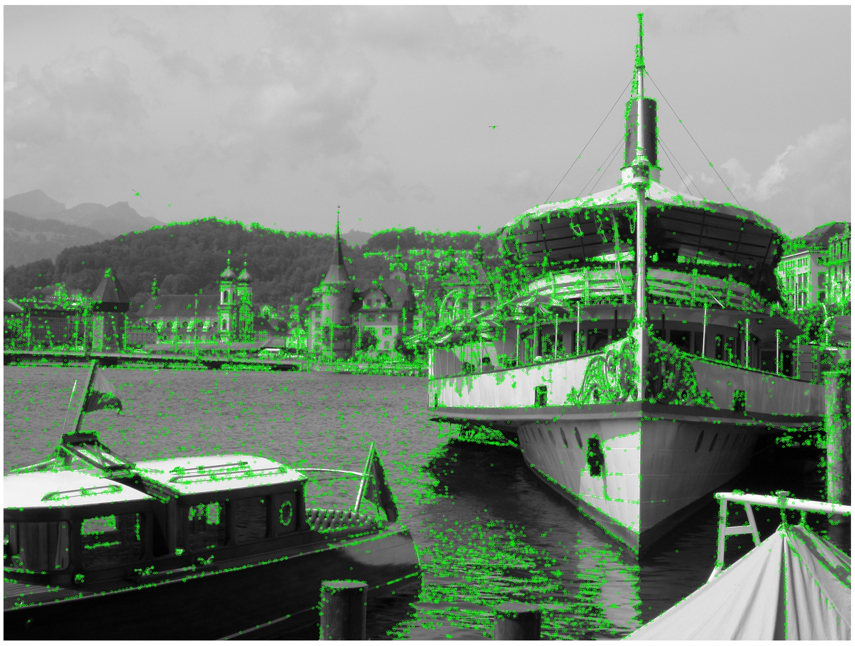


Figure 5.12: 2560x1920 image from [11]: 24432 keypoints detected.

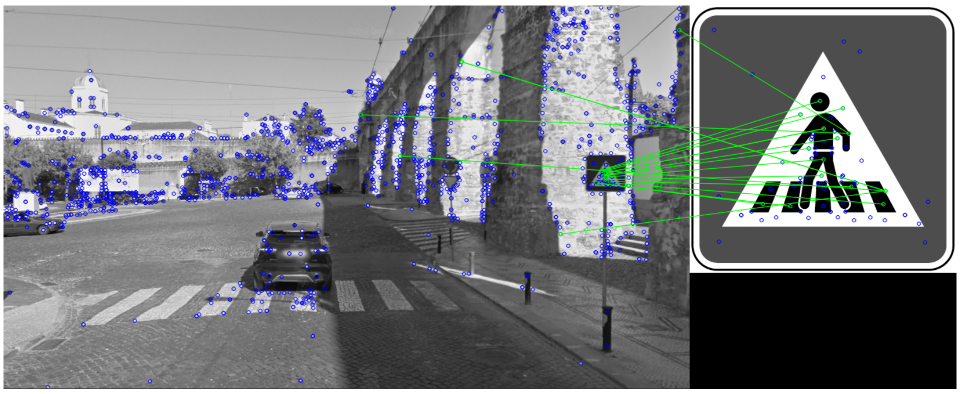


Figure 5.13: Example of scale invariance in matching.

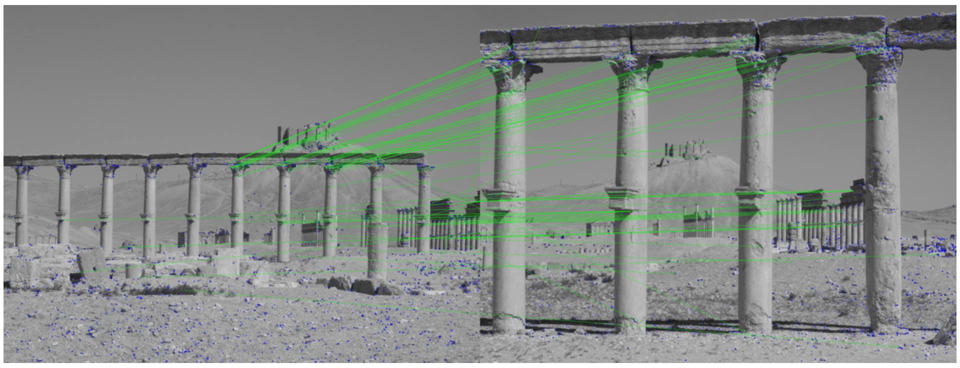


Figure 5.14: Another example of scale invariance in matching

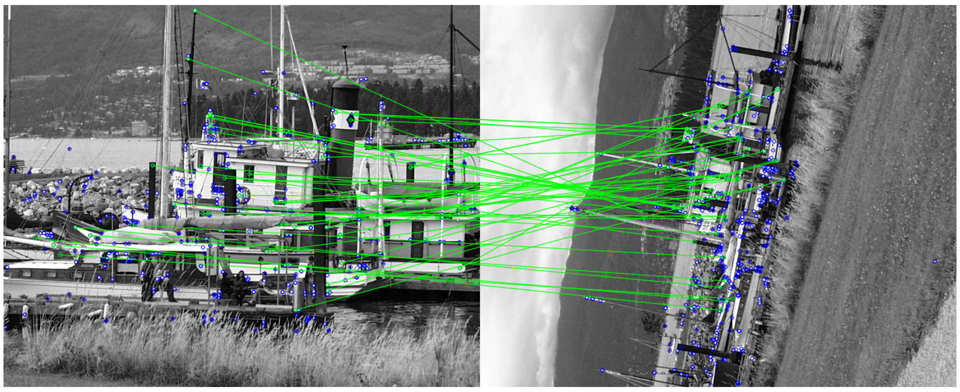


Figure 5.15: Example of rotation invariance in matching.

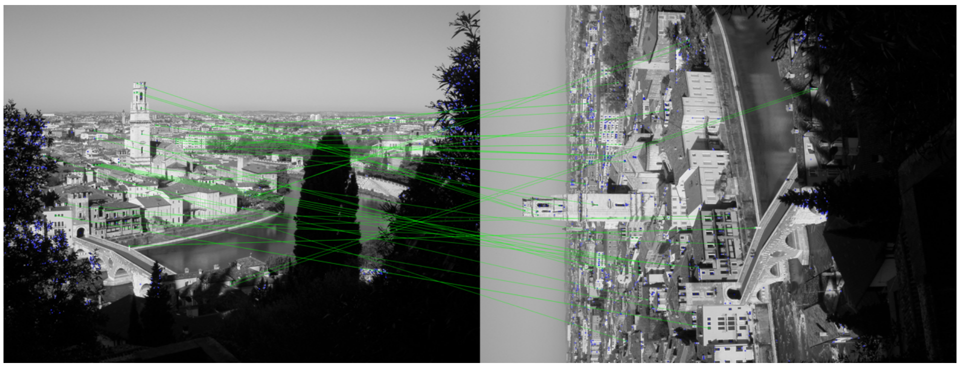


Figure 5.16: Another example of rotation invariance in matching

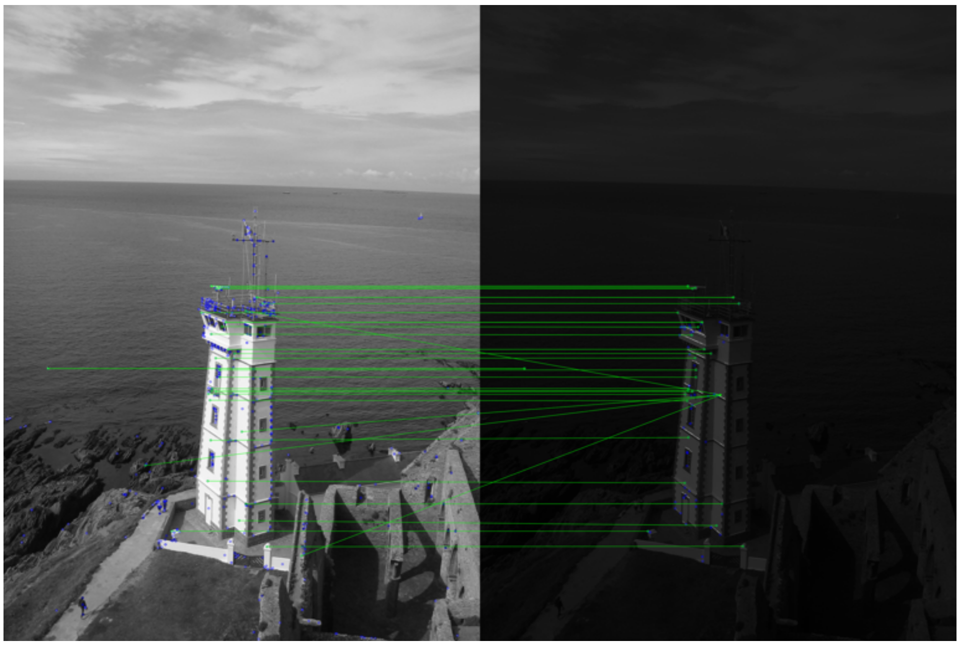


Figure 5.17: Example of illumination invariance in matching.

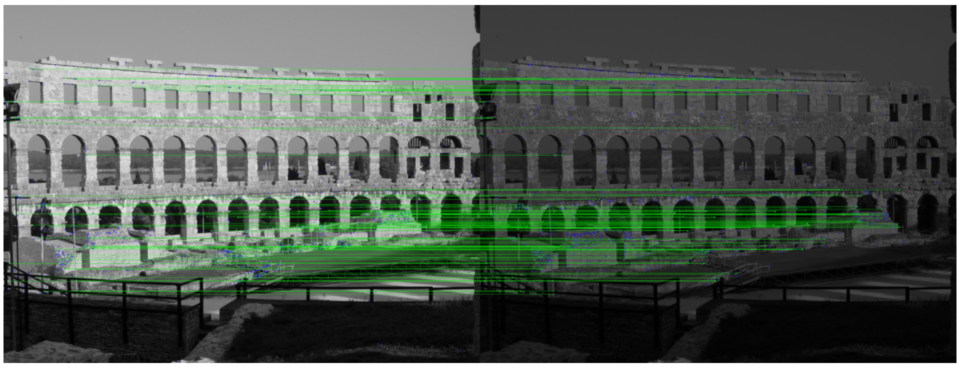


Figure 5.18: Another example of illumination invariance in matching.

Like any other detection algorithm, false positives or matching errors are inevitable (best seen in Figure 5.13). Although it is nearly impossible to eliminate all of them without affecting other matches, it is still desirable to use the parameters suggested by D. Lowe to help minimize the number of outliers and improve the quality of the extracted features.

In order to assess the robustness of the present work, a series of tests were carried out on three separate images: *Graf.pgm*, *Boat.pgm* and *Bikes.pgm*. For each of these images three scenarios were applied. A rotational scenario, a down-scaling scenario and a illumination scenario. Next, each is addressed in more detail.

Rotation Robustness: Rotation invariance suggests that the algorithm is capable of correctly matching keypoints between a given image and a rotated version of that same image or scene. In order to evaluate the rotational robustness of the proposed solution, a known rotation must be first applied to the original image. Then, once the algorithm is executed on the original, a ground-truth can be created for the rotated image (since keypoint location can be predicted). After applying the algorithm to the rotated figure and finding all the corresponding matches, the ground-truth can be used to check the accuracy of its matches. In this case, from 10^o to 60^o more than 90% of matches are correct (see Figure 5.19).

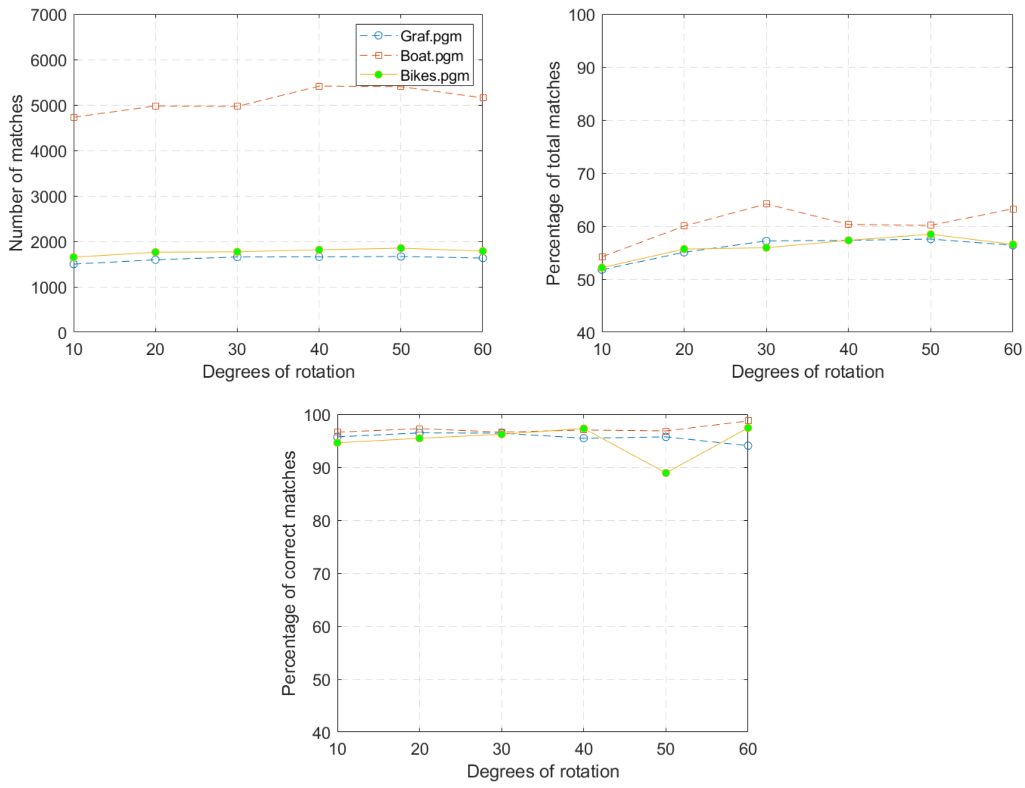


Figure 5.19: Rotation invariance analysis. The reason why 'boat.pgm' has more matches than the other two images is mostly due to a greater number of detected keypoints during the extraction phase.

Scale Robustness: Scale invariance implies that the algorithm is capable of correctly matching keypoints between a given image and a down-scaled version of that same image or scene. In order to verify the scale robustness of the proposed implementation, a similar method to the above is used. First, the image is scaled down by a known factor, then keypoints are extracted from the original and are used as a ground-truth for future keypoints of the downscaled image. Once the algorithm runs the second image and finds the corresponding matches, a comparison with the predicted keypoints can be used to determine the accuracy of the matches. In this case, more than 80% of matches are still correct after a downscale by a factor of 5$\times$ (see Figure 5.20).

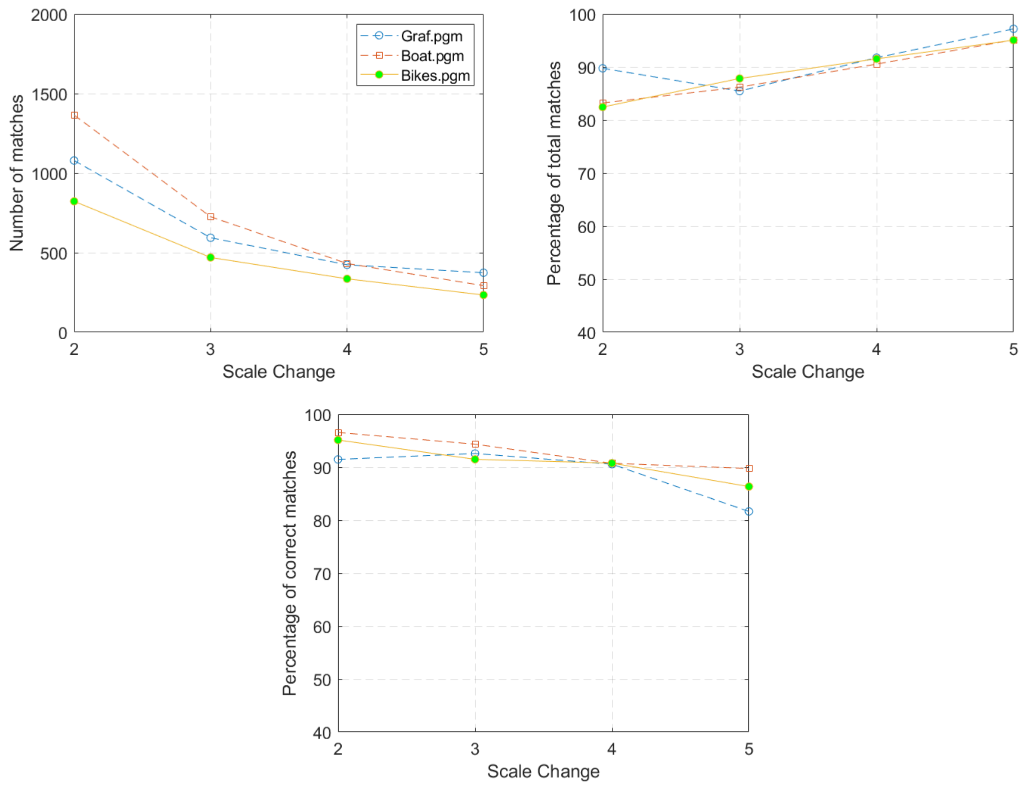


Figure 5.20: Scale invariance analysis. Any time an image scales down in size, the amount of information contained in the image decreases. For this reason, the number of keypoints and subsequent matches is expected to decline as well.

Illumination Robustness: Illumination invariance indicates that the algorithm is capable of correctly matching keypoints between a given image and a darker or brighter version of that same image or scene. In order to check the illumination robustness of the current work, no affine transformation is required, as compared to previous methods. In this part, the algorithm only needs to compare the differences between the keypoints found in each image. Results show that the algorithm is largely stable with more than 95% accurate matches up to the 40% brightness stage, where it begins to decline sharply (see Figure 5.21). The differences seen in the *boat.pgm* throughout the different scenarios are largely due to the fact that it has more detected keypoints and is a brighter image in general.

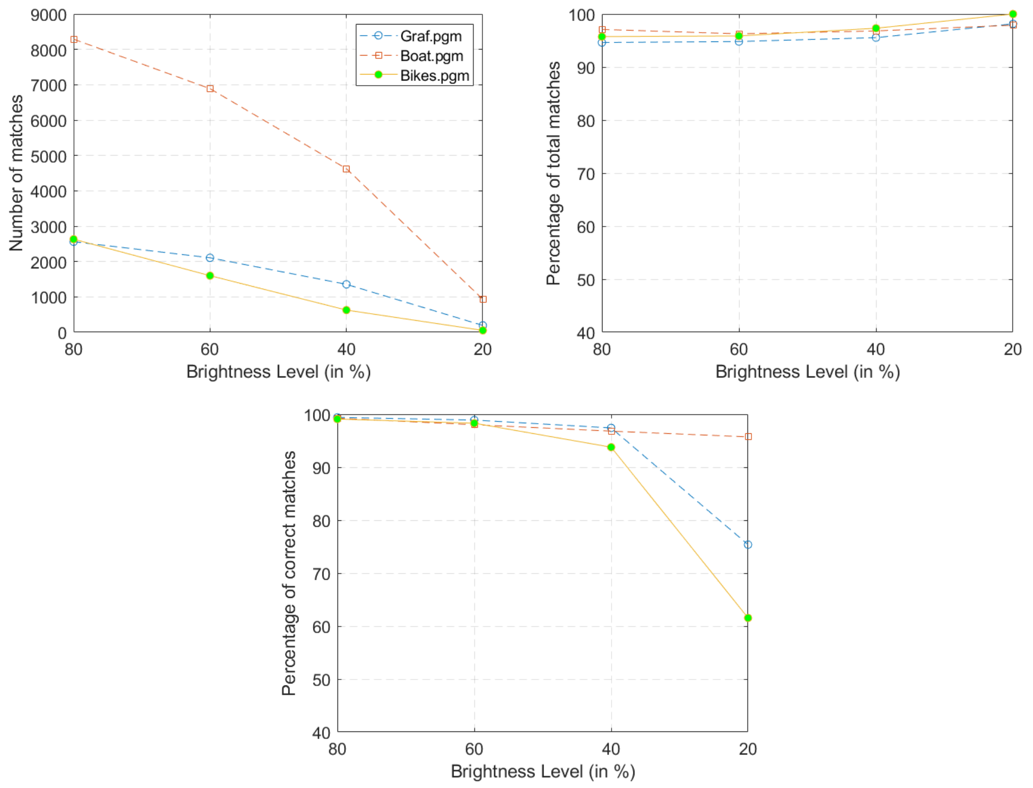


Figure 5.21: Illumination invariance analysis. When the level of brightness is close to 20%, the number of detected keypoints and matches drops significantly. As a result, the quality of remaining matches becomes uncertain as the number of correct matches decreases.

6 Conclusions

Contents

By introducing the concept of GPGPU to feature extraction algorithms, it becomes possible to obtain massive gains when compared to optimized serial implementations, especially on larger scale inputs, as long as the hardware has enough resources and computational power to do so. The proposed implementation is on par with the state of art and can achieve 24 FPS on a 2560x1920 image with a GTX TITAN X.

The use of dedicated low-power GPUs also brings a new paradigm to feature extraction algorithms. Despite the current solution not being able to replicate the efficiency of FPGAs and ASICs, it is still interesting from a development-effort perspective. Besides, the gap between GPUs and FPGAs is shrinking as the demand for more efficient smartphones continues to increase.

To conclude, the present work explores the use of emerging parallel hardware and its feasibility for future signal processing applications.

6.1 Future Work

The results obtained are great and encourage to continue investigation on GPGPU applications. However, for the present work there are still some steps that can be further improved, to name a few:

- Using of interpolation that is integrated into the hardware in order to find the accurate location for keypoints faster.
- Testing Convolution Neural Networks as a possible replacement of extrema detection.
- Expanding the number of image formats supported.
- Testing the present solution in other GPUs

Bibliography

[1] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[2] D. G. Lowe, “Object recognition from local scale-invariant features,” in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, Sep. 1999, pp. 1150–1157 vol.2.

[3] G. Baltazar, “Cpu vs gpu in machine learning,” September 2018. [Online]. Available: https://blogs.oracle.com/datascience/cpu-vs-gpu-in-machine-learning

[4] NVIDIA Corporation, “NVIDIA CUDA C programming guide,” 2019, version 10.2.

[5] M. Harris, “How to optimize data transfers in cuda c/c++,” December 2012. [Online]. Available: https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

[6] ——, “How to overlap data transfers in cuda c/c++,” December 2012. [Online]. Available: https://devblogs.nvidia.com/how-overlap-data-transfers-cuda-cc/

[7] ——, “How to access global memory efficiently in cuda c/c++ kernels,” January 2013. [Online]. Available: https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

[8] E. Domazet, M. Gusev, and S. Ristov, *Optimizing High-Performance CUDA DSP Filter for ECG Signals*, 01 2016, pp. 0623–0632.

[9] V. Podlozhnyuk, “Image convolution with cuda,” September 2013.

[10] Visual geometry group, university of oxford, affine covariant features. available: http://www. robots. ox. ac. uk/ vgg/research/affine/.

[11] M. D. Herve Jegou and C. Schmid, "Hamming embedding and weak geometry consistency for large scale image search," October 2008.

[12] Mahesh and M. V. Subramanyam, "Automatic feature based image registration using sift algorithm," in *2012 Third International Conference on Computing, Communication and Networking Technologies (ICCCNT'12)*, July 2012, pp. 1–5.

[13] A. Ahsan and D. Mohamad, "Features extraction for object detection based on interest point," *TELKOMNIKA Indonesian Journal of Electrical Engineering*, vol. 11, May 2013.

[14] H. Zhou, Y. Yuan, and C. Shi, "Object tracking using sift features and mean shift," *Computer Vision and Image Understanding*, vol. 113, pp. 345–352, March 2009.

[15] F. Yu, H. Luo, Z. Lu, and P. Wang, *3D Model Feature Extraction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 161–235. [Online]. Available: https://doi.org/10.1007/978-3-642-12651-2_3

[16] S. Greengard, "The future of data storage," April 2019. [Online]. Available: https://cacm.acm.org/magazines/2019/4/235573-the-future-of-data-storage/fulltext

[17] G. Falcao, V. Silva, L. Sousa, and J. Andrade, "Portable ldpc decoding on multicores using opencl," *IEEE Signal Processing Magazine*, vol. 29, pp. 81–109, 07 2012.

[18] E. Karami, S. Prasad, and M. Shehata, "Image matching using sift, surf, brief and orb: Performance comparison for distorted images," 11 2015.

[19] M. S. Nixon and A. S. Aguado, "Feature extraction and image processing. academic press," 2008.

[20] T. Lindeberg, "Scale Invariant Feature Transform," *Scholarpedia*, vol. 7, no. 5, p. 10491, 2012, revision #153939.

[21] ——, "Image matching using generalized scale-space interest points," *Journal of Mathematical Imaging and Vision*, vol. 52, no. 1, pp. 3–36, 2015. [Online]. Available: https://doi.org/10.1007/s10851-014-0541-0

[22] H. Fassold and J. Rosner, "A real-time gpu implementation of the sift algorithm for large-scale video analysis tasks," in *Proc. Real-time Image and Video Processing*, 2015.

[23] J. X. Zhou Yonglong, Mei Kuizhi and D. Peixiang, "Parallelization and optimization of sift on gpu using cuda," in *2013 IEEE 10th International Conference on High Performance Computing and Communications and 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC)*, November 2013, pp. 1351–1358.

[24] A. A. M. Ahmed Mehrez and E. E. Hemayed, "Speeding up spatiotemporal feature extraction using gpu," January 2018.

[25] J. Sloup, M. Perd'och, [U+FFFD] Obdržálek, and J. Matas, "Hessian interest points on gpu," 2016.

[26] C. Jiang, Z.-x. Geng, X.-f. Wei, and C. Shen, "Sift implementation based on gpu," 08 2013, p. 891304.

[27] Z. Yonglong, K. Mei, J. Xiang, and D. Peixiang, "Parallelization and optimization of sift on gpu using cuda," 11 2013, pp. 1351–1358.

[28] M. Björkman, N. Bergström, and D. Kragic, "Detecting, segmenting and tracking unknown objects using multi-label mrf inference," *Computer Vision and Image Understanding*, vol. 118, pp. 111 – 127, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S107731421300194X

[29] K. Mizuno, H. Noguchi, G. He, Y. Terachi, T. Kamino, H. Kawaguchi, and M. Yoshimoto, "Fast and low-memory-bandwidth architecture of sift descriptor generation with scalability on speed and accuracy for vga video," in *2010 International Conference on Field Programmable Logic and Applications*, Aug 2010, pp. 608–611.

[30] W. Pang, H. Huang, F. An, and H. Yu, "Low-power and real-time computer vision on-chip," in *2016 International SoC Design Conference (ISOCC)*, Oct 2016, pp. 43–44.

[31] L. Yao, H. Feng, Y. Zhu, Z. Jiang, D. Zhao, and W. Feng, "An architecture of optimised sift feature detection for an fpga implementation of an image matcher," 01 2010, pp. 30 – 37.

[32] D. C. Le, E. Y. Oh, J. H. Jeong, S. H. Kim, M. Jeon, J. Jang, and C. Youn, "An opencl-based sift accelerator for image features extraction on fpga in mobile edge computing environment," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, Oct 2018, pp. 1406–1410.

[33] G. Wang, B. Rister, and J. R. Cavallaro, "Workload analysis and efficient opencl-based implementation of sift algorithm on a smartphone," in *2013 IEEE Global Conference on Signal and Information Processing*, Dec 2013, pp. 759–762.

[34] "Ibm's server processors: The rs64 and the power," January 2011. [Online]. Available: http://www.cpushack.com/2011/01/24/ibms-server-processors-the-rs64-and-the-power/

[35] J. Clark, "Intel: Why a 1,000-core chip is feasible," August 2010. [Online]. Available: https://www.zdnet.com/article/intel-why-a-1000-core-chip-is-feasible/

[36] J. Hruska, "How l1 and l2 cpu caches work, and why they're an essential part of modern chips," August 2018. [Online]. Available: https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips

[37] T. Ozaki, T. Otsuka, and S. Shimizu, "3d-cg system with video texturing for personal computers," 1997.

[38] NVIDIA, "Nvidia turing gpu architecture," 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[39] NVIDIA Corporation, "NVIDIA CUDA C++ best practices guide," 2019, version 10.2.

[40] R. Mafi and S. Sirouspour, "Gpu-based acceleration of computations in nonlinear finite element deformation analysis," *International journal for numerical methods in biomedical engineering*, vol. 30, 03 2014.

[41] K. Chaudhury and S. Sanyal, "Improvements on "fast space-variant elliptical filtering using box splines"," *IEEE transactions on image processing : a publication of the IEEE Signal Processing Society*, vol. 21, September 2011.

[42] Z. Li, H. Jia, and Y. Zhang, “Hartsift: A high-accuracy and real-time sift based on gpu,” 12 2017.

[43] N. Fauzia, L. Pouchet, and P. Sadayappan, “Characterizing and enhancing global memory data coalescing on gpus,” in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2015, pp. 12–22.

[44] I. Rey Otero and M. Delbracio, “Anatomy of the SIFT Method,” *Image Processing On Line*, vol. 4, pp. 370–396, 2014.

[45] P. Kovesi, “Fast almost-gaussian filtering,” in *2010 International Conference on Digital Image Computing: Techniques and Applications*, Dec 2010, pp. 121–125.

[46] P. Ouyang, S. Yin, L. Liu, Y. Zhang, W. Zhao, and S. Wei, “A fast and power-efficient hardware architecture for visual feature detection in affine-sift,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 10, pp. 3362–3375, Oct 2018.