Efficient and Portable SIFT Algorithms Using High-Level Parallel Programming Models

Nikolai Johannessen

Thesis submitted for the degree of
Master in Informatics: Programming and System Architecture
60 credits

Institute for Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2022

Efficient and Portable SIFT Algorithms Using High-Level Parallel Programming Models

Nikolai Johannessen

Efficient and Portable SIFT Algorithms Using High-Level Parallel Programming Models

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

With the advent of powerful secondary accelerators and heterogeneous computer architectures, different ways to utilize these processing units for general-purpose computing were developed. While a secondary unit such as a GPU was initially designed to excel in a specific area, frameworks that expose functions for general-purpose use now existed. These programming models are commonly used to offload general computations to a secondary processing unit through parallelism. GPUs are especially proficient at computing multiple independent values in parallel. In an attempt to abstract upon these parallel programming models further, several high-level programming models now exist. This thesis presents and utilizes one such model, namely the Open Accelerator or OpenACC for short. This thesis evaluates the performance and productivity impacts of using OpenACC to parallelize algorithms, using the famous feature extraction and description algorithm SIFT as an example. Through profiling and analysis, this thesis demonstrates how portable solutions can be written with high speedup relative to the programming effort required. The results of this demonstration suggests, in conjunction with prior study and literature, that high-level parallel programming models are superior in certain aspects compared to more traditional lower-level models.

Sammendrag

Grunnet økende popularitet og beregningskraft på sekundære prosesseringsenheter slik som grafikkkort, har en rekke programmeringsmodeller blitt utviklet med hensikten å utvinne noe av denne kraften til mer generelle beregninger. Selv om et grafikkort originalt er designet til å tjene et bestemt formål, så gjør disse programmeringsmodellene det mulig å bruke lavere-nivå metoder for utvinne kortets kraft til mer generell komputasjon. Måten disse modellene effektivt bruker sekundære prosesseringsenheter i sammenheng med hovedprosesseringsenheten (CPU), er ved å utnytte avanasert tråd-programmering og parallelisme. Da de tidligere programmeringsmodellene ofte var mer lavnivå, er det nå blitt et større fokus på å designe programmeringmodeller som er lettere å forstå, og som fungerer på et høyere abstraksjonsnivå. Denne oppgaven presenterer og demonstrerer bruken av én slik model, nemlig Open Accelerator, også kalt OpenACC. Oppgaven evaluerer effekten på både ytelse og produktivitet ved å optimalisere en kjent algoritme som identifiserer og beskriver nøkkeltrekk i bilder som et eksempel. Denne algoritmen heter SIFT, og oppgaven vil demonstrere en parallelisert løsning basert på profilering og analyse av den original algoritmen. Oppgaven forsøker også å vise hvordan man kan skrive løsninger som både har høy ytelse, og er tilgjengelige for et mangfold av platformer samtidig. I kombinasjon med oppgavens resultater og tidligere forskning på området, merkes det en tendens til at høy-nivå programmeringsmodeller er bedre eller likeså bra på visse områder enn tradisjonelle lav-nivå modeller.

Acknowledgments

I would like to thank my supervisor Professor Carsten Griwodz for introducing me to the field of computer vision, and suggesting and supporting this thesis.

Additionally, i want to thank my parents for supporting and believing in me. I also want to thank my brother for initially encouraging me to pursue an academic degree in Informatics.

-Nikolai Johannessen

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

The process of feature extraction is a key component of most, if not all, image matching algorithms and software. In the field of computer vision, being able to efficiently extract and describe keypoints in images remains important. This extends to more practical cases as well, such as detecting diseased grapevines in vineyards [44], in unmanned autonomous vehicles[19], and in 3D reconstruction of images. The features we extract from an image should ideally be recognizable from multiple angles, be scale-invariant, and identifiable in most lighting environments. This is why Lowe's SIFT[25, 26] is still such a populare feature extractor and descriptor, with several variants[1, 32, 56] and competing algorithms[3, 18, 47]. Especially in recent years, utilizing the extremely parallel nature of Graphical Processing Units (GPU's) to effectively do feature extraction has been an important topic in the field of computer vision. Conversely, one such GPU implementation is popsift[12]. This is a faithful implementation of the original SIFT algorithm which achieves high speedup using NVIDIA's CUDA(Compute Unified Device Architecture) programming model. Another SIFT variant using CUDA is CudaSift[6]. However, my thesis will focus mostly on how parallel programming models may be used to accelerate algorithms, using SIFT,popsift and the CUDA framework as points of reference. In addition, I will present and discuss alternative parallel programming models which may provide higher degrees of portability than CUDA does. This is both to see the effect and performance penalty of using another framework than strictly

just CUDA, and to measure and quantify the impact a higher-level model has on productivity.

In association with the AliceVision project, found at https://alicevision.org/, popsift is a key part of the 3D image reconstruction that can be used to render virtual elements for Mixed Reality(MR). MR is the combination of real-world environments and virtual elements, and these virtual elements often act as an extension or an integrated part of the real world. One important thing to consider when rendering such virtual elements into the real-world environment is depth. Being able to compute depth quickly to form a seamless combination of a physical surrounding and some virtual element is one of the main goals of depth estimation in MR. Moreover, this process should work in real-time, as anything else will not result in a successful MR experience. This is where GPGPU and multi-core CPU parallelization techniques become important. I will explain what GPGPU is, as well as some concepts surround it, in section 2.3. While popsift generally achieves extraction of frames and descriptors of 1080p images in real-time with CUDA[12, section 6.2], I want to examine how other programming models perform in similar scenarios, and if it is possible to approach the same kind of speeds and speedup.

I will also demonstrate how to optimize sequential algorithms with a high-level programming model, using a sequential SIFT algorithm as an example. The alternative programming model I have chosen to use for my demonstration is OpenACC[41]. I will go into more depth on the details of this programming model in section 3.1. While considering alternative frameworks and APIs for my demonstrative implemenation, the OpenCL[1] framework was considered as well. However, the decision was made to use OpenACC, as it is a more interesting and high-level approach to offloading and accelerating applications than OpenCL is. CUDA and OpenCL share many of the same features and programming paradigms, and are therefore quite alike. On the other side, OpenACC favours ease of implementation over pure performance[30], except if very careful manual optimizations are done[17].

[1] https://www.khronos.org/opencl/

1.1 Thesis Goals

My thesis will attempt to answer the following two research questions

- RQ1: How much speedup is realistically possible when implementing an algorithm such as SIFT in a high-level programming model like OpenACC, while still maintaining portability?
- RQ2: What are the overall benefits of using a higher-abstraction programming paradigm in terms of ease-of-implementation and productivity?

RQ1 requires profiling and measurements of differing execution speeds in sequential and parallel implementations of the same algorithm, as well as some analysis of memory movement and management. Additionally, when I analyse parallelized solutions, portability should be prioritized over performance in most cases. Put simply, the optimized parallel implementation should not make too many platform-specific optimizations, as to not break the platform portability.

RQ2 will be answered by comparing the amount of code in a faithful sequential SIFT algorithm with how much code needs to be added or re-factored when parallelizing with the OpenACC programming model. Productivity and programming effort will also be analysed and presented through my own experience with porting and implementing the algorithm, as well as through study of prior articles and research.

1.2 Thesis Overview

Chapter 2 presents and explains some relevant background information on the SIFT algorithm, GPGPU and CUDA. Then, chapter 3 explains the OpenACC parallel programming model in detail. Chapters 4 and 5 deal with my actual demonstrative implementation, and the methods and tools i used to parallelize the code.

Chapter 2

Background and Relevant Literature

In this chapter I will first explain the SIFT algorithm and its different steps, then explore some key differences between CPU and GPU architecture. Afterwards I will present some concepts of general-purpose computing on graphical processing units (GPGPU) with examples using CUDA, while also explaining the CUDA programming model. Lastly, I will explain my reasoning for wanting to find an alternative framework suitable for parallel optimization other than CUDA.

2.1 The SIFT algorithm

The Scale-Invariant-Feature-Transform algorithm, or SIFT for short, is an algorithm originally conceived by David G. Lowe in a 1999 conference paper[26], and later expanded upon in a scientific article [25]. The SIFT algorithm handles an extremely important step of most computer vision and image processing software, namely feature description and extraction. This process can be explained concisely as identifying points of interest in an image (keypoints) and then assigning them unique signatures. These unique signatures are then stored as SIFT feature descriptors, which are commonly used to recognize the same points of interest in another image. SIFT aims to be invariant against brightness, rotation, and scale. Ideally, SIFT will always recognize the same features

if they are present in two images, no matter how far away, what the lighting conditions are, or how different the roation is. These qualities are what makes SIFT such a good feature extractor and descriptor for image matching

Because of its robustness, and due to not being too computationally heavy, especially with more modern and improved versions, SIFT sees use in many different disciplines. Some examples include 3D reconstruction software[13] with AliceVision, face recognition and authentication[5, 27], object tracking[58], image processing[19, 24, 44], and even to detect buildings and urban areas from satelite pictures[51]. The widespread use of SIFT might also explain its ever-growing number of variations and alternatives, and in our case, SIFT algorithms which exploit the GPU[6, 12, 16].

In order to understand how to optimize the SIFT algorithm for parallel computing, knowing its steps is important. The goal of my demonstration with OpenACC will be to optimize the steps with the largest bottlenecks, and we therefore need an understanding of where they may occur.I will put less emphasis on the specific maths and computations which go into these steps, as the important part is mostly to see how a parallel programming model may affect the large-scale execution speeds and memory efficiency when implemented. Figure 2.1 also illustrates the process, somewhat simplifying the process of finding dominant orientations, as well as feature description.

2.1.1 Upscaling

The first step is to upscale the input image. This upscaling is traditionally performed through bilinear interpolation [1] with a scale factor of 2. The upscaled image will be two times larger both in width and height. This step is to ensure that we make full use of the input image, as it increases the number of stable keypoints[25, p. 10]. Additionally, it allows for sub-pixelic refinement of the original image.

[1] https://en.wikipedia.org/wiki/Bilinear_interpolation

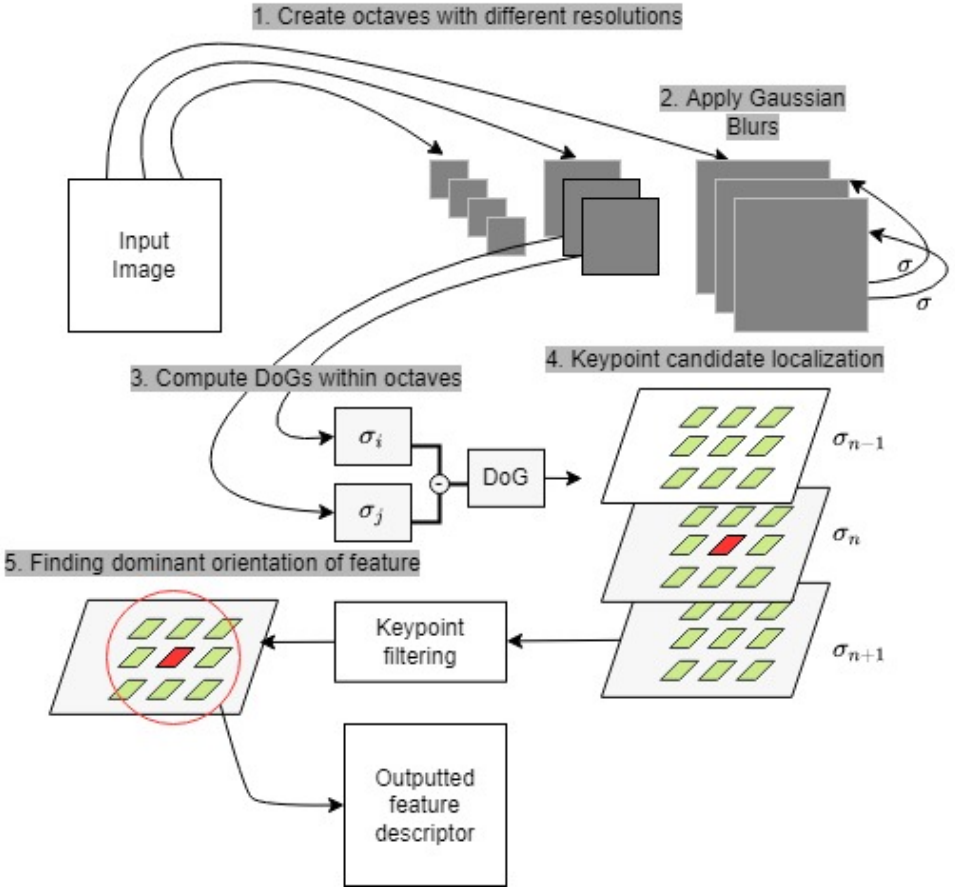

Figure 2.1: A simplified illustration of the SIFT algorithm

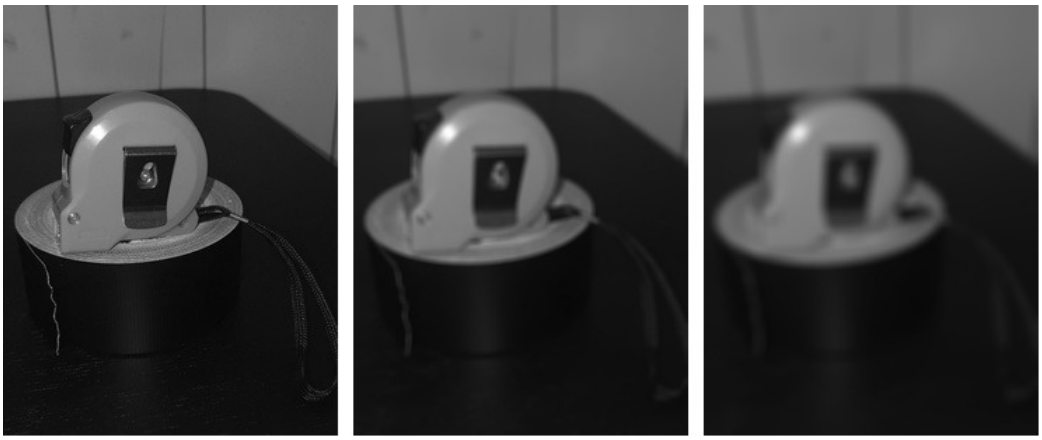

Figure 2.2: Gaussian blur illustrated. The original image, image blurred with $\sigma = 5$, and image blurred with $\sigma = 11$

2.1.2 Creating a Gaussian Pyramid

Gaussian Blur

A common tool in the field of computer vision and image processing, is the Gaussian blur. Such a blur is commonly achieved by using a Gaussian function to comb over an image, and apply some transformation to each individual pixel. This transformation usually consists of changing the value or color of a pixel to that of a weighted average of its eight neighbours. The result is an image which looks blurred and less detailed, see Figure 2.2. In the SIFT algorithm, a Gaussian blur is applied in order to make the detection and extraction of keypoints scale-invariant. The blur attempts to simulate how an image would look if you zoomed further and further out. Logically, it follows that if an image feature is identifiable and visible even through several Gaussian convolutions, it must be a stable feature.

Octaves and Levels

After the image has been upscaled, the algorithm will start creating groups of increasingly blurred images with the same resolution. These groups are referred to as octaves. Within an octave, the image is blurred using a Gaussian convolution with a blur factor σ. This blurring occurs several times, and the differently-blurred images within an octave are referred to as levels, as they represent different levels of Gaussian blur.

The first octave has the same resolution as the upscaled input image, and every subsequent octave is downscaled from the third-last level of the previous octave, halving its resolution. This process of halving the resolution is repeated until the image is too small to be useful. This collection of octaves with levels is called a *scale space* [23]. The blur factor σ,the amount of octaves, and the depth of levels may depend on the implementation, as well as whatever parameters will yield the best results for specific image resolutions.

2.1.3 Computing the Difference-of-Gaussian (DoG)

In this step, we want to determine how much an image has changed from one blur level to another, and more importantly where those changes occur. The most accurate way of determining the per-pixel differences between two levels would be to use a Laplace-of-Gaussian, as proposed by Tony Lindeberg [23]. However, Lowe [25] proposes a more computationally efficient approximation of the Laplacian interest points; the Difference-of-Gaussian method. This method retains the scale invariance, as well as the rotational invariance of the original Laplacian gradient method.

Within our scale space, each DoG is computed with the following formula:

$$DoG(x, y, \sigma) = L(x, y, k_i\sigma) - L(x, y, k_j\sigma) \tag{2.1}$$

Formula 2.1 states that the Difference-of-Gaussian images are produced by subtracting the image L with a blur level j from the image L with a blur level I. These images are adjacent in the scale space, being images of the same resolution, but with different blur levels.

2.1.4 Detecting keypoints

After all the octaves and their DoGs have been computed, the algorithm starts looking for potential keypoint candidates. Every pixel in the DoGs is processed by the algorithm, and potential keypoints are found

by looking at pixels which are an extremum in its neighbourhood. An extremum is either an absolute minimum or a maximum. At this point in the algorithm, adjacent DoG's are layered on top of each other to create a 3D space, with the axes being x,y and σ. Therefore a pixel has 26 neighbours, and if a particular pixel is the extremum of its neighbourhood, a keypoint search close to it is initiated. The result of the search will be a keypoint candidate, which may be assigned SIFT Feature Descriptors if it qualifies as a keypoint. Whether the candidate qualifies as a keypoint or not is determined by requiring it to pass certain tests.

2.1.5 Computing the keypoint orientations

After identifying keypoints, the dominant orientation of the feature or features present at the extrememum must be determined. In simple terms, which way the feature is pointing in the image. This is to make the descriptors invariant to image orientation. This orientation is represented as a SIFT feature descriptor, and a keypoint may have multiple descriptors with different dominant orientations. The dominant orientation is decided by seeing in which 2D-direction the strongest luminance/brightness change occurs.

2.1.6 Extracting descriptors

As previously mentioned, a keypoint may have several dominant orientation, and therefore several SIFT descriptors attached to it. These descriptors function like a unique identifier for a feature, and they are composed of 128-element vectors, with either decimal or integer values, depending on the implementation and further image matching functions which take the descriptors as input. The next-to-last step in feature descriptor extraction is to calculate the orientation histograms. These are grouped around the keypoint as a 16 square grid, with each square having 8 vectors describing gradient change.

2.1.7 Normalizing descriptors

In the standard SIFT algorithm, the descriptor vectors are normalzied through the L2 norm to improve image matching with the descriptors.

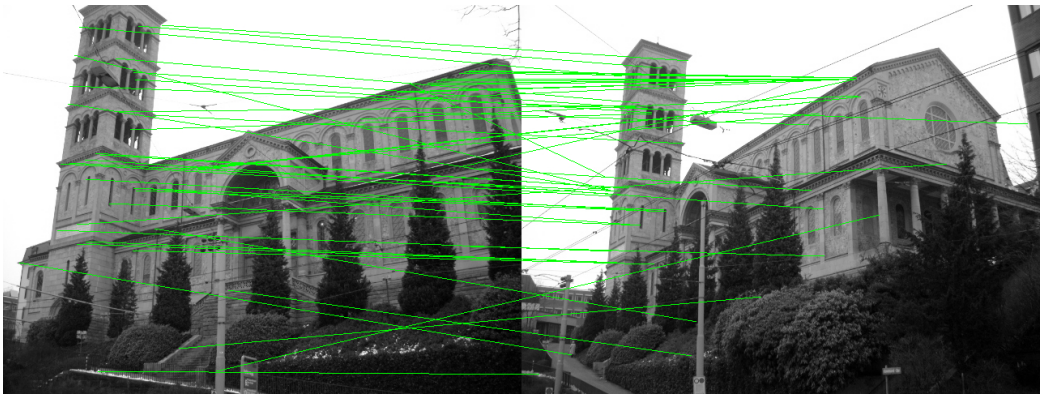

Figure 2.3: SIFT Image matching illustrated. Photos with different angles from the Zurich dataset

The L2 norm value is a common operation when dealing with vector computations, and is in short a value which describes the shortest distance from the space origin to the vector. The value is defined as the root of the sum of the squares of the components of the vector. Popsift also implements a newer form of proposed normalization called RootSIFT[2].

2.1.8 SIFT In Practice: Extracting and Matching

The GPU-implementation of SIFT known as popsift is a key part of a larger software known as Meshroom, a 3D reconstruction software based on the AliceVision framework. In this software, SIFT is responsible for both feature extraction,description, as well as a part of actual image matching. In the case of meshroom, images are matched together in order to create a 3D texture based on the images provided. An example of feature matching with sift can be seen in figure 2.3.

2.2 GPU Architecture

Before I present the core concepts and operations of general-purpose programming on GPU, I would like to give an overview of GPU architecture in general. I will be using Nvidia's Pascal[37] as an example. I have chosen Pascal as an example because it is relatively new, and it is the architecture my demonstrative SIFT optimizations are primarily tested on. Currently, the Pascal architecture has been succeeded by the

Turing[39], Volta[38] and Ampere[35] architectures.

2.2.1 Latency and Throughput

Regarding GPU architecture, the most important thing to remember is that GPUs are made with throughput in mind, moreso than latency. While a CPU might typically only have a core count below 20, GPUs often have four- or three-digit amount of cores. Naturally, one could assume that core count is the most important part when considering performance, but this is not always accurate. In truth, the internal architecture of the cores, as well as how those cores are arranged matter just as much. However, it is true that the high core count in a GPU allows for a high degree of parallelization, which can be much better performance-wise if utilized correctly.

CPU cores traditionally focus more on low-latency memory access, and are best suited for sequential compute-heavy processes. On the other hand, GPU's are made to put all of its cores to use simultaneously, favouring parallel code. The main difference in latency lies in how data is cached, and how large those caches are. CPU's usually have larger and faster caches near the cores, and if data is not contained in those caches, it will be fetched from L3 caches, or from RAM. GPU's operate with smaller caches for each cluster of cores commonly called Streaming Multiprocessors, and a larger cache for shared data. As illustrated by figure 2.4, Pascal has 20 SM's and a large L2 Cache in the middle. Additionally, 2.4 shows how many cores the architecture truly has, with every green square representing one core, each SM having 128 cores. Additionally, instead of each core having its own L1 cache, each SM has two Texture/L1 Caches, colored light-blue in the figure. Each SM also contains 96KB of shared memory.

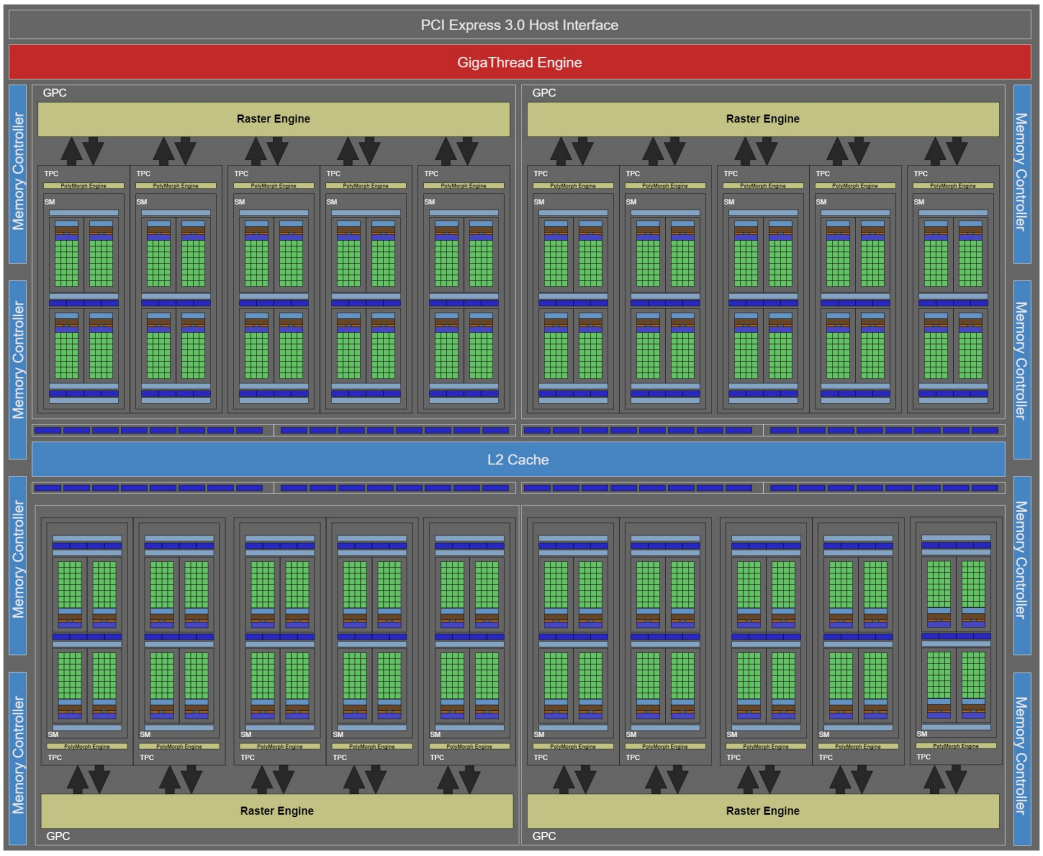

Figure 2.4: A block diagram of the Pascal Architecture, specifically the GP104 structure. Based on an NVIDIA model

2.3 GPGPU

2.3.1 Core concepts: GPU vs CPU

General Philosophy

General-purpose programming on a GPU differs greatly from traditional single-core CPU programming, or even multi-core CPU programming. Concepts from multi-threaded CPU programming are applicable, but the scale of the parallelization itself is much larger, with a larger number of threads or streams working simultaneously. Additionally, GPGPU usually requires a good knowledge of low-level memory structures and memory operations. Even CUDA, the largest GPGPU API requires the programmer to have this deep knowledge. However, attempts[4, 14, 45], have been made to further abstract upon the CUDA standard, making it more high-level, without sacrificing performance. Thrust[4] is an example of this, and is also used as an optional feature in popsift. Similarly a paper by Han&Abdelrahman[14] attempts to abstract the memory instructions of CUDA to a directive-based model, not unlike OpenACC[41]

Textures vs Arrays

Textures as the primary representation of data is a core concept of GPGPU. While the native data layout of CPUs are one-dimensional arrays, GPUs use two-dimensional arrays called textures instead. Textures are used as the most basic form of data storage, and are the most efficient way for the GPU to manipulate data. They essentially replace traditional CPU arrays. While accessing array data on a CPU is often done using array indices, on the GPU, data is accessed through *texture coordinates*. These are 2-dimensional coordinates which indicates where a texture pixel, or texel, is located on a texture map. The dimensions of a texture are typically restricted to maximum values depending on the GPU chip. Typical values are 2048 or 4096 per dimension.

Kernels vs Loops

Loops are a very common tool for data manipulation on the CPU. A standard CPU implementation of a problem might initiate a single loop to modify the values of an array, even if those values were completely independent of each other. When a loop is initiated, there also has to be a loop counter that continually updates as the data is being processed, and is then used in each loop to access the array at the appropriate indices. However, this is technically inefficient, and unrolling loops is a common optimization trick. This is done to reduce loop-initialization overhead and unnecessary memory operations. Now, in these situations where the array elements are at distinct memory locations, and there exists no dependencies between elements, GPU kernels are usually much more effective.

A kernel in GPGPU represents a block of code that will run in parallel on multiple processing cores. Kernels are a staple feature of GPU parallel processing, and is the most important tool for a GPGPU programmer. In short, kernels are the primary way of executing code on a GPU, but has some key restrictions when compared to thread-programming on the CPU. Therefore, this feature is usually best utilized in SIMD (single instruction, multiple data) scenarios. The larger the amount of data, e.g. the amount of parallel computations available, the more efficient a GPU kernel will be relative to a CPU implementation of single- or nested loops. Oftentimes, the performance will be better than a thread-based CPU approach as well. However, writing efficient kernels is usually much harder than writing efficient threads, due to the massively parallelizable nature of GPU threads.

Precise number calculation

Another key difference between CPU and GPU is the way precise many-decimal numbers are calculated. While CUDA strives to produce exactly the same results as a CPU calculation for basic operations, there will still be some small differences due to a variety of reasons which are hard to control. This becomes especially relevant when comparing results between CPU and GPGPU implementations of the same algorithm. Due to the difference in precision, it is important to maintain a certain margin

of error for precise decimal values.

2.3.2 History and Hardware Utilization

In the earlier days of computation, specialized processing units aside from the CPU were not as prevalent. Display adapters and graphics accelerators did exist, but were nowhere near powerful enough. Therefore, a lot of programs and programmers today often only utilize the CPU efficiently, while somewhat neglecting GPU use. However, with the advancement of both integrated graphics, as well as bigger graphical processing units such as NVIDIA or AMD graphics cards, there now existed another powerful processor in many computers. This lead to the creation of general-purpose programming languages focused on accelerators with high potential for memory sharing and parallelization, such as OpenCL[34], OpenMP[8] and CUDA. GPGPU also sees a high degree of use in HPC situations, where the amount of data points often reach staggeringly high numbers.

There are a many reasons why we would want to do general computations on the GPU at all, and one of them is hardware utilization. Logically, it would be wasteful to not utilize every part of your machine if possible. If most software left powerful secondary processing units completely unused, then that clearly diminishes their usefulness. Essentially, it is a matter of how much value you get out of each part of your machine. It would probably be beneficial for both consumers and manufacturers if their GPU's were utilized more in general, as it would feel like that part of your machine is not at any moment a waste.

Of course, we wouldn't want to over-utilize the GPU for general-purpose instructions either, hindering its usefulness in graphics and user interface rendering. Assuredly, a middle ground between almost zero utilization and some utilization does exist. While it might seem wasteful to not utilize GPUs more often, general purpose processing on GPU is still a relatively new concept, with both CUDA and OpenCL having their initial releases in 2007 and 2009 respectively. Additionally, the circumstances where the GPU will be more efficient are less common.

2.3.3 GPU In Tandem With CPU

Heterogeneous architecture also allows for your main processing unit to be supported by the secondary accelerator. Thus reducing the load on the CPU, and ideally allowing for parallel processing with both units. Alternatively, coordinating which unit will be the most optimal for a certain problem[52] is also a valid option. Another interesting idea would be to somehow fuse GPU and CPU architecture into a homogeneous unit on the same chip[57]. While Yang et al. proves that this approach provides significant speedup (up to 113%), it is mostly a proof-of-concept, and is unlikely to see popularity.

If a programmer recognizes that a certain problem could be split into an amount of sub-problems far outweighing the amount of threads a parallel CPU solution would use, that problem might be suitable for the GPU instead. Of course, recognizing such a problem is not always easy, and requires deep knowledge of the problem at hand. However, if it is recognized as such, the programmer might turn to a parallel general-purpose computing platform such as CUDA. Depending on the specifications of the processing units, the performance gain itself could also be large. Additionally, the program now has the option to let the CPU work on some other problem instead, while the GPU does its own work. An instance where this might be particularly useful, is in programs where I/O operations are done frequently. For example, the CPU could read something which it sends to the GPU for processing, and then immediately read the next data, all while the GPU is processing. This is an example of effectively utilizing both the CPU and the GPU at the same time.

2.3.4 Performance

Lastly, in some cases, a good GPU implementation with a high-end unit simply outperforms a traditional CPU implementation. As is the case with popsift[12] and other GPU implementations[6, 7, 55], they are often many times faster than CPU implementations done on similarly high-end CPUs. Regarding the SIFT algorithm, the speedup is due to the nature of per-pixel image processing. Generally, the larger the problem, the easier

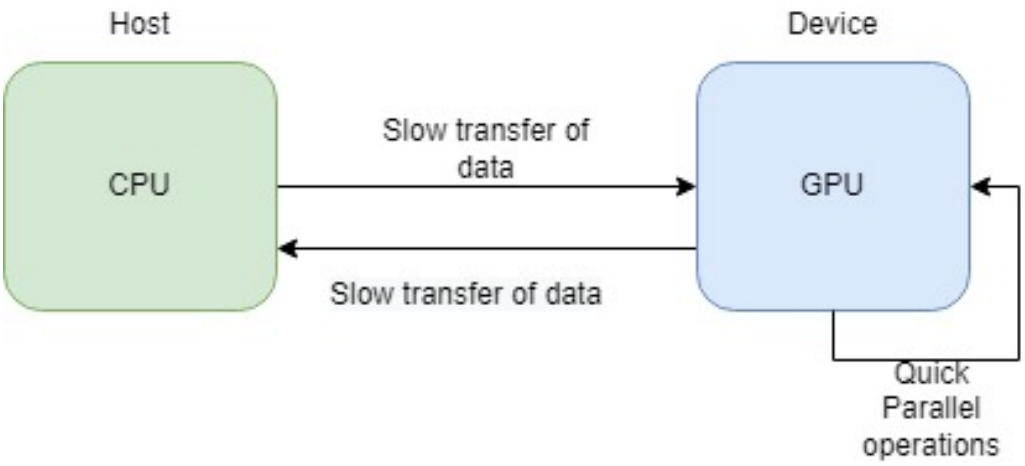

Figure 2.5: A simple diagram showing how host and device send data between each other

it is for the GPU to be more efficient. Consequentially, image processing and feature extraction problems are faster on the GPU, simply because the amount of calculations that need to be done is immensely high.

Additionally, as illustrated in figure 2.5, while the parallel GPU operations themselves are quite fast, transferring data between host and device is not. Therefore, when considering how to attain maximum performance with GPGPU, the programmer must have a deep knowledge of the memory spaces and hierarchies which exist in the different architectures. The most important thing to consider is data movement between host and device, re-using memory structures where possible, and generally avoiding unnecessary data movements.

2.4 CUDA For General-purpose Programming

One of the most widely used platforms for general-purpose computing on GPUs is CUDA[36]. This programming model and computing platform is developed and maintained by NVIDIA, and is targeted specifically for their own processing units. CUDA provides extensions to C, C++ and Fortran, although the list of supported existing languages is expanding. The reason for CUDA being so dominant on the GPGPU front, is attributed to the fact that they are the leading developer and manufacturer of graphics cards, and have been for the last years.

Combine this with the fact that they poured resources into developing the CUDA platform, and it suddenly becomes the biggest alternative for computing on the GPU. If what you are looking for is the absolute best performing GPU computing on an NVIDIA graphics card, then CUDA is your go-to platform. As I alluded to earlier, a good GPU implementation will sometimes strictly be better, and this is often the case with a well-implemented CUDA solution.

2.4.1 HPC and Cross-Discipline

GPU computing and CUDA in particular has a lot of different applications, mostly in high-performance computing(HPC) situations, as well as many different scientific disciplines. According to NVIDIA themselves[2], CUDA is utilized in a wide range of domains. For example computational chemistry, bioinformatics, data science, computational fluid dynamics, as well as weather and climate predictions. With CUDA already having such a large reportoir of disciplines, and applications for these fields already developed or in development, it looks like CUDA is here to stay. Unless a competing GPU manufacturer or some different group comes up with a platform which is either even easier to use than CUDA, or has better performance, there is no reason to believe that CUDA will not continue to be popular. Eventually, as an increasing number of companies and researchers utilize CUDA for their high-performance computing, even if a realistic competitor arose, it would probably take a lot of effort to refactor and re-implement existing CUDA code. Essentially, the longer CUDA is popular, the harder it will be for another GPGPU to break through to the mainstream.

2.4.2 CUDA's Design

Regarding ease of implementation and overall readability, CUDA strikes a nice balance between readability, abstraction, and implementation. It is quite readable, and does not require a tremendous amount of effort to understand and implement. One of the key features of CUDA is its ability to launch and execute CUDA kernels. Kernels, as mentioned

[2]https://www.nvidia.com/en-us/gpu-accelerated-applications/

previously, are an essential part of general-purpose GPU programming. The reason being that it is here the parallel code itself is written. Code written in a kernel is executed in parallel by blocks containing threads. More specifically, when launching a kernel in CUDA, you must specify the amount of blocks to use, and how many threads you want to utilize within that block. The platform also allows you direct access to memory allocation and copying between the GPU and CPU explicitly. In newer CUDA releases, the Unified Memory [3] feature is also available, which allows for some more flexibility regarding memory transfers. This is achieved by making a shared pool of memory between the GPU and CPU, which data may be allocated to and accessed by both units. I will talk a bit more about Unified Memory throughout this thesis when relevant.

2.4.3 CUDA Kernels

Using the examples from the official NVIDIA CUDA C programming guide[4], I will explain some concepts regarding kernels in CUDA. In CUDA, a kernel can be written as such:

```
// 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 N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Listing 2.1: CUDA kernel definition

In the above snippet, the function "VecAdd" is a CUDA kernel, signified

[3] https://developer.nvidia.com/blog/maximizing-unified-memory-performance-cuda/

[4] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

by the keyword "__global__". This function now acts as the block of code which will be executed in every specified thread of the GPU. Invoking the kernel and starting threads is done through CUDA's specific execution syntax on line 12. A CUDA kernel invocation is essentially just calling the function with parameters, and adding the triple angle brackets before the parameters. The first angle-bracket parameter is the amount of blocks per grid, while the second parameter is the amount of threads each block has. Both values can be either a *int* value, or a *dim3* value. The amount of blocks and threads per block can be assigned 1-, 2- or 3-dimensional values as well. Depending on the problem at hand, it may be beneficial to not use the same number of dimensions for both parameters.

2.4.4 Memory Hierarchy

CUDA operates with several different types of memory, most of them specialized to serve a specific purpose. Accessing and storing data in the correct memory space is an important part of making efficient CUDA code. The different types of memory are as follows:

- **Registers**. Thread-specific and extremely fast memory close to the cores.
- **Shared Memory**. Can be used for fast communication between threads in the same block.
- **Texture Memory**. Utilizes the texture engine of the GPU, allowing for fast parallel row and column accesses.
- **Constant Memory**. Used to store constant values.
- **Local Memory**. Local memory is a part of every SM, and is used when registers start spilling data. Local memory is Very slow compared to the shared memory and registers, and should be avoided.
- **Global Memory**. The most general-purpose of the memory spaces, mostly used for intermittent transfers from host to device if direct writing is not available, i.e for shared memory, local memory and the registers. Host-data is then transferred from the global memory

to a more appropriate memory-space.

2.5 CUDA Drawbacks

2.5.1 Vendor Lock

If CUDA is seemingly so efficient, easy to implement and widely used, why would we want to look for an alternative to it? Well, CUDA has one massive drawback. It is entirely vendor-locked behind having an NVIDIA GPU chip with CUDA support. While it is true that CUDA offers a good combination of performance and readability compared to other frameworks, that would not matter if your GPU simply cannot run CUDA programs. In the example of popsift, porting it to another programming model would be beneficial, and strengthen its overall outreach and portability. Having a SIFT implementation that only supports one type of secondary high-performance accelerator is not optimal. Therefore, the optimal solution would be to enable users without CUDA-capable accelerators to use this SIFT solution, without having to sacrifice any efficiency for users already using the CUDA implementation.

According to JPR 's market research report from 2020[43], NVIDIA's market share on the discrete GPU market was about 80% in the second quarter of 2020. In the report, the market shares of the three main GPU manufacturers are displayed. While Intel here is reported to have a 0% market share, the other graphic in the report states that Intel has the majority of the market share on the GPU market overall. However, this is not especially relevant for us, as this share is largely carried by Intel's manufacturing of integrated graphics units, which are usually no where near powerful enough for efficient GPGPU. We are much more interested in discrete GPUs, as they are many times more powerful. As illustrated by the figure in the article, even though Nvidia controls the market with 80% market share, there is still a large portion of users with powerful (and not powerful) discrete AMD GPUs.

2.5.2 Ease of Implementation

While I previously praised the relative ease of implementation and readability of CUDA, when compared to other programming models, CUDA is quite low-level and difficult to grasp. There are opposing views on whether or not higher-level abstraction paradigms such as Thrust actually provide users with an easier to learn model however, at least relative to the maximum performance they can achieve. Daleiden et al.[9] asserts that based on their results with the specific task they gave students, higher-level abstraction paradigms are actually harder to learn. Of course, the validity of that claim is to be disputed, and only true in the case of Thrust vs CUDA.

Contrary to the claim that higher-level GPU programming models are harder to learn, Li et al.[22] suggests that OpenACC is easier to learn, and a viable parallel solution was produced faster with OpenACC than with CUDA. However, OpenACC solutions were always slower performance-wise, with CUDA solutions having 9x shorter execution times[22, p. 13]. In another article[30], multiple GPU programming standards are compared to each other in terms of performance, producitvity and more. Here, they conclude that the programming effort required to implement certain things in OpenCL is higher than both CUDA and OpenACC.

Altogether, these few examples show that there are simply a lot of factors at play, both in terms of performance and productivity. However, it seems plausible that OpenACC is the programming model with the gentlest learning curve and lowest programming effort required, although at the cost of performance.

2.6 Parallel Programming On CPU

Achieving the same kind of speeds as a good GPU implementation with desktop CPUs would be difficult, and probably impossible. GPU's are very well suited for exactly such a problem as extraction and description of image keypoints. This task is so massively parallelizable, making it hard to think of a multi-threaded CPU implementation that would be faster, while also remaining faithful to the SIFT algorithm.

2.6.1 Manycore CPUs

A way to technically make parallel CPU solutions perform at the same level as parallel GPU solutions would be through the use of manycore CPUS as a secondary accelerator. The term "manycore" usually refers to CPUs with core counts in the double digits, but the definition may also include supercomputers with several thousand cores as well. An example of a manycore CPU fit for explicit parallelism are the CPUs in Intel's Xeon Phi [5] series. These CPUs usually have about 60-70 cores, depending on the specific series model.

[5]https://en.wikipedia.org/wiki/Xeon_Phi

Chapter 3

Framework and Optimization

In this chapter I will present the OpenACC programming model, which is used as the programming model for my demonstration of parallelism with SIFT, as well as the most important part of this thesis's goal. The goal being to evaluate the performance penalty of porting CUDA or sequential code to OpenACC, while also analysing its impact on productivity and readability. First, I will explain the model and its functionality, as well as how to utilize it efficiently. Then, I will compare it to some other GPGPU frameworks and models, in order to explain how OpenACC differs from them, and what advantages it offers. These advantages might take the form of pure performance gains, easier implementation, increased readability or increased portability. In this chapter, i will be evaluating the frameworks mostly based on prior research, while my own findings will be included more in Chapter 4.

3.1 OpenACC

OpenACC is a directive-based and portable parallel programming model, developed by Cray, CAPS, NVIDIA and PGI. As I have mentioned earlier, OpenACC works by annotating C, C++ or Fortran code with simple directives to initiate massively parallel instructions. This is either done on the GPU, or a multi-core CPU. This makes the code highly portable, seeing use on many popular vendors's GPUs, as well as CPUs. The fact that only simple directives or annotations are needed,

makes the model a very high-abstraction programming platform.

Compared to other models, OpenACC has convincing performance-to-effort ratios[54] and offers increased programmer productivity[15]. However, the actual performance portability across platforms[48] might be overstated. Brodman&Tu proposes that without platform-specific adjustments, achieving similar performances across platforms with OpenACC is difficult. However, their analyses were made using an older compiler called OpenARC[21].

Similar to CUDA, OpenACC operates with a host (typically the CPU) and a device (typically the GPU). In OpenACC, the device is commonly referred to as the accelerator. Figure 3.1 illustrates the OpenACC's Abstract Accelerator Model. Although represented as two separate entities in the model, this is simply an abstraction in order to increase portability and support. In reality, the accelerator and host may be the same physical unit. Whether they share memory spaces or not, OpenACC will treat the host and accelerator with this abstract separation in mind. In addition, if they are in fact separate architectures with separate memory spaces, OpenACC will automatically handle most memory management and data transfers as well. However, when it comes down to actual programming with OpenACC, one should treat variables as existing in the same memory space.

To explain how to use OpenACC, I will mostly refer to and comment on the official "OpenACC Programming and Best Practices Guide"[40].

3.1.1 Analysis

The most important step is almost always pre-analysis and profiling of the program you want to port to OpenACC. Simple or extensive profiling through tools such as the NVIDIA Nsight Systems or gprof will reveal if your problem should be ported or not. Additionally, such profiling tools are important to use throughout the porting process as well, in order to handle memory properly, and to gauge how many parallel threads should be used.

As I've previosly mentioned, GPU solutions are only better in certain specific cases. Oftentimes you're better off just using a standard CPU

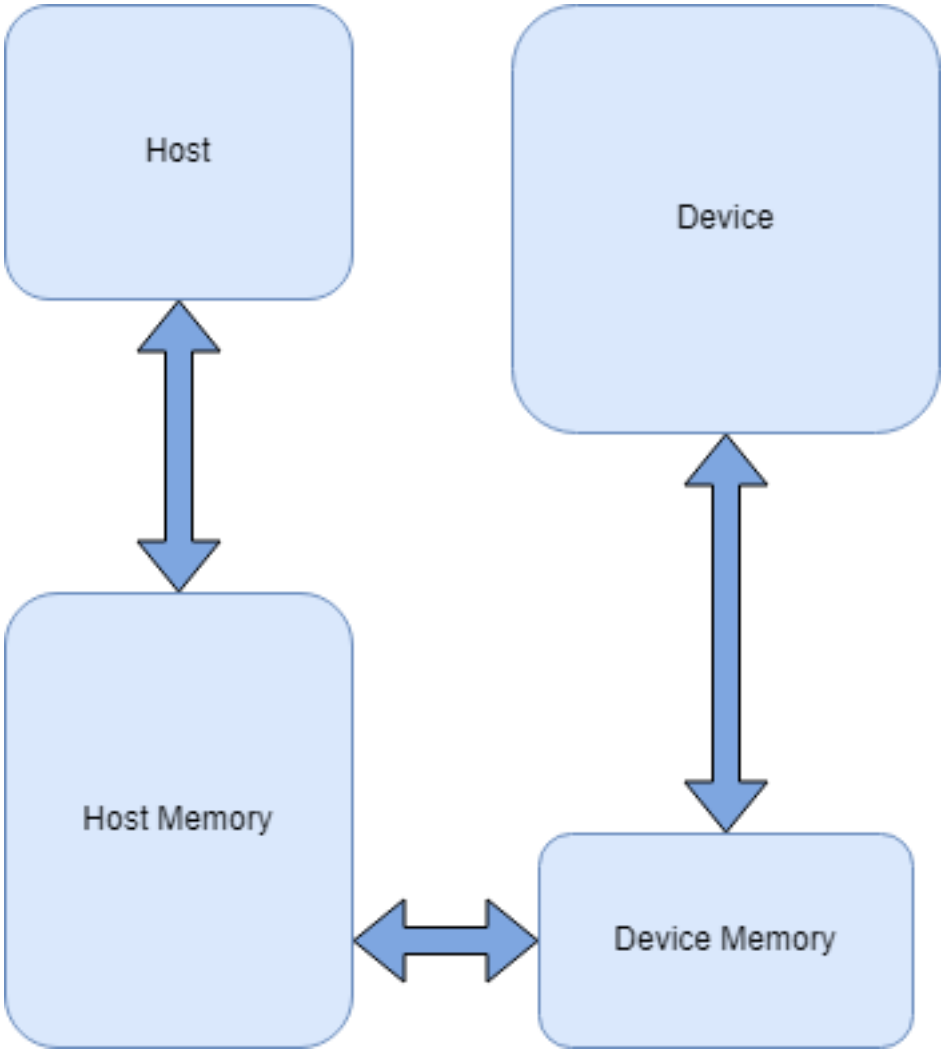

Figure 3.1: OpenACC's Abstract Accelerator Model. Based on the model in the OpenACC Programming Guide

implementation. However, this is the reason why analysis is important. For example, if you are working with a large piece of software with multiple different data-processing scenarios, only some of those scenarios may be suitable for parallel execution on an accelerator. Recognizing situations where massive parallelization might result in higher performance is a paramount skill to have as a GPGPU programmer.

3.1.2 Directives

At the core of the OpenACC programming model is the directive syntax. I will be presenting the directive syntax in C/C++, as it is the language my demonstration is written in. Keep in mind that OpenACC supports directives in Fortran as well. In C and C++, directives take the form of a pragma, commonly used to provide hints to the compiler. In that sense, OpenACC can mostly be interpreted as a set of advanced compiler hints.

```
1 #pragma acc kernels
```

Listing 3.1: OpenACC pragma

The above example is a directive in OpenACC. First, the #pragma to indicate a compiler hint, then the sentinal *acc* which signals that the next part is acc code. Lastly, the main part of the directive, the *construct*, in this case *kernels*. The construct may also be followed by additional clauses to the same construct, or a different construct.

The *kernels* construct signifies to the compiler that the following block of code, specified by curly brackets {}, should be analyzed for parallelization by the compiler, and possibly offloaded to the accelerator. However, this directive relies heavily on the compiler to correctly analyse and automatically parallelize the block of code. It is up to the programmer to determine if this code region will benefit from pallelization or not. An example of the *kernels* construct may look like this:

```
1 #pragma acc kernels
2 {
3     for (i=0; i<N; i++)
4     {
```

```
5       y[i] = 0.0f;
6       x[i] = (float)(i+1);
7    }
8
9    for (i=0; i<N; i++)
10   {
11      y[i] = 2.0f * x[i] + y[i];
12   }
13}
```

Listing 3.2: Kernels directive

Another construct is *parallel*. This construct does not practically do that much by itself, and is commonly paired with other constructs, such as the *loop* construct. Parallel signifies that the specified region of code will be parallelized across OpenACC gangs. This terminology is unique to OpenACC, but a gang is essentially a collection of workers that do vector computations in parallel, while being able to share some data with each other.

While the *kernels* construct can be placed before a large block of code containting multiple loops, the combined *parallel loop* construct must be placed directly in front of the loop that should be parallelized. The difference between the *parallel* and *kernels* directives might not be readily apparent, but the former gives maximum control over to the compiler, while the latter is an assertion by the programmer that the specified loop is safe and desirable to parallelize.

```
1 #pragma acc parallel loop
2 for (i=0; i<N; i++)
3 {
4    y[i] = 0.0f;
5    x[i] = (float)(i+1);
6 }
7
8 #pragma acc parallel loop
9 for (i=0; i<N; i++)
10 {
11    y[i] = 2.0f * x[i] + y[i];
12 }
```

Listing 3.3: Parallel loop directive

As mentioned earlier, constructs may have clauses which provide additional instructions. For the *loop* construct, clauses are used either to ensure correctness, or to optimize the loop. I will not go into these clauses in detail, but they will be explained as they are used in my demonstration.

Routine Directive

In most programs it is necessary to call functions or subroutines from within regions which are parallelized by acc. However, the compiler is not able to recognize loops or other parallelizable regions within that function. To address this, functions which are to be called from within parallelized loops must have a directive with the *routine* construct decorating it:

```
1  class float3 {
2     public:
3      float x,y,z;
4
5     #pragma acc routine seq
6     void set(const float3 *f) {
7      x=f->x;
8      y=f->y;
9      z=f->z;
10    }
11
12 };
```

Listing 3.4: Routine directive

Now, the set function will be callable from within a parallel region of code. Additionally, the function is declared a *seq* through the directive, signifying that it is a sequential function.

Atomic Operations

In order to ensure correctness when an element in memory is accessed across multiple parallel loops, it is important to avoid races whenever possible. A race condition may occur if that specific value is modified by one instance of the loop, while another instance is simultaneously trying to access it. This leads to unpredictable and undefined behaviour, with

results being different upon each execution of the program. Needless to say, this is undesirable and should be avoided. In some cases, using the aforementioned loop directives for correctness will solve this issue, but another slightly more complex way to do so is by using an *atomic* directive. The *atomic* directive may be followed by one of the following four clauses:

- **Read**: ensures that no two loop iterations will read from the region at the same time.
- **Write**: ensures that no two loop iterations will write to the region at the same time.
- **Update**: a combination of read and write.
- **Capture**: performs an update, and also saves the value calculated in that region to be used in the following code.

If neither clause is given, the directive defaults to the **update** clause. Example:

```
#pragma acc parallel loop
for(int i=0;i<HN;i++)
  h[i]=0;

#pragma acc parallel loop
for(int i=0;i<N;i++) {
  #pragma acc atomic update
  h[a[i]]+=1;
}
```

Listing 3.5: Atomic update directive

With the directives and clauses I have mentioned so far, it is possible to parallelize most problems. However, a common occurrence is that the program now actually takes longer to execute. The reason for this might simply be that the problem is not a good fit for massive parallelization, but the primary reason is likely overhead due to data movement. Put simply, the program spends more time moving data between host and accelerator than it does actually solving the problem. In order to minimize this effect, it is important to give additional information to the compiler. This information also comes in the form of directives, the

specifics of which will be discussed in the next section.

3.1.3 Data Locality

In order for the compiler to be safe and robust, it will usually copy data between devices many more times than necessary. This is because it is difficult for the compiler to inherently know when and where data will be needed. Therefore, the compiler cautiously copies over data frequently, just in case it might be needed. However, human programmers will always know when and where data should be used, and we can therefore exploit the data locality of the program. By data locality, I am referring to data which should remain in either host or device memory for as long as it is needed. Optimizing reuse and relocation of data through OpenACC directives is a key component of attaining higher efficiency with the framework.

Data Construct

To allow data to be shared between multiple parallel regions, the *data* construct must be used. This construct may decorate one or more parallel regions in the same function or alternatively placed at a higher/outer level to allow data to be shared between multiple functions. Adding a *data* region to a previous example:

```
1  #pragma acc data
2  {
3     #pragma acc parallel loop
4     for (i=0; i<N; i++)
5     {
6        y[i] = 0.0f;
7        x[i] = (float)(i+1);
8     }
9
10    #pragma acc parallel loop
11    for (i=0; i<N; i++)
12    {
13       y[i] = 2.0f * x[i] + y[i];
14    }
15 }
```

Listing 3.6: data directive

Now, data (the arrays x and y) may be shared between the two parallel regions. However, this is still not optimal in terms of data movement, so additional clauses are neccessary. The clauses available to use with the *data* construct are:

- **Copy**: Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region, copy the results back to the host at the end of the region, and finally release the space on the device when done.
- **Copyin**: Create space for the listed variables on the device, initialize the variable by copying data to the device at the beginning of the region, and release the space on the device when done without copying the data back the the host.
- **Copyout**: Create space for the listed variables on the device but do not initialize them. At the end of the region, copy the results back to the host and release the space on the device.
- **Create**: Create space for the listed variables and release it at the end of the region, but do not copy to or from the device.
- **Present**: The listed variables are already present on the device, so no further action needs to be taken. This is most frequently used when a data region exists in a higher-level routine
- **Deviceptr**: The listed variables use device memory that has been managed outside of OpenACC, therefore the variables should be used on the device without any address translation. This clause is generally used when OpenACC is mixed with another programming model.

There also special data clauses for more complex data structures such as C++ classes or C structs. These directives are slightly more advanced, and will be explained when I utilize them in my demonstration.

Array Shaping

The final piece of additional compiler info i will be presenting is called array shaping. This feature allows you to tell the compiler the size and shape of an array. Array shaping is done through a clause to the *data*

construct. In C/C++ you may describe the shape and size of an array as such: x[start:count], where start is the first element to be copied, and count signifies the number of elements to copy. For dynamically allocated arrays, array shaping is exceptionally useful, as those will not have known sizes or shapes at compile time.

Using the previous examples, we know that both x and y will be filled with data on the accelerator, therefore neither needs to copy data from the host. Additionally, we may use the *create* clause to create space for the x array on the device, in conjunction with a copyout clause for the y array. This is because the y-array contains the final values we want to store in host memory. In code, it would look like this, where N is the x and y array sizes:

```
#pragma acc data create(x[0:N]) copyout(y[0:N])
{
   #pragma acc parallel loop
   for (i=0; i<N; i++)
   {
      y[i] = 0.0f;
      x[i] = (float)(i+1);
   }

   #pragma acc parallel loop
   for (i=0; i<N; i++)
   {
      y[i] = 2.0f * x[i] + y[i];
   }
}
```

Listing 3.7: Array shaping with create and copyout directives

It is technically unneccessary to include the 0 in the *create* and *copyout* clauses, but they are added for readability. OpenACC supports a syntax where the 0 can be replaced by a blank: create(x[:N]).

There are more advanced features of OpenACC which i will not explain in detail here, but rather explain them when they are naturally brought up in my solution.

3.2 Comparing GPGPU Models

In this section I want to discuss the apparent drawbacks and upsides with choosing one of the different GPGPU models. Things I will consider are: performance, portability, ease of implementation and readability. Portability is the main thing I consider the most important. Second to portability will be performance, with the expectation that most other programming models will perform worse than CUDA already does, with the exception of optimized OpenCL implementations.

3.2.1 Performance Comparison

On the topic of performance, there are not a lot of existing GPU implementations of SIFT in OpenCL or OpenACC that report speedup. Most of them are understandably written in CUDA. There are some [7, 31, 55] using OpenCL, and Condello et al. reports that implementing SIFT on the GPU was slightly inefficient due to the high latency of global memory. Keep in mind that the article is from 2013, so they did not have access to the some of the high-end and consumer-available GPUs we have today. Their testing was done on a NVIDIA's GTX275 which launched in 2009. Additionally, they discuss trying to use a tree-based search algorithm similar to a classic CPU implementation of the Nearest Neighbour Problem, but that approach is not ported to the GPU. Condello et al. describe the branching nature of those CPU implementations as being extremely performance-hampering on parallel code, and the CUDA programming guide [36] advises against threads that diverge too much.

The other article, Yan[55], is not even really on SIFT, but rather on the Speeded-Up Robust Feature algorithm (SURF). However, they achieved a speedup of 37% and 64% over CUDA implementations on a GTX660 and GTX460SE respectively. The GTX660 approach is also on average 22 times faster than its original CPU version.

Another approach to GPU sift is Moren&Göhringer[31]. Their portable and multi-device OpenCL SIFT implementation has a significant performance advantage over similar CPU implementations. Their overall speedup is 2.69 times as fast. Their implementation is also platform-

independent, theoritically able to be run on CPU's, GPU's or other heterogeneous architectures.

For OpenACC , I was not able to find any existing papers or articles on a SIFT implementation using OpenACC, not considering Fassold's [11] article about the different frameworks and platforms for GPGPU in general. I attribute this fact simply to OpenACC being relatively new, having its first stable release of the 2.0 version in 2013 [1]. This means that I will have to do some performance testing myself, to find out how useful OpenACC is for SIFT specifically.

3.2.2 Portability Comparison

Here, I want to discuss the differences in portability of some parallel programming models. At first glance, I believe it will be much easier to port an existing algorithm and its code into OpenACC. My reasoning for this is intertwined with the relative ease of implementation of OpenACC as well. Annotating code is a lot easier than writing a whole kernel, as is the case with CUDA and OpenCL. However, I am not convinced that OpenACC is as simple as just annotating some code either. Porting or implementing an solution with OpenACC will still require deep knowledge of not only the GPU architecture, but the behind-the-scenes memory movement and management these annotations execute as well. While OpenCL is considered similar to CUDA, I believe that it would be much harder to completely rewrite several kernels from CUDA to OpenCL.

In both ease of implementation and readability, OpenACC clearly wins out in my opinion. Overall, I am convinced that OpenACC will be the best way to successfully port the SIFT algorithm to other GPUs and accelerator devices. I am expecting a hit in overall performance compared to CUDA solutions overall, but hopefully not so much as to render OpenACC completely unviable as a high-level parallel progamming option. As mentioned earlier, the main focus will be on finding a suitable parallel programming model for platform-agnostic solutions that can be run on other GPUs than only those manufactured

[1] https://www.openacc.org/sites/default/files/inline-files/OpenACC_2_0_specification.pdf

by NVIDIA, as well as being able to run on many- or multicore CPUs.

An important thing to keep in mind when writing portable OpenACC code however, is that when compiling with the NVHPC compiler nvc++ while having an NVIDIA GPU, the compiler will translate OpenACC directives into CUDA kernels. As most CUDA-ready devices nowadays have access to Unified Memory, it is important to turn this option off when compiling. To be brief, this is to ensure a certain degree of portability, as solutions written and tested with Unified Memory enabled will probably not work as expected on other architectures. I will explain this with more detail in section 4.2.

3.3 OpenCL

OpenCL is an open programming standard and interface for multi-core heterogenous systems, developed and maintained by the Khronos Group. Khronos is an open, non-profit consortium of over 150 companies. Some other projects maintained by Khronos include the Vulkan, OpenGL and WebGL renderers, as well as the programming standard aimed at cross-platform VR and AR development; OpenXR. As mentioned earlier, OpenCL is functionally and programatically similiar to CUDA. OpenCL uses kernels, and provides many of the same extensions. Additionally, OpenCL is far more portable than CUDA. However, OpenCL requires an even better knowledge of low-level programming paradigms, as kernels require far more lines of codes to implement with this interface.

Regarding portability of performance, OpenCL should behave similarly to OpenACC. Writing catch-all portable code is certainly feasible and should offer guaranteed correctness, but tweaking solutions for specific architectures is beneficial for performance. This is indeed similar to OpenACC, where much of the selling point is the ease-of-implementation, even though it will still require fine-tuning to the specific architecture to achieve maximum performance.

3.4 OpenMP

OpenMP [8] is another extremely popular multi-processor and device offloading programming model. It is very similar to OpenACC in terms of implementation, being directive-based as well. Initially, only multi-thread CPU offloading was possible, but they fairly recently (2013) added some support for offloading to GPUs and FPGAs as well. While the two models seems very similar, OpenMP is a bit more general. While OpenACC allows for intimate management of memory through its directives, OpenMP directives are mostly used to initiate specific parallel code execution. Additionally, OpenACC is much more focused on letting the compiler make optimizations and improvements, even in areas where the programmer initially didn't put any directives.

Since i primarily want to focus on the GPGPU and multi-core offloading aspects of high-level optimization models, OpenMP simply does not fit well enough for an efficient GPU implementation. While OpenMP might be more straigthforward [53], OpenACC offers a much more interesting approach to parallel code execution.

Chapter 4

Method and Implementation

This chapter aims to outline and explain my OpenACC demonstration. First an overview of the development environment and hardware specs. Then, I will present parts of the sequential SIFT implementation i used as a starting point, and how to offload these parts and computations to the GPU or multi-core CPU with OpenACC directives. After explaining the basic implementation, i will present the steps i have taken to further optimize the directives, using the features and methodology i outlined in section 3.1. I will be providing code snippets of my implementation and relevant theory throughout this chapter, which will explain and support my chosen methods.

4.1 Method

In this section I will discuss some of the design philosophies and methods of implementation i will be using throughout the parallelization cycle.

4.1.1 Implementation Cycle

In general, the way I will parallelize and re-write the sequential implementation follows three steps as outlined in figure 4.1. Firstly, I will do some basic profiling of the program, focusing on regions where I suspect there is the biggest room for improvement. Secondly, based on the basic profiling, I will implement a naive parallel solution which may

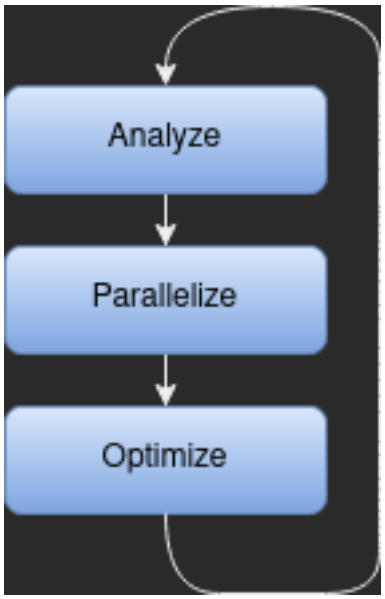

Figure 4.1: The parallel programming porting cycle

or may not achieve speedup. When dealing with a high-level framework such as OpenACC, i am unlikely to achieve significant speedup through a naive implementation. Lastly, I will profile the proposed parallel optimizations, making improvements and tweaks based on the data. This is the most important step, and requires knowledge of memory movement, as well as the correct utilities.

4.2 Development Environment

Having a proper development environment is important when porting a larger algorithm. Therefore, i will very briefly explain what technologies, i.e languages,frameworks,tools, or standards I used during the implementation phase.

4.2.1 Language Standards and Tools

The solution is written in C++ standard C++17, using CMake [1] minimum version 3.14 as a compilation controller, linker and organizer. The parallel OpenACC solution is written with OpenACC version 2.7+ in mind, and without the use of CUDA unified/managed memory. This is an

[1] cmake.org

extremely important thing to keep in mind when developing with OpenACC on a CUDA-enabled GPU, as the code will not work properly on other devices with this option turned on. Disabling managed memory comes with some extra challenges however, which i will discuss later in Chapter 6.

4.2.2 Profiling Utilities

I have used a combination of the C++ std::chrono library to measure elapsed real time, also known as wall time, and NVIDIA Nsight Systems to profile my parallel optimizations. It is important to note that the execution speed values from these two tool might not always line up either, due to a number of factors. The most important one being overhead due to profiler initialization. Nsight is mostly to profile CUDA and GPU code, and will not work as well for profiling the sequential solution.

4.2.3 Debugging Utilites

My main debugging utility is the command line program "compute-sanitizer" (formerly cuda-memcheck) for GPU code. This utility comes bundled with the CUDA toolkit. In addition, I also use valgrind for more basic memory access and movement runtime errors.

4.2.4 GPGPU Compilers

There are two main ways of compiling C++ code with OpenACC directives enabled: GCC(g++) with offloading capabilities enabled, or NVIDIA HPC SDK's nvc++.

GCC

GCC has support for offloading computations to a secondary processor, providing the programmer with access to both OpenMP and OpenACC directives. However, i found their implementation to be prone to errors and more difficult to get running, so i decided to not choose this compiler.

```
gaussian_blur(const Image &, float):
    293, Generating enter data copyin(img[:1],img->data[:img->height*img->width])
    299, Generating enter data copyin(kernel)
    310, Generating enter data copyin(kernel.data[:size])
    315, Generating present(img[:1],kernel)
         Generating copyout(temps[:sz]) [if not already present]
         Generating NVIDIA GPU code
        319, #pragma acc loop gang collapse(2) /* blockIdx.x */
        320,   /* blockIdx.x collapsed */
        323, #pragma acc loop vector(128) /* threadIdx.x */
             Generating reduction(+:sum)
```

Figure 4.2: Example of -Minfo=accel output

NVHPC

The NVHPC compiler is the commercial NVIDIA-supported compiler for all things HPC and GPGPU-related. For C++, the compiler is called nvc++, but used to be called pgc++, before the Portland group was acquired by NVIDIA in 2013[2]. An important feature of the NVCHP compiler is the abundance of flags you can set. The only strictly mandatory flag for compiling OpenACC code however, is the "-acc" flag. From my experience with OpenACC during this thesis, i found the "-Minfo" flag set to "-Minfo=accel" the most helpful. This flag will display possible parallelization, your implemented parallelizations and implicit optimizations that the compiler automatically does with accelerator code.

4.3 Sequential Implementation

The baseline sequential solution i will be partly referring to when demonstrating parallel optimizations with OpenACC, is a C++ implementation at https://github.com/dbarac/sift-cpp. To reiterate, I did not write this sequential solution myself, and I am not presenting the sequential solution as my own. I am simply using it as a baseline to demonstrate the effects of parallel optimizations.

The solution is faithful to the original algorithm, differing only in the descriptor normalization and utilization in actual image matching. This solution is written to closely follow an implementation outlined in the article called "Anatomy of the SIFT Method" [46].

Since the solution is a faithful implementation, it follows the SIFT steps

[2]https://www.theregister.com/2013/07/30/nvidia_buys_the_portland_group/

exactly, featuring 6 main functions:

- **generate_gaussian_pyramid**(): Upscales the base image by a factor of 2 through bilinear interpolation, then applies a blur with $\sigma = 1.6$ with default settings. Then, sigma values are calculated for the remaining blur levels, and the construction of the scale space pyramid begins. The gaussian pyramid is stored in a ScaleSpacePyramid struct.
- **generate_dog_pyramid**(): Constructs the DoG pyramid from the original pyramid by subtracting the layers. This pyramid is also stored in a ScaleSpacePyramid struct.
- **find_keypoints**(): Loops through the DoG pyramid to find extremums, and starts testing possible keypoint features. This is done by refining the keypoint through the contrast and edge-tests described in the original algorithm.
- **generate_gradient_pyramid**(): Generates a new scale space pyramid with the keypoints found in the DoG images layered onto the input images, containing the surrounding gradient information.
- **find_keypoint_orientations**(): First, this function checks if any keypoints are too close to the edge, and discards them if they are. Then, it loops through the gradient pyramid and accumulates them in directional bins, which are then smoothed with a 6x convolve box filter. Finally, the dominant orientation of the feature is computed.
- **compute_keypoint_descriptor**(): This function computes the actual descriptors for the keypoints. The function finds the image coordinates of the features from the gradient pyramid, and starts computing the orientation histograms, of which there are 8 in 16 regions surrounding the keypoint feature. Lastly, the orientation histograms are transformed into feature vectors.

Images are stored as structs containing various information such as width,height and the actual data. Some member functions to get and set specific pixels are declared as well.

```
1 struct Image {
```

```
    explicit Image(std::string file_path);
    Image(int w, int h, int c);
    Image();
    ~Image();
    Image(const Image& other);
    Image& operator=(const Image& other);
    Image(Image&& other);
    Image& operator=(Image&& other);
    int width;
    int height;
    int channels;
    int size;
    float *data;
    bool save(std::string file_path);
    void set_pixel(int x, int y, int c, float val);
    float get_pixel(int x, int y, int c) const;
    void clamp();
    Image resize(int new_w, int new_h, Interpolation method =
   BILINEAR) const;
};
```

Listing 4.1: Image struct

In addition to the Image struct, the Keypoint struct stores information about the coordinates of a keypoint (including DoG-coordinates). In addition, a 128-element array is allocated as the container for the actual descriptor of the keypoint feature.

```
struct Keypoint {
    // discrete coordinates
    int i;
    int j;
    int octave;
    int scale; //index of gaussian image inside the octave

    // continuous coordinates (interpolated)
    float x;
    float y;
    float sigma;
    float extremum_val; //value of interpolated DoG extremum

    std::array<float, 128> descriptor;
```

```
15 };
```

Listing 4.2: Keypoint Struct

4.3.1 About Porting popsift Directly

The ideal way to compare an OpenACC implementation of SIFT to popsift would be to un-CUDAfy the solution, turning it into a sequential solution, and then use OpenACC to parallelize again. However, this would take a tremendous amount of programming effort, and I believe it would not be necessary in order to answer the goals proposed in this thesis. The largest contributor to the amount of programming effort required is the inability to use the CUDA texture engine with OpenACC. Popsift utilizes this feature extensively, and is built from the ground up with it in mind. I will talk a bit more about the texture engine in Chapter 6 as well. If my goal was to write a replacement for popsift with OpenACC, I should either have written a solution from scratch using popsifts methodology, or rewritten popsift as a sequential solution. Alas, it seemed like a better idea to instead refocus my efforts on learning OpenACC through porting an already sequential solution, as well as studying available research and literature on the topic.

4.3.2 Profiling

To get an idea of which functions takes the longest to execute, I perform initial profiling on the sequential solution. On a single run-through with a 1920x1080 sized image containing about 12,000 features, the time spent in each function was as follows:

Function	Time Spent(s)	Relative Time Spent(%)
generate_gaussian_pyramid()	2.98	28.27
generate_dog_pyramid()	0.11	1.09
find_keypoints()	0.53	5.05
generate_gradient_pyramid()	2.12	20.00
find_keypoint_orientations()	0.29	2.81
compute_keypoint_descriptor()	4.50	42.66
TOTAL	10.55	100

Table 4.1: Execution Time by Function

As illustrated by table 4.1, the majority of execution time is spent computing keypoint descriptors. This is not surprising, considering the amount of features in an image of that size. Considering the image has about 12,000 features, and each feature is described by 16 square regions containing 8 histograms, the total number of computations is about 1.5 million. Similarly, the gaussian pyramid function also has to do an enormous amount of calculations, especially on the first upscaled image. To demonstrate paralleization with OpenACC, these are the two regions i will be optimizing further.

4.4 Parallel Implementation

With a very rough idea of where I should start, I can now begin actually implementing parallel solutions.

4.4.1 OpenACC Directive-Based Implementation

Gaussian Pyramid

The key part of this step is the actual gaussian blurring of the scaled images. The gaussian_blur function is by far the most called during the construction of the scale space and gaussian pyramid. Code listing 4.3 is a display of my proposed naive parallel optimization using the *parallel loop* directive.

In the sequential solution, the primary step of convolving the original image with the gaussian kernel is done inside two triple-nested loops. The outer two loops are simply iterating through every pixel in the current-scale image, while the innermost loop is the actual gaussian calculation:

```
1    // convolve vertical
2    #pragma acc parallel loop independent collapse(2) present(
     kernel, img) copyout(temps[:sz])
3    for (int x = 0; x < img.width; x++) {
4        for (int y = 0; y < img.height; y++) {
5            float sum = 0;
6            #pragma acc loop reduction(+:sum)
7            for (int k = 0; k < size; k++) {
```

```
8                   int dy = -center + k;
9                   sum += img.get_pixel(x, y+dy, 0) * kernel.data[k
    ];
10              }
11              /* tmp.set_pixel_routine(x, y, 0, sum, failed); */
12              temps[ y*img.width + x] = sum;
13          }
14      }
15      std::copy(temps, temps + sz, tmp.data);
```

Listing 4.3: Vertical Gaussian Convolution

This is the first of the two convolution loops. There two separate loops because the gaussian convolution has to happen in both vertical and horizontal directions. The second loop is also dependent on data from the first loop. The next loop, which convolves in the horizontal direction, is mostly identical. As seen in listing 4.3, I have added some OpenACC directives to initialize parallelization. First, the loop is made parallel through the *parallel* and *loop* directives. These two are the most important constructs. Then, the *independent* clause is an additional assertion that overrides the compiler analysis. This is essentially a guarantee from me to the compiler that the loop iterations are completely data-independent. However, that assertion comes with one caveat, which can be seen with the directive decorating the innermost loop. The *reduction* clause is used to reduce the loop into sum before moving on. This is in order to ensure data correctness when assigning the value in the second loop.

Going back to the outer loop, I have added some additional constructs and clauses. The *collapse* construct allows the compiler to flatten the two first loops, giving it more room for parallelization. The *present* construct allows me to signify to the compiler that these values are already present on the device. These values have been added as unstructured data earlier in the function:

```
#pragma acc enter data copyin(img, img.data[:img.width*img.
    height])
int size = std::ceil(6 * sigma);
if (size % 2 == 0)
    size++;
int center = size / 2;
Image kernel(size, 1, 1);
```

```
#pragma acc enter data copyin(kernel)
float sum = 0;
for (int k = -size/2; k <= size/2; k++) {
    float val = std::exp(-(k*k) / (2*sigma*sigma));
    kernel.set_pixel(center+k, 0, 0, val);
    sum += val;
}
for (int k = 0; k < size; k++) {
    kernel.data[k] /= sum;
}
#pragma acc enter data copyin(kernel.data[:size])
```

Listing 4.4: Unstructered Data by using the enter data directive

As i briefly mentioned when explaining the OpenACC data directives, there exists an option for more complex data structures. This directive comes in the form of the *enter data* directive that can bee seen in listing 4.4. Whenever this directive appears, the data which is copied or created on the device stays there until a corresponding *exit data* directive is used.

Finally, the *copyout* data clause allows the compiler to allocate space on the device for the array "temps", which is initially empty before the loop, as well as knowing its size "sz". Then, when the parallel loop region is finished, the data which has been stored in the device-side version of the temps array will be copied back to host memory. I then store the temporary data in a host array, as the data will be used in the next triple-nested loop.

For now, this naive implementation will stand, and I will go over the profiling and optimization in section 4.4.2.

Computing Keypoint Descriptors

As outlined earlier, the description of keypoint features is done in the **compute_keypoint_descriptor**() function. In it, the orientation histograms are computed in a double-nested loop. Finally, the data from those histograms are transformed into 128-element feature vectors. While the previous naive implementation of gaussian pyramid construction didn't incur any particular speedup or slowdown, this naive implementation causes the program to slow down significantly.

Without placing any significant importance on data movement and just implementing a parallel solution that provides near-identical resuls to the original, i propose the following naive optimization:

```
1  //accumulate samples into histograms
2  #pragma acc parallel loop  independent collapse(2) present(
       img_grad) copy(histograms[:N_HIST][:N_HIST][:N_ORI])
3  for (int m = x_start; m <= x_end; m++) {
4      for (int n = y_start; n <= y_end; n++) {
5          // find normalized coords w.r.t. kp position and
       reference orientation
6          float x = ((m*pix_dist - kp.x)*cos_t
7                  +(n*pix_dist - kp.y)*sin_t) / kp.sigma;
8          float y = (-(m*pix_dist - kp.x)*sin_t
9                  +(n*pix_dist - kp.y)*cos_t) / kp.sigma;
10
11         // verify (x, y) is inside the description patch
12         if (std::max(std::abs(x), std::abs(y)) > lambda_desc*(
       N_HIST+1.)/N_HIST)
13             continue;
14         float gx = img_grad.get_pixel_routine(m, n, 0), gy =
       img_grad.get_pixel_routine(m, n, 1);
15         float theta_mn = std::fmod(std::atan2(gy, gx)-theta+4*
       M_PI, 2*M_PI);
16         float grad_norm = std::sqrt(gx*gx + gy*gy);
17         float weight = std::exp(-(std::pow(m*pix_dist-kp.x, 2)+
       std::pow(n*pix_dist-kp.y, 2))
18                                     /(2*patch_sigma*patch_sigma)
       );
19         float contribution = weight * grad_norm;
20         update_histograms(histograms, x, y, contribution,
       theta_mn, lambda_desc);
21         }
22 }
```

Listing 4.5: Naive implementation of feature descriptor computation

Code listing 4.5 is the main loop of the feature descriptor computation function, and has been decorated with OpenACC pragma directives. Note that many of the constructs used here are the same as the previous parallel naive solution in listing 4.3. The present clause here states that an element img_grad should be present on the device. This is assured by an "enter data copyin" directive earlier in the function. The element

img_grad is an Image containing the current gradient image being looped through, depending on the current scale and octave. It is likely that the entering of img_grad is the biggest cause of slowdown, which I will look into more in the optimization and profiling section.

Furthermore the *copy* data clause is used here. This is in order to ensure that the array histograms (containing the orientation histograms) is copied into the device with its intial values, which are previously set to 0. Then, the data in histograms is copied over to the host after being modified by the calls of update_histogram() in the parallel code. In addition, since update_histograms is being used in a parallel region, it must be declared as an acc routine:

```
#pragma acc routine
void update_histograms(float hist[N_HIST][N_HIST][N_ORI], float
    x, float y,
                       float contrib, float theta_mn, float
    lambda_desc)
{
            *** truncated ***
            for (int k = 1; k <= N_ORI; k++) {
                float theta_k = 2*M_PI*(k-1)/N_ORI;
                float theta_diff = std::fmod(theta_k-theta_mn+2*
    M_PI, 2*M_PI);
                if (std::abs(theta_diff) >= 2*M_PI/N_ORI)
                    continue;
                float bin_weight = 1 - N_ORI*0.5/M_PI*std::abs(
    theta_diff);
                #pragma acc atomic update
                hist[i-1][j-1][k-1] += hist_weight*bin_weight*
    contrib;
            }
        }
    }
}
```

Listing 4.6: Update Histogram routine function

Another extremely important thing to include in this function is the *atomic update* directive. Usually, subroutines are easily handled by simply adding the routine seq information, so in cases such as these, it is important to keep data correctness in mind. Without this directive,

data correctness cannot be assured. This is because several threads will try to access and write into the histogram-memory, and cause undefined behaviour.

4.4.2 Optimizing Directives and Data Movement

In this section i will investigate my naive parallel implementations further, in order to find out where optimizations in data movement and other general improvements may be done. As outlined earlier, i will be doing this in three steps. Firstly, i will profile the solution using Nvidia Nsight Systems. In general, profiling with Nsight Systems is done through a CLI: "nsys-profile -t cuda,openacc <executable>". This creates a report which can be opened via an advanced GUI, showing desired trace information. The -t flag specifies which traces we would like to include in the report.

Secondly, i will make modifications which i believe will alleviate the problems shown through profiling. Finally i will evaluate performance once again through profiling, and decide if the proposed modification was beneficial to overall performance.

Optimizing Gaussian Pyramid Construction

For this section, I will simply revert all other code than the Gaussian Pyramid function to sequential code, in order to more clearly understand the profiling results. Profiling with the nsys CLI and opening the report in software interface yields the result in figure 4.3.

In this figure, we are presented with several ways to process the results. Firstly, there are visual boxes representing the different GPGPU API calls that the program does during execution. On the OpenACC row, the basic breakdown of API calls is differentiated by color. Red represents memory management, in our case mostly "exit data" and "enter data". Since I am compiling for an NVIDIA GPU, those API calls are in turn translated to CUDA API calls. These are the beige boxes in the CUDA API row. These CUDA API calls to allocate memory are in turn executed on the hardware, in the memory row. This execution of API calls is displayed in turquoise or pink, and is either HtoD(Host to Device) or DtoH(Device to

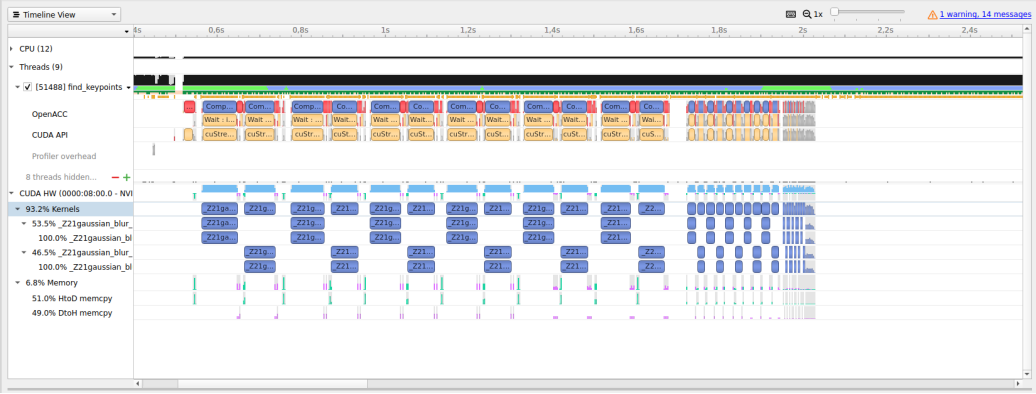

Figure 4.3: Results from the Nvida Nsight Systems profiling of the Gaussian Pyramid naive parallel implementation

Host) respectively.

In addition to memory management, figure 4.3 also displays the initialization of OpenACC compute constructs, and the respective CUDA API calls to start and execute CUDA kernels. Kernel initialization and execution is shown here as blue boxes, and the lighter blue above the CUDA kernels signify how efficient the kernels are. As we can see, in most cases except for the very short kernels, efficiency is maxed out. For the smaller and shorter kernels there are certain valleys present, but this is mostly due to kernel initialization overhead being a larger percentage, and the results being smoothed out. If I were to zoom further in, we would see that the sub-optimal efficiency numbers are simply at the beginning and the end of each kernel execution.

Interpreting the actual data, we see that there is a certain degree of memory movement overhead between the kernel initializations. However, the ratio of memory transfer to actual execution time is very good. The reason is likely the fact that I am intentionally restricting myself to not have access to the CUDA Unified Memory. This forces me to properly manage memory from the beginning. Additionally, this function is quite simple, and the order in which data is transferred is more or less impossible to change. If possible, I would like to use async loops here, but the strictness of the SIFT algorithm requires that i first compute at least the third-to-last image in an octave before moving on to the next one.

Remember that in the function, we copy both the base image and its data into the device, as well as the kernel. An approach where less data is copied onto the host and back would be preferable. A potential alternative may be to instead make a host copy of only the data, and pass that to the loop. This should effectively remove the need for both enter and exit data directives. However, due to being a sufficiently efficient function now, I will not make further modifications.

Optimizing Computing Keypoint Descriptors

For this step i am keeping the optimizations done in the previous section, as they are for the most part entirely separate. In addition, this will make the profiling process itself take a shorter amount of time. As i mentioned earlier when outlining the naive implementation of this function, the parallel solution caused a severe slowdown of about 70 seconds. Profiling will reveal where and why this slowdown occurs.

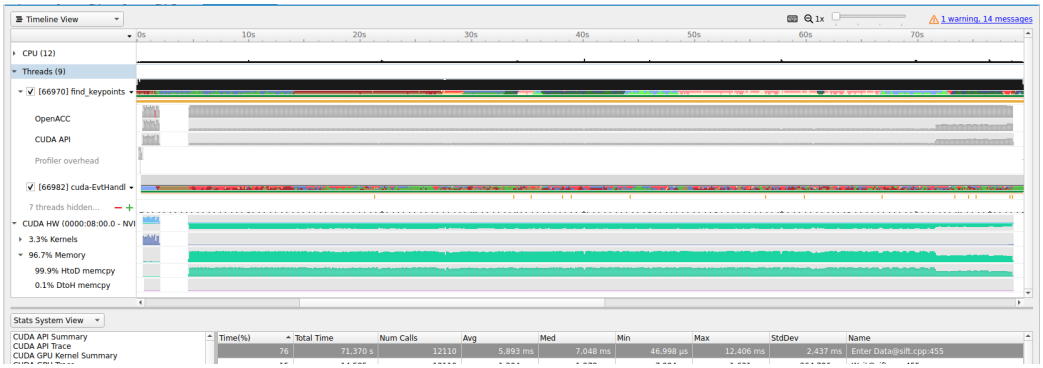

Figure 4.4: Results from the Nvida Nsight Systems profiling of the Compute Keypoint Descriptors naive implementation

Figure 4.4 is the NVIDIA Nsight Systems report visualized. Compared to the previous function, it is visibly apparent what is wrong. The ratio of memory management to actual kernel execution is extremely skewed towards memory management. Essentially, when the program is managing memory, it could have been doing something actually productive. As can be seen on the table at the bottom of the figure, 71 seconds is spent entering data. It is clear that the amount of memory movement must be lessened somehow. Just for reference, figure 4.5 is the same code, but compiling with managed/unified memory on. The ratio

of memory movement vs kernel execution is completely different, and much more healthy. This is an example of how programming without managed memory truly is more convenient, albeit for a very specific architecture.

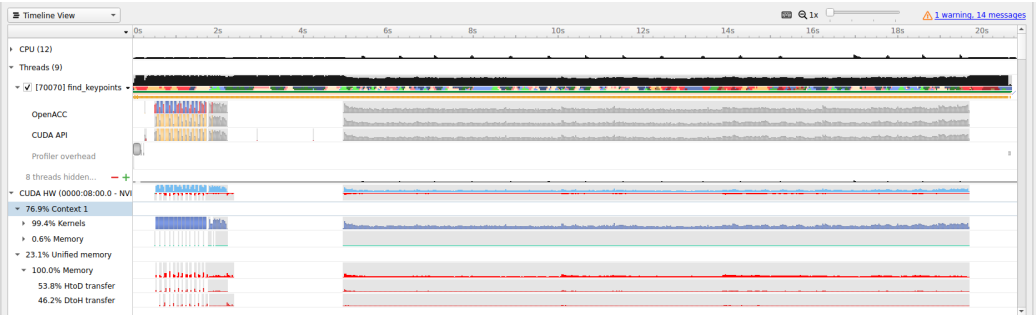

Figure 4.5: Results from the Nvida Nsight Systems profiling of the Compute Keypoint Descriptors naive implementation with unified memory enabled

However, as i aim to evaluate what good portable code looks like, i will not be using unified memory. Additionally, as i suspected earlier when implementing the parallel solution naively, entering and exiting the gradient image data is what takes the most time. However, one way to solve this is to not use the *exit data* directive directly after the parallel loop. To assure no data leaks, we still have to delete the data some time later in the code. By only entering data within the function, and not exiting it, we keep the data in memory for multiple function calls.

The reason this approach works, and does not lead to some kind of overflow or duplication of data is due to how the copyin data clause works. Technically, all data clauses contain a hidden "if_present" check, stating that if the element being moved is already present at the destination, it does not need to be moved again. Therefore, this works even though the compute function is called for each detected keypoint whose location may be in any of the octaves or levels of the gradient pyramid. Put simply, the first time a keypoint appears in a certain gradient image, that gradient image is transferred to the device, where it stays for the remainder of the program.

```
1 #pragma acc enter data copyin(img_grad, img_grad.data[:sz])
2     float cos_t = std::cos(theta), sin_t = std::sin(theta);
3     float patch_sigma = lambda_desc * kp.sigma;
```

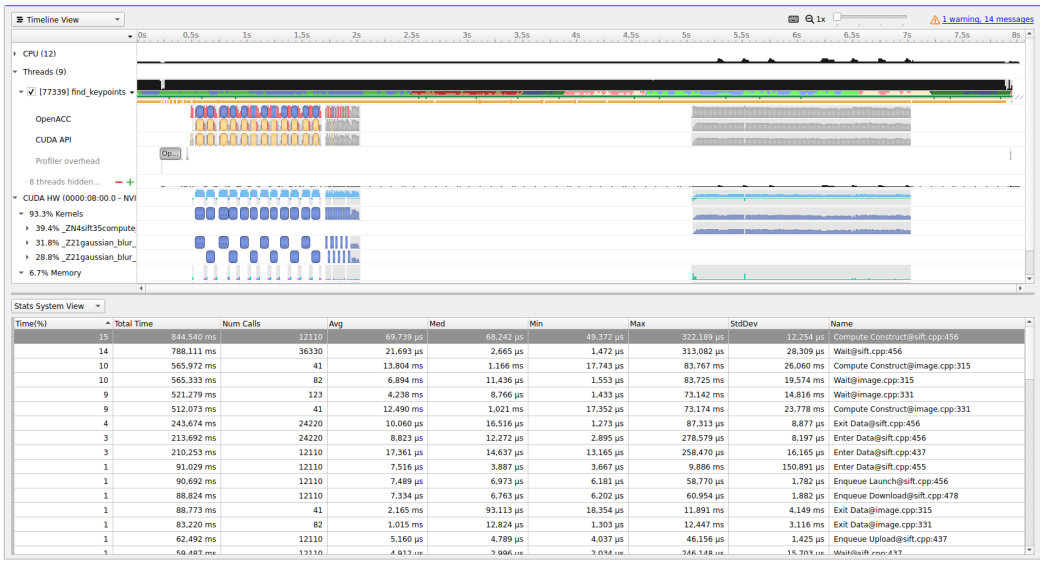

Figure 4.6: Results from the Nvida Nsight Systems profiling of the Compute Keypoint Descriptors optimized parallel solution

```
4     //accumulate samples into histograms
5     #pragma acc parallel loop  independent collapse(2) present(
      img_grad) copy(histograms[:N_HIST][:N_HIST][:N_ORI])
6     for (int m = x_start; m <= x_end; m++) {
7         for (int n = y_start; n <= y_end; n++) {
```

Listing 4.7: Entering data before the main feature descriptor computation loop

Implementing this optimization step has a drastic effect on the runtime and overall frequency of memory transfers in comparison to the original naive implementation. Looking at figure 4.6, we can see that the ratio of memory movement to kernel execution is at a much more sane level.

Chapter 5

Results

The results from my parallel optimizations will be presented in this chapter. Additionally, i will present and explain general considerations and conditions that might influence the data, such as testing computer specifications, differences in keypoint amounts in images, as well as image sizes. Since the optimizations are meant to be as platform portable as possible, i will present results from executing the program both on a GPU, as well as a multi-core CPU.

5.1 Testing Environment and Setup

5.1.1 Computer Specs

Name	AMD® Ryzen 5 2600x
# Of CPU Cores	6
# Of Threads	12
Base Clock	3.6GHz
L1 Cache	576 KB
L2 Cache	3MB
L3 Cache	16MB

Table 5.1: CPU capabilities of the testing machine

CUDA Compute Capability

An important thing to remember is that since my testing machine has an NVIDIA GPU, OpenACC will actually offload the code by translating

Name	NVIDIA Corporation GP102 [GeForce GTX 1080 Ti]
# Of CUDA Cores	3584
# Graphics Clock (MHz)	1480
Processor Clock (MHz)	1582
Standard Memory Config	11 GB GDDR5X
Memory Interface Width	352-bit
Memory Bandwidth	11 Gbps

Table 5.2: GPU capabilities of the testing machine

it to CUDA. Therefore, the CUDA compute capabilities and details of my graphics card is of significance. This info is available in the appendix table A.1, and is obtained by running the command *nvaccelinfo*, which is a utility that is part of the CUDA drivers.

5.2 Time Performance Evaluation

In this section I will present the resulting speedup from my parallel optimizations compared to the original sequential implementation. I will only evaluate the timings of the feature extraction and description step, and ignore the execution time spent on surrounding processes such as image loading and descriptor output I/O operations. Additionally, i will not evaluate the timings of actual image matching with the parallel optimizations, as this is essentially just the extraction and description step performed two times. Arguably, the image matching itself is an important part of most processes utilizing SIFT, but it is not the primary focus of this thesis. Therefore, the only qualitative evaluation I will be doing is evaluation with regards to the number of feature descriptors extracted per algorithm in section 5.3.

To evaluate the timings and potential speedup of the parallel optimizations, I will run both the sequential and parallelized solution through two sets of images. The first is an excerpt containing nine 640x480 png images from the Zurich buildings dataset [50], while the second is a collection of nine 1920x1080 jpg images i downloaded from Google Images. The timings will be computed as the average value of these 9 runs. This is to ensure that the results are more correct and robust with regards to images of varying sizes, as well as images containing differing amounts

of feature keypoints and descriptors. Keep in mind that the timings are measured as wall-clock time and not CPU time, as our program has a lot of non-cpu operations as well as a large degree of parallelism.

For the parallel solution, i will be compiling the code into two different binaries with the nvc++ compiler flags "-acc=gpu" and "-acc=multicore". This will produce two different binaries which are targeting GPU and multicore CPU architectures respectively. This is to produce results which will help me answer RQ1 regarding portability.

5.2.1 Zurich Dataset

Running the three solutions through the Zurich dataset excerpt produces the results shown in figure 5.1. This is a simple presentation of the average runtime explained earlier. As expected, the sequential solution is the slowest, while the GPU and multi-core CPU implementations are very close in terms of execution speed. Taking into consideration that we have not made any platform-specific optimizations, the closeness in speed of the two binaries is an expected result. An additional consideration here is general program overhead, as the Zurich images are quite small. Therefore, general program overhead likely plays a factor when looking at these speeds.

Figure 5.2 is a bit more interesting. First off, my definition of speedup is as follows:

$$speedup = \frac{sequential\_solution\_speed}{parallel\_solution\_speed} \tag{5.1}$$

The speedup value is used to determine the actual amount of times faster the parallelized solution is. For example, for the average execution speed values, the achieved speedup for the GPU version is about 1.57. We want to compute a speedup value in order to further combine it with other data, creating more complex data points.

Back to figure 5.2, this graph represents the efficiency of the parallelized solution on the two different architectures. It is important to note that this graph does not start at 0 y-value, this is in order to see the differences

Figure 5.1: Execution times in ms for the Zurich Dataset

in speedup more clearly, as they are quite small. From the graph we observe that the multi-core binary produces higher speedup at lower keypoint amounts. However, the GPU-accelerated solution quickly becomes more efficient at about 2600 keypoints.

5.2.2 Google Images Dataset

Running the three solutions through the images I downloaded from Google Images, the resulting execution speeds are shown in figure 5.3. Remember that these images are 1920x1080 resolution, and are therefore a bit more representative of common SIFT situations, such as video tracking or image matching. Again, the sequential solution is the slowest, and the GPU-accelerated solution is slightly faster than the multi-core one. Keeping some of the results from the previous section in mind, it is expected that these larger images would be faster on the GPU. This is due to an overall larger amount of features per image.

In figure 5.4, we see a similar graph to 5.2. The key difference here is that even the most feature-sparse image in this dataset contains more keypoints than the most feature-dense in the Zurich dataset. Therefore, the GPU-solution starts off with being the superior solution in terms

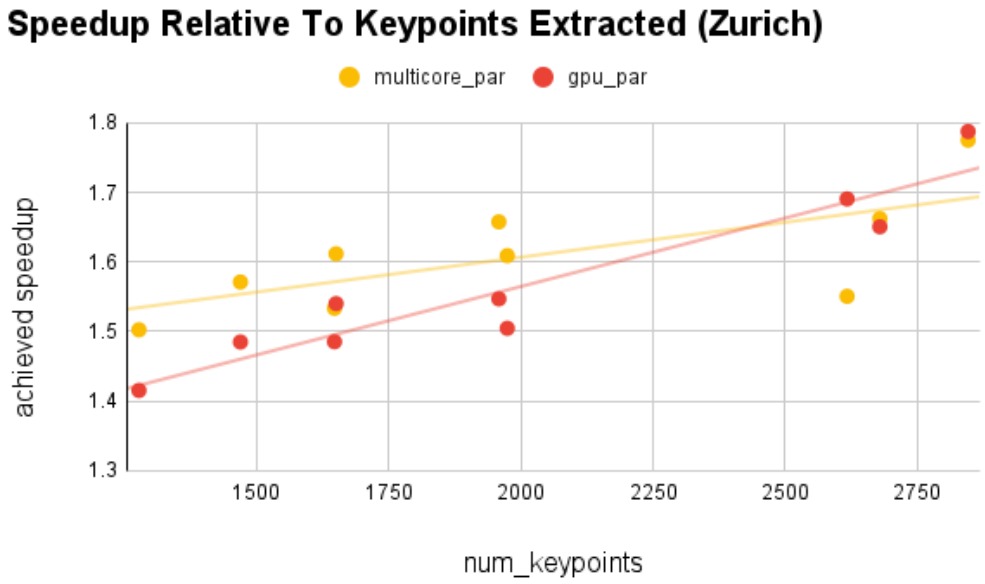

Figure 5.2: Speedup achieved from the sequential solution compared to keypoint descriptor amounts (Zurich dataset)

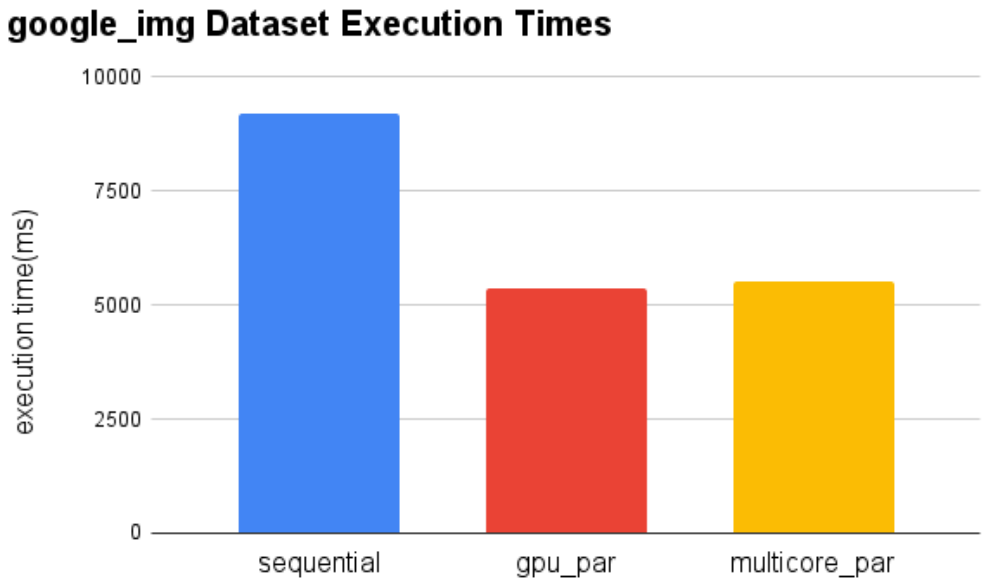

Figure 5.3: Execution times in ms for the Google Images dataset

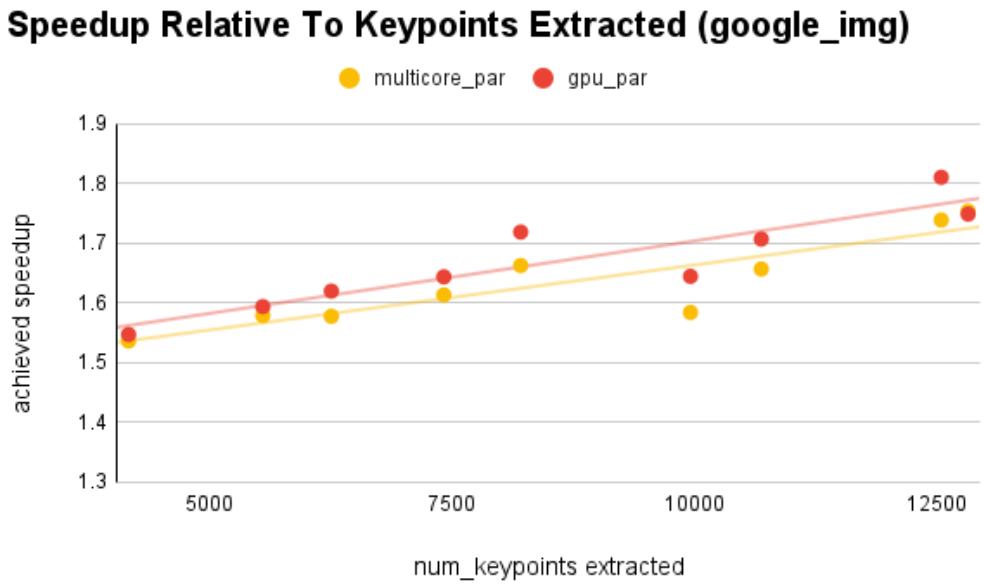

Figure 5.4: Speedup achieved from the sequential solution compared to keypoint descriptor amounts (Google Images dataset)

of speed. Looking at the drawn trend lines, we can see that the GPU-solution grows slightly faster than the CPU-solution. This trend is likely to continue, as GPUs generally perform better the larger the problem is. Another thing that should be considered here is hardware, which i will discuss more in Chapter 6.

5.3 Number of Feature Descriptors Extracted

In addition to time performance evaluations, i will also evaluate one qualititative aspect of the optimization demonstration. In order to compare how many feature descriptors are extracted, i will use an image from the zurich dataset, and compare the amount of feature descriptors extracted using the sequential solution, popsift, and the parallel solution. The image i will use for this can be found in appendix B.1. Additionally, figure 5.5 is the image with keypoints from the parallel solution drawn on using the stb_image_write and stb_image libraries.

The different numbers of feature descriptors extracted from the Zurich building image is shown in table 5.3. Note how the parallel solution

Figure 5.5: Zurich city building with keypoints drawn

Implementation	Descriptors Extracted
popsift	2656
sequential	1817
parallel	1958

Table 5.3: Amount of extracted descriptors per implementation

finds more descriptors than the sequential solution, even though they are using the same functions and criteria for detecting keypoints. This difference is likely due to the imprecision in precise decimal values that inherently occur when computing values on a GPU and transferring them to the CPU back and forth. In this way, small changes in the values may allow for some keypoints to pass contrast and edge tests even though the same keypoints did not pass those tests in the original algorithm.

5.4 Evaluating Programming Effort

5.4.1 Quantifying Programming Effort

To answer RQ2 about what the overall advantages of using such a high-level programming model, I need to quantify the programming effort my parallel optimizations have had. An easy way to do this is to compare how many directives, e.g lines of code, i had to write with how many lines of code are in the sequential implementation already. This will give me a value which i call parallelEffort, and it is defined as such:

$$parallelEffort = \frac{totalDirectives}{totalCode} \tag{5.2}$$

Thus, the defintion of parallelEffort states how many lines of code are parallel optimization. This can also be combined with the speedup value, making it possible to quantify how much speedup per measure of effort. The resulting number is then a quantifiable property of how much speedup is gained per unit of effort.

Applying this to my solution, i get the following result:

```
totalCode = 1130
totalDirectives = 19
parallelEffort = 19/1130
parallelEffort = 0.016

speedupPerEffort = 1.57/0.016
speedupPerEffort = 98
```

Now i want to compare this value to a similar one, using popsifts code as a base. I measure the speedup of popsift over the sequential solution to be about 18x using the same zurich excerpt image. I will also approximate the amount of code that popsift runs through with default configurations, as well as the lines of code to implement those CUDA kernels. My very rough estimation of the total code executed in popsift, excluding kernels, is about 4000. Additionally, i estimate about 1700 lines

of kernel code. Then, doing the same calculations with popsift:

```
totalCode = 5700
totalKernels = 1700
parallelEffort = 1700/5700
parallelEffort = 0.298

speedupPerEffort = 18/0.298
speedupPerEffort = 60.40
```

Interpreting the results, the initial parallelEffort value states that about 30% of the executed popsift code is kernel code. Keep in mind this number is calculated with a rough estimation based on analysis of the popsift codebase and program flow. Then, computing the speedupPerEffort value which turns out to be about 60.40.

5.4.2 Productivity Comparison

Additionally, comparing the two values, we can get a rough idea of how the two models differ in productivity and ease-of-implementation. Essentially, OpenACC has about 50% more speedup per effort than CUDA. Of course, this is not an exact value to draw conclusions from by itself, and there are several factors which might influence how many lines of code are necessary for a certain problem. Also, not every bit of optimization comes from parallel code, it may also come from differing implementations or clever optimizing tricks. In order to glean any meaningful information from these values, I will have to look at prior research on productivity with OpenACC, and assess if the findings support or contradict my own results.

In addition, this is all under the assumption that all lines of code require the same amount of programming effort to write, which is simply not true. However, it is simply unfeasible to compute an estimation of each line of code without conducting extensive producitivity studies such as Daleiden et. al [9] and Li et al.[22]. Essentially, the computed speedupPerEffort values should only be treated as a representation of

how the two frameworks differ, and how we can further investigate this difference going forward. This discussion and analysis will be done in section 6.2.

Chapter 6

Discussion

In this chapter i want to properly discuss and explain my findings based on the results i have generated in the previous chapter, linking them up to previous research and studies. I will also discuss how my results answer the research questions i posed at the beginning of the thesis. Additionally, I will talk about some broader issues and considerations i found during the implementation process. Finally, i will be analyzing my methodology and providing alternatives based on prior findings within this subject.

6.1 Speedup and Time Performance

It is hard to evaluate the speedup of my parallel optimizations on the SIFT algorithm compared to other research papers and experiments, simply because there is not a lot of research on this specific topic. What we can do however, is look at OpenACC being used to replace or supplement CUDA or some other framework in other fields of computation. For example Searles et.al[49] which describes OpenACC as a parallelization framework for wavefront algorithms, e.g. algorithms that require the results to be computed in stages. This is not dissimilar to certain parts of the SIFT algorithm, where a computation in the construction of the gaussian scale space pyramid depends on images from the previous octaves. Searles et. al report a whopping 85x speedup using OpenACC over the associated sequential implementation. In

addition, this speedup is faster than their results with CUDA, where they report a speedup of 83.72. Thus, this shows that certain problems are extremely parallelizable to great effect, or the previous sequential implementation was extremely slow.

A similar speedup result can be found in a paper[33] on the DS-DMAS algorithm for photoacoustic image reconstruction. Here, a speedup of 74x is achieved for 1024x1024 images. In it, they also assert that the larger the image they run through the algorithm is, the larger the GPU speedup is. This supports my own results, and suggests that this might be true for other image-processing algorithms than just SIFT.

In another field, namely nuclear physic simulation [28], OpenACC is used to accelerate the solution of large eigenvalue problems. This paper presents a translation of OpenMP code into Openacc, and their speedup ranges from 15-27 for differently size input. Their solution is written in Fortran OpenACC, and features many GPU-specific optimizations, as they are targeting them specifically. Two other OpenACC implementations which report speedup factors on the lower-end are Otero et.al[42] and Martelli et.al[29], with values of 3.1 and 4.56 respectively.

As we can glean from the papers i have presented above, attaining very high speedup numbers is possible with OpenACC, but it does require more fine-tuning to specific architectures. The amount of speedup you can reach is mostly a matter of how much data movement optimization and platform-specific fine tuning you are willing to do. If I wanted to create an implementation that would rival popsift in speed, i would need to do a lot more optimization. This fact is probably the reason why my parallel optimizations do not achieve a big speedup like other implementations. Additionally, I did not make optimizations in every part of the program, and really only focused on some hotspots. However, my optimizations were mostly to illustrate how OpenACC will parallelize data, as well as generate some productivity statistics.

Regarding hardware, the processor i used for testing is no where near the core amount of something like a Xeon Phi, where the core count is usually between 60-70, depending on the series. These processors are

designed with parallel models such as OpenMP in mind, and would be perfect for OpenACC multicore solutions as well. If i had such hardware available, I am certain that my multi-core solution would have much larger speedup. Alas, i have not had the chance to test my solution on a proper manycore processor.

6.2 Productivity in Abstraction

The push for high-level programming models is fairly recent, with the Thrust CUDA framework having its 1.0 release in May 2020[1], OpenACC's 2.5 in 2016, while an older model such as OpenMP is still only from 1997. It seems that the need for productivity and high-level abstraction in the parallel programming discipline was quickly being realized by these programming models. It is not surprising that a model such as CUDA gained popularity, being a good combination of readable, efficient and customizable.

Additionally, due to rise of the GPU powerhouse known as NVIDIA, CUDA also benefits as a model due to having support in a wide number of computers and high-performance workbenches. However, CUDA still suffers from being a bit too low-level and complicated, as supported by Li et. al[22]. Also, while CUDA does have a large user base, the truth remains that only NVIDIA GPUs can run it, limiting its user base ever so slightly. This is why models such as OpenACC and OpenCL have gained popularity as well, both for different reasons.

While OpenACC boasts minimal implementations for maximum performance, the truth is that without careful optimization, high-level directives are not always as effective as the model would lead you to believe. Looking at my own results and Daleiden et. al's study[9], there is actually quite a learning curve to OpenACC as well. I believe that programmers might be fooled into thinking that optimizations with OpenACC will be severely easier, and that might be true in terms of actual programming effort, but the knowledge required to actually implement an efficient solution is a hurdle that needs to be cleared.

[1] https://github.com/NVIDIA/thrust/releases/tag/1.0.0

My results seem to suggest that there is indeed a benefit to a high-level programming model however, seeing how my estimated quantification of speedup per programming effort shows OpenACC being clearly favoured over CUDA. My demonstration just goes to show that with just a few lines of code, it is actually possible to optimize a solution, albeit with a small speedup, in a relatively short amount of time and effort. Remember that even though the achieved speedup is small, the point still stands that the speedup-to-effort ratio should be much higher, as is supported by several other OpenACC implementations as well [15, 28, 33].

6.3 Losing CUDA-Specific Technology

As my development and testing has mostly occurred on an NVIDIA GPU, the OpenACC directives implicitly translate to CUDA code. However, the abstraction level that OpenACC provides does not allow for some more advanced CUDA features to be initiated and executed. If it did, then the required programming effort would probably rise. The entire point of OpenACC is to write portable and easy-to-implement code, and being too specific in your implementation with regards to target architecture might be a slippery slope.

Consider a simple loop that you want to optimize through OpenACC. With the current implementation, it is possible to write a single, maybe two or three lines of codes with directives to very efficiently offload this loop to an accelerator. Now consider what it would be like if for every accelerator type, you knew that there are several fine-grained and detailed ways to optimize the loop. This would result in programmers creating loads of directives, each targeting their own specific architecture. In all likelihood, this would lead to better performance portability, but would absolutely lead to a manyfold increase in programming effort.

6.3.1 Texture Engine

A big reason why popsift[12] is such an efficient SIFT algorithm has a lot to do with its use of the CUDA texture engine. As I have explained earlier, a texture is essentially a GPU array, with its default

configuration being 2-dimensional. This layout of data is such a perfect fit for SIFT, and image processing purposes in general. After all, in a CPU implementation, an image will probably be stored as 2-Dimensional plane with each element containing pixel data anyway. This makes the translation and representation of images as a CUDA texture very easy.

As OpenACC does not expose the CUDA API directly, and neither does it implement a directive or API call for texture creation, there is simply no way to take advantage of this feature with OpenACC. Therefore we are forced to first create suitable data structures for storing 2-Dimensional planes, and copy these over to the accelerator. Then again, if what you wanted was to exactly call CUDA APIs through a high-level model, there are other options available [4, 14].

There are advantages to this limitation as well, one being that you will not be restricted to storing complex data as textures. Essentially, this allows the programmer to be a bit more flexible and not spend programming effort on converting certain complex data structures into structures which are suitable to be represented with a CUDA texture.

6.3.2 Unified/Managed Memory

In CUDA 6.0, a feature called Unified Memory or Managed Memory was introduced. The feature aimed to simplify memory management in CUDA programs. It achieves this by allowing the programmer to instead allocate data to one shared pool of memory. This memory will be available for use in both the host and device. Having this feature enabled has a profound impact on the overall ease-of-implementation of solutions using CUDA. Where you previously had to allocate memory on host, transfer over, and then transfer back again, you could now simply allocate to this shared pool, and access from both units. Usually, in combination with OpenACC this leads to an even greater degree of productivity, but in larger and more complex programs, the combination of the two might be problematic [10, Section 5]. This is due to complicated allocated data objects that may contain members that point to static data.

As I have already mentioned throughout the thesis, my implementation

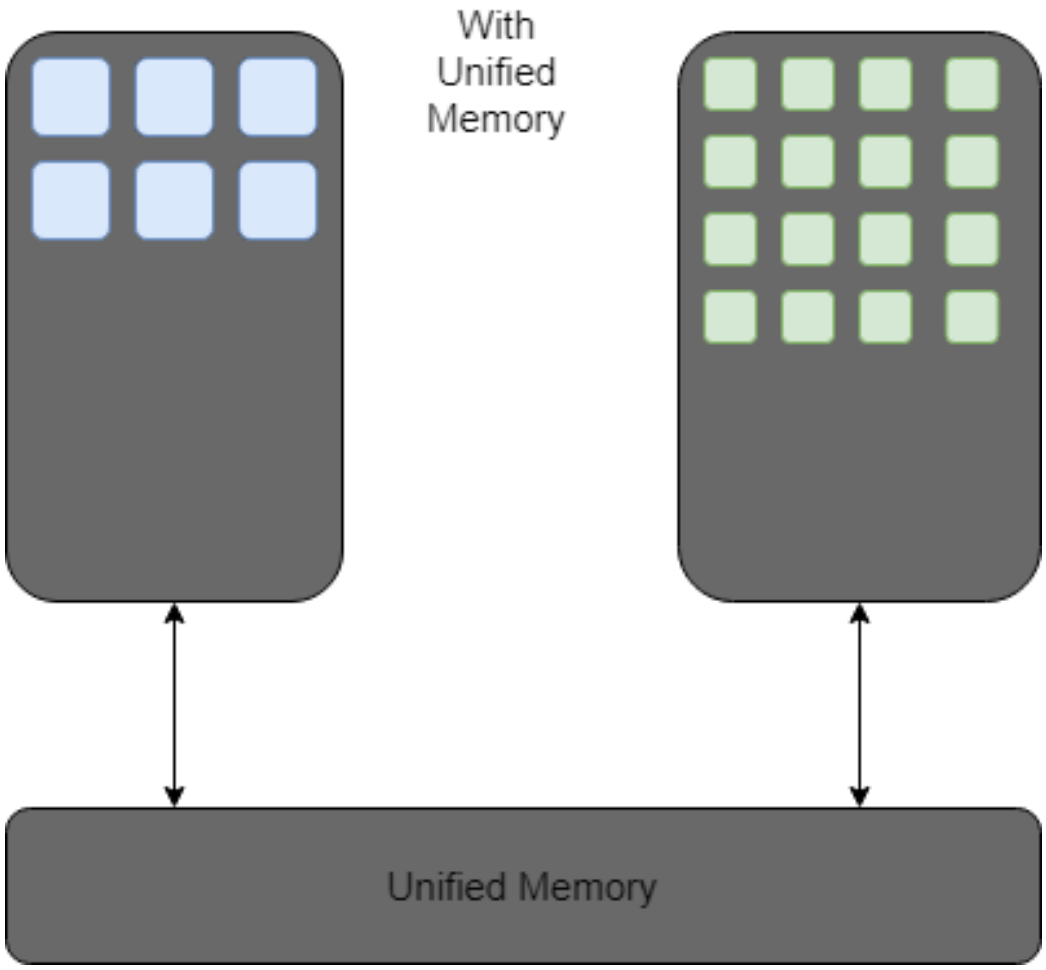

Figure 6.1: CUDA memory with unified memory

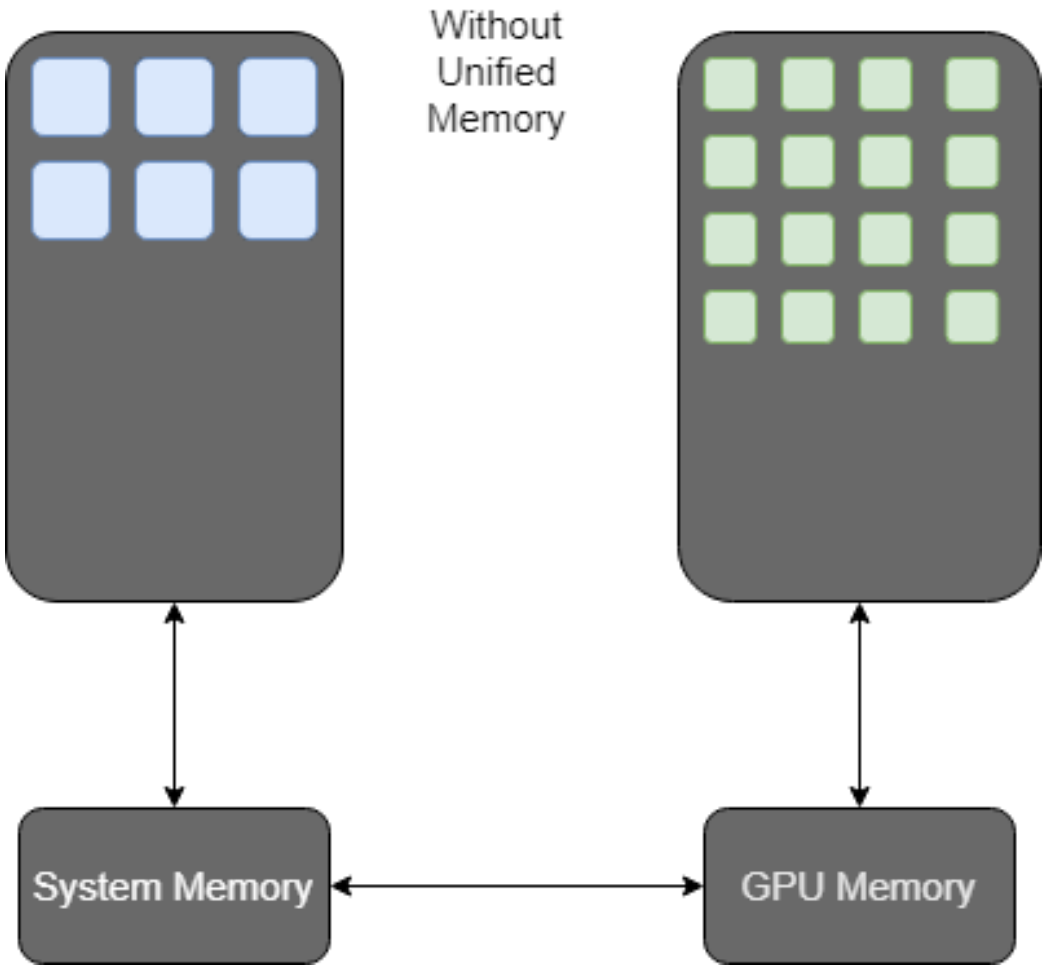

Figure 6.2: CUDA memory without unified memory

is written without CUDA unified memory enabled. This means that I had to be extra careful with handling data to and from the device. This does result in OpenACC requiring slightly more effort than strictly necessary for NVIDIA GPU devices, but may also yield higher performance [10] in some cases. Conversely, using unified memory without proper understanding of memory spaces and management will often result in performance loss due to the CUDA API attempting to do what it thinks is most optimal. However, as Knap&Czarnul[20] states in their paper, the use of Unified Memory may actually always be beneficial in some scenarios. These scenarios being multi-image processing with streams, as well as fluid dynamic simulations. In summary, the main areas where Unified Memory is more efficient, is when the amount computation time far outweighs the amount of memory transfers needed.

Even though i originally wanted to move away from CUDA altogether due to its vendor lock, the acquisition of the Portland Group by NVIDIA seems to have pushed yet more emphasis on CUDA in the GPGPU community. On the NVIDIA forums, there are plenty of cases where the approved answer to a problem is simply "have you tried turning on unified memory?". While convenient, I do think that this further corners the available and currently supported commercial compilers for GPGPU code through high-level models. Realistically, it seems like the two de facto frameworks are OpenCL for non-NVIDIA GPUs and CUDA.

Chapter 7

Conclusion

This thesis has explained and shown the different steps of the SIFT algorithm, given background on the CUDA and OpenACC programming models, and presented and discussed results from an accelerated version of the SIFT algorithm. I have also discussed and compared several different parallelization platforms, with portability being the highest weighted factor. By optimizing the SIFT algorithm with the parallel programming model OpenACC, this thesis has shown and discussed the performance potential and productivity potential of high-abstraction programming paradigms.

I have achieved the goal of RQ1 by analysing accelerated solutions with regards to not only my own solution, but several other solutions that accelerate their algorithms with a high-level parallel programming model. Through the use of profiling utilities I have optimized computational hotspots and presented the results from the optimization. By referring to relevant literature and studies, this thesis has explored how different computational problems have varying degrees of potential for parallelism. The proposed solution also retains a high degree of portability, while achieving a speedup factor of about 1.57.

The thesis also achieves the goal of RQ2 by quantifying and comparing the ease-of-implementation and productivity benefits of using the high-abstraction programming model OpenACC. This quantification is done by estimating ease-of-implementation as an equation relating the amount of total code to the amount of code needed to parallelize. The final

quantifiable result with the attained speedup included showed that the speedup per unit of programming effort was about 50% more with OpenACC than CUDA.

Chapter 8

Future work

8.1 Increased Optimization

There are several places in my proposed algorithm where there is room for improvement, namely the calculation of the gradient scale space pyramid, and more optimized gaussian pyramid creation and descriptor computation. If I were to suggest OpenACC as a replacement to CUDA in popsift, the solution would have to be refined a lot more. However, I did not write a solution with that goal in mind. However, it would be interesting to actually provide support for other parallel programming models other than CUDA in popsift. This would not need to even be OpenACC specifically, OpenCL and OpenMP are potential candidates for suitable frameworks.

8.2 Thorough Qualitative Performance Analysis

A more detailed and thorough analysis of the qualitative performance of my proposed solution is sorely needed. Ideally, this would be similar to the qualitative analysis in the popsift paper, where correctness measures such as true correspondence, repeatability rate and number of correct matches are evaluated. The reason this is not already included in the thesis is a matter of prioritization, as my main focus was on the conceptualization of the OpenACC model as well as productivity measurements.

8.3 Platform-Specific Optimizations

As I have stated several times in this master thesis, I prioritized portability over anything, and I therefore had to impose certain limitations of the actual performance portability of my solution. In the future, I would like to improve the solution by adding platform-specific optimizations which take effect when the program is run on a certain architecture. There are OpenACC API calls that do facilitate this according to the 2.7 specifications[1].

[1] https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf

Bibliography

[1] A.E. Abdel-Hakim and A.A. Farag. "CSIFT: A SIFT Descriptor with Color Invariant Characteristics". In: *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2 (CVPR'06)*. 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2 (CVPR'06). Vol. 2. New York, NY, USA: IEEE, 2006, pp. 1978–1983. ISBN: 978-0-7695-2597-6. DOI: 10.1109/CVPR.2006.95. URL: http://ieeexplore.ieee.org/document/1640995/ (visited on 04/29/2022).

[2] R. Arandjelovic and A. Zisserman. "Three things everyone should know to improve object retrieval". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition*. 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). Providence, RI: IEEE, June 2012, pp. 2911–2918. ISBN: 978-1-4673-1228-8. DOI: 10.1109/CVPR.2012.6248018. URL: http://ieeexplore.ieee.org/document/6248018/ (visited on 05/13/2022).

[3] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "SURF: Speeded Up Robust Features". In: *Computer Vision – ECCV 2006*. Ed. by Aleš Leonardis, Horst Bischof, and Axel Pinz. Vol. 3951. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 404–417. ISBN: 978-3-540-33832-1. DOI: 10.1007/11744023_32. URL: http://link.springer.com/10.1007/11744023_32 (visited on 04/29/2022).

[4] Nathan Bell and Jared Hoberock. "Thrust". In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 359–371. ISBN: 978-0-12-385963-1. DOI: 10.1016/B978-0-12-385963-1.00026-5. URL: https://linkinghub.elsevier.com/retrieve/pii/B9780123859631000265 (visited on 04/29/2022).

[5] M. Bicego et al. “On the Use of SIFT Features for Face Authentication”. In: *2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06)*. 2006 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'06). New York, NY, USA: IEEE, 2006, pp. 35–35. ISBN: 978-0-7695-2646-1. DOI: 10.1109/CVPRW.2006.149. URL: http://ieeexplore.ieee.org/document/1640475/ (visited on 04/29/2022).

[6] Mårten Björkman, Niklas Bergström, and Danica Kragic. “Detecting, segmenting and tracking unknown objects using multi-label MRF inference”. In: *Computer Vision and Image Understanding* 118 (Jan. 2014), pp. 111–127. ISSN: 10773142. DOI: 10.1016/j.cviu.2013.10.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S107731421300194X (visited on 04/21/2022).

[7] Giovanni Condello et al. “An OpenCL-based feature matcher”. en. In: *Signal Processing: Image Communication* 28.4 (Apr. 2013), pp. 345–350. ISSN: 09235965. DOI: 10.1016/j.image.2012.06.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0923596512001117 (visited on 06/01/2021).

[8] L. Dagum and R. Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE Computational Science and Engineering* 5.1 (Mar. 1998), pp. 46–55. ISSN: 10709924. DOI: 10.1109/99.660313. URL: http://ieeexplore.ieee.org/document/660313/ (visited on 04/29/2022).

[9] Patrick Daleiden, Andreas Stefik, and Philip Merlin Uesbeck. “GPU Programming Productivity in Different Abstraction Paradigms: A Randomized Controlled Trial Comparing CUDA and Thrust”. In: *ACM Transactions on Computing Education* 20.4 (Dec. 31, 2020), pp. 1–27. ISSN: 1946-6226. DOI: 10.1145/3418301. URL: https://dl.acm.org/doi/10.1145/3418301 (visited on 04/29/2022).

[10] Sebastien Deldon, James Beyer, and Douglas Miles. “OpenACC and CUDA unified memory”. In: *Proc. Cray User Group (CUG), Stockholm, Sweden* (2018).

[11] Hannes Fassold. “Computer vision on the GPU — Tools, algorithms and frameworks”. In: *2016 IEEE 20th Jubilee International Conference on Intelligent Engineering Systems (INES)*. Budapest,

Hungary: IEEE, June 2016, pp. 245–250. ISBN: 978-1-5090-1216-9. DOI: 10.1109/INES.2016.7555129. URL: http://ieeexplore.ieee.org/document/7555129/ (visited on 02/15/2021).

[12] Carsten Griwodz, Lilian Calvet, and Pål Halvorsen. "Popsift: a faithful SIFT implementation for real-time applications". en. In: *Proceedings of the 9th ACM Multimedia Systems Conference*. Amsterdam Netherlands: ACM, June 2018, pp. 415–420. ISBN: 978-1-4503-5192-8. DOI: 10.1145/3204949.3208136. URL: https://dl.acm.org/doi/10.1145/3204949.3208136 (visited on 02/11/2021).

[13] Carsten Griwodz et al. "AliceVision Meshroom: An open-source 3D reconstruction pipeline". In: *Proceedings of the 12th ACM Multimedia Systems Conference*. MMSys '21: 12th ACM Multimedia Systems Conference. Istanbul Turkey: ACM, June 24, 2021, pp. 241–247. ISBN: 978-1-4503-8434-6. DOI: 10.1145/3458305.3478443. URL: https://dl.acm.org/doi/10.1145/3458305.3478443 (visited on 04/29/2022).

[14] Tianyi David Han and Tarek S. Abdelrahman. "hiCUDA: High-Level GPGPU Programming". In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (Jan. 2011), pp. 78–90. ISSN: 1045-9219. DOI: 10.1109/TPDS.2010.62. URL: http://ieeexplore.ieee.org/document/5445082/ (visited on 04/21/2022).

[15] J. A. Herdman et al. "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 2012 SC Companion: High-Performance Computing, Networking, Storage and Analysis (SCC). Salt Lake City, UT: IEEE, Nov. 2012, pp. 465–471. ISBN: 978-0-7695-4956-9. DOI: 10.1109/SC.Companion.2012.66. URL: http://ieeexplore.ieee.org/document/6495848/ (visited on 04/29/2022).

[16] S Heymann et al. "SIFT implementation and optimization for general-purpose GPU". In: (Jan. 2007). URL: https://www.researchgate.net/publication/228622667_SIFT_implementation_and_optimization_for_general-purpose_GPU (visited on 04/29/2022).

[17] T. Hoshino et al. "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application". In: *2013 13th IEEE/ACM International Symposium on Cluster,*

Cloud, and Grid Computing. 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). Delft: IEEE, May 2013, pp. 136–143. ISBN: 978-0-7695-4996-5. DOI: 10.1109/CCGrid.2013.12. URL: http://ieeexplore.ieee.org/document/6546071/ (visited on 04/29/2022).

[18] David Hutchison et al. "BRIEF: Binary Robust Independent Elementary Features". In: *Computer Vision – ECCV 2010*. Ed. by Kostas Daniilidis, Petros Maragos, and Nikos Paragios. Vol. 6314. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 778–792. ISBN: 978-3-642-15560-4. DOI: 10.1007/978-3-642-15561-1_56. URL: http://link.springer.com/10.1007/978-3-642-15561-1_56 (visited on 04/29/2022).

[19] Liu Jianfang, Zheng Hao, and Gao Jingli. "A novel fast target tracking method for UAV aerial image". In: *Open Physics* 15.1 (June 2017), pp. 420–426. ISSN: 2391-5471. DOI: 10.1515/phys-2017-0046. URL: https://www.degruyter.com/document/doi/10.1515/phys-2017-0046/html (visited on 05/31/2021).

[20] Marcin Knap and Paweł Czarnul. "Performance evaluation of Unified Memory with prefetching and oversubscription for selected parallel CUDA applications on NVIDIA Pascal and Volta GPUs". In: *The Journal of Supercomputing* 75.11 (Nov. 2019), pp. 7625–7645. ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-019-02966-8. URL: http://link.springer.com/10.1007/s11227-019-02966-8 (visited on 05/13/2022).

[21] Seyong Lee and Jeffrey S. Vetter. "OpenARC: open accelerator research compiler for directive-based, efficient heterogeneous computing". In: *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing - HPDC '14*. the 23rd international symposium. Vancouver, BC, Canada: ACM Press, 2014, pp. 115–120. ISBN: 978-1-4503-2749-7. DOI: 10.1145/2600212.2600704. URL: http://dl.acm.org/citation.cfm?doid=2600212.2600704 (visited on 04/29/2022).

[22] Xuechao Li et al. "Comparing Programmer Productivity in Openacc and Cuda : An Empirical Investigation". In: *International*

Journal of Computer Science, Engineering and Applications 6.5 (Oct. 31, 2016), pp. 1–15. ISSN: 22310088, 22309616. DOI: 10.5121/ijcsea.2016.6501. URL: http://www.aircconline.com/ijcsea/V6N5/6516ijcsea01.pdf (visited on 04/29/2022).

[23] Tony Lindeberg. "Scale-space theory: a basic tool for analyzing structures at different scales". In: *Journal of Applied Statistics* 21.1 (Jan. 1994), pp. 225–270. ISSN: 0266-4763, 1360-0532. DOI: 10.1080/757582976. URL: https://www.tandfonline.com/doi/full/10.1080/757582976 (visited on 04/21/2022).

[24] Yu Liu, Shuping Liu, and Zengfu Wang. "Multi-focus image fusion with dense SIFT". In: *Information Fusion* 23 (May 2015), pp. 139–155. ISSN: 15662535. DOI: 10.1016/j.inffus.2014.05.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S1566253514000670 (visited on 04/29/2022).

[25] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". en. In: *International Journal of Computer Vision* 60.2 (Nov. 2004), pp. 91–110. ISSN: 0920-5691. DOI: 10.1023/B:VISI.0000029664.99615.94. URL: http://link.springer.com/10.1023/B:VISI.0000029664.99615.94 (visited on 05/31/2021).

[26] David G. Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Proceedings of the Seventh IEEE International Conference on Computer Vision. Kerkyra, Greece: IEEE, 1999, 1150–1157 vol.2. ISBN: 978-0-7695-0164-2. DOI: 10.1109/ICCV.1999.790410. URL: http://ieeexplore.ieee.org/document/790410/ (visited on 04/21/2022).

[27] Jun Luo et al. "Person-Specific SIFT Features for Face Recognition". In: *2007 IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '07*. 2007 IEEE International Conference on Acoustics, Speech, and Signal Processing. Honolulu, HI: IEEE, Apr. 2007, pp. II–593–II–596. ISBN: 978-1-4244-0727-9. DOI: 10.1109/ICASSP.2007.366305. URL: https://ieeexplore.ieee.org/document/4217478/ (visited on 04/29/2022).

[28] Pieter Maris et al. “Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on GPUs using OpenACC”. In: *Journal of Computational Science* 59 (Mar. 2022), p. 101554. ISSN: 18777503. DOI: 10.1016/j.jocs.2021.101554. URL: https://linkinghub.elsevier.com/retrieve/pii/S1877750321002076 (visited on 05/13/2022).

[29] Maxime Martelli et al. “GPU Acceleration : OpenACC for Radar Processing Simulation”. In: *2019 International Radar Conference (RADAR)*. 2019 International Radar Conference (RADAR). TOULON, France: IEEE, Sept. 2019, pp. 1–6. ISBN: 978-1-72812-660-9. DOI: 10.1109/RADAR41533.2019.171296. URL: https://ieeexplore.ieee.org/document/9078942/ (visited on 05/13/2022).

[30] Suejb Memeti et al. “Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: Programming Productivity, Performance, and Energy Consumption”. In: *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing - ARMS-CC '17*. the 2017 Workshop. Washington, DC, USA: ACM Press, 2017, pp. 1–6. ISBN: 978-1-4503-5116-4. DOI: 10.1145/3110355.3110356. URL: http://dl.acm.org/citation.cfm?doid=3110355.3110356 (visited on 04/29/2022).

[31] Konrad Moren and Diana Göhringer. “A framework for accelerating local feature extraction with OpenCL on multi-core CPUs and co-processors”. In: *Journal of Real-Time Image Processing* 16.4 (Aug. 2019), pp. 901–918. ISSN: 1861-8200, 1861-8219. DOI: 10.1007/s11554-016-0576-0. URL: http://link.springer.com/10.1007/s11554-016-0576-0 (visited on 05/02/2022).

[32] E.N. Mortensen, Hongli Deng, and L. Shapiro. “A SIFT Descriptor with Global Context”. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05). Vol. 1. San Diego, CA, USA: IEEE, 2005, pp. 184–190. ISBN: 978-0-7695-2372-9. DOI: 10.1109/CVPR.2005.45. URL: http://ieeexplore.ieee.org/document/1467266/ (visited on 04/29/2022).

[33] Moein Mozaffarzadeh et al. “OpenACC GPU implementation of double-stage delay-multiply-and-sum algorithm: toward enhanced real-time linear-array photoacoustic tomography”. In: *Photons Plus Ultrasound: Imaging and Sensing 2019*. Photons Plus Ultrasound: Imaging and Sensing 2019. Ed. by Alexander A. Oraevsky and Lihong V. Wang. San Francisco, United States: SPIE, Feb. 27, 2019, p. 195. ISBN: 978-1-5106-2398-9. DOI: 10.1117/12.2511115. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/10878/2511115/OpenACC-GPU-implementation-of-double-stage-delay-multiply-and-sum/10.1117/12.2511115.full (visited on 05/13/2022).

[34] Aaftab Munshi. “The OpenCL specification”. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009 IEEE Hot Chips 21 Symposium (HCS). Stanford, CA: IEEE, Aug. 2009, pp. 1–314. ISBN: 978-1-4673-8873-3. DOI: 10.1109/HOTCHIPS.2009.7478342. URL: http://ieeexplore.ieee.org/document/7478342/ (visited on 04/29/2022).

[35] NVIDIA. *NVIDIA A100 Tensor Core GPU Architecture*. 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf (visited on 04/22/2022).

[36] NVIDIA. *NVIDIA CUDA Toolkit Release Notes*. 2022. URL: https://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html (visited on 04/22/2022).

[37] NVIDIA. *NVIDIA GeForce GTX 1080*. 2016. URL: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf (visited on 04/22/2022).

[38] NVIDIA. *NVIDIA TESLA V100 GPU ARCHITECTURE*. 2017. URL: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (visited on 04/22/2022).

[39] NVIDIA. *NVIDIA Turing Architecture Whitepaper*. 2018. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf (visited on 04/22/2022).

[40] openacc-standard.org. *OpenACC Programming and Best Practices Guide*. 2021. URL: https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0_0.pdf (visited on 04/29/2022).

[41] OpenACC.org. *OpenACC homepage*. 2022. URL: http://openacc.org (visited on 04/22/2022).

[42] Evelyn Otero et al. "OpenACC acceleration for the P N – P N - 2 algorithm in Nek5000". In: *Journal of Parallel and Distributed Computing* 132 (Oct. 2019), pp. 69–78. ISSN: 07437315. DOI: 10.1016/j.jpdc.2019.05.010. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731518305549 (visited on 05/13/2022).

[43] Jon Peddie. *Pandemic distorts global GPU market results*. 2020. URL: https://www.jonpeddie.com/press-releases/pandemic-distorts-global-gpu-market-results (visited on 04/22/2022).

[44] Florian Rançon et al. "Comparison of SIFT Encoded and Deep Learning Features for the Classification and Detection of Esca Disease in Bordeaux Vineyards". en. In: *Remote Sensing* 11.1 (Dec. 2018), p. 1. ISSN: 2072-4292. DOI: 10.3390/rs11010001. URL: http://www.mdpi.com/2072-4292/11/1/1 (visited on 05/31/2021).

[45] Ari Rasch et al. "dOCAL: high-level distributed programming with OpenCL and CUDA". In: *The Journal of Supercomputing* 76.7 (July 2020), pp. 5117–5138. ISSN: 0920-8542, 1573-0484. DOI: 10.1007/s11227-019-02829-2. URL: http://link.springer.com/10.1007/s11227-019-02829-2 (visited on 04/29/2022).

[46] Ives Rey Otero and Mauricio Delbracio. "Anatomy of the SIFT Method". In: *Image Processing On Line* 4 (Dec. 22, 2014), pp. 370–396. ISSN: 2105-1232. DOI: 10.5201/ipol.2014.82. URL: https://www.ipol.im/pub/art/2014/82/?utm_source=doi (visited on 05/13/2022).

[47] Ethan Rublee et al. "ORB: An efficient alternative to SIFT or SURF". In: *2011 International Conference on Computer Vision*. 2011 IEEE International Conference on Computer Vision (ICCV). Barcelona, Spain: IEEE, Nov. 2011, pp. 2564–2571. ISBN: 978-1-4577-1102-2. DOI: 10.1109/ICCV.2011.6126544. URL: http://ieeexplore.ieee.org/document/6126544/ (visited on 04/29/2022).

[48] Amit Sabne et al. “Evaluating Performance Portability of OpenACC”. In: *Languages and Compilers for Parallel Computing*. Ed. by James Brodman and Peng Tu. Vol. 8967. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 51–66. ISBN: 978-3-319-17473-0. DOI: 10.1007/978-3-319-17473-0_4. URL: http://link.springer.com/10.1007/978-3-319-17473-0_4 (visited on 04/29/2022).

[49] Robert Searles et al. “MPI + OpenACC: Accelerating radiation transport mini-application, minisweep, on heterogeneous systems”. In: *Computer Physics Communications* 236 (Mar. 2019), pp. 176–187. ISSN: 00104655. DOI: 10.1016/j.cpc.2018.10.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S0010465518303552 (visited on 05/13/2022).

[50] H. Shao, T. Svoboda, and L. Van Gool. *Datasets - Computer Vision Group*. 2003. URL: https://icu.ee.ethz.ch/research/datsets.html (visited on 01/20/2022).

[51] B. Sirmacek and C. Unsalan. “Urban-Area and Building Detection Using SIFT Keypoints and Graph Theory”. In: *IEEE Transactions on Geoscience and Remote Sensing* 47.4 (Apr. 2009), pp. 1156–1167. ISSN: 0196-2892, 1558-0644. DOI: 10.1109/TGRS.2008.2008440. URL: http://ieeexplore.ieee.org/document/4799121/ (visited on 04/29/2022).

[52] George Teodoro et al. “Coordinating the use of GPU and CPU for improving performance of compute intensive applications”. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009 IEEE International Conference on Cluster Computing and Workshops. New Orleans, LA, USA: IEEE, 2009, pp. 1–10. ISBN: 978-1-4244-5011-4. DOI: 10.1109/CLUSTR.2009.5289193. URL: http://ieeexplore.ieee.org/document/5289193/ (visited on 04/29/2022).

[53] Seth Warn et al. “Accelerating SIFT on parallel architectures”. In: *2009 IEEE International Conference on Cluster Computing and Workshops*. 2009 IEEE International Conference on Cluster Computing and Workshops. New Orleans, LA, USA: IEEE, 2009, pp. 1–4. ISBN: 978-1-4244-5011-4. DOI: 10.1109/CLUSTR.2009.

5289155. URL: http://ieeexplore.ieee.org/document/5289155/ (visited on 05/13/2022).

[54] Sandra Wienke et al. "OpenACC — First Experiences with Real-World Applications". In: *Euro-Par 2012 Parallel Processing*. Ed. by Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis. Red. by David Hutchison et al. Vol. 7484. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–870. ISBN: 978-3-642-32820-6. DOI: 10.1007/978-3-642-32820-6_85. URL: http://link.springer.com/10.1007/978-3-642-32820-6_85 (visited on 04/29/2022).

[55] Wanglong Yan et al. "Computing OpenSURF on OpenCL and General Purpose GPU". en. In: *International Journal of Advanced Robotic Systems* 10.10 (Oct. 2013), p. 375. ISSN: 1729-8814, 1729-8814. DOI: 10.5772/57057. URL: http://journals.sagepub.com/doi/10.5772/57057 (visited on 06/01/2021).

[56] Yan Ke and R. Sukthankar. "PCA-SIFT: a more distinctive representation for local image descriptors". In: *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.* Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004. Vol. 2. Washington, DC, USA: IEEE, 2004, pp. 506–513. ISBN: 978-0-7695-2158-9. DOI: 10.1109/CVPR.2004.1315206. URL: http://ieeexplore.ieee.org/document/1315206/ (visited on 04/29/2022).

[57] Yi Yang et al. "CPU-assisted GPGPU on fused CPU-GPU architectures". In: *IEEE International Symposium on High-Performance Comp Architecture*. 2012 IEEE 18th International Symposium on High Performance Computer Architecture (HPCA). New Orleans, LA, USA: IEEE, Feb. 2012, pp. 1–12. ISBN: 978-1-4673-0825-0. DOI: 10.1109/HPCA.2012.6168948. URL: http://ieeexplore.ieee.org/document/6168948/ (visited on 04/29/2022).

[58] Huiyu Zhou, Yuan Yuan, and Chunmei Shi. "Object tracking using SIFT features and mean shift". In: *Computer Vision and Image Understanding* 113.3 (Mar. 2009), pp. 345–352. ISSN: 10773142. DOI:

10.1016/j.cviu.2008.08.006. URL: https://linkinghub.elsevier.com/retrieve/pii/S1077314208001331 (visited on 04/29/2022).

Appendix A

Tables

CUDA Driver Version:	11060
NVRM version:	Kernel Module 510.47.03
Device Number:	0
Device Name:	NVIDIA GeForce GTX 1080 Ti
Device Revision Number:	6.1
Global Memory Size:	11713118208
Number of Multiprocessors:	28
Concurrent Copy and Execution:	Yes
Total Constant Memory:	65536
Total Shared Memory per Block:	49152
Registers per Block:	65536
Warp Size:	32
Maximum Threads per Block:	1024
Maximum Block Dimensions:	1024, 1024, 64
Maximum Grid Dimensions:	2147483647 x 65535 x 65535
Maximum Memory Pitch:	2147483647B
Texture Alignment:	512B
Clock Rate:	1607 MHz
Execution Timeout:	Yes
Integrated Device:	No
Can Map Host Memory:	Yes
Compute Mode:	default
Concurrent Kernels:	Yes
ECC Enabled:	No
Memory Clock Rate:	5505 MHz
Memory Bus Width:	352 bits
L2 Cache Size:	2883584 bytes
Max Threads Per SMP:	2048
Async Engines:	2
Unified Addressing:	Yes
Managed Memory:	Yes
Concurrent Managed Memory:	Yes
Preemption Supported:	Yes
Cooperative Launch:	Yes
Multi-Device:	Yes
Default Target:	cc61

Table A.1: CUDA Compute Capability of the testing machine

Appendix B

Images

Figure B.1: object0001.view01.png

Appendix C

Code

C.1 Naive Parallel Gaussian Blur Function

```
// separable 2D gaussian blur for 1 channel image
Image gaussian_blur(const Image& img, float sigma)
{
    assert(img.channels == 1);
    bool failed = false;

    #pragma acc enter data copyin(img, img.data[:img.width*img.
    height])

    int size = std::ceil(6 * sigma);
    if (size % 2 == 0)
        size++;
    int center = size / 2;
    Image kernel(size, 1, 1);
    #pragma acc enter data copyin(kernel)
    float sum = 0;
    for (int k = -size/2; k <= size/2; k++) {
        float val = std::exp(-(k*k) / (2*sigma*sigma));
        kernel.set_pixel(center+k, 0, 0, val);
        sum += val;
    }
    for (int k = 0; k < size; k++) {
        kernel.data[k] /= sum;
    }
    #pragma acc enter data copyin(kernel.data[:size])

```

```
26      int sz = img.width * img.height;
27      Image tmp(img.width, img.height, 1);
28      Image filtered(img.width, img.height, 1);
29
30      float * filt = new float[sz];
31      float * temps = new float[sz];
32
33      #pragma acc parallel loop independent collapse(2) present(
        kernel, img) copyout(temps[:sz])
34      for (int x = 0; x < img.width; x++) {
35          for (int y = 0; y < img.height; y++) {
36              float sum = 0;
37              #pragma acc loop reduction(+:sum)
38              for (int k = 0; k < size; k++) {
39                  int dy = -center + k;
40                  sum += img.get_pixel(x, y+dy, 0) * kernel.data[k
        ];
41              }
42              /* tmp.set_pixel_routine(x, y, 0, sum, failed); */
43              temps[ y*img.width + x] = sum;
44          }
45      }
46
47      std::copy(temps, temps + sz, tmp.data);
48  /*      #pragma acc update self(tmp.data[:sz]) */
49
50      #pragma acc parallel loop independent collapse(2) present(
        kernel) copyin(tmp,tmp.data[:sz]) copyout(filt[:sz])
51      for (int x = 0; x < img.width; x++) {
52          for (int y = 0; y < img.height; y++) {
53              float sum = 0;
54              #pragma acc loop reduction(+:sum)
55              for (int k = 0; k < size; k++) {
56                  int dx = -center + k;
57                  sum += tmp.get_pixel(x+dx, y, 0) * kernel.data[k
        ];
58              }
59  /*              filtered.set_pixel_routine(x, y, 0, sum,failed);
        */
60                  filt[ y*img.width + x] = sum;
61              }
62          }
63      #pragma acc exit data delete(kernel, kernel.data[:size])
```

```
64      #pragma acc exit data delete(img, img.data[:img.width*img.
    height])
65      std::copy(filt, filt + sz, filtered.data);
66      delete [] filt;
67      delete [] temps;
68      return filtered;
69  }
```

Listing C.1: Gaussian Blur

C.2 generate_gaussian_pyramid

```
1   ScaleSpacePyramid generate_gaussian_pyramid(const Image& img,
        float sigma_min,
2                                               int num_octaves, int
        scales_per_octave)
3   {
4       // assume initial sigma is 1.0 (after resizing) and smooth
5       // the image with sigma_diff to reach requried base_sigma
6       float base_sigma = sigma_min / MIN_PIX_DIST;
7
8       clock_t timer = clock();
9       Image base_img = img.resize(img.width*2, img.height*2,
        Interpolation::BILINEAR);
10      float resizing = ((float)(clock() - timer))  /
        CLOCKS_PER_SEC;
11
12      timer = clock();
13      float sigma_diff = std::sqrt(base_sigma*base_sigma - 1.0f);
14      base_img = gaussian_blur(base_img, sigma_diff);
15      float blur_base = ((float)(clock() - timer))  /
        CLOCKS_PER_SEC;
16
17      if (base_img.height == 0 || base_img.width == 0 || base_img.
        channels == 0) {
18              std::cerr << "***Blurring bad***\n";
19              std::exit(1);
20      }
21
22      int imgs_per_octave = scales_per_octave + 3;
23
24      // determine sigma values for bluring
25      timer = clock();
```

```
26      float k = std::pow(2, 1.0/scales_per_octave);
27      std::vector<float> sigma_vals {base_sigma};
28      for (int i = 1; i < imgs_per_octave; i++) {
29          float sigma_prev = base_sigma * std::pow(k, i-1);
30          float sigma_total = k * sigma_prev;
31          sigma_vals.push_back(std::sqrt(sigma_total*sigma_total -
        sigma_prev*sigma_prev));
32      }
33      float determine_sigma = ((float)(clock() - timer))  /
    CLOCKS_PER_SEC;
34
35      // create a scale space pyramid of gaussian images
36      // images in each octave are half the size of images in the
    previous one
37      timer = clock();
38      ScaleSpacePyramid pyramid = {
39          num_octaves,
40          imgs_per_octave,
41          std::vector<std::vector<Image>>(num_octaves)
42
43      };
44      for (int i = 0; i < num_octaves; i++) {
45          pyramid.octaves[i].reserve(imgs_per_octave);
46          pyramid.octaves[i].push_back(std::move(base_img));
47          for (int j = 1; j < sigma_vals.size(); j++) {
48              const Image& prev_img = pyramid.octaves[i].back();
49              if (prev_img.height == 0 || prev_img.width == 0 ||
    prev_img.channels == 0) {
50                  std::cerr << "***Image something wrong***\n";
51              }
52              pyramid.octaves[i].push_back(gaussian_blur(prev_img,
        sigma_vals[j]));
53          }
54          // prepare base image for next octave
55          const Image& next_base_img = pyramid.octaves[i][
    imgs_per_octave-3];
56          base_img = next_base_img.resize(next_base_img.width/2,
    next_base_img.height/2,
57                                          Interpolation::NEAREST);
58      }
59      float scale_space = ((float)(clock() - timer))  /
    CLOCKS_PER_SEC;
60      std::cout << "Resizing:" << resizing << " BlurBase:" <<
```

```
    blur_base << " DetSigma:" << determine_sigma << " ScaleSpace:
    " << scale_space << "\n";
61     return pyramid;
62 }
```

Listing C.2: Gaussian Pyramid

C.3 Naive Parallel Feature Descriptor Computation

```
1  #pragma acc routine
2  void update_histograms(float hist[N_HIST][N_HIST][N_ORI], float
       x, float y,
3                         float contrib, float theta_mn, float
       lambda_desc)
4  {
5      float x_i, y_j;
6      for (int i = 1; i <= N_HIST; i++) {
7          x_i = (i-(1+(float)N_HIST)/2) * 2*lambda_desc/N_HIST;
8          if (std::abs(x_i-x) > 2*lambda_desc/N_HIST)
9              continue;
10         for (int j = 1; j <= N_HIST; j++) {
11             y_j = (j-(1+(float)N_HIST)/2) * 2*lambda_desc/N_HIST
       ;
12             if (std::abs(y_j-y) > 2*lambda_desc/N_HIST)
13                 continue;
14
15             float hist_weight = (1 - N_HIST*0.5/lambda_desc*std
       ::abs(x_i-x))
16                                 *(1 - N_HIST*0.5/lambda_desc*std
       ::abs(y_j-y));
17
18             for (int k = 1; k <= N_ORI; k++) {
19                 float theta_k = 2*M_PI*(k-1)/N_ORI;
20                 float theta_diff = std::fmod(theta_k-theta_mn+2*
       M_PI, 2*M_PI);
21                 if (std::abs(theta_diff) >= 2*M_PI/N_ORI)
22                     continue;
23                 float bin_weight = 1 - N_ORI*0.5/M_PI*std::abs(
       theta_diff);
24                 #pragma acc atomic update
```

```
                hist[i-1][j-1][k-1] += hist_weight*bin_weight*
    contrib;
            }
        }
    }
}

void hists_to_vec(float histograms[N_HIST][N_HIST][N_ORI], std::
    array<float, 128>& feature_vec)
{
    int size = N_HIST*N_HIST*N_ORI;
    float *hist = reinterpret_cast<float *>(histograms);

    float norm = 0;
    for (int i = 0; i < size; i++) {
        norm += hist[i] * hist[i];
    }
    norm = std::sqrt(norm);
    float norm2 = 0;
    for (int i = 0; i < size; i++) {
        hist[i] = std::min(hist[i], 0.2f*norm);
        norm2 += hist[i] * hist[i];
    }
    norm2 = std::sqrt(norm2);
    for (int i = 0; i < size; i++) {
/*          float val = std::floor(512*hist[i]/norm2); */
        float val = (512*hist[i]/norm2);
        feature_vec[i] = std::min(val, (float)255);
    }
}
void compute_keypoint_descriptor(Keypoint& kp, float theta,
                                 const ScaleSpacePyramid&
    grad_pyramid,
                                 float lambda_desc)
{

    #pragma acc enter data copyin(kp)
    float pix_dist = MIN_PIX_DIST * std::pow(2, kp.octave);
    const Image& img_grad = grad_pyramid.octaves[kp.octave][kp.
    scale];

    float histograms[N_HIST][N_HIST][N_ORI] = {0};

```

```
64      int h = img_grad.height, w = img_grad.width;
65      int sz = w*h * img_grad.channels;
66
67      //find start and end coords for loops over image patch
68      float half_size = std::sqrt(2)*lambda_desc*kp.sigma*(N_HIST
    +1.)/N_HIST;
69      int x_start = std::round((kp.x-half_size) / pix_dist);
70      int x_end = std::round((kp.x+half_size) / pix_dist);
71      int y_start = std::round((kp.y-half_size) / pix_dist);
72      int y_end = std::round((kp.y+half_size) / pix_dist);
73
74
75      #pragma acc enter data copyin(img_grad, img_grad.data[:sz])
76
77      float cos_t = std::cos(theta), sin_t = std::sin(theta);
78      float patch_sigma = lambda_desc * kp.sigma;
79      //accumulate samples into histograms
80      #pragma acc parallel loop  independent collapse(2) present(
    img_grad) copy(histograms[:N_HIST][:N_HIST][:N_ORI])
81      for (int m = x_start; m <= x_end; m++) {
82          for (int n = y_start; n <= y_end; n++) {
83              // find normalized coords w.r.t. kp position and
    reference orientation
84              float x = ((m*pix_dist - kp.x)*cos_t
85                      +(n*pix_dist - kp.y)*sin_t) / kp.sigma;
86              float y = (-(m*pix_dist - kp.x)*sin_t
87                      +(n*pix_dist - kp.y)*cos_t) / kp.sigma;
88
89              // verify (x, y) is inside the description patch
90              if (std::max(std::abs(x), std::abs(y)) > lambda_desc
    *(N_HIST+1.)/N_HIST)
91                  continue;
92              float gx = img_grad.get_pixel_routine(m, n, 0), gy =
     img_grad.get_pixel_routine(m, n, 1);
93              float theta_mn = std::fmod(std::atan2(gy, gx)-theta
    +4*M_PI, 2*M_PI);
94              float grad_norm = std::sqrt(gx*gx + gy*gy);
95              float weight = std::exp(-(std::pow(m*pix_dist-kp.x,
    2)+std::pow(n*pix_dist-kp.y, 2))
96                                      /(2*patch_sigma*
    patch_sigma));
97              float contribution = weight * grad_norm;
98              update_histograms(histograms, x, y, contribution,
```

```
    theta_mn, lambda_desc);
99              }
100     }
101
102     // build feature vector (descriptor) from histograms
103     hists_to_vec(histograms, kp.descriptor);
104
105     #pragma acc exit data delete(kp)
106     #pragma acc exit data delete(img_grad, img_grad.data[:sz])
107 }
```

Listing C.3: Feature Descriptor Computation