Low Complexity Feature Point Detection and Tracking using CUDA

Rajat Phull
Low Complexity Feature Point Detection and Tracking using CUDA

Master’s Thesis in Computer Engineering

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, Netherlands

NVision group
Smart Systems and Energy Technology, Interuniversity Microelectronics Centre (IMEC), Belgium

Supervisors

Henk Sips
Chair and Professor, Parallel and Distributed Group, Delft University of Technology

Qiong Yang
Senior Researcher, NVision group, IMEC

Rajat Phull

13th July 2010
Author
Rajat Phull

Title
Low Complexity Feature Point Detection and Tracking using CUDA

MSc presentation
14th July 2010

Graduation Committee
Prof. dr. ir. H. J. Sips (chair) Delft University of Technology
Dr. Q. Yang Interuniversitair Micro-Elektronica Centrum
Dr. ir. E. Hendriks Delft University of Technology
Abstract

High speed feature point detection and tracking is very demanding for many real-time computer vision applications. In existing work, the commonly used feature point detection algorithms like Harris and KLT (Kanade-Lucas-Tomasi) and feature tracking algorithm (Pyramidal-KLT) were redesigned to increase the performance by reducing the algorithmic complexity, resulting in the Low Complexity Corner detector (LOCOCO) and Robust Low Complexity Feature tracking (RLCT) algorithms. To attain further speedup, this report proposes the implementation of these low complexity detection and tracking algorithms on a massively parallel architecture of the modern graphics processing units (GPUs) using Compute Unified Device Architecture (CUDA).

In the computing domain, due to semiconductor scaling limits and associated power and thermal challenges, combined with the difficulty of exploiting greater levels of instruction level parallelism, a paradigm shift is happening from a single core to many-core processors and massively multi-processing platforms. High performance is now available on single-chip commodity GPUs. Moreover, GPUs are no longer limited to graphics applications, but are emerging as usable general purpose computing devices. Advancement in such platforms, are making many computational intensive problems that were solvable only on supercomputing systems, to be computed on desktop systems, at a reduced price, and lower power requirements. The arrival of this new generation of low-cost high performance computing platforms presents both numerous opportunities and challenges.

In this report, we present the use of such high performance many-core GPU platforms to obtain speedup by mapping general purpose computations to massively parallel architectures. It is observed, when properly executed, GPU adaptation of algorithms can result in significant savings in computation times. For an image size of 960x960 pixels, the low complexity corner detector and robust low complexity feature tracking algorithms are factor of 16 and 25 times faster on a GeForce 280 GTX GPU than the corresponding implementation on an Intel Core 2 Duo, 2.66 GHz, and 2GB RAM CPU.
## Contents

1 Introduction 1
   1.1 Contributions ............................................. 3
   1.2 Organization ............................................. 3

2 Applications 5
   2.1 Panoramic Mosaics from Video Sequences .............. 6
   2.2 Computation of camera motion and 3D reconstruction of a scene .... 7

3 Hybrid CPU-GPU systems 9
   3.1 GPGPU programming ...................................... 10
   3.2 CUDA .................................................. 12
   3.3 GPU Architecture ..................................... 13
   3.4 Hardware specificities .................................. 14
      3.4.1 CPU-GPU communication overhead .................. 14
      3.4.2 Bank conflicts ..................................... 14
      3.4.3 Control Flow overhead .............................. 15
   3.5 CUDA Good Practice Rules for Building Optimized Code ..... 15

4 Feature Detector 17
   4.1 Harris and KLT Corner Detector ......................... 17
   4.2 Low Complexity Corner Detector Algorithm ............ 19
      4.2.1 Integral Image ..................................... 19
      4.2.2 Gaussian Derivative .............................. 20
      4.2.3 Cornerness Response .............................. 20
      4.2.4 Non-Maximum Suppression ......................... 21
   4.3 Mapping Low Complexity Corner Detector on GPU ...... 22
      4.3.1 Parallelize Low Complexity Corner Detector algorithm .... 22
      4.3.2 CUDA Kernels for Low Complexity Corner Detector .... 23
   4.4 Experimental Results and Discussion ................... 29

5 Feature Tracking 35
   5.1 Kanade-Lucas-Tomasi Algorithm ......................... 35
   5.2 Pyramidal Kanade-Lucas-Tomasi Algorithm ............. 37
   5.3 Robust Low Complexity Tracking Algorithm ............ 37
5.3.1 Coarse to fine feature tracking ...................................... 38
5.3.2 Initializing the homography and predicting the feature location .................................................. 38
5.4 Mapping Robust Low Complexity Tracking on GPU .......... 39
5.4.1 Parallelize Robust Low Complexity Tracking algorithm . 39
5.4.2 Thread organization and memory access pattern .......... 40
5.4.3 CUDA Kernels for feature point tracking ................. 40
5.5 Experimental Results and Discussion ......................... 43
5.6 Application to Image Registration ......................... 46

6 Conclusion and Future Work ........................................... 49

List of Publications .............................................. 53
Chapter 1

Introduction

Real-time feature detection and tracking is a fundamental problem of many computer vision applications such as robot navigation, mosaicing, video stabilization etc. For instance, as shown in figure 1.1, the detection and tracking of feature points in successive endoscopic images captured during laparoscopic surgery is the basic processing step for image registration, realignment or stabilization. It may eventually be used in multi-image stitching for a panoramic viewing experience, providing a higher level of ergonomics in delicate, life-saving surgery practices.

Figure 1.1: Feature tracking on laparoscopic medical images (a) Feature point tracking with RLCT algorithm (green = inlier, red= outlier) (b) image registration

Feature detection and tracking being a low-level, intensive DSP processing step applied on millions/billions of pixels per second. The target of achieving real-time and/or low/moderate-power video tracking processing in constrained application setups (autonomous/mobile cameras, no active cooling for DSP processing in surgery rooms, affordable video grid computing, etc.) is still challenging today.

Both the Harris [13] and KLT [7] algorithms are the most widely used corner detection algorithms. Several implementations of these algorithms exist on GPUs. Sinha et. al. [33] proposed an implementation of the KLT corner detector for GPUs. However, the last step of non-maximum suppression of the cornerness response was performed on the CPU; this limits the potential speedup that can be
obtained by this corner detector algorithm implementation. Teixeira et. al [34] proposed their own implementation of non-maximum suppression for GPUs. They attained a significant speedup but introduced an imprecision of one pixel in the localization of corner points. Moreover, their method used a Prewitt filter instead of a gaussian filter to compute the image gradients. Both algorithms [33, 34] were implemented using the traditional OpenGL GPGPU API.

For tracking of feature points, Pyramidal Kanade-Lucas-Tomasi(P-KLT) is often utilized to track feature points with a large displacement. Sinha et. al. [33] proposed a GPU implementations of (P-KLT) tracking [5, 2]. Their work was accelerated and extended in [38]. One of the reasons for the improved performance was the adoption a small binomial kernel instead of a Gaussian kernel for down-sampling. These implementations use the traditional GPGPU APIs such as Cg/OpenGL, where – for performance reasons and to avoid workload variations – the iterative tracking equation solver is constrained to a fixed number of iterations. This creates an overkill for small feature point displacements, while jeopardizing the tracking quality in larger scene/feature displacements.

For modern GPU architectures, the CUDA [1] framework supported by Nvidia GPUs for general purpose parallel computing. Compared to traditional GPUs and APIs such as Direct 3D or OpenGL, CUDA provides much more flexibility to manage and utilize GPU resources to fully exploit the data parallelism in an application. Moreover, CUDA provides a high level programming model and a straightforward method of writing scalable parallel programs to execute on the GPU. Due to their increasing programmability and computational power, GPUs are increasingly used for real-time image processing.

The Low Complexity Corner detector algorithm (LOCOCO) [19] reduces the complexity of the Harris and KLT corner detectors by using a box kernel, integral image, and efficient non-maximum suppression. It achieves a complexity reduction by a factor of 8 on a CPU platform. In contrast to the P-KLT algorithm, the Robust Low Complexity Tracking (RLCT) algorithm [20] adopts a varying integration window approach and predicts the coarse location for tracking with an approximated homography over a subset of feature points, which guarantees faster convergence within a few iterations. The robustness of tracking is improved due to a good initial estimate of the feature point locations for tracking, and a low execution time is obtained by avoiding the painstaking image down-sampling filtering overhead of competitor P-KLT algorithms. This report presents scalable GPU implementations of a low complexity corner detector, and a non-pyramidal, though efficient and Robust Low Complexity Tracking (RLCT) algorithm using CUDA. For LOCOCO, the complete algorithm is made to run on the GPU, but for RLCT a CPU-GPU partitioning approach is utilized wherein tracking is performed on the GPU and approximate image registration/alignment is estimated on the CPU.
1.1 Contributions

The major contributions of the thesis work are as follows:

1. We analyze the state-of-the-art low complexity corner detector and tracking algorithms. We propose a solution to exploit data parallelism at each stage of the algorithm. As a result, the algorithms are efficiently parallelized on a Nvidia’s parallel computing GPU architecture.

2. We observed that compared to traditional GPUs, CUDA architecture provides more flexibility in managing and utilizing GPU resources while maintaining huge computational power. We exploit various CUDA tricks and optimization strategies, and compared to the CPU counterpart, our designs for low complexity detector and tracking algorithms attains a speedup factor of 16 and 25, respectively.

3. We demonstrate that the execution time of our work outperforms the existing state-of-the-art implementations of corner detector and tracking algorithm on the GPUs.

4. The work performed on Robust low complexity corner tracking using CUDA is accepted for an IEEE workshop on signal processing systems [25]. We are in process of submitting the work on low complexity corner detection using CUDA [24] to an appropriate conference.

1.2 Organization

The remaining report is organized as follows: Chapter 2 presents the application scenarios where feature detection and tracking is often used. Chapter 3 gives a general overview of Hybrid CPU-GPU systems. It presents the paradigm shift from single core microprocessors to multicore and manycore microprocessors. Then, it compares the traditional APIs and next generation APIs for general purpose computations on GPUs, followed by specificities and limitations of CUDA. The chapter also lists the challenges when programming on such systems and concludes by listing good practices and rules for building optimized code using CUDA. Chapter 4 presents the overview of commonly known corner detectors like Harris and KLT. Then, it presents the low complexity corner detector algorithm, followed by strategy to map LOCOCO algorithm on GPU. The chapter ends by presenting the experimental results of LOCOCO on GPU. Chapter 5 follows the similar structure as Chapter 4. First, it briefly discusses the commonly known tracking algorithms. Then, it shows how RLCT reduces the complexity of P-KLT algorithm, followed by strategy to map RLCT algorithm on GPU. The chapter ends by presenting the experimental results and also demonstrates the visual results of feature detection and tracking with it’s application to image registration. Chapter 6 concludes the report.
Chapter 2

Applications

Feature point detection is a process of determining salient features in an image. A feature, or an interest point, is a point or a set of points where an algorithm can look and follow the motion through frames. Feature tracking is one of the most popular ways of extracting motion information from an image sequence. Feature tracking techniques extract a set of features from each frame (typically corner-like or blob-like features), and then attempt to establish correspondences between both sets of features. An example of feature detection and feature tracking is shown in Figure 2.1. The result of feature detection is shown by highlighted corner points in 2.1 (a), and the result of feature tracking is shown by horizontal displacements of the feature points in 2.1 (b).

![Figure 2.1: (a) Detection of feature points, (b) Tracking of feature points in the consecutive frame sequence](image)

Feature detection and tracking in real time is a fundamental problem of many computer vision applications such as robot navigation, mosaicing, video stabilization etc. Detection and tracking is very low level processing and thus must be
performed in very less time to provide more room for higher level of processing. Some of the applications requiring feature detection and tracking are as follows:

2.1 Panoramic Mosaics from Video Sequences

Figure 2.2: Six frames from a benthic, 150-frame sequence acquired in the Pacific Ocean, and panoramic mosaic of the whole sequence

Feature point detection and tracking is used in panoramic mosaics. As shown in 2.3, to form panoramic mosaics, reliable frame-to-frame correspondences are computed by using feature point detection and tracking of point in subsequent frames. The correspondence results are fed into module that estimates the best plane homo-
Figure 2.3: Video mosaic

graphy and aligns two consecutive frames. The homography is then used to warp the current frame to the reference one, i.e., the current image is warped to form an image plane coplanar with that of the reference frame. To guarantee seamless frame-to-frame borders, image blending is implemented by locally averaging the values of all the pixels mapped to the same pixel in the final mosaic. An example of panoramic mosaic is presented in [36], and Figure 2.2 shows the mosaic of multiple frames.

2.2 Computation of camera motion and 3D reconstruction of a scene

Figure 2.4: Reconstruction of a 3D scene

Feature point detection and tracking can be used in 3D reconstruction of a scene from an image sequence. The technique for three-dimensional endoscopy is pre-
The described approach estimates the camera motion and uses the estimated camera motion to reconstruct the 3D scene. The estimation of camera motion consists of three steps. First, the feature points are detected in the image sequence. Second, the correspondences are established for the feature points between the image sequence. Finally, the corresponding feature points are used to estimate the camera motion parameters.

Figure 2.5: First two images of the sequence

Figure 2.6: Reconstructed 3D surface model from two different views with and without textures.

The technique for automatic reconstruction of 3D scene from an image sequence is shown in 2.4. The 3D reconstruction process consist of three steps: First, 3D coordinates are calculated for each detected inlier feature point which gives a cloud of 3D feature points. Second, the cloud of 3D points is converted into triangle mesh. Finally, texturing is performed to achieve natural look for the 3D construction. For more details on each of the step, the reader may refer to [35]. An example of 3D reconstruction is demonstrated in Figure 2.5 and 2.6.
Chapter 3

Hybrid CPU-GPU systems

Semiconductor scaling limits and associated power and thermal challenges, combined with the difficulty of exploiting greater levels of instruction level parallelism, have combined to limit performance growth for single core microprocessors. This has led most microprocessor vendors to turn instead to multicore chip organizations, even though the benefits of multiple cores can only be realized if the programmer or compiler explicitly parallelizes the software. In this context, graphics processors (GPUs) have become attractive because they offer extensive resources even for non-visual, general-purpose computations: massive parallelism, high memory bandwidth, and a general purpose instruction set, including support for both single and double precision IEEE floating point arithmetic (albeit with some limitations on rounding). In fact, GPUs are really manycore processors, with hundreds of processing elements. A Hybrid CPU-GPU system is a system in which one or several GPU(s) act(s) as massively data-parallel co-processor(s) for a CPU. NVIDIA’s CUDA [1] programming framework is utilised to exploit parallelism among GPU cores. The methodology proposed is in fact a set of programming rules based on the type of code to be mapped.

The advent of general purpose computing on GPUs makes it important to understand when GPUs are preferable to conventional, multicore CPUs. As new parallel computing platforms such as GPUs and multicore CPUs have come to dominate the market, it also becomes important to revisit parallel programming models and to find the best balance between programming convenience and hardware implementation costs. Che et al. [8] explore the extent to which traditionally CPU domain problems can be mapped to GPU architectures using current parallel programming models. A recent report from Berkeley [11] argued that successful parallel architectures should perform well over a set of 13 representative classes of problems, termed dwarves, which each capture a body of related problems. In [8], noting an apparent architectural convergence of CPUs and GPUs, the authors examine the effectiveness of CUDA [1] as a tool to express parallel computation with differing sets of performance characteristics problems from different dwarves on GPUs.

The remaining chapter is organized as follows: Section 3.1 compares the usage
of traditional APIs and next generation APIs for general purpose computations on GPUs. Section 3.2 describes the specificities and limitations of CUDA. Section 3.3 describes the NVIDIA’s unified computing architecture. Section 3.4 highlights the challenges when programming on such hybrid systems. Section 3.5 lists good practices and rules for building optimized code.

3.1 GPGPU programming

For years, people are exploiting the powerful computational resources of the GPU for general-purpose computations (GPGPU). Previously, the only way to harness these resources, was the traditional way by exploiting the graphics pipeline through its respective API, i.e. Direct3D or OpenGL [21]. Due to the tremendous interest in the common public, next-generation APIs such as CUDA [1] and Brook+ [6] have been developed to further facilitate the acceleration of generic computations on GPUs.

![CUDA architecture and memory organization](image)

Figure 3.1: CUDA architecture and memory organization
The NVIDIA CUDA programming model [1] was created for developing applications for NVIDIA GPUs. In this model, the system consists of a host that is a traditional CPU and one or more compute devices that are massively data-parallel coprocessors. Each CUDA device processor supports the Single-Program Multiple-Data (SPMD) model, widely available in parallel processing systems, where all concurrent threads are based on the same code, although they may not follow exactly the same path of execution. All threads share the same global address space. CUDA programming is done with standard ANSI C extended with keywords that designate data-parallel functions, called kernels, and their associated data structures to the compute devices. These kernels describe simultaneously the work of a single thread and typically are invoked on thousands of threads (see Figure 3.1). These threads can, within developer-defined bundles, termed thread blocks, share their data and synchronize their actions through built-in primitives. Unlike in legacy GPGPU programming, memory can be accessed in C style: per rows, per columns, or per tiles (see Figure 3.2). This is particularly suited for image processing. The CUDA runtime also provides library functions for device memory management and data transfers between the host and the compute devices. One can view CUDA as a programming environment that enables software developers to isolate program components that are rich in data parallelism for execution on a coprocessor specialized for exploiting massive data parallelism. An overview of the CUDA programming model can be found in [1]. These APIs provide an alternative means – parallel to the traditional graphics APIs – to access the computational resources on the graphics hardware, in a more generic and familiar way as in CPU programming. However, people either stick to the traditional approach, or start porting their implementations completely to these next-generation paradigms. It appears however that both legacy GPGPU programming and CUDA programming enable different kinds of code optimizations or parallel programming model. Indeed pipeline-based parallelism can be processed using a legacy GPGPU language but not with CUDA. Conversely, only CUDA provides access to a user-managed local cache per block (the shared memory) which enables data re-use exploration. It must be noted that it is also possible to build an application with kernels coded with legacy GPGPU and kernels coded with CUDA.
Figure 3.3: Shared memory enables data re-use in CUDA contrarily to legacy GPGPU programming

The choice of the programming framework should then be made on the basis of the kernel characteristics (see Figure 3.3). The remainder of the report, the focus will be on the CUDA framework as most of the kernels developed for feature detection and tracking fitted with that framework.

3.2 CUDA

As already introduced in the previous section, kernels consist of conventional, scalar C code representing the work to be done at a single point in the domain. CUDA’s extensions to the C programming language are fairly minor. A function declaration can include a modifier specifying whether the function will execute on the CPU or the GPU, and each variable declaration in a GPU function can include a type qualifier specifying where in the GPU’s memory hierarchy the variable will be stored. Kernels also have special thread-identification variables automatically defined to allow threads to identify their location in the domain and work on separate parts of a data set.

The domain is actually defined with a 5-dimensional structure, in the form of a 2D grid of 3D thread blocks. Thread blocks are limited to 512 total threads. The significance of the thread block construct is that each thread block is assigned in its entirety to a single streaming multiprocessor (SM) and runs as a unit to completion without preemption. All threads within the thread block are simultaneously live and the threads are temporally multiplexed onto the processing elements in a fine-grained, time-sliced manner, but their resources cannot be reclaimed until the entire block of threads completes. The number of thread blocks in a grid can greatly exceed the hardware resources, in which case fresh thread blocks are assigned to SMs as previous thread blocks retire. In addition to global shared memory,
each thread block has available a private, per-block shared memory (PBSM) that is only visible to threads within that thread block. The amount of this PBSM that will be used must be defined by the kernel but is limited to 16 KB because it is implemented using fast SRAM, similar to a first-level cache. The PBSM allows threads within a thread block to cooperate in a fine-grained fashion by sharing data among themselves with low latency. Data can also be shared between thread blocks through global memory, which is generally not cached, but the latency is of course much longer.

Synchronization within a thread block is entirely managed in hardware. Synchronization among thread blocks is achieved by allowing a kernel to complete and starting a new kernel; in effect, a global barrier. It is important to note that the order in which thread blocks are assigned to SMs is arbitrary. Because order of execution among thread blocks within a grid is non-deterministic, and because thread blocks run to completion, it is important to note that thread blocks should never have a producer-consumer relationship due to the risk of deadlock. Producer-consumer relationships must be confined within thread blocks or separated across global barriers (i.e., back-to-back kernels) [8]. By separating the size of the domain from the underlying hardware, CUDA allows the programmer to focus on available parallelism. The restrictions on communication among thread blocks define a virtual machine so that the same CUDA program will run on a wide variety of parallel platforms. Indeed, nothing in the CUDA specification prevents CUDA applications from running effectively on other platforms.

### 3.3 GPU Architecture

In NVIDIA’s unified computing architecture, programmable processing elements share a common, very general-purpose instruction set that is used by both graphics and general-purpose computation. Each processing element (PE) supports 128 concurrent thread contexts, allowing a very simple pipeline. Latencies are simply tolerated by switching threads. Current GPUs can support up to 30720 concurrent threads.

Each SM consists of 8 processing elements, called Stream Processors or SPs. To maximize the number of processing elements that can be accommodated within the GPU die, these 8 SPs operate in SIMD fashion under the control of a single instruction sequencer. The threads in a thread block (up to 512) are time-sliced onto these 8 SPs in groups of 32 called warps. Each warp of 32 threads operates in lockstep and these 32 threads are quadpumped on the 8 SPs. Multithreading is then achieved through a hardware thread scheduler in each SM. Every cycle this scheduler selects the next warp to execute. Divergent threads are handled using hardware masking until they re-converge. Different warps in a thread block need not operate in lockstep, but if threads within a warp follow divergent paths, only threads on the same path can be executed simultaneously.

When a kernel is launched, the driver notifies the GPU’s work distributor of
the kernel’s starting PC and its grid configuration. As soon as an SM has sufficient thread and PBSM resources to accommodate a new thread block, a hardware scheduler randomly assigns a new thread block and the SM’s hardware controller initializes the state for all threads (up to 512) in that thread block. Workloads with relatively little temporal data locality and only much localized data reuse are supported. As a consequence, it does not provide large hardware caches which are shared among multiple cores, as is the case on modern CPUs. In fact, there is no cache in the conventional sense: variables that do not fit in a thread’s register file are spilled to global memory. Instead, in addition to the PBSM, each SM has two small, private data caches, both of which only hold read-only data: the texture cache and the constant cache (The name texture comes from 3D graphics, where images which are mapped onto polygons are called textures). Data structures must be explicitly allocated into the PBSM, constant, and texture memory spaces. The texture cache allows arbitrary access patterns at full performance. It is useful for achieving maximum performance on coalesced access patterns with arbitrary offsets. The constant cache is optimized for broadcasting values to all PEs in an SM and performance degrades linearly if PEs request multiple addresses in a given cycle. This limitation makes it primarily useful for small data structures which are accessed in a uniform manner by many threads in a warp. For a more detailed presentation of these features the reader may refer to [1, 28, 8]

3.4 Hardware specificities

Effective CUDA programming requires knowledge of the GPU’s underlying hardware architecture. Specific features of the GPU architecture, such as memory transfer overhead, shared memory bank conflicts, and the impact of control flow need to be considered when programming. Programmers can reduce the overhead and improve the performance of their applications by tailoring their algorithms specifically for execution on the GPU.

3.4.1 CPU-GPU communication overhead

CPU and GPU are usually connected via a PCI-Express bus and a North Bridge chip. Even though the data transfer speed is relatively high (1.47 GHz) and the data transfer time evolves linearly with the amount of data. This might be a significant constraint for some applications where the ratio between memory communication and computational complexity intensities is high.

3.4.2 Bank conflicts

In the GPU, GTX 200 series, each multiprocessor has a 16 KB, on-chip, software-controlled shared memory which enables efficient data-sharing among threads within a thread block. Physically, each shared memory unit is organized into 16 banks, with successive 32-bit words mapped onto successive banks. Simultaneous
accesses to different banks occur in parallel while simultaneous accesses to different addresses within the same bank must be serialized. The existence of bank conflicts approximately doubles the kernel’s execution time [8]. Dealing with bank conflicts is generally lower priority than maximizing parallelism and data locality.

3.4.3 Control Flow overhead

In CUDA, control flow instructions, such as those generated from if and switch statements, can significantly impact the instruction throughput if threads within the same warp follow different branches. When executing divergent branches, either the execution of each path must be serialized or all threads within the warp must execute each instruction, with predication used to mask out the effects of instructions that should not be executed. The overhead of divergent control flow increases linearly as the number of divergent threads increases [8]. Programmers should try to avoid excessive use of control flow instructions, or ensure that the value of the controlling condition is the same across the entire warp.

3.5 CUDA Good Practice Rules for Building Optimized Code

The optimization rules and strategies observed with CUDA programming are as follows:

1. Identify parallelization opportunities and programming pattern for each functional block.
2. Estimate the amount of data transfers between CPU and GPU and minimize it.
3. Optimize the way functional blocks or kernels read and write data and select the best memory type (global, texture, constant) for each of them.
4. If possible exploit data re-use through shared memory.
5. Select the best read mode access from global memory when this is used.
6. Minimize control flow and synchronization signals when not necessary.
7. Avoid using non-trivial mathematical functions (sqrt, modulo, sine, cosine)
8. Optimize occupancy through domain specification (number of threads) and register use. In case of tile access to data, tune the tile size accordingly.

The first rule concerns the algorithm chosen. Is it suitable for parallel programming and particularly for CUDA? If not the algorithm might have to be re-engineered in order to get accelerations from a GPU. Does all the application fit well with the CUDA programming model or only a significant part of it? A good example of this is H.264 intra prediction. At first sight it seems that every macroblock could be processed in parallel (SIMD) with a simple data access pattern (tile access). But it turns out that when neglecting the producer-consumer behavior of the intra prediction (the results of the first macroblocks are needed in order to launch computations for next macroblocks), the subsequent drop in quality is not acceptable.
In this example, the user has to choose between a low quality but well accelerated CUDA version or a high quality but very slow CUDA version of the code with almost no trade-off in between. Another option is to use legacy GPGPU programming in order to exploit the task level parallelism.

The second rule relates to the application level and/or component level. It must be noted that the memory bandwidth between CPU and GPU is 100 times slower than the internal global memory bandwidth. It is therefore more efficient to port the entirety of an application onto the GPU even though each sub component does not comply well with CUDA programming features if this can significantly reduce the data transfer amount between host and device [28]. Sending compressed data to the GPU also significantly speeds up the application by lowering the data transfer amount. For example, in the pipeline comprised of feature detection and feature tracking, the best solution turns out to keep all intermediate results on global memory instead of sending them back to CPU. Sending compressed data to the GPU also significantly speeds up the application by lowering the data transfer amount.

The third rule concerns the choice between texture, global and constant memory. Texture cache is a nice tool for irregular access patterns and hardware optimized linear interpolations. However, the texture binding time is often much more important than memory allocation and copy in global memory if shared memory can be used afterwards.

The fourth and fifth rules are very linked in the sense that shared memory must be filled in following a given memory access pattern. The idea here is to minimize the register usage and fully exploit the collaboration between threads inside the same block in order to make computations in shared memory. Avoiding bank conflicts is then a key issue in order to avoid warp serializations.

The sixth rule is a further optimization of the previous ones related to shared memory. The idea is also to avoid warp serializations when divergent branches are created.

The seventh rule stems for integer low accuracy mathematical functions rather than floating point precision ones. It highly depends on the compute capability of the GPU chosen.

Finally the last rule stems for a benchmark of the domain parameters in order to find the best acceleration. Ryoo et al. [28] show that taking symmetric thread organizations (number of threads in x and y are equal) results in unexpected higher throughput.
Chapter 4

Feature Detector

Many applications require two or more image sequences to be related in order to extract necessary information from them. The brute-force method of comparing each pixel between two images is computationally intensive and not feasible for most of the applications. Intuitively, the two images can be related by comparing the points which are interesting in some way. The points of interest are termed as feature points and are detected using feature point detector. The feature points are used to find relationship between the images and drastically reduces the required computation time. Corners [13, 7, 27] and blobs [16, 4] are typical salient features in an image to detect. The Harris and Kanade-Lucas-Tomasi (KLT) are two most important corner detectors which have wide applications.

4.1 Harris and KLT Corner Detector

Harris and KLT feature detectors are based on the local autocorrelation function within a small window of each pixel, which measures the local change of intensities due to shifts in a local window. For a shift \((\delta x, \delta y)\) and a point \((x, y)\), the autocorrelation function is as follows:

\[
c(x_0, y_0) = \sum_{x \in W} [I(x) - I(x + u)]^2
\]  

(4.1)

where \(x \in (x, y)\) and \(u \in (\delta x, \delta y)\). The shifted image is approximated by a Taylor’s series expansion truncated to the linear term as shown in Eq. (4.2).

\[
I(x + u) = I(x) + u \frac{\partial I(x)}{\partial x}
\]

(4.2)

substituting in Eq. (4.6) yields:

\[
c(x_0, y_0) = \sum_{x \in W} ([I(x) - I(x) - u \frac{\partial I(x)}{\partial x}])^2
\]

(4.3)

\[= \sum_{x \in W} ([u \frac{\partial I(x)}{\partial x}])^2\]

(4.4)
where \( \frac{\partial I(x)}{\partial x} = \delta x \cdot \frac{\partial I(x,y)}{\partial x} + \delta y \cdot \frac{\partial I(x,y)}{\partial y} \). Therefore, Eq. (4.4) can be written as:

\[
c(x_0, y_0) = \sum_{x,y \in W} \left( \frac{\partial I(x,y)}{\partial x} \right)^2 \left( \frac{\partial I(x,y)}{\partial y} \right)^2 \sum_{x,y \in W} \left( \frac{\partial I(x,y)}{\partial x} \frac{\partial I(x,y)}{\partial y} \right) \delta x \delta y
\]

As a first step, the image is convolved with a Gaussian filter that is equivalent to \( g \otimes I \).

\[
\partial(g \otimes I) = \partial(g) \otimes I = g_i, i \in (x, y)
\]

Therefore, Eq. (4.6) can be written as:

\[
c(x_0, y_0) = \sum_{x \in W} \left\{ \begin{bmatrix} g_x(x) & g_x(x) g_y(x) \\
g_y(x) & g_y^2(x) \end{bmatrix} \times v(x) \right\} \begin{bmatrix} \delta x \\ \delta y \end{bmatrix}
\]

\[
C(x_0, y_0) = \sum_{x \in W} \left\{ \begin{bmatrix} g_x^2(x) & g_x(x) g_y(x) \\
g_x(x) g_y(x) & g_y^2(x) \end{bmatrix} \times v(x) \right\} = \begin{bmatrix} G_{xx} & G_{xy} \\ G_{xy} & G_{yy} \end{bmatrix}
\]

where \( v(x) \) is a weighting function which is usually Gaussian or uniform, \( W \) is a window centered at \((x_0, y_0)\), \( I \) is the original image, \( g \) is Gaussian, and \( g_x \) and \( g_y \) are the convolution of \( g \) with the Gaussian first order partial derivative with \( I \) in \( x \) and \( y \) direction at point \((x_0, y_0)\), respectively.

Harris and KLT corner detectors differ in evaluating the measure of cornerness \( R \). As in Eq. (4.10), Harris corner detector evaluates the cornerness without explicit eigenvalue decomposition. The image pixel is a corner if the eigenvalues for the matrix \( C(x_0, y_0) \) are large and thus the value of \( R \) is large and above specified threshold.

\[
R = |C| - k \times (\text{trace}(C))^2
\]

In contrast, KLT explicitly calculates the eigenvalues for matrix \( C(x_0, y_0) \). The corner points are selected such that the minimum eigenvalue as in Eq. (4.11) is greater than a specific threshold [7].

\[
R = \lambda_{\min} = \min(\lambda_1, \lambda_2)
\]

\[
\lambda_{\min} = \frac{1}{2} \left( G_{xx} + G_{yy} - \sqrt{(G_{xx} - G_{yy})^2 + 4 \times G_{xy}^2} \right)
\]
Once the cornerness measure is evaluated for each pixel, both Harris and KLT use a combination of quick-sort and non-maximum suppression to suppress the less strong points locally.

4.2 Low Complexity Corner Detector Algorithm

The Low Complexity Corner Detector Algorithm (LOCOCO) [19] reduces the computational complexity of the Harris and KLT algorithms in each step. First, by using the integral image and box kernel the computational cost of gradients is reduced. The box kernel is obtained by approximating the first order gaussian derivative kernel. Second, many repeated calculations for computation of cornerness according to the Eq. (4.9) are reduced by the use of the integral image. Finally, the combination of sorting (to rank cornerness responses) and non-maximum suppression is replaced by the efficient non-maximum suppression [22]. The algorithm is discussed in detail in the following sections:

4.2.1 Integral Image

The integral image provides a fast and efficient way of generating the sum of values in a rectangular subset of a grid. The integral image at point \((x, y)\) is the sum of all the pixels above and to the left of \((x, y)\), inclusive as in Eq. (4.12).

\[
ii(x, y) = I_1(x) = \sum_{y' \leq y} \sum_{x' \leq x} I(x', y')
\] (4.12)

As shown in Eq. (4.13), the integral image can be efficiently computed in a single pass over the image.

\[
ii(x, y) = i(x, y) + ii(x - 1, y) + ii(x, y - 1) - ii(x - 1, y - 1)
\] (4.13)

Once the integral image is computed, the summation of pixel values within a rectangle can be accomplished in constant time with just four array references [37] as shown in Figure 4.1.
4.2.2 Gaussian Derivative

As shown in Eq. (4.9), the first step for detection of a corner feature point requires the computation of the partial derivative \((g_x\) and \(g_y\)) of an image in the \(x\) and \(y\) directions, respectively. The image is convolved with the Gaussian derivative kernel to achieve robustness against noise as shown in Eq. (4.7). Convolving of Image with Gaussian kernel is a computational intensive task. LOCOCO reduces the complexity at this step by approximating first order Gaussian Derivative by a Box Kernel. As illustrated in Figure 4.2, the first order Gaussian derivative kernel is approximated in \(x\)-direction. The approximation is performed by setting the black and white regions to -1 and +1 respectively and the grey region is set to zero. Hence, with the combination of the integral image the first order Gaussian derivative of an image can be computed at lower computational cost. The summation of areas in both the white and black regions can be performed in 3 operations and 4 memory accesses as described in Section 4.2.1. Thus, computation of gradient in each direction requires only 7 operations and 8 memory accesses.

![Figure 4.2: (a) Discrete Gaussian partial first order x derivative ; and, (b) Box kernel for the partial first order x derivative \(\sigma=1.2\). The kernel size is \(9\times9(4\sigma=9)\)](image)

4.2.3 Cornerness Response

The measure of cornerness for each pixel is calculated by summation of squares and products of gradients within an integration window centered at the pixel as in Eq.(4.9). This stage is accelerated by using integral images. Integral images for elements in Eq.(4.9), \(g_x^2\), \(g_y^2\) and \(g_xg_y\) are calculated as follows:

\[
ii_{xx}(x,y) = \sum_{x' \leq x, y' \leq y} g_x^2(x',y') 
\]

\[
ii_{yy}(x,y) = \sum_{x' \leq x, y' \leq y} g_y^2(x',y') 
\]

\[
ii_{xy}(x,y) = \sum_{x' \leq x, y' \leq y} g_x(x',y')g_y(x',y') 
\]

Once the integral image is created, it requires only 3 operations and 4 memory accesses to compute the summations \(G_{xx}\), \(G_{yy}\), and \(G_{xy}\) in the Eq.(4.9). Thus, the
repeated operations for each pixel within the integration window $W$ are replaced by the creation of an integral image and the lower number of operations for each pixel.

### 4.2.4 Non-Maximum Suppression

Non maximum suppression can be described as search for local maximum, where a local maximum is greater than all its neighbors. A single location for each feature point is retained by using Non-maximum suppression over the cornerness response image. At first, the KLT feature detector [7, 2] sorts the cornerness measure $R$ over the image in descending order. Later, it picks up the feature points from the top of the sorted list and performs a non-maximum suppression in the region $(2d + 1 \times 2d + 1)$ to remove the less strong response points successive in the list. Consequently, the method enforces a minimum distance of $d$ between the selected feature points with a final set of selected feature points arranged in descending order of cornerness response. LOCOCO achieves a similar result by changing the order of the steps, i.e. At first, the Non-Maximum points are suppressed in a region $(2d + 1 \times 2d + 1)$. Later, the sorting is performed on the Non-suppressed feature points. As a result, LOCOCO reduces the complexity of this step by performing sorting on a lower number of feature points.

![Figure 4.3: Efficient Non-maximum suppression algorithm.](image)

To find the local maximum point by using a naive implementation of non-maximum suppression, for each pixel, the comparison is performed with all the pixels lying within a window $(2d + 1 \times 2d + 1)$. This approach is computationally intensive. Instead of adopting this approach, LOCOCO uses efficient non-maximum suppression (E-NMS) method proposed in [22]. As illustrated in Figure 4.3, the E-NMS algorithm consists of two steps. First, the input cornerness image is partitioned into blocks of size $d \times d$ and greatest element within each block is evaluated, which is possibly the local maximum candidate. Second, full neighborhood of the candi-
date maximum is tested to determine if the candidate is the maximum in the region $(2d + 1 \times 2d + 1)$. The candidate maximum is retained if it passes the local maximum test and is larger than the specified threshold.

4.3 Mapping Low Complexity Corner Detector on GPU

This section explains the mapping of feature detection algorithm on the GPU using CUDA. Section 4.3.1 presents the overview of the algorithm running on the GPU. Section 4.3.2 describes processing kernels for detecting corner points in an image.

4.3.1 Parallelize Low Complexity Corner Detector algorithm

Figure 4.4 shows the organization of the memory and CUDA kernels for the implementation of the LOCOCO algorithm on GPU. The input image and the image-level intermediate data are stored in global memory. There are mainly four function kernels, i.e., calculating integral image, computing gradients in $x$ and $y$ directions, computing cornerness response, and efficient non-maximum suppression. First, the input image is transferred to the GPU, then integral image is computed and
stored in the global memory. The gradient kernel uses the integral image to compute the gradients in \( x \) and \( y \) directions, respectively. After this step, the gradients are used to compute cornerness response image. Then, the kernel for Efficient non-maximum suppression is executed to suppress the less strong points in the cornerness response image. Finally, the feature points are downloaded back to the CPU.

### 4.3.2 CUDA Kernels for Low Complexity Corner Detector

#### Integral Image

LOCOCO makes extensive use of the integral image. This section presents an efficient method to map the computation of the integral image on the GPU. The computation of the integral image can be separated in two stages. As shown in Figure 4.5(a), the prefix sum is calculated for each row. After completing the processing on rows, as shown in Figure 4.5(b), the prefix sum is applied to each column, thus resulting into an integral image.

![Figure 4.5: (a) Prefix sum on rows (b) Prefix sum on columns](image)

To compute the prefix sum for all the rows in parallel, the efficient parallel scan algorithm for GPUs [30] is utilized for all the rows. The key idea of this algorithm is to divide a block of data into warp-sized chunks and all scan primitives are built upon the set of primitive intra-warp scan routines. The GPU architecture by NVIDIA executes the threads of a block in SIMT(single instruction, multiple thread) group of 32 called warps. A warp of threads can take any execution path but all the threads execute the same instruction at a time therefore synchronization is not needed in order to share data within a warp. Thus, the intra-warp scan routine performs scan operation over a warp of 32 threads and computes the prefix sum for 32 elements without requiring any synchronization operation. Figure 4.6 demonstrates the working of scan algorithm for a warp.

The array of data is divided into thread-blocks, and each thread-block is comprised of warps. The algorithm requires synchronization across the warps to complete
scan operation for a block, in the same way the algorithm requires global synchronization across the blocks to complete scan operation for the complete array. The thread block consists of 128 threads and as each thread handles 8 elements, global synchronization across the block is only required if the number of elements to scan are more than 1024. The scan operation is described in Algorithms 1 and 2 respectively. As shown in Figure 4.7, to compute prefix sum on a row, if a row corresponds to a thread block then the block is divided into warp-sized chunks. All the warps are scanned in parallel using an intra-warp scan routine. Next, the partial results of each scan are accumulated and adjusted to get the scan for the complete row. The reduced number of synchronization steps and various optimizations, such as efficiently exploiting shared memory and performing an initial serial scan of multiple input elements when read from global memory, makes it one of the fastest scans yet designed for the GPUs [30]. In order to evaluate the subsequent prefix sum on the columns, the prefix sum result of the rows is transposed and a new row-based scan is launched. Transpose between the two steps help to maintain coalesced access to the global memory [1]. The resultant integral image is not transposed back again to correct the orientation, since the computation of integral image in the subsequent step of computing the cornerness response leads to another transpose, yielding the restored image.
Algorithm 1 Scan operation for a block

1: The intra-warp scan runs in parallel for all the warps in the block.
2: The last partial result from each warp is recorded and a single warp performs
   intra-warp scan on the recorded partial results.
3: The results obtained in second step are used by threads of each warp and accu-
   mulated to the results of first step to complete the scan operation for a block.

Algorithm 2 Scan operation for array of arbitrary length

1: The intra-block scan runs in parallel for all the block.
2: The last partial result from each block is recorded.
3: The scan operation is performed on the recorded partial results.
4: The results obtained in third step are used by threads of each block and ac-
   cumulated to the results of first step to complete the scan operation for the
   complete array.

Box Kernel

After computation of the integral image, the partial derivatives are computed for the
\( x \) and \( y \) directions. The strategy used to parallelize this step is based upon creating
many threads to exploit the large number of cores and hide global memory latency
by computations. As shown in Figure 4.8, for an image size of \( W \times H \), the grid
is formulated with blocks \((W/16) \times (H/16)\). Each thread in that block can be
mapped to a pixel location and computes the gradient value corresponding to that pixel of the image. Therefore for the box kernel, the computations performed by the threads in a block are independent of each other. The CUDA kernel is launched wherein each thread performs 8 memory accesses and 7 operations in parallel to calculate \( g_x \) or \( g_y \), corresponding to each pixel. The logic executed by each thread for computation of \( g_x \) is shown in Figure 4.9 and Listing 4.1. The step is complete when the gradients are computed for all the pixels in the image.

Figure 4.8: Arrangement of grid

\[
g_x = (ii_e + ii_h - ii_f - ii_g) - (ii_a + ii_d - ii_a - ii_c)
\]

Figure 4.9: Partial derivative in \( x \) direction
void partial_derivative_x()
{
    /* Compute the indices for the block corresponding to the thread */
xBlock = blockDim.x * blockIdx.x;
yBlock = blockDim.y * blockIdx.y;

    /* Compute the pixel location */
index = pitch * (yBlock + threadIdx.y) + xBlock + threadIdx.x;

    /* Compute partial derivative */
region1 = ii_a + ii_d - ii_b - ii_c;
region2 = ii_e + ii_h - ii_f - ii_g;
out[index] = (region2 - region1)/(WIN*WIN);
}

Cornerness Response

The strategy used to compute the integral image for $g_x^2$, $g_y^2$, and $g_{xy}$ is the same as described in the section of computation of Integral Image. To obtain $g_x^2$, $g_y^2$, and $g_{xy}$, the scan algorithm is modified such that each element of $g_x$ and $g_y$ is squared or multiplied with each other when fetched from global memory to shared memory.

To compute the summation within a window and cornerness response corresponding to each pixel, a similar strategy as described in computation of Box kernel. The grid arrangement is done such that each thread in every block can be mapped to a pixel location in the image. The CUDA kernel is launched wherein each thread performs 4 memory accesses and 3 operations in parallel to calculate the window sum corresponding to each pixel followed by the computation of cornerness response. The logic executed by each thread for computation of cornerness response is shown in Figure 4.10 and Listing 4.2. The step is complete when the cornerness response is calculated for all the pixels in the image.

$$G_{x/y/xy} = \sum_{w} ii_{x^2+y^2;xy} = (ii_a + ii_d - ii_b - ii_c)$$

Figure 4.10: Cornerness response
Listing 4.2: Pseudo code for Cornerness response (executed by all the threads)

```c
void cornerness_response()
{
  // Compute the indices for the block corresponding to the thread
  xBlock = blockDim.x * blockIdx.x;
  yBlock = blockDim.y * blockIdx.y;

  // Compute the pixel location
  index = pitch * (yBlock + threadIdx.y) + xBlock + threadIdx.x;

  // Compute cornerness
  Gxx = ii_a + ii_d - ii_b - ii_c;
  Gyy = ii_a + ii_d - ii_b - ii_c;
  Gxy = ii_a + ii_d - ii_b - ii_c;
  out[index] = Gxx*Gyy - Gxy*Gxy - .04*(Gxx+Gyy)*(Gxx+Gyy);
}
```

Efficient Non-maximum Suppresion

Access to off-chip global memory is slow and requires 200 to 300 cycles per access. This latency can be hidden by launching a massive number of active threads [28]. But this technique does not give enough speedup for algorithms that have repeated calculations or are bounded by memory accesses. Another technique to attain speedup is to use low latency on-chip shared memory and reuse data among all the threads in a thread block to reduce the number of accesses to the global memory [28]. As non-maximum suppression incorporates repeated calculations on a small region of pixels, shared memory is therefore exploited to reduce the number of accesses to the global memory.

![Efficient non-maximum suppression](image)

Figure 4.11: Efficient non-maximum suppression

The suppression algorithm is implemented for a neighborhood. The kernel is
launched wherein threads in each thread block fetch pixels from global memory to the shared memory. At this point the contents of the shared memory can be visualized as four sub-blocks as shown in Figure 4.11(a). The kernel is implemented in such a way that the threads in a thread block compute the maximum value of the cornerness response in each of the sub-blocks in parallel. These maximum values are termed as candidate local-maximas (max1, max2, max3, and max4). The maximum value in each of the sub-blocks is calculated using parallel reduction [14] where the add operation is replaced with a comparison operator. For each candidate maxima, the threads in a thread block fetch the local neighborhood pixels to the shared memory if the value is greater than the predefined threshold, as shown in Figure 4.11(b). The maximum value is computed in each of the blocks using [14]. If the candidate maxima remains maximum in the local neighborhood then it is marked as corner point else the point is suppressed.

4.4 Experimental Results and Discussion

The execution time of LOCOCO is evaluated on a CPU and on a GPU. To measure the effectiveness of LOCOCO on GPUs, the execution time are compared with CUDA based implementation of the Harris corner detector on the same GPU. For implementing the Harris corner detector, the gaussian derivative is implemented by using the separable gaussian convolution kernel [26] which requires less computations compared to the 2D convolution. To measure the cornerness response, the gradients are squared and multiplied with each other and the summation within the window is implemented by utilizing the separable convolution. The separable filter can be used to sum the pixels within the window by setting the coefficients of separable filters to 1; this method of implementation runs much faster than computing the naive sum of all pixels within a window. As described in [33], the sorting for cornerness response is performed on the CPU and this involves transferring the complete cornerness image back to the CPU. Instead of adopting this approach, the sorting is performed on the GPU by using an efficient sorting algorithm as presented in [29]. After this step, non-maximum suppression is performed on the GPU. Thus, the Harris algorithm runs completely on the GPU and this implementation is taken to have a fair comparison with LOCOCO implementation on the GPU.

As shown in Figure 4.12, for image size of $960 \times 960$ the GPU implementation of LOCOCO is around 16 times faster than the corresponding CPU implementation. The speedup is mainly due to the fact that computation of the integral image and efficient non-maximum suppression is efficiently parallelized using CUDA.

The comparison of execution time of both LOCOCO and Harris on GPU is shown in Figure 4.13 and 4.14 for different image and kernel sizes, respectively. As shown in figure 4.13, for various image sizes and a fixed kernel size of $9 \times 9$, the LOCOCO implementation on GPU is around 2 times faster than the Harris corner detector on GPU. The original Harris algorithm uses Gaussian convolution
Figure 4.12: Execution time of LOCOCO on CPU and GPU (CPU: Intel Core 2 Duo, 2.66 GHz and 2 GB RAM. GPU: Nvidia GeForce GTX 280)

Table 4.1: Complexity of convolution with the first order Gaussian derivative filter and the Box filter

<table>
<thead>
<tr>
<th></th>
<th>Additions</th>
<th>Multiplications</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Convolution</td>
<td>2N</td>
<td>2N</td>
<td>4N</td>
</tr>
<tr>
<td>Box Filter</td>
<td>2+3*2</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

instead of Integral image computation and box kernel approximations; contrary to CPU programming the execution time of the GPU implementation of convolution for small kernel size (9 × 9) is comparable to the time taken by the computation of the integral image and the box kernel approximation. With the box kernel approximation, the speedup for a kernel size of 9x9 is mainly because LOCOCO replaces the combination of feature sorting and non-maximum suppression in Harris by efficient non-maximum suppression. As shown in Figure 4.14, for a kernel size of 31 × 31, LOCOCO is 3 times faster than the Harris. This is because, as shown in Table 4.1, when the kernel size increases, the computation of the integral image and box kernel approximation remains unaffected but the execution time for the
convolution increases significantly. For applications that require multi-scale estimation, the convolution must be computed for each scale, while only one execution of the integral image allows for the computation of all the scales. In that case, the
LOCOCO algorithm turns out to be much more efficient than the Harris algorithm.

Table 4.2: Comparison with other methods

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Image Size</th>
<th>Time (ms)</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCOCO CUDA</td>
<td>640×480</td>
<td>2.4</td>
<td>GeForce 280 GTX</td>
</tr>
<tr>
<td>L. Teixeira [34]</td>
<td>640×480</td>
<td>7.3</td>
<td>GeForce 8800 GTX</td>
</tr>
<tr>
<td>Sinha [33]</td>
<td>720×576</td>
<td>61.7</td>
<td>GeForce 8800 GTX + AMD Athlon 64 X2 Dual Core 4400 (one core used)</td>
</tr>
</tbody>
</table>

Table 4.2 presents the comparison of LOCOCO using CUDA with other state-of-the-art implementations of corner detectors. The Accelerated corner detector presented in [5] provides a full implementation of the Harris corner detector on GPU by proposing its own version of non-maximum suppression. The proposed non-maximum suppression has two different variants, one in which the cornerness response image is compressed and the other in which the cornerness response image is not compressed. The lossy compression of the cornerness response image introduces a precision error of one pixel in localization of the corner points whereas our method does not introduce any precision error or compression of the cornerness response image. A comparison is made with the version of the accelerated corner detector in which the cornerness response image is not compressed. Moreover, the accelerated corner detector uses a Prewitt filter instead of a gaussian filter used in our implementation. Sinha et al [4] proposed an implementation of the KLT corner detector with the computation of the gradients and the corner response mapped on the GPU and the further steps of sorting and non-maximum suppression mapped on the CPU. The time taken by this implementation is reported in [13]. The timings presented in Table 4.2 include the time to transfer data between the GPU and the CPU. It can be inferred that our method is the fastest compared to state-of-the-art implementations of corner detectors reported till now. Figure 4.15 shows the feature point detection results for various data set obtained using low complexity corner detector. It was observed that the feature point detection results produced by LOCOCO on GPU are similar to the results generated by execution of LOCOCO on CPU [19].
Figure 4.15: Feature points detection for various data sets
Chapter 5

Feature Tracking

The pixel position of the feature points differs between frames because of the relative motion between the camera and the objects. Feature tracking is the process of finding the correspondence between pixel positions of salient features between two frames. Let $I$ and $J$ be two consecutive 2D grey scale image sequences, $I(x)$ and $J(x)$ give the respective intensity value at each location $x$ in the two images, where $x = [x, y]^T$. As shown in Figure 5.1, if the pixel position $x_1$ is known in $I$, then tracking is performed to find the corresponding pixel location $x_1'$ in $J$. The image sequences can get affected by translation, rotation, or scaling transformations. These effects should be considered while tracking.

**Figure 5.1: Tracking of a feature point for consecutive image sequence**

5.1 Kanade-Lucas-Tomasi Algorithm

The Kanade-Lucas-Tomasi (KLT) algorithm [17] tracks feature points when the displacement and brightness variation between frames is quite small. This is done by computing a displacement vector $d$ so that $J(x) = I(x - d)$. However, it is impossible that a single displacement $d$ correlates all the regions of $I$ and $J$ correctly. The points on the surface of different objects or points on the background can have different displacements. The problem is solved by dividing the images in
small regions and the displacement is computed for every region. The displacement vector \( \mathbf{d} = [d_x, d_y]^T \) between the images is chosen so as to minimize the residue error defined by the summation over a window \( W \) as in Eq.(5.1):

\[
\epsilon = \sum_{x \in W} [I(x - \mathbf{d}) - J(x)]^2 w(x) \tag{5.1}
\]

where \( w \) is a weighting function. \( w \) can be set to 1 for a simple case or it could be a Gaussian-like function to emphasize the central area of the window.

When the displacement vector \( \mathbf{d} \) is small, the intensity function can be approximated by its Taylor series truncated to the linear term as shown in Eq.(5.2):

\[
I(x - \mathbf{d}) = I(x) - \mathbf{g} \cdot \mathbf{d} \tag{5.2}
\]

the residue defined in Eq.(5.1) can be rewritten as:

\[
\epsilon = \sum_{x \in W} [I(x) - \mathbf{g} \cdot \mathbf{d} - J(x)]^2 w(x) = \sum_{x \in W} (h - \mathbf{g} \cdot \mathbf{d})^2 w(x) \tag{5.3}
\]

where \( h = I(x) - J(x) \).

On differentiating the last expression in Eq.(5.1) with respect to \( \mathbf{d} \) and setting to zero results in following:

\[
\sum_{x \in W} (h - \mathbf{g} \cdot \mathbf{d}) g w(x) = 0 \tag{5.4}
\]

since \((\mathbf{g} \cdot \mathbf{d})\mathbf{g} = (\mathbf{g g}^T)\mathbf{d} \), and \( \mathbf{d} \) is assumed to be fixed within a window \( W \), the Eq.(5.4) can be written as:

\[
(\sum_{x \in W} \mathbf{g g}^T w(x)) \mathbf{d} = \sum_{x \in W} h g w(x) \tag{5.5}
\]

Eq.(5.5) is a system of two scalar equations in two unknowns and can be rewritten as:

\[
\mathbf{G d} = \mathbf{e}
\]

\[
\mathbf{d} = \mathbf{G}^{-1} \mathbf{e} \tag{5.6}
\]

where \( \mathbf{G} = \sum_{x \in W} \begin{bmatrix} g_x^2 & g_x g_y \\ g_x g_y & g_y^2 \end{bmatrix}, \mathbf{e} = \sum_{x \in W} (I(x) - J(x)) \mathbf{g} \) and \( w = 1 \).

\( \mathbf{g} = g_i = \partial_i (g \otimes I) = (\partial_i g) \otimes I, i \in (x, y) \), where \( \mathbf{g} \) is a gaussian function and \( W \) is an integration window.

Eq.(5.6) is solved until convergence [5, 32] and leads to following convergence scheme:
\begin{equation}
\begin{align*}
d_0 &= 0 \\
d_{n+1} &= d_n + G^{-1}e 
\end{align*}
\tag{5.7}
\end{equation}

After each iteration step, the image \( J(x) \) is adjusted by the computed displacement before the system of equations is evaluated again. This Newton-Rhapson iteration scheme converges quickly, usually five to ten iterations are enough to compute the displacement with sufficient subpixel precision. Due to Taylor series approximation, the above method is suitable for tracking when the displacement between frames is small compared to the size of integration window i.e. \( d = [d_x, d_y] \leq W/2 \). If the displacement is not small, then the tracking convergence fails to converge to accurate solution.

### 5.2 Pyramidal Kanade-Lucas-Tomasi Algorithm

Pyramidal Kanade-Lucas-Tomasi Algorithm (P-KLT) \cite{5} was introduced to track feature points with larger displacement. The method involves the construction of an image pyramid where the image is represented at different scale levels. Let \( I(x) \) be an image, then \( I^L(x) \) represents the image at different scales with \( L \in \{0, 1, 2, ..., n\} \). \( I^0(x) \) is the base level with highest resolution. The width and height of the next level (\( I^1(x) \)) is half of the previous level, and so on. Image at level \( m + 1 \) is obtained by smoothing the image at level \( m \) by a gaussian kernel and then sampling the result at every alternate row and column.

In order to track feature points from image \( I(x) \) to \( J(x) \), the first step involves building of image pyramid for both the images. The number of levels in image pyramid \( n \) is 3 or 4. The coordinates of a feature point in image \( I(x) \) are propagated down to the lowest level of pyramid by dividing the coordinates by a factor of 2\(^n\). The tracking is performed at the lowest level (level \( n \)) and the tracked points are propagated to the next higher level (level \( n - 1 \)). The tracking process in level \( n - 1 \) uses the tracked offset returned from level \( n \). The process is repeated until the feature points are tracked at the highest pyramid level (level \( 0 \)). As shown in Eq. (5.8), the feature tracking is performed with the same window size in all the levels of pyramid, thus providing a larger tracking range.

\begin{equation}
\varepsilon(d^L) = \sum_{x \in W} (I^L(x, y) - J^L(x + p_x^L + d_x^L, y + p_y^L + d_y^L))^2 
\tag{5.8}
\end{equation}

In Eq. (5.8) an initial point \( p = [p_x^L, p_y^L]^T \) is supplied to level \( L \) from a level \( L + 1 \), after multiplying with the up-sampling factor of the image pyramid.

### 5.3 Robust Low Complexity Tracking Algorithm

Robust Low Complexity Tracking Algorithm is functionally equivalently to P-KLT algorithm and is designed to be a low complexity version of P-KLT algorithm. As
shown in Section 5.2, there is a fixed overhead for the tracking algorithm with the pyramidal approach in constructing the image pyramid before tracking. In contrast, the Robust Low Complexity Tracking (RLCT) algorithm [20] developed at IMEC (Interuniversity Microelectronics Centre) does not require building the image pyramid. The RLCT algorithm consists of two parts. First, a subset of feature points is tracked by coupling a varying integration window with the KLT tracking algorithm. Second, an approximate motion model is estimated by using RANSAC on a subset of feature points to provide initial tracking point locations for the tracking of all remaining feature points. Then those points are tracked using a small integration window. The RLCT is described briefly in following sections.

5.3.1 Coarse to fine feature tracking

The coarse to fine feature point tracking for RLCT works by coupling the KLT tracking with a varying integration window, instead of performing it through out the image pyramid. The varying integration window feature tracking algorithm works as follows. Lets assume that there are \( m \) integration windows of size \( W_{m-1}, W_{m-2}, \ldots, W_0 \). First the KLT tracking is performed with largest integration window \( W_m \). Then, the result of this tracking is propagated as an initial guess of the pixel displacement \( i^m = (i^m_x, i^m_y) \) to the next phase of the KLT tracking with a smaller integration window of size \( W_{m-1} \), and the process is repeated iteratively until the smallest integration window \( W_0 \) is reached:

\[
\varepsilon(d^m) = \sum_{x \in W_m} (I(x,y) - J(x + i^m_x + d_x, x + i^m_y + d_y))^2 \quad (5.9)
\]

By solving Eq. (5.9) as in Section 5.1 gives:

\[
\sum_{x \in W_m} Gd = \sum_{x \in W_m} e \quad (5.10)
\]

The matrix \( G \) is evaluated for the first image and is assumed to be constant [5, 3].

5.3.2 Initializing the homography and predicting the feature location

Often features are tracked independently and later RANSAC is applied to estimate the correspondence between images by homography. The feature points may be tracked with a small integration window if the initial location of the feature point for the KLT tracking algorithm in the region around the final target can be predicted. To exploit this, RLCT utilizes the RANSAC much earlier by tracking a small subset of feature points to estimate the approximated homography between the frames. This approximated homography is used to estimate the location of the remaining feature points. Hence, the remaining points are tracked at low computational cost with a small integration window, without using the image pyramid and without varying the integration window size. The projective homography \( H_{init} \) is solved by using the normalized DLT algorithm [15], with RANSAC [12] and
SVD decomposition. The initial location for the tracking of the remaining feature points is estimated using $H_{\text{init}}$ and tracking is performed with the window as in Eq. (5.11).

$$\bar{x} = xH_{\text{init}}$$

$$\varepsilon(d) = \sum_{x, \bar{x} \in W_0} (I(x) - J(\bar{x} + d))^2$$

(5.11)

On solving Eq. (5.11) as in Section 5.1:

$$\sum_{x \in W_0} Gd = \sum_{x, \bar{x} \in W_0} e, \text{ where } e = \sum_{x, \bar{x} \in W_0} (I(x) - J(\bar{x})) \cdot g$$

(5.12)

Eq. (5.12) is solved iteratively for each predicted feature point. The initial location for tracking in the second image is $\bar{x}$ instead of $x$, which is near to the target, hence enabling tracking with the small integration window $W_0$. This highly improves the behavior of KLT which uses the SSD approach to find similar regions and hence - without this initial location estimation - would exhibit a higher risk of diverging into the outlier region, especially in low texture regions.

5.4 Mapping Robust Low Complexity Tracking on GPU

5.4.1 Parallelize Robust Low Complexity Tracking algorithm

Figure 5.2 shows the organization of the memory and CUDA kernels for the implementation of the RLCT algorithm on GPU. First, N feature points are transferred to the GPU, then these N points are tracked with varying integration window as explained in Section 5.3.1, and the tracked results are downloaded back to the CPU. The approximate homography between the frames is estimated on the CPU and the location of the remaining feature points is predicted. The predicted feature point locations are transferred again to the GPU and tracking is performed with a single integration window of size as described in Section 5.3.2. The GPU implementation of RLCT on GPU can be summarized as follows:

1. Upload images $I$ and $J$, transfer coordinates of the N feature points and memory space for offset $d$ to the global memory.
2. Invoke the convolution kernel [26] to compute gradients $g_x$ and $g_y$ for image $I$.
3. Bind $I$, $J$, $g_x$, and $g_y$ to the texture memory.
4. For each feature point (up to N):
   
   (a) For each window of size $W_i$, $i = m - 1$ to 0, invoke the kernel to compute $G_{xx}$, $G_{yy}$ and $G_{xy}$ within the window $W_i$ centered at feature point location.
(b) For each window of size $W$, $i = m - 1$ to 0, invoke the kernel which uses the results from previous step and solve the equation $d = G^{-1}e$ iteratively until convergence as in Eq.(5.10).

5. Download the tracked offset for the feature points from the global memory to CPU.

6. Estimate $H_{\text{init}}$ on the CPU from the correspondence set of feature points.

7. Predict location of remaining feature points on the CPU.

8. Upload predicted point locations to GPU.

9. For each remaining feature point:

   (a) Invoke the kernel to compute $G_{xx}$, $G_{yy}$, and $G_{xy}$ within the window $W_0$ centered at the feature point location.

   (b) Invoke the kernel which uses the results from previous step and solve the equation $d = G^{-1}e$ iteratively until convergence as in Eq.(5.12).

10. Download the tracked offset $d$ for the feature points from the global memory to CPU.

### 5.4.2 Thread organization and memory access pattern

The tracking algorithm involves iteratively solving the KLT tracking equation around each feature point location. The feature points can be independently tracked. Hence, we propose an approach wherein each feature point is processed by a thread-block rather than a single thread. Therefore, threads within a block cooperate to track a feature point and at the same time blocks can be processed in any order. Additionally, using CUDA for tracking makes it scalable to execute efficiently on CUDA enabled future graphics processor families.

Since the input and gradient images are frequently accessed and not modified during the whole algorithm, we have chosen to bind them to texture memory, which provides hardware implemented caching scheme with a 2D spatial locality [31], perfectly matching the 2D access pattern within the RLCT integration window centered at each feature point. Additionally, texture memory provides dedicated interpolation hardware, which is efficiently used in calculating - by bilinear interpolation - the sub-pixel locations in feature tracking. Consequently, the hardware caching and interpolation mechanism of GPU’s texture memory is fully exploited in the RLCT implementation.

### 5.4.3 CUDA Kernels for feature point tracking

The general framework used in solving the tracking equation for a feature point within an integration window is shown in Figure 5.3. Each block in the grid corresponds to the feature point to be tracked. The integration window centered at the
feature point is divided into regions $R_1, R_2, ..., R_t$. The threads within a block read a region from the texture memory and perform the required computations. The intermediate results are stored in the registers of each thread. This process is repeated for all the regions and finally the accumulated results by each thread are stored into shared memory. A single thread adds the content of shared memory and produces the final result for a block corresponding to a feature point. We create two kernels to track feature points. The first kernel computes and for each integration window corresponding to all the feature points. The second kernel solves the tracking equation iteratively until convergence. The kernels are as follows:

**Kernel for sum of products**

This kernel computes $G_{xx}$, $G_{yy}$, and $G_{xy}$ within an integration window for all the feature points. The implementation of the kernel is summarized as follows:

1. The location of feature point for the corresponding block is read from global memory to the shared memory.
2. The threads in a block access texture memory for $g_x$ and $g_y$ to compute $G_{xx}$, $G_{yy}$ and $G_{xy}$ for each integration window centered at the feature location.
Figure 5.3: Processing performed by a block in solving the tracking equation within an integration window

**Kernel for tracking**

Tracking kernel solves the tracking equation iteratively until convergence. In contrast to state-of-the-art methods which solve the equation for a fixed number of iterations, we allow the tracking kernel to return if it converges earlier. The implementation of the tracking kernel for an integration window is summarized as follows:

1. The location of feature point, offset $d$ and $G_{xx}$, $G_{yy}$ and $G_{xy}$ for the corresponding block are read by a single thread from global memory to the shared memory.

2. Repeat the following until convergence:
   
   (a) The threads in the block access texture memory for $I$, $g_x$, $g_y$, and $J$ around the feature point location and compute $e$ as in equation 5.6.
   
   (b) A single thread computes $G^{-1}e$, updates the value of offset $d$ in the shared memory and evaluates the condition for convergence.

3. Finally, the offset $d$ and tracking status are updated in the global memory.

We tried to maintain high GPU occupancy for our kernels. GPU occupancy is defined as a ratio of the number of active warps per multi-processor to the maximum number of active warps. The execution resources in a Streaming Multiprocessor, or SM, include registers, thread block slots, and thread slots. These resources are dynamically partitioned and assigned to threads to support their execution. CUDA occupancy is calculated by taking into account the factors like number of threads
per block, number of registers per thread and shared memory per block. In our implementation, for GTX 280 GPU, each SM has 1024 thread slots. The thread slots are partitioned and assigned to thread blocks during runtime. If each thread block contains 128 threads, the threads slots are partitioned and assigned to 8 blocks (which is also the maximum number of possible blocks per SM). The register file is another dynamically partitioned resource. In GTX 280, there is an 16384 entry register file in each SM. If we have 19 registers per thread, and the block contains 128 threads, then the number of thread needed for each block is $19 \times 128 = 2304$. The number of registers required by 8 blocks is 19456. The number of registers as 19456, exceeds the limit of 16384. Hence, the limitation allows only 6 blocks to run on each SM, and the GPU occupancy gets reduced. To increase the occupancy for this scenario, the number of registers should be made as 16. The register pressure was reduced by shifting some of the common variables to shared memory, and hence we managed to achieve the GPU occupancy as 1.

5.5 Experimental Results and Discussion

The performance of the GPU implementation of RLCT is compared with its CPU implementation and competitive P-KLT state-of-the-art implementations. The computation time of RLCT on the CPU platform is used from existing work [20]. The low complexity version of the corner detector on GPU is used to detect feature points.

The RLCT method uses integration windows $25 \times 25$, $13 \times 13$, and $7 \times 7$ to be functionally equivalent to the tracking results produced with three levels of P-KLT with an integration window of size $7 \times 7$. To obtain the computation time of P-KLT on the CPU platform, the implementation in [2] is modified such that $G$ is derived from a single frame and the tracking uses three levels of the pyramid as proposed in [5].

The CPU version of RLCT algorithm [20] uses a subset size of 20 feature points ($N$) for estimating the approximate homography. As demonstrated in [20], increasing the subset size does not substantially improve the final tracking accuracy, while drastically increasing the execution time. However, the same statement does not hold for the GPU implementation. Therefore, an experiment was performed to determine $N$ by measuring the execution time with the varying integration window approach, for an increasing number of feature points, as shown in Figure 5.4. The tracking execution time in the GPU remains almost constant from 20 to 125 points. This is due to a large number of cores available in the GPU which allows executing the tracking of a large number of feature points in parallel, in contrast to a CPU. Thus, $N$ is chosen to be 125 for this version of GPU implementation. It is to be noted that $N$ can be scaled to a higher number for future more advanced versions of the GPUs. The RLCT algorithm for $N$ number of feature points consists of tracking only with varying window approach.

As shown in Table 5.1, the GPU implementation of RLCT is 25 times faster.
Figure 5.4: Feature point tracking time with only varying integration window (without gradients)

Table 5.1: Execution time of RLCT on CPU and GPU for Image size=960×960 and Feature count=1000 (CPU: Intel Core 2 Duo, 2.66 GHz and 2 GB RAM. GPU: Nvidia GeForce GTX 280)

<table>
<thead>
<tr>
<th></th>
<th>RLCT CPU Time (ms)</th>
<th>RLCT GPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Derivative</td>
<td>31</td>
<td>0.65</td>
</tr>
<tr>
<td>Varying window tracking</td>
<td>7.15</td>
<td>0.51</td>
</tr>
<tr>
<td>RANSAC</td>
<td>0.55</td>
<td>0.55</td>
</tr>
<tr>
<td>Tracking</td>
<td>16.05</td>
<td>0.48</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>54.75</strong></td>
<td><strong>2.19</strong></td>
</tr>
</tbody>
</table>

than the corresponding CPU implementation. The speedup is mainly due to the fact that the GPU implementation fully exploits the inherent data parallelism in the feature tracking kernel.

Figure 5.5 shows the execution time for varying number of feature points. For a large number of feature points, the RLCT algorithm starts leveling off, due to the small number of iterations required for convergence. Figure 5.6 shows the execution time for varying image sizes. The feature tracking time increases minimally with an increasing image size. This is because the RLCT algorithm does not require building up the image pyramid for tracking. Thus, RLCT partly removes the dependency on the image size.

Table 5.2 presents the comparison of the execution time of RLCT with other
state-of-the-art implementations of the pyramidal KLT tracking algorithm on the
Table 5.2: Comparison with other methods

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Image Size</th>
<th>Feature Count</th>
<th>Time (ms)</th>
<th>Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLCT CUDA</td>
<td>720×576</td>
<td>100</td>
<td>0.95</td>
<td>GeForce 280 GTX</td>
</tr>
<tr>
<td></td>
<td>720×576</td>
<td>1000</td>
<td>1.96</td>
<td>+ Intel Core</td>
</tr>
<tr>
<td></td>
<td>1024×768</td>
<td>1000</td>
<td>2.15</td>
<td>2 Duo, 2.66 GHz</td>
</tr>
<tr>
<td>Ohmer et al [23]</td>
<td>720×576</td>
<td>100</td>
<td>10.8</td>
<td>GeForce 8800 GTX</td>
</tr>
<tr>
<td>Zach et al [38]</td>
<td>720×576</td>
<td>1000</td>
<td>3.2</td>
<td>GeForce 8800 Ultra</td>
</tr>
<tr>
<td>Sinha et al [33]</td>
<td>1024×768</td>
<td>1000</td>
<td>47</td>
<td>NVIDIA 7900 GT</td>
</tr>
<tr>
<td>P-KLT</td>
<td>1024×768</td>
<td>1000</td>
<td>508.9</td>
<td>Intel Core 2 Duo, 2.66 GHz</td>
</tr>
</tbody>
</table>

GPU. Other methods perform tracking on all the levels of the pyramid by using an integration window size of 7×7. For performance reasons, the image pyramid in [38] is constructed by using a small binomial kernel instead of a Gaussian kernel as implemented in CPU version of P-KLT [5]. Both [33, 38] settle the fixed number of iterations to 5. The implementation in [23] follows the same key approach used in [38] with two levels in the pyramid and the number of iterations fixed at 10. In contrast, the GPU based implementation of RLCT method does not run for a fixed number of iterations, but only restricts the maximum number of iterations to 10. Moreover, the RLCT is performed for three sizes of integration window which is functionally equivalent to the tracking performed in three levels of the pyramid. RLCT algorithm on GPU is 236 times faster than the original P-KLT algorithm on CPU. Table 5.2 shows that the GPU based implementation of RLCT is the fastest compared to state-of-the-art implementations of the pyramidal KLT tracking. The large speedup compared to other state-of-the-art methods is mainly due to three reasons. First, the RLCT algorithm does not require the construction of an image pyramid and hence saves upon computations required for down-sampling and gradients calculations in these down-sampled images. Second, the iterations for tracking are terminated once the convergence is reached instead of performing it for fixed number of iterations. Finally, even though there is extra computation because of larger window sizes utilized up to 125 points, at the same time it provides better 2D data locality and hence provides efficient utilization of texture cache.

The most interesting result is that the GPU implementation of RLCT reaches 1000 fps in tracking 125 feature points, which would never be possible with the P-KLT approach due to its fixed overhead of constructing the image pyramid before tracking.

5.6 Application to Image Registration

Feature point detection and tracking is used for registering video frames, which is the basic step for the applications such as video stabilization and video frame
Figure 5.7: Image registration of the successive images of Laparoscopic surgery video: (a), (d), (g), (j) Feature points detected by Low complexity corner detector; (b), (e), (h), (k) Robust low complexity corner tracking of the feature points in consecutive frame (black inliers and green outliers); (c), (f), (i), (l) Registration of the second image with the first image (second frame transparent and at the back)

mosaicing. Figure 5.7 and Figure 5.8 show feature detection, tracking and image registration on medical images and outdoor scene. The data set of medical images and outdoor video is derived from [18, 9, 10].
Figure 5.8: Image registration of successive video frames of University Library of Leuven, Arenberg Castle Leuven and Aerial view of parking respectively: (a), (d), (g) Feature points detected by Low complexity corner detector; (b), (e), (h) Robust low complexity corner tracking of the feature points in consecutive frame (black inliers and green outliers); (c), (f), (i) Registration of the second image with the first image (second frame transparent and at the back)
Chapter 6

Conclusion and Future Work

This report proposed an efficient implementation of low complexity corner detector and robust low complexity tracking algorithm using CUDA.

For an image size of 960×960, our method on low complexity corner detector using CUDA achieves a speedup of factor 16 times over an equivalent CPU implementation, and even outperforms the execution time of state-of-the-art implementations. The speedup is obtained from following aspects. First, we propose an efficient technique to parallelize computation of the Integral image on the GPU. Second, in contrast to other corner detector on GPUs, our method completely runs on the GPU, and hence tries to achieve the potential speedup that can be obtained by a corner detector algorithm. Finally, use of various optimizations at different stages such as launching magnitude of thread to hide memory latency by computations, efficient utilization of on-chip shared memory and coalesced access to global memory leads to an efficient solution.

Robust low complexity feature tracking using CUDA greatly exploits the data parallelism and achieves a speedup factor of 25 over an equivalent CPU implementation, and even outperforms the execution time of state-of-the-art implementations. A large speed-up is achieved from following aspects. First, our method does not require building the image pyramid, thus saving the associated computational cost of down-sampling and multilevel gradient computation. Second, it predicts the initial motion of the feature points based on a subset of them and hence reduces the number of iterations for convergence. Third, we terminate the tracking once convergence is reached instead of executing for a fixed number of iterations. Finally, the use of various optimizations like exploiting texture memory - which is cached and optimized for 2D data locality and also provides hardware based interpolation, using shared memory to reduce register pressure on each thread, and maintaining high GPU occupancy improves the overall performance. The most interesting result is that the GPU implementation of RLCT reaches 1000 fps in tracking 125 feature points, which would never be possible with the P-KLT approach due to its fixed overhead of constructing the image pyramid before tracking.

The design flow with desirable generality suggests that, compared to the traditio-
nal GPU architectures, CUDA provides more design flexibility while maintaining the huge computational power. As new generation graphics cards evolve, our implementations would be compatible and would run even faster. This now makes it possible to perform feature detection and tracking on high resolution video in real-time on most modern computers without heading to the need for special-purpose hardware solutions.

The future work can be broadly classified in the following directions:
1. It will be interesting to integrate LOCOCO and RLCT on GPU to real time applications like Video stabilization and Image moisacing. As our GPU implementations are considerably faster than optimized CPU versions making it possible to process high resolution video within real-time vision applications.
2. LOCOCO algorithm is designed to operate with single kernel size. For certain applications like object recognition, the feature detector should be robust to changes in image scale, and hence scale-invariant feature detector is required. As shown in experimental results, with the increase of kernel scale size, the execution time of LOCOCO remains close to constant, it will be interesting to extend the LOCOCO method to high speed scale-invariance corner feature point detection.
Bibliography


[26] Victor Podlozhnyuk. *Image Convolution with CUDA*. NVIDIA CUDA SDK.


52
List of Publications
