URBAN
Uitmeten, Reconstrueren, Bekijken, Animeren en Navigeren
van stedelijke omgevingen



WP1 Digital Surveying

Task 1.2 Processing on the servers
Deliverable D.1.2.3: Report on methodology for mapping on processors of mixed types

IBBT vzw, Interdisciplinair Instituut voor Breedbandtechnologie
Gaston Crommenlaan 8 (bus 102), 9050 Gent, België • E info@ibbt.be • T +32 9 331 48 00 • F +32 9 331 48 05 • www.ibbt.be

P. Rondao Alface

Maarten Vergauwen, GeoAutomation	Project Leader
Luc Van Gool, K.U.Leuven-ESAT/PSI-VISICS	Research Leader
Carolien Maertens, IMEC-NES	Work Package Leader WP1
Patrice Rondao Alface, IMEC-NES	Task Leader Responsible WP1.2
Klaas Tack, IMEC-NES	

Revision History :

Version	Reviewer	Date
V1.0	P. Rondao Alface	24 November 2009

Contents

Abstract 5
Glossary 5
1. Introduction 5
2. Hybrid CPU-GPU systems 6
2.1.1. GPGPU programming 7
2.1.2. CUDA 9
2.1.3. GPU Architecture 10
2.1.4. Hardware specificities 11
2.1.5. Dwarves 12
2.1.6. CUDA Good Practice Rules for Building Optimized Code 13
3. Selected Components of the pipeline 15
3.1. AVC decoding 15
3.2. Feature (corner) point detection 15
3.3. Feature Matching 17
3.3.1. Spatial Domain 18
3.3.2. Spectral Domain 19
4. CUDA implementation 21
4.1. Feature point detection 21
4.2. Feature point matching 24
4.3. AVC decoding 26
4.4. Debayering 26
4.5. Summary 26
5. Prototype 27
6. Conclusion 28
7. Bibliography 28

List of Figures

Figure 1. CUDA architecture and memory organization. 7
Figure 2. CUDA memory accesses (a) Traversal along rows (b) Traversal along columns (c) Traversal along tiles 8
Figure 3. Shared memory enables data re-use in CUDA contrarily to legacy GPGPU programming. 8
Figure 4. Dwarves: from "The landscape of parallel computing reasearch: A view from Berkeley" [8] 12
Figure 5. Low complexity corner (Lococo) point detection algorithm [4] 16
Figure 6. Corner points detected with the Lococo detector [4] 17
Figure 7. The blue interrogation window from image I is cross-correlated to a green interrogation window of image J regarding a displacement vector (u; v) in white (picture from [5]). 18
Figure 8. Integral image and area computation. 21
Figure 9. Scan algorithm [12]. On the left, Up Sweep and on the right, Down Sweep 22

Figure 10. Convolution and integral image speed in function of the image size. 23
Figure 12. From irregular access to aligned feature strips. 25
Figure 13. Pipeline for the processing on the servers..... 27

List of Tables

Table 1. Hardware specification used for the experiments 21
Table 2. Timing measures for the feature point detection kernel..... 23
Table 3. Timing measures for the feature matching kernel..... 25
Table 4. Timing measures for the video decoder 26
Table 5. Timing measures for the debayering kernel..... 26
Table 6. Summary table of the speed-ups obtained per kernel 26

Abstract

This report presents the methodology used for mapping code on processors of mixed types i.e. the combination of CPUs and GPUs.

Glossary

AVC: H.264 Advanced Video Coding
CUDA: NVIDIA Compute Unified Device Architecture

1. Introduction

This deliverable presents the methodology used for mapping the selected components of the URBAN pipeline on hybrid CPU-GPU systems. This deliverable mostly covers the work done in D122 of work package 1. In this work package, 3D data is extracted for the masses of buildings in cities, with an emphasis on the precision. The goal is to allow cartographers at SPC to determine the 3D positions of the different objects needed by FGIA, and this within the prescribed precisions. GeoAutomation has developed a solution through which these measurements can be largely brought into the office, from behind a desktop. GeoAutomation has developed a van with 8 cameras, and the necessary processing pipeline to extract 3D coordinates for points clicked on in an image. The traditional method would be to measure the points with GPS and photogrammetry, as field work. Now, only sparsely distributed reference points are measured on the field, to geo-reference the 3D data obtained from the imagery. Field work therefore is reduced to a minimum. This said the actual data capture as well as the processing can be made more efficient. This is the main goal of this Task: applying IMEC's know-how on speeding-up algorithms and systems to the existing GeoAutomation pipeline. This collaboration has been focused on two different areas: on-board the GeoAutomation recording van and later on during the processing on the computing servers. This deliverable is devoted to the latter.

The processing pipeline in use by GeoAutomation has already been optimized to some extent to be run in parallel on several general-purpose CPUs. As presented in the project definition, we have investigated some components of this pipeline in order to know whether they might be replaced by algorithms that are more suitable for parallelism or sped-up through implementation on GPUs with Nvidia's CUDA framework:

- decompression of the images compressed with the AVC encoder developed in Task 1 of this work package
- de-bayering (a.k.a. demosaicking) of the camera's output images
- detection of specific image features (corner point detection)
- matching the features between images (NCC)

We report here the results of our implementation of these optimized components and will further describe them in deliverable D.1.2.3.

2. 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) become attractive because they offer extensive resources even for non-visual, general-purpose computations: massive parallelism, high memory bandwidth, and general purpose instruction sets, 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.

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.* [15] 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 [8] 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 [15], noting an apparent architectural convergence of CPUs and GPUs, the authors examine the effectiveness of CUDA [9] as a tool to express parallel computation with differing sets of performance characteristics "problems from different dwarves" on GPUs.

In this report, we denote as hybrid CPU-GPU system a system in which one or several GPU(s) act(s) as massively data-parallel co-processor(s) for multicore CPUs. We make use of NVIDIA's CUDA [9] programming framework as well as of OpenMP in order to program and exploit parallelism among respectively GPU cores and CPU cores or devices. The methodology proposed is in fact a set of programming rules based on the type of code to be mapped. We highlight the main challenges when programming on such hybrid systems: *e.g.* limited communication bandwidth between CPU and GPU devices, programming constraints for efficient memory access on GPUs, parallel programming paradigms (and their corresponding relevant dwarf) that best fit on multicore CPUs or on GPUs, parallelization opportunities that require or not algorithmic re-engineering in order to enable effective accelerations.

2.1.1. 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 [11]. Due to the tremendous interest in the common public, next-generation APIs such as CUDA [9] and Brook+ [10] have been developed to further facilitate the acceleration of generic computations on GPUs.

The NVIDIA CUDA programming model [9] 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 [14], 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.

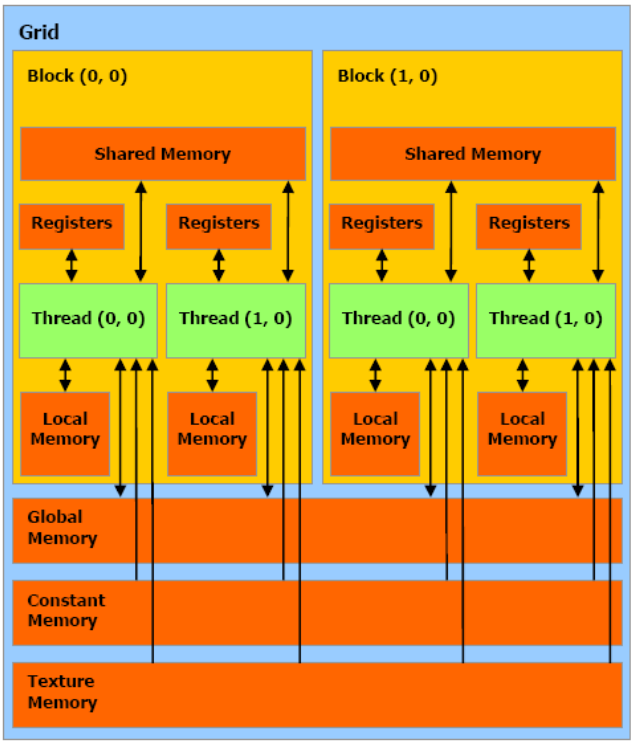


Figure 1. CUDA architecture and memory organization.

These kernels describe the work of a single thread and typically are invoked on thousands of threads (see Figure 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 2). This is particularly suited for image processing.

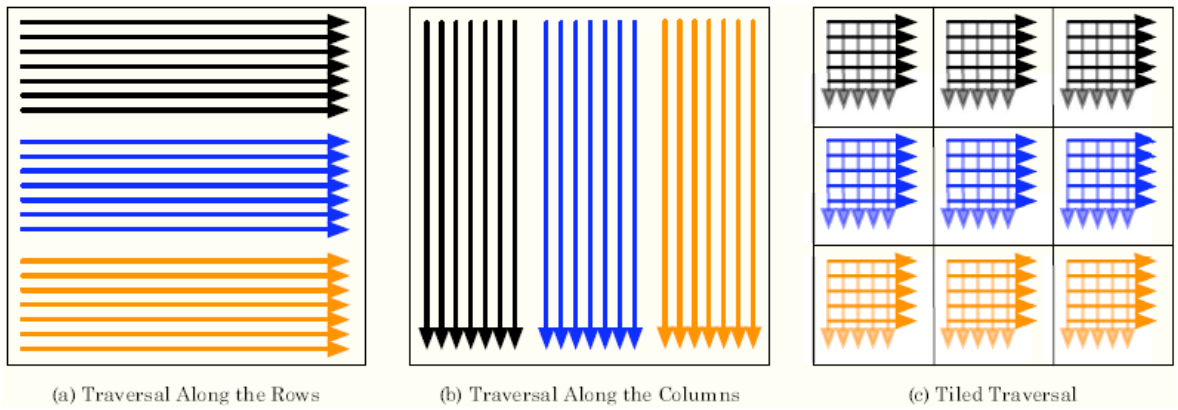


Figure 2. CUDA memory accesses (a) Traversal along rows (b) Traversal along columns (c) Traversal along tiles

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 [9].

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 compared to CPU programming. However, we notice that people either stick to the traditional approach, or start porting their implementations completely to these next-generation paradigms.

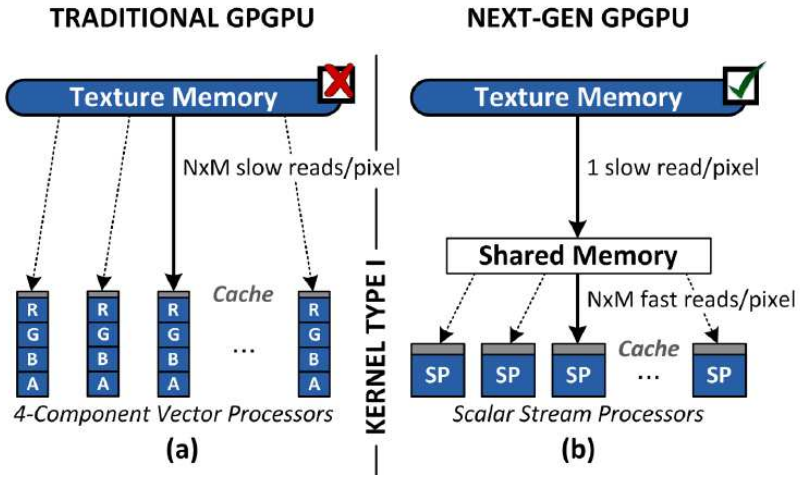


Figure 3. Shared memory enables data re-use in CUDA contrarily to legacy GPGPU programming.

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 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. The choice of the programming framework should then be made on the basis of the kernel characteristics (see Figure 3). For the remainder of the report we will however focus more on the CUDA framework as most of the kernels developed in URBAN better fitted with that framework.

2.1.2. CUDA

In this subsection we present with more details the specificities and limitations of 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 [9].

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) [15].

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.

Recent research has shown that CUDA programs can be compiled to execute efficiently on multi-core CPUs [15]. For CPUs, of course, specialized hardware, such as support for texturing, must be implemented in software.

2.1.3. 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 quad-pumped 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 [9, 14, 15].

2.1.4. 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.

2.1.4.1. 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.

2.1.4.2. Bank conflicts

In the GPU we tested, GTX 260, 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 [15]. Dealing with bank conflicts is generally lower priority than maximizing parallelism and data locality.

2.1.4.3. 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 [15]. 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.

2.1.5. Dwarves

Berkeley's dwarf taxonomy [8] is an interesting starting point for considering applications to be ported on hybrid CPU-GPU systems.

Each dwarf represents a set of algorithms with similar computation and data movement. Structured Grid, Unstructured Grid, Combinational Logic, Dense Linear Algebra, Fast Fourier Transform (FFT), N-Body, Dynamic Programming, and Monte Carlo have been covered in previous work. We feel that the kernel taxonomy should also be looked not only at the application level but also at the kernel (or component) level. Indeed reading image data might always be linked to Structured Grid or Sparse Matrix while their variety is very important in terms of memory access and computations.

Applications usually exhibit different parallelism and data-sharing characteristics, and thus take advantage of the GPU's parallel computing resources to different degrees. The level of programmer effort required to achieve satisfactory performance also varies widely across different applications. As Figure 4 illustrates, the application access patterns range from bit-level parallelism to row-level parallelism.

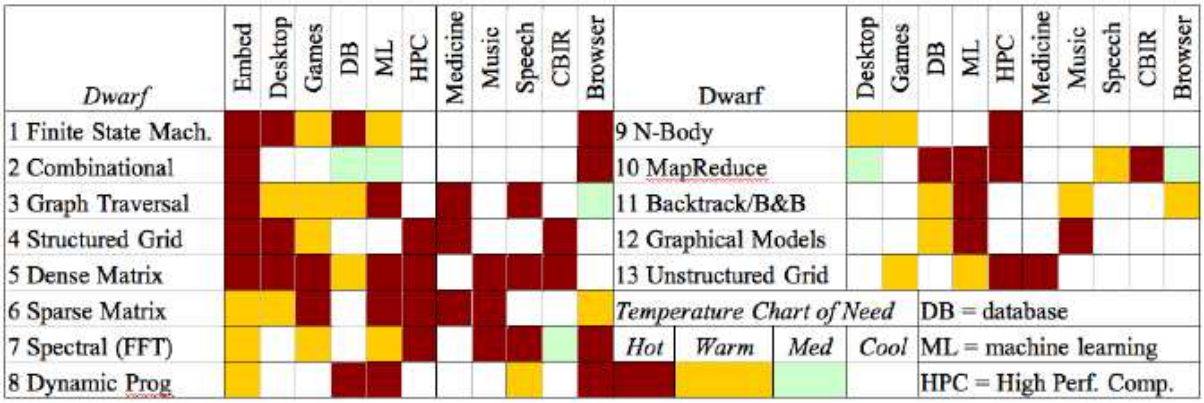


Figure 4. Dwarves: from "The landscape of parallel computing reasearch: A view from Berkeley" [8]

2.1.6. CUDA Good Practice Rules for Building Optimized Code

Here follow the optimization rules and strategy we could observe with CUDA programming.

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 on 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 [14]. For example, in the sever pipeline composed of video decoding, feature detection and feature matching, 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 really 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. Scan algorithm or image convolution are good examples of this strategy. 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.* [14] show that taking symmetric thread organizations (number of threads in x and y are equal) results in unexpected higher throughput.

We now present the different applications and kernels we mapped in the context of URBAN, the optimizations performed and the resulting accelerations when compared to CPU-only coding.

3. Selected Components of the pipeline

In D.1.2.1 we have already introduced CPU-GPU hybrid systems and an overview of the tasks to be done on the processing servers of GeoAutomation. The selection of components to be optimized has also already been described in that deliverable as well as the optimization of the debayering component. Therefore we propose to only focus here on GPU-based AVC decoding, feature point detection and matching.

3.1. AVC decoding

In this framework, the decoding of frames has to be as fast as possible. The algorithm used is based on the algorithm choice made in deliverables D.1.1.1, D.1.1.2 and D.1.1.3: H.264 Baseline Profile Intra Coding.

Interestingly, new generation Nvidia's GPUs offer a hardware accelerated H.264 decoding API through the so-called VP2 video processor [7]. This API enables developers to decode video streams on the GPU and process the decoded uncompressed surfaces within CUDA programs. The decoded surfaces can be transferred back to system memory using CUDA's fast asynchronous read-backs, or the application can use CUDA's 3D interoperability features to render the surfaces using a 3D API (OpenGL or DirectX). We report our experiments using the streams composed of GeoAutomation's data encoded with the AVC encoder delivered by IMEC in this work package. The speed-ups obtained compared to a CPU-based decoder are given in Section 3. As an alternative, NVIDIA® PureVideo™ HD Technology can also be used at this aim [7]. This is the only kernel run in legacy GPGPU in the context of this project, the others have all been coded in CUDA.

3.2. Feature (corner) point detection

In this section we present state-of-the-art corner points detection algorithms and propose a modified algorithm with lower complexity that has been mapped on GPU.

The most famous recent corner points detector are certainly Harris [1] and KLT [2, 3] feature detectors. They 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:

$$C(\mathbf{p}) = \sum_{\mathbf{x}\in W} \left\{ \begin{bmatrix} g_x^2(\mathbf{x}) & g_x(\mathbf{x})g_y(\mathbf{x}) \\ g_x(\mathbf{x})g_y(\mathbf{x}) & g_y^2(\mathbf{x}) \end{bmatrix} \times v(\mathbf{x}) \right\} = \begin{bmatrix} G_{xx} & G_{xy} \\ G_{xy} & G_{yy} \end{bmatrix}$$

$$\mathbf{x} = (x, y)$$

Here,

$$g_i = \partial_i (g \otimes I) = (\partial_i g) \otimes I, i \in (x, y)$$

with and v(x) is a weighting function which is usually Gaussian or uniform, p is a center pixel, W is a window centered at p, I is the image, g is Gaussian, gx and gy are convolution of Gaussian first order partial derivative with I in x and y directions respectively at point (x, y).
The Harris corner detector evaluates the cornerness without explicit eigenvalue decomposition:

$$R = |C| - k \times (trace(C))^2$$

The image pixel is corner, if it has large positive eigenvalues and thus large value in R and R is above a specified threshold. KLT explicitly calculates the eigenvalue of C. It selects those points for which the minimum eigenvalue is larger than some threshold [2].

$$R = \lambda_{\min} = \min(\lambda_1 \lambda_2)$$

$$\lambda_{\min} = \frac{1}{2}(G_{xx} + G_{yy} - \sqrt{(G_{xx} - G_{yy})^2 + 4 \times G_{xy}^2})$$

The detection of features by KLT or Harris is divided into three main steps [1, 2]: a) the Gaussian derivates of the original image, where gx and gy are calculated by convolution; b) the autocorrelation matrix C and cornerness measure R, which are evaluated for each pixel individually; c) Quick-Sort and non-maximum suppression, which are used to suppress the less strong points locally.

In this deliverable, we follow the strategy proposed in [4] which proposes to reduce the complexity in two aspects as described in Figure 5. First, the integral image is used to lower the complexity of convolution and evaluation of autocorrelation. Second, efficient non-maximum suppression is adopted.

1. Calculate the integral image of the original image I
2. Approximate the Gaussian derivative kernel by the Box kernel, and compute g_x and g_y by using the integral image
3. Create the integral image of g_x^2, g_y^2 and $g_x g_y$
4. Evaluate R
5. Efficient non-maximum suppression

Figure 5. Low complexity corner (Lococo) point detection algorithm [4]

More details on the algorithm and its repeatability (robustness of the feature point detection against image or light condition transformations) are to be found in reference [4]. An example of the simplified corner detector output for the URBAN data is illustrated on Figure 6. Results are given in Section 3.



Figure 6. Corner points detected with the Lococo detector [4]

3.3. Feature Matching

Many metrics exist in order to estimate feature matches (sum of square differences, normalized cross correlation, mutual information…). Here we focus on the method preferred by GeoAutomation for its low complexity and robustness to illumination changes: normalized cross correlation (NCC) and analyze ways of optimizing this technique using CUDA.

Normalized cross-correlation divides both images into interrogation windows of the same size (see). All interrogation windows either do not overlap or overlap regularly. We denote the size of an interrogation window with m (in all dimensions).

Typically, an interrogation window size is 32 x 32. The number of interrogation windows is denoted k = n/m. Note, that the same division pattern is used for both images.

Regardless of the algorithm, preprocessing of the interrogation window data in both images helps improving the displacement results. It is advantageous to subtract the mean values μI and μJ from both windows in order to shift the intensity values to the common basis, where

$$\mu_I = \frac{1}{m^2} \sum_{i=1}^{m} \sum_{j=1}^{m} I(x+i, y+j)$$

and μJ analogously. If the result of the subtraction gets smaller than 0, we clamp it to 0. Implementation-wise, each image is stored in a texture.

In the following, we discuss the spatial domain cross-correlation and the frequency domain cross-correlation

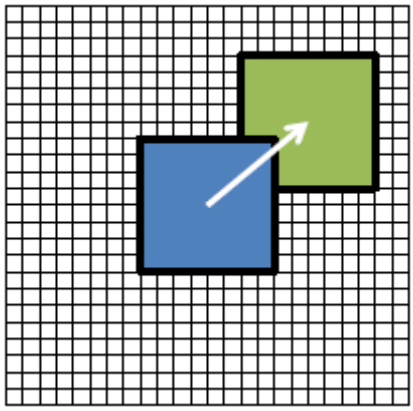


Figure 7. The blue interrogation window from image I is cross-correlated to a green interrogation window of image J regarding a displacement vector (u; v) in white (picture from [5]).

3.3.1. Spatial Domain

Given a 2D displacement vector (u; v), the cross-correlation between image I and J can be computed as

$$C_{spatial}(u,v) = \sum_{i=-m/2}^{m/2} \sum_{j=-m/2}^{m/2} I(i,j) \times J(u+i, v+j),$$

if m is an even number. In order to find the displacement vector for each interrogation window, the correlation for each displacement vector (u; v) living inside the search window s is computed. In Figure 3, the search window size is s = 2m, since the search window size is measured from the center of the interrogation windows. The search window size s can be chosen arbitrarily, but larger search windows account for a more global solution than small ones. Assumptions about the maximum displacement help optimizing the search window size, since the larger the search window the slower the computation.

Finally, the displacement vector (u; v) with the largest cross-correlation in the search window is selected as the resulting displacement vector for each interrogation window. The likelihood of false maxima can be reduced by normalizing the previous equation with division by

$$\sqrt{\sum_{i=-m/2}^{m/2}\sum_{j=-m/2}^{m/2}(I(i,j)-\mu_I)}-\sqrt{\sum_{i=-m/2}^{m/2}\sum_{j=-m/2}^{m/2}(J(i+u,j+v)-\mu_J)},$$

where μI and μJ are the mean values of the current interrogation window of the images I and J. The spatial domain cross-correlation has complexity O(kd md sd), where d is the number of dimensions, here d = 2. Assuming that the search window size s is of similar size as the interrogation window size m, the complexity of the spatial domain cross-correlation is O(kd md+d). In 2D, the dominant term is m, so the complexity is O(m4).

This algorithm implies many texture or global memory fetches in CUDA as in the case of full search motion estimation in AVC/H.264 video coding (deliverable D.1.1.1). In this context, using shared memory reduces the memory loads latency by using data locality as well as using a reduction strategy. Reduction is a well-known parallel programming design pattern where sub-results are computed recursively [6]. More details on our optimizations will be given in next deliverable D.1.2.3.

3.3.2. Spectral Domain

In order to reduce the amount of computation time, the cross-correlation can also be computed in frequency domain. Using the Wiener-Khinchin Theorem, the cross-correlation can be formulated in frequency domain as

$$C_{frequency}(w)=\max(\mathrm{F}^{-1}(\mathrm{F}(I_w)\cdot\mathrm{F}^{*}(J_w))),$$

where F denotes the Fourier transform, F-1 the inverse Fourier transform, F* the complex conjugate of the Fourier transform, Iw and Jw denote an arbitrary pair of interrogation windows, and the max operator finds the maximum value (peak) in the window m x m. The max operator is often extended to account for sub-pixel accuracy. Given an interrogation window in image I, and an interrogation window at the same position in image J, the cross-correlation can be found by transforming both windows to the frequency domain, complex conjugate multiplying both, and transforming the result back to spatial domain, and, finally, searching for the maximum value. The computational complexity of this algorithm is significantly lower than the spatial domain algorithm with complexity O(m4). The frequency domain approach is dominated by the two Fourier transforms. A 2D FFT has the computational complexity O(m2 logm) if applied to a window of size m x m. The complex conjugate multiplication and the search for the maximum are all O(m2). So, the total complexity is O(m2 logm).

It is obvious that Cspatial ≠ Cfrequency. Overall the quality of the frequency domain approach does not match the spatial domain counterpart. The frequency domain approach relies on the periodicity assumption of the Fourier transform which is not fulfilled. Nevertheless, the frequency domain approach is much faster on CPU, and the results still convincing.

Now, we discuss the GPU implementation of this approach. CUDA offers Fast Fourier Transform in its cuFFT library. We measured the optimality of this library in this context. It clearly appeared that cuFFT is optimal only for

large window sizes (typically above 512x512). Transforming all our small interrogation functions (9x9) sequentially is not efficient and turns to be slower than the CPU.

Furthermore, cross-correlation is a complex conjugate multiplication of the previously transformed windowed images. Since two complex values are multiplied to one, two values are repacked in order to exploit the parallelism of the graphics hardware architecture. Using the same library as described above, the inverse windowed FFT is executed in order to transform the frequency domain response back to spatial domain. This procedure has still to be done sequentially which is in practice much slower than the spatial method for which the full parallelism of the GPU can be exploited.

Finally, the displacement vector of each block is determined by finding the peak (maximum value) in each window. Once again, a reduction operator seems to be the best choice in order to find the position of the maximum in each block.

For further peak accuracy improvements, sub-pixel displacement has to be taken into account. The neighbourhood around the previously found peak contributes to the correct solution. We have implemented the most common fitting functions centre of mass and Gauss fit. In 1D, the functions are as follows if vi is the peak location.

$$\text{Center of mass: } \frac{v_{i-1}-v_{i+1}}{v_{i-1}+v_i+v_{i+1}} \quad \text{Gauss fit: } \frac{\ln v_{i-1}-\ln v_{i+1}}{2\cdot(v_{i-1}+2v_i+v_{i+1})}$$

Once the displacement vectors of all interrogation windows have been found the set of vectors forms a vector field. In the following, we optimize each vector with respect to its neighbourhood. Typically, smooth vector fields are desirable. An outlier vector is a vector that does not share similar orientation or length with regard to its neighbours. A simple method to correct outliers is to replace them by the average vector in its neighbourhood while the support can be chosen arbitrarily. This process is efficiently implemented using CUDA. More sophisticated algorithms to correct outliers can, of course, also be implemented.

In order to increase the density of the vector field, additional cross-correlation passes with equally shifted interrogation windows can be computed. Now, the interrogation windows no longer are disjoint but overlap and special care must be taken at boundaries of the images. This strategy has however not been explored as cuFFT does not enable for an efficient parallel processing of the feature match estimations.

4. CUDA implementation

This section is devoted to the results of our implementation using a state-of-the-art CPU-GPU system (Table 1.).

Table 1. Hardware specification used for the experiments

CPU	Intel® Core™ i7 Processor: 3.06 GHz 8 MB of Intel® Smart Cache 3 Channels of DDR3 1066 MHz memory 4 GB RAM
GPU	Nvidia's GeForce GTX 280: 1.296 GHz clock speed 240 CUDA cores 1 GB global memory 16 KB shared memory per core 65535 maximum number of threads Compute capability 1.3

4.1. Feature point detection

The followed strategy is the following. The application is here composed of mainly five steps.

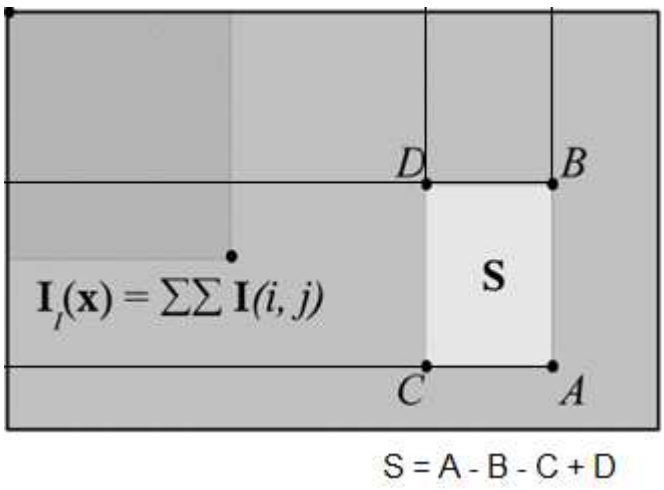


Figure 8. Integral image and area computation.

The first one consists in computing an integral image (see Figure 8) and the second in simple pixel differences in order to evaluate the gradients. The integral image is easy to port on GPU by making use of the scan

algorithm [12]. The way the shared memory is exploited in this context is illustrated on
Figure 9. Rows can efficiently be processed in parallel in shared memory. In order to propagate the can on columns, it is better to store this information in global memory and transpose this resulting image. Therefore a new row-based scan can be launched avoiding any bank conflict.

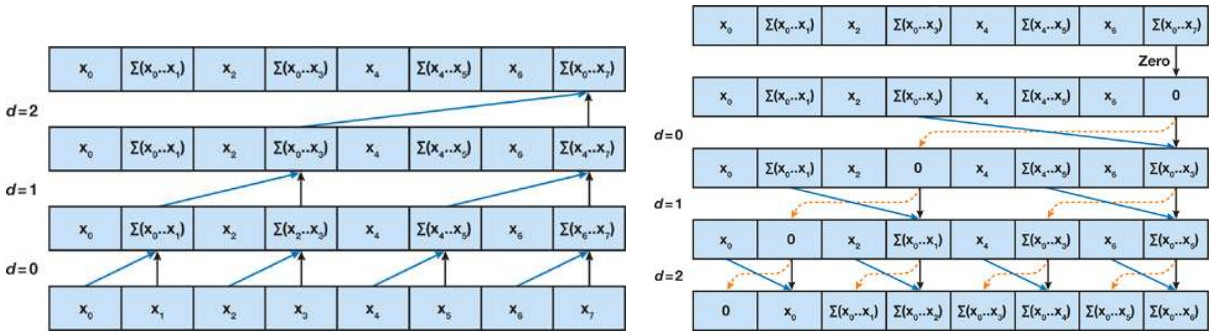


Figure 9. Scan algorithm [12]. On the left, Up Sweep and on the right, Down Sweep

The original Harris algorithm uses Gaussian convolution instead and contrarily to CPU programming the GPU implementation of convolution can be faster than launching two scans and one image transpose. It depends in fact on the scale of the Gaussian kernel used (see Figure 10). It appears that a convolution kernel size (which is equal to four times the sigma value corresponding to the chosen scale) above 16 is always slower than the integral image computation. At larger image resolutions, larger scales make sense depending on the application needs. However, it can be seen that for low scales, convolution is faster. One point can here promote the integer image: in the case of a multi-scale estimation, the convolution must be computed for each scale while only one execution of the integral image allows for computing all scales. In that case, integral image and the Lococo algorithm are much more efficient than the strategy used in the original Harris algorithm.

Table 2 illustrates the acceleration of the feature point detection for 1632*1248 images. The CPU results are given for the Lococo detector which accelerates KLT by a factor of 8. Our mapping on the GPU gives with no loss in quality a further speed-up of 7.2 or 12 if non-maximum suppression is included or not respectively. Indeed, this step is the bottleneck of the GPU implementation du to the highly controlled nature of a radix sorting algorithm. This step is however less an issue for the CPU and we can decide to re-transmit the cornerness results to the CPU in order to process the sorting there. The other possibility is to avoid this communication and store the features normalized with their neighborhood on the GPU global memory in order to minimize the CPU-GPU communication bandwidth usage and optimize the read-out configuration for the feature matching kernel. This is the solution we selected.

The kernel has to store 8 times the equivalent of the memory needed for one frame (2.03 MB) in order to store intermediate results of each step of the algorithm. This means that a maximum of 10 images can be processed in parallel on the GPU using our implementation at a speed of 40 frames per second.

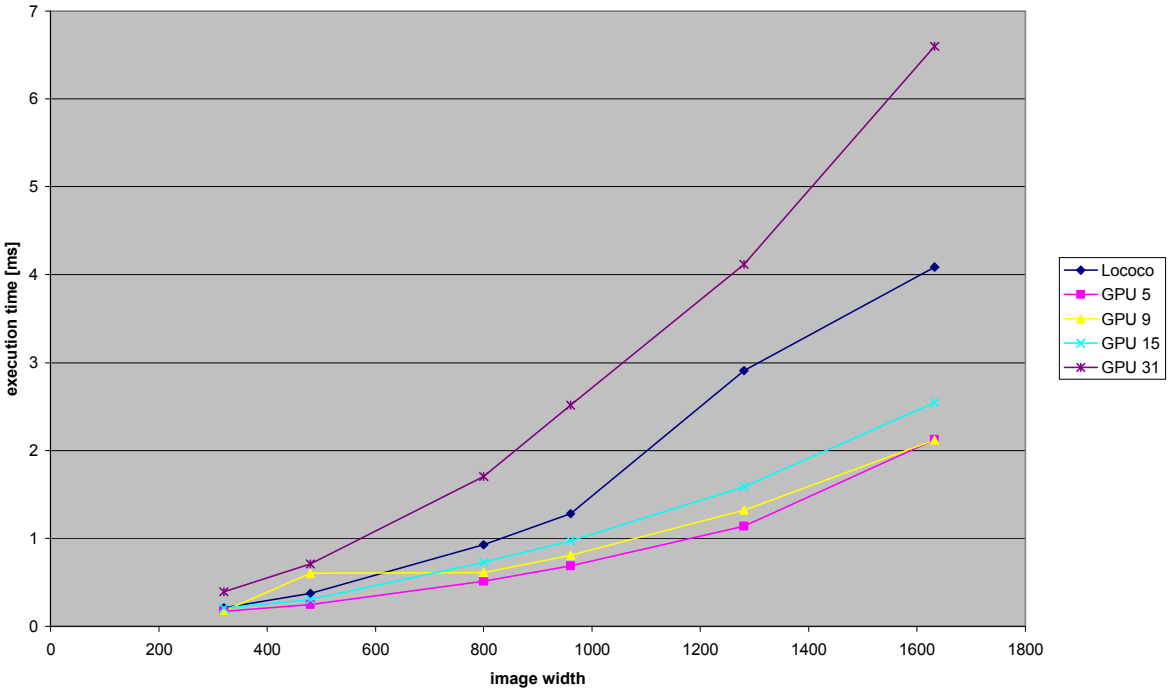


Figure 10. Convolution and integral image speed in function of the image size.

The CPU-GPU communication is limited to the transmission of the input image at a speed of 4.1 GB/sec (434 frames per second if represented in float). The output is stored on the GPU global memory and can be communicated back to the CPU or kept on-board for the next kernel. The amount of memory needed is proportional to the number of features detected (typically around 1000 features per frame which results in 400KB per frame).

Table 2. Timing measures for the feature point detection kernel

Feature detection	CPU (ms)	GPU (ms)	Speed-up
Integral Image	18.3	1.41	13
Gradient	45.4	1.26	36
Integral Gradient	75.1	7.5	10
Cornerness	17.6	1.1	16
Non maximum suppression (NMS)	22.7	3.2	7.1
Total time	179.7	25	12.42
Total time without NMS	157	13.2	13.9

A more detailed comparison of the resulting accelerations for each component of the feature detection is given on Figure 11. The Integral gradient consists in computing the cross products of the gradients and summing them on a 9x9 neighborhood. The cornerness estimation is then a

pixel-based estimation of a value depending on the sums of the cross-products of the gradients. NMS stands for Non-Maximum Suppression and consists in sorting the cornerness values inside tiles of the image and then detecting whether these maximums are also the maximum value inside a neighborhood centered on their position. This is done by an extensive use of shared memory again as well as a rapid sorting algorithm [13].

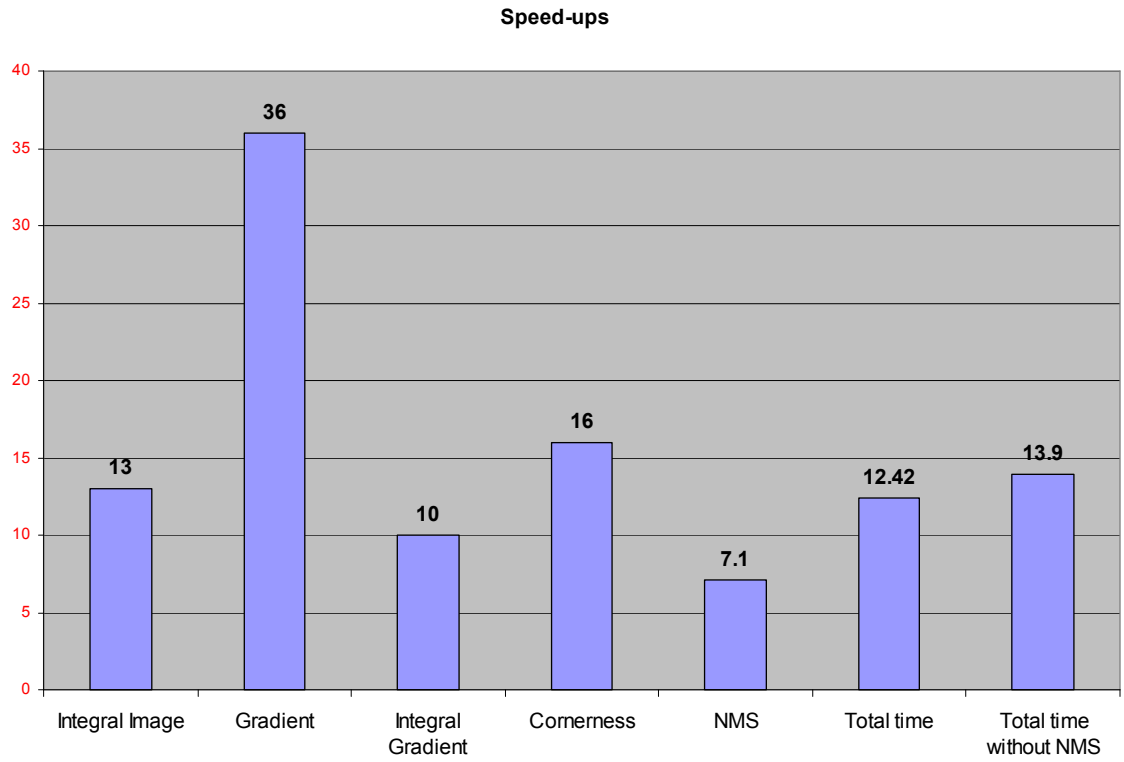


Figure 11. Acceleration of the feature detection application.

4.2. Feature point matching

As discussed above, the spatial method for computing NCC enables to take full advantage of the GPU parallelism. As the features are already stored in global memory during the feature detection kernel, no CPU to GPU communication is needed in principle. The kernel simply consists in a correlation and a quick sort of the matching results for each feature.

There are here some choices that can be made in order to find the best feature matches pairs.

First, full search comparison enables to match each feature of one image A with each feature of another image B. This is what is timed in Table 3 as 1000x1000 features match. The bottleneck here is clearly the quick sort as for the feature detection. Anyway, nice accelerations can be observed.

Table 3. Timing measures for the feature matching kernel

Feature matching (NCC)	CPU (ms)	GPU (ms)	Speed-up
1000 x 1000 features	398	12.5	31.84
1000 x 16 features	6.05	0.11	54.5
16x16 features	0.11	0.010	10

Another possibility is to select for each feature of image A given set of features from image B which lie in a research window or along a given row, column or line that has to be pre-computed by the CPU. This means that the feature information should be sent back to the CPU before the matching kernel in case complex computations are needed in order to reduce the number of possible candidates per feature. We only measure here the time needed in order to match each of the 1000 features of image A with 16 randomly selected features of image B (which are not the same for different features of A). The time needed in order to compare 16 features with 16 other features is also given in Table 3 as an indication.

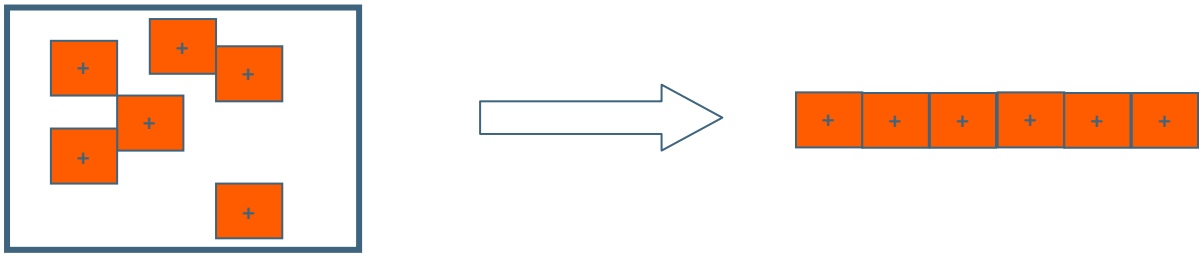


Figure 12. From irregular access to aligned feature strips.

The strategy used here in order to accelerate the feature matching process has been the following. As the average values of the feature neighborhood have to be computed first and that the feature position distribution is generally not uniformly organized in the image, the data access as well as the synchronization issues prevent from getting the best possible acceleration. It is however possible to solve this issue by modifying the output of the feature detection application. Indeed if the features and their neighborhoods are stored in global memory as aligned strips of data, it becomes possible to access them in a fast and uniform way (Figure 12). Furthermore, it is also possible to already "center" these neighborhoods (this means subtracting the average of the feature neighborhood to each pixel of the neighborhood) so that the NCC computation does not need anymore to compute the neighborhood average. This is efficiently done at the feature detection side as the integral image is available. This explains why very good accelerations can be obtained without modifying the results of the feature detection since no CPU-GPU communication is involved.

4.3. AVC decoding

AVC decoding thanks to the hardware accelerated features of VP2 is surprisingly not so efficient compared to a CPU decoder. This is explained by the fact that streams are composed of only one frame intra coded. Intra prediction is very efficient at the encoder side but more complex at the decoder side. Also, the decoder achieves better performances on streams containing multiple frames. Anyway, decoding on the GPU enables to perform other tasks on the CPU. Times given in Table 4 take the CPU-GPU communication into account.

Table 4. Timing measures for the video decoder

AVC decoding	CPU (ms)	GPU (ms)	Speed-up
1632x1248 intra coded frames	111	56	1.98

However, if we consider the whole application, this AVC decoding enables to significantly reduce the CPU-GPU data transfer amount. It turns out that it is still more interesting to run the decoding on the GPU as no raw image communication is needed (see Figure 13).

4.4. Debayering

Debayering is intended to be processed on a GPU on-board of the GeoAutomation's van. This simple kernel already presented in deliverable D.1.2.1 maps particularly well on a GPU as illustrated on Table 5.

Table 5. Timing measures for the debayering kernel

Debayering	CPU (ms)	GPU (ms)	Speed-up
Edge-adaptive	0.025	0.002	12.25
Non edge-adaptive	0.028	0.002	13.8

4.5. Summary

Finally Table 6 recaps the measures by giving the speed of each kernel in frames per second.

Table 6. Summary table of the speed-ups obtained per kernel

Summary	CPU (fps)	GPU (fps)
Feature detection	5.6	40
Feature matching	2.5	80-9000*
AVC decoding	9	17.8
Debayering	40	490

* frame pairs per second

5. Prototype

The implementation of these kernels can be tested on simple request by email. The final pipeline is represented on Figure 13.

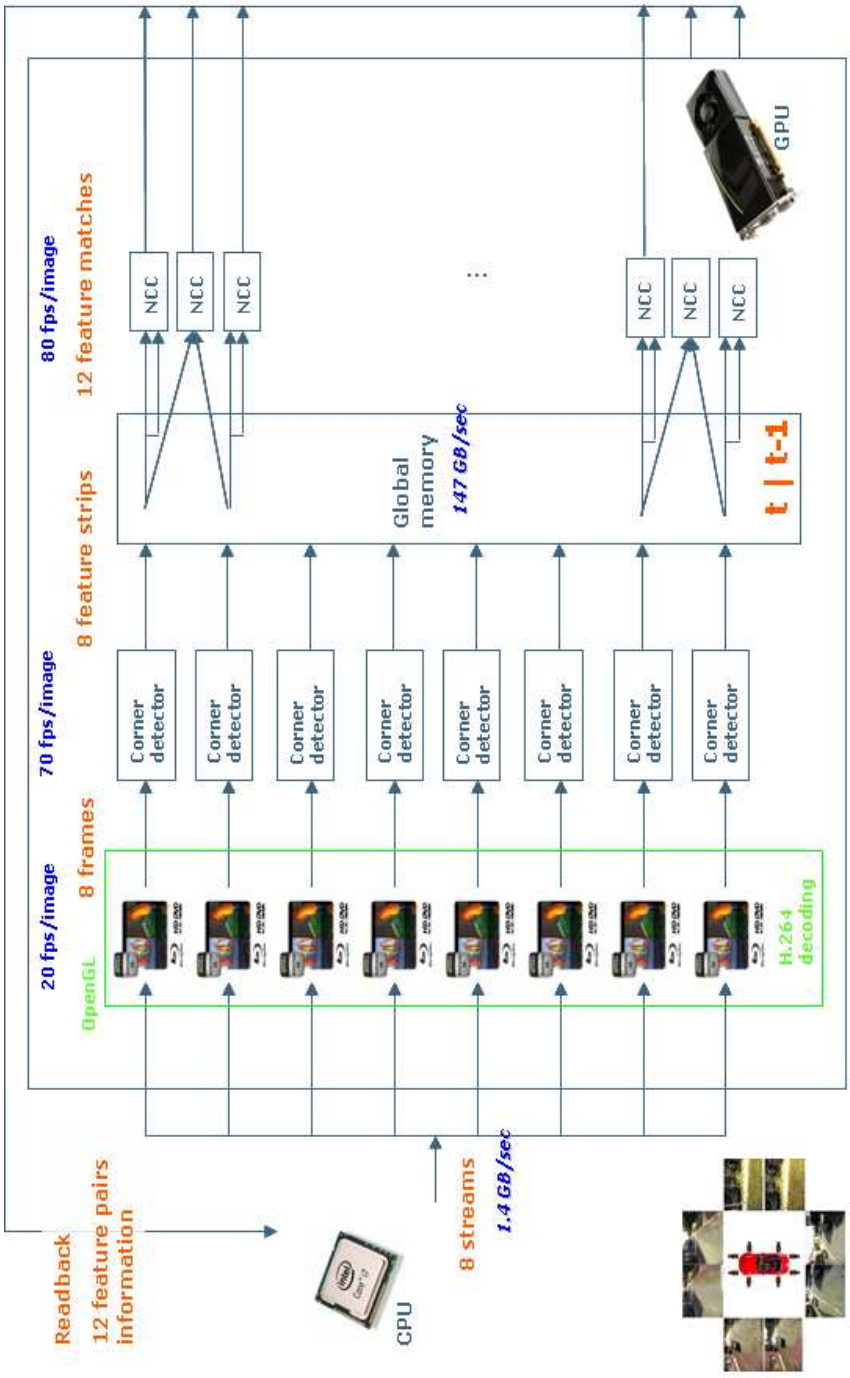


Figure 13. Pipeline for the processing on the servers

6. Conclusion

In this deliverable, we have presented the methodology used in order to build an optimized CUDA implementation of the selected components of the GeoAutomation server's pipeline.

7. Bibliography

[1] C. Harris and M. Stephen, "A Combined corner and edge detector" In Proce. of Alvey Vision Conf., pp. 147-151, 1988.
[2] Tomasi C. and T. Kanade, "Detection and tracking of point features", Technical Report CMU-CS-91-132, Carnegie Melon University, Pittsburgh, April 1991.
[3] J. Shi and C. Tomasi, "Good Features to Track", in CVPR, 1994
[4] P. Mainali, Q. Yang, G. Lafruit, R. Lauwereins and L. Van Gool, "LOCOCO: LOW COMPLEXITY CORNER DETECTOR", submitted to ICASSP 2010.
[5] T. Schiwietz, "Acceleration of Medical Imaging Algorithms Using Programmable Graphics Hardware", PhD thesis, Institut fur Informatik Technische Universitat Munchen, 2008.
[6] G. E. Blelloch, .Prefix sums and their applications, in Synthesis of Parallel Algorithms, J. H. Reif, Ed. Morgan Kaufmann, 1993.
[7] http://www.nvidia.co.uk/page/purevideo_HD.html
[8] K. Asavic et al., The Landscape of Parallel Computing Research: A View from Berkeley, Technical report no. UCB/EECD-2006-183, available at http://eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html
[9] CUDA, http://developer.nvidia.com/object/cuda.html
[10] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, Brook for GPUs: Stream Computing on Graphics Hardware, In SIGGRAPH 04, Computer Graphics Proceedings, 2004
[11] M. McCool, Z. Qin and T. Popa, Shader metaprogramming, In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2002
[12] M. Harris, S. Sengupta, and J. D. Owens. "Parallel Prefix Sum (Scan) with CUDA". In Hubert Nguyen, editor, GPU Gems 3, chapter 39, pages 851–876. Addison Wesley, August 2007
[13] N. Satish, M. Harris, and M. Garland. "Designing Efficient Sorting Algorithms for Manycore GPUs". Proc. 23rd IEEE International Parallel & Distributed Processing Symposium, May 2009.
[14] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk, W.-M. Hwu. "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA". Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 73--82, February, 2008
[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA", Journal of Parallel and Distributed Computing,Volume 68, Issue 10,October 2008, pp. 1370--1380