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



WP1 Digital Surveying

Deliverable D1.2.1: Report on the requirements, properties and parallelization opportunities for the processing pipeline

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 WP1.1
Klaas Tack, IMEC-NES	

Revision History :

Version	Reviewer	Date
V1.0	P. Rondao Alface	27 June 2008
V1.1	M. Vergauwen	27 June 2008

Contents

Abstract 4
Glossary 4
1. Introduction 4
2. Parallelization strategies 6
  2.1. GPGPU – CUDA 7
  2.2. Multi-core 8
3. Processing on the van 8
  3.1. De-bayering (a.k.a. demosaicking or demosaicing) 9
    3.1.1. Interpolation-based demosaicking without edge detection 10
    3.1.2. Edge-adaptive demosaicking 11
    3.1.3. Experiments on CPU and GPU 12
  3.2. Compression 14
  3.3. Interactions between De-bayering and compression 14
4. Processing on the servers 14
  4.1. Overview 14
  4.2. Image decoding 15
  4.3. Feature detection and matching 15
  4.4. Plane-Sweeping 16
5. Conclusion 16
6. Bibliography 16

List of Figures

Figure 1 System composed of the pipeline on the van and the pipeline on the servers of GeoAutomation. Above: previous solution with raw image storage on the van. Below: on-board de-bayering and compression on GPU performed on the van. 6
Figure 2 Floating-point operations per second on GPU and CPU 7
Figure 3 Differences between a CPU and a GPU 8
Figure 4 Four possible bayer types 9
Figure 5 Zipper effect. On the left, the zipper effect is generated by demosaicking without edge-adaptive interpolation. On the right, edge-adaptive demosaicking reduces these distortions [LJL07]. 10
Figure 6 Processing on the servers: 3D reconstruction and georeferencing 15

Abstract

In this report, an analysis of the requirements, properties and parallelization opportunities for the processing pipeline is presented. Since some parts of the process are actually to be moved from the servers to the van, the report focuses accordingly on both processing pipelines. The description of the requirements and the possible parallelization strategies that could be developed are then proposed for each of the pipelines with their specific requirements. In some way, this report overlaps then Task 1 which is dedicated to the processing on the van, even though this deliverable was originally meant for Task 2 whose topic is the processing on the servers.

Glossary

AVC: H.264 Advanced Video Coding
CUDA: NVIDIA Compute Unified Device Architecture
DCT: Discrete Cosine Transform
DWT: Discrete Wavelet Transform
GPU: Graphics Processing Unit
MVC: H.264 extension to Multi-View Coding
PSNR: Peak Signal-to-Noise Ratio

1. Introduction

This first work package extracts 3D data 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 WP. IMEC's know-how on speeding-up algorithms and systems will be applied to the existing GeoAutomation pipeline. This collaboration will be focused on two different areas: on-board the GeoAutomation recording van and later on during the processing on the computing servers.

As illustrated by the processing pipeline of GeoAutomation, there is a clear trend and need for implementation on parallel systems. In the past, parallel implementations were not needed because technology scaling enabled doubling the performance of mono-processor systems every 18 months. This law was true for almost 30 years, but in 2006, the gain is clearly slowing

down (by a factor of three). The main reasons for this decreased growth are:

1. The ILP wall: in the past, parallelism was exploited by adding more instruction level parallelism to a single CPU. The extra performance gain because of this is however coming to its end [HP07].
2. The Memory wall: the increasing gap between performance of the memory and processor slows down the overall performance of the system [M04].
3. The Power wall: Power consumption and heat removal have become first order limiters to Moore's law [N05].

Multi-core processor technology is currently seen as the solution to overcome the ILP, Memory and Power walls. Although Multicore processors help to accelerate a mix of independent, but sequential tasks, it is not seen as the ultimate solution to overcome the ILP, power and memory walls. Asavic et al. [A06] claim that only many-core (> 1000 processors) technology will really help to overcome the red brick wall of the computing industry. One of the great challenges for the many-core processor era is to enable the easy programming of such systems.

Graphics Processing Units (GPUs) are massive parallel and programmable systems for the acceleration of algorithms that can be rewritten in the streaming programming model. Because the streaming programming model is one of the programming models that will be used in the many-core future, it is an interesting target architecture to further explore the easy mapping of multimedia algorithms. We will therefore research which algorithms from the GeoAutomation pipeline can be rewritten in a streaming programming model and efficiently mapped on a GPU using the CUDA framework [C08].

The processing pipeline in use by GeoAutomation has already been optimized to some extent to be run in parallel on several general-purpose CPUs. It can be investigated if some components might be replaced by algorithms that are more suitable for parallelism or sped-up through implementation on GPUs:

- decompression of the images, e.g. motion compensation, IDCT, …
- detection of specific image features
- matching the features between images (SSD, NCC, …)
- trying out and verifying hypotheses for the camera transformations

Several problems can appear here:

- What is the architecture of the software and more importantly of the data-flow? The overhead of data transport can annihilate possible speed gain from the GPU.
- Too many if-then-else paths make it near-to impossible to speed-up the software.

Many researchers have already explored the use of the GPU for general purpose tasks. Some of them have even developed high-level languages that hide the specifics of graphics programming. However, it is still difficult to judge whether: (i) an algorithm can be efficiently rewritten using the streaming programming model and (ii) it can be efficiently implemented on a CPU/GPU system with a communication bottleneck between the two. In URBAN, we want to develop a methodology to solve these issues.

The topic of this deliverable is to report the requirements, properties and parallelization opportunities for the processing pipeline on the van and on the servers. As explained in the deliverable D1.1.1. and illustrated on Figure 1, it can be useful to move the de-bayering and compression from the

servers to the van in order to increase the size of 3D scenes that can be reconstructed from a single recording session of the van. This part of the pipeline is the main focus of this deliverable. Indeed, it has received most of our attention until now. The conclusions of this analysis will be integrated into a more systematic approach and will be the basis of the methodology that will be used for the optimization of the processing on the servers.

The organization of the deliverable is as follows. A first section gives an overview on parallelization strategies on two kinds of platforms: GPU and multi-core. Section 3 is dedicated to the processing on the van and Section 4 introduces the processing on the servers.

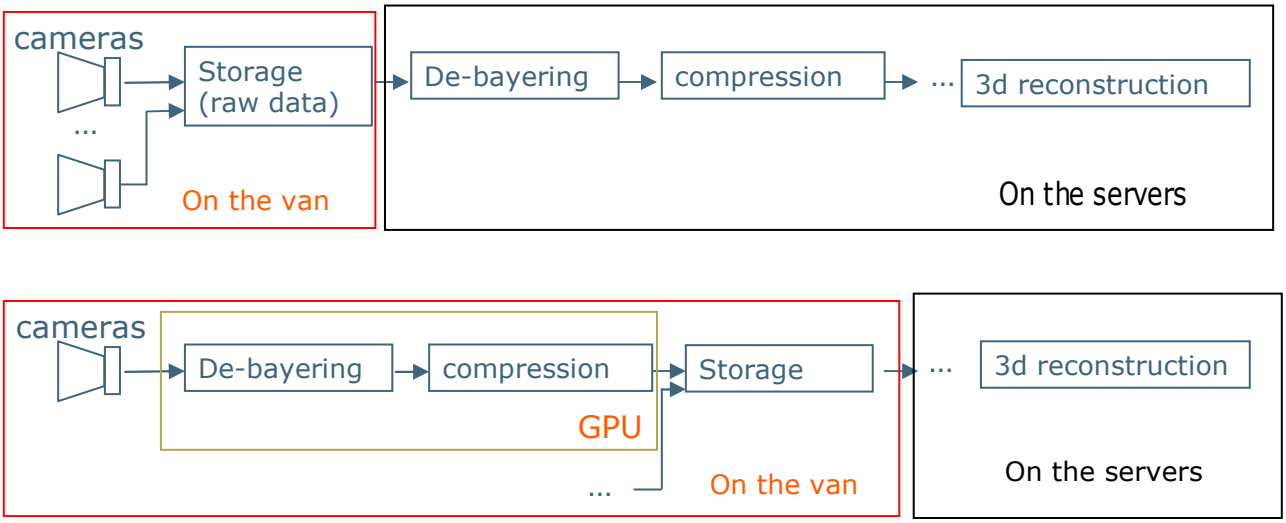

Figure 1 System composed of the pipeline on the van and the pipeline on the servers of GeoAutomation. Above: previous solution with raw image storage on the van. Below: on-board de-bayering and compression on GPU performed on the van.

2. Parallelization strategies

Currently, there is a trend in the semiconductor industry to move from single-core designs in General Purpose Processors (GPPs) to multi-core design. Another trend is using a Graphics Processing Unit (GPU) for general purpose programming (GPGPU). Both architectures offer a lot of processing power employing multiple processing units which can be used in parallel.

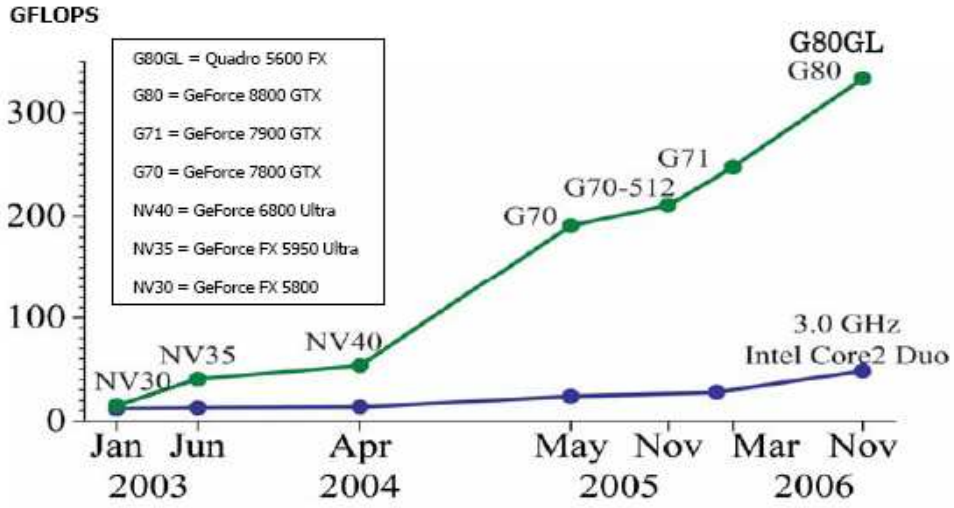

Figure 2 Floating-point operations per second on GPU and CPU

2.1. GPGPU – CUDA

As illustrated on Figure 2, with multiple cores driven by very high memory bandwidth, today's GPUs offer resources outperforming CPUs for both graphics and non-graphics processing. This can be explained by the fact that the GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore is designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 3. Parallel computing on GPUs is principally based on Data-level parallelism. This means that not every algorithm will perfectly fit in this framework. However, many image processing algorithms can be efficiently programmed following this generic model.

The most recent and promising GPGPU framework is offered by the NVIDIA's Compute Unified Device Architecture (CUDA). In CUDA, programs are expressed as kernels. Kernels have a Single Program Multiple Data (SPMD) programming model, which is essentially Single Instruction Multiple Data (SIMD) programming model that allows limited divergence in execution. A part of the application that is executed many times, but independently on different elements of a dataset, can be isolated into a kernel that is executed on the GPU in the form of many different threads. Kernels run on a grid, which is an array of blocks; and each block is an array of threads. Blocks are mapped to multiprocessors within the G80 architecture, and each thread is mapped to a single processor. Threads within a block can share memory on a multiprocessor. But two threads from two different blocks cannot cooperate.

The GPU hardware performs switching of threads on multiprocessors to keep processors busy and hide memory latency. Thus, thousands of threads can be in flight at the same time, and CUDA kernels are executed on all elements of the dataset in parallel. It is also possible to increase the dataset size (e.g. image or video resolution) with no effect on the shared memory usage. This is because, to deal with larger images, we only have to

increase the number of blocks and keep the shared memory allocations in a thread as well as the number of threads in a block the same.

2.2. Multi-core

A good introduction and survey of parallel computing on multi-core platforms can be found in [A06]. In this section we will focus on GPPs but these considerations can be easily extended to other multi-core or many-core hardware. In this context, we can cite as a good example Intel Performance Primitives (IPP). These primitives are functions that are designed to deliver performance by matching the function algorithms to low-level optimizations based on the processor's available features such as Streaming SIMD Extensions (SSE, SSE2, SSE3, SSSE3, and SSE4). These primitives can be used for testing some basic functional blocks (demosaicking, AVC decoder and encoder blocks, color conversion etc) in order to know how well they scale in function of the number of cores when compared to GPGPU implementations.

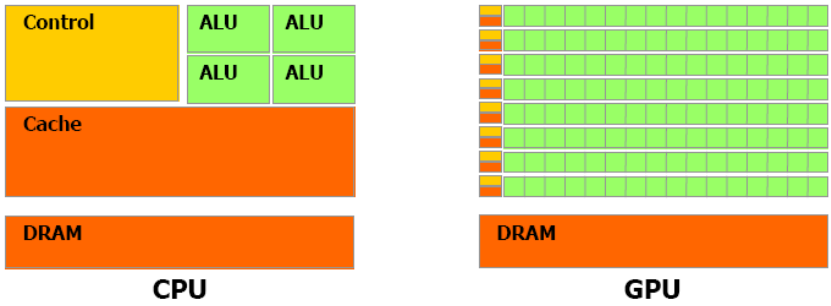

Figure 3 Differences between a CPU and a GPU

Compared to GPUs, multi-core CPUs allow for more general parallel programming models and do not present the GPU's bottleneck in communication between host and device. In a few words, if the algorithm can be programmed in a data level parallelism mode and that the ratio between number of operations and amount of data to transfer is high, then it is more suitable for GPUs; otherwise it would better fit in multi-core or many-core platforms.

3. Processing on the van

This section presents our analysis of the processing on the van.

The pipeline of the van is composed of eight cameras outputting 12 1628x1236 bayered frames per second. These frames have to be de-bayered and compressed in real-time to be stored while recording on the van. Another possibility is to encode the raw bayered images and only perform the demosaicking at the decoder side. This can increase the van recording capacity and ease the encoder task to meet the time requirements. However, this has a certain impact for the processing on the servers because many decoding operations are needed and that they should also be followed by a

demosaicking algorithm in this case. The resulting quality of the decoded images should also be assessed.

In the following subsections, we describe the de-bayering and compression functional blocks of the pipeline on the van. A detailed analysis of compression algorithms, requirements and tests is given in deliverable D1.1.1. We then focus more on possible interactions between compression and demosaicking in this deliverable.

3.1. De-bayering (a.k.a. demosaicking or demosaicing)

Most digital still cameras in consumer electronics have a single image sensor (e.g., CCD or CMOS) to reduce costs and the size of the camera. In the single-chip sensor camera, color images are encoded by a color filter array and a subsequent interpolation process produces full-color images from sparsely sampled images. Figure 4 shows the Bayer color filter array (CFA) pattern [B76], which is widely used in digital still cameras because of its excellent color signal sensitivity and good color restoration.



Figure 4 Four possible bayer types

Early demosaicking methods include bilinear interpolation [APS98], cubic spline interpolation [U99, LCAS04], and neural networks [GSL00]. However, these methods often produce undesirable results such as blurred boundaries. In order to obtain improved performance, demosaicking methods which combine the information from each color plane have been proposed. These methods take advantage of high correlations between the red, green and blue planes in the local regions of natural scene images. For example, the effective color plane interpolation method in [PT03] uses spatial correlations of color difference planes while the smooth hue transition method in [C87] uses spatial correlations of hue planes.

However, they still tend to generate zipper effects along edges since they do not consider object boundaries. To reduce these zipper effects and smearing of sharp edges, an edge-adaptive demosaicking method was proposed by Hamilton et al. [H97]. This method uses edge indicators to find an edge direction and interpolates missing color pixels along the detected edge direction. Various demosaicking methods using gradients [K99], directional information [HP05, MMKMS06] and weighted edge interpolation [MP05, ZW05] also use edge information to reduce artifacts along edges. Finally, Lee et al. [LJL07] propose an edge-adaptive interpolation followed by a post-processing algorithm that performs refining and calibration operations.



Figure 5 Zipper effect. On the left, the zipper effect is generated by demosaicking without edge-adaptive interpolation. On the right, edge-adaptive demosaicking reduces these distortions [LJL07].

In the sequel, we compare two demosaicking algorithms: the interpolation algorithm used by GeoAutomation without any edge-adaptive considerations and an edge-adaptive algorithm which is close to [H97]. One aspect to be studied is that for the processing on the servers, the feature detection block is a corner detector which is very sensitive to zipper effects (see Figure 5). Some investigation is needed in order to determine whether choosing an edge-adaptive demosaicking method will improve or decrease the reliability of the corner detector. On the one hand, the zipper effects could intuitively create too many false positive responses while the edge-adaptive method could increase the false negative probability score by erasing features that should be detected.

3.1.1. Interpolation-based demosaicking without edge detection

Demosaicking without edge detection can be considered when high interpolation precision is needed and if-then-else branches due to edges should be avoided. The filters used to interpolate missing color components are given by Equation 1 and Equation 2 for green-blue lines and red-blue lines, respectively. Both equations are given only for the green position (at index (2, 2)) but they are exactly the same when the pixel processed is blue or red.

$$G = I_{22}$$

$$R = \left\lfloor \frac{5I_{22} + 4(I_{21} + I_{23}) - (I_{20} + I_{11} + I_{31} + I_{33} + I_{24}) + \left\lfloor \frac{I_{02} + I_{42} + 1}{2} \right\rfloor + 4}{8} \right\rfloor$$

$$B = \left\lfloor \frac{5I_{22} + 4(I_{12} + I_{32}) - (I_{20} + I_{11} + I_{31} + I_{33} + I_{24}) + \left\lfloor \frac{I_{20} + I_{24} + 1}{2} \right\rfloor + 4}{8} \right\rfloor$$

Equation 1 interpolation on blue-green lines

$$G = I_{22}$$
$$R = \left\lfloor \frac{6I_{22} + 2(I_{11} + I_{13} + I_{31} + I_{33}) - \left\lfloor \frac{3(I_{20} + I_{02} + I_{42} + I_{24}) + 1}{2} \right\rfloor + 4}{8} \right\rfloor$$
$$B = \left\lfloor \frac{4I_{22} + 2(I_{12} + I_{21} + I_{32} + I_{23}) - (I_{20} + I_{02} + I_{24} + I_{42}) + 4}{8} \right\rfloor$$

Equation 2 interpolation on red-green lines

As it can be seen, the interpolation filters are non-linear and processing four consecutive pixels on a line needs to read 28 neighboring pixels for both equations.

3.1.2. Edge-adaptive demosaicking

In this case, the interpolation filter is much simpler (bilinear) but contains an if-then-else branch. This can limit performances in a parallelization strategy context.
The demosaicking filters are given by equations 3 and 4. In order to compute missing red and blue components, the bilinear interpolation takes the closest available red and blue components. In case the given component is red or blue, four neighbors are needed to compute the missing blue/red component respectively. In case the given component is green, only two neighbors are needed for red and blue component interpolation. The edge-adaptive interpolation only concerns the interpolation of the green component. In this case, one detects the presence of vertical and horizontal edges. The bilinear interpolation then only takes the two pixels along the edge normal. In this case, for processing four consecutive pixels, 21 neighboring pixels are needed. This means that there are less memory reads for this algorithm for an equivalent data re-use. However, there is here a data-dependent decision branch that can hardly be predicted for optimization purposes.

$$G = I_{22}, \quad R = \frac{I_{12} + I_{32}}{2}, \quad B = \frac{I_{21} + I_{23}}{2}$$

Equation 3

$$R = I_{22}, \quad B = \frac{I_{11} + I_{33} + I_{13} + I_{31}}{4}, \quad G = \begin{cases} \frac{I_{12} + I_{32}}{2}, & if \ dv > dh \\ \frac{I_{21} + I_{23}}{2}, & else \end{cases}$$
$$where \quad dv = |I_{02} - I_{42}|, \quad and \quad dh = |I_{20} - I_{24}|.$$

Equation 4

3.1.3. Experiments on CPU and GPU

We programmed both demosaicking algorithms in C. For comparison purposes, we implemented both algorithms in CUDA kernels and we tested the IPP primitives (only available for the edge-adaptive demosaicking): ippiCFAToRGB_8u_C1C3R and ippiCFAToRGB_16u_C1C3R.

The CUDA kernels have been designed base on ConvolutionSeparable and ConvolutionTexture projects of the CUDA SDK. These two projects exploit two different strategies.

ConvolutionTexture exploits the automatic caching system provided by the texture memory. Bayer pixels are also processed as words composed of four consecutive pixels to speed-up the memory transfers between GPU and CPU as well as increasing the amount of data re-use. Typically, each CUDA block processes a set of rows of the input bayer image. This means that iteratively, each thread writes a RGB or YUV triplet at a position (i, j) and then at a position (i+block_size, j), where block_size is the number of threads in the block, until the whole row is processed.

The ConvolutionSeparable implementation exploits the texture caching as well as the available shared memory in blocks. Here also bayer pixels are glued in 32-bits words. The usage of the shared memory is done as follows. The bayer image is decomposed in macroblocks that will be processed by a CUDA block instead of a full row in the case of ConvolutionTexture. In a first step, each thread loads a pixel from the global or texture memory into the shared memory. Then a synchronization message is waited from all threads to ensure that the shared memory loading stage is finished. Finally each thread writes a RGB or YUV word by performing the interpolation on the shared memory array. The sizes of the macroblocks and the number of threads depend on the available shared memory size and are parameters that have to be carefully set in order to ensure a correct and efficient execution. It should also be mentioned that both demosaicking convolutions are not separable.

Since in CUDA, memory reads from the global memory (no caching system) or texture memory (automatic cache) take between 400 and 600 clock cycles on average (significantly less if the data fits in he texture cache) and that multiplications and additions (MADDs) or accesses to the shared memory are performed in 4 clock cycles, data re-use is of crucial importance for increasing the performances. This implies that the strategies used for reading input pixels have to be designed in order to maximize the data re-use, i.e. avoid as much as possible to read several times the same location in the texture or global memory.

Based on these considerations, we can already assume that, neither ConvolutionSeparable nor ConvolutionTexture strategies for memory reads are optimal with respect to the demosaicking algorithms since their convolution masks (the neighboring pixels needed for processing one input pixel) are not nxn squares but more cross-shaped masks which are the union of a small 3x3 square and the four vertical and horizontal pixels at a distance of 2 from the central pixel to be processed. Furthermore, knowing that, for a word of four input pixels, we need 21 and 28 pixels for the two different demosaicking algorithms tested respectively, we can foresee that the edge-adaptive algorithm will show better performances provided the edge dependent branching is efficiently programmed.

Simulation results can be seen on Table 1. The platforms used are for the CPU an Intel Core2 Quad CPU, 2.4 GHz with 2 GB of RAM and for the GPU, Nvidia 8800 GT. It can be observed that IPP and CUDA significantly improve the speed of the CPU implementation. Performances for the edge-adaptive and non-edge adaptive demosaicking are the same. It turns out that the if-then-else branch effect is compensated by the difference of memory complexity. This table also illustrates that the example of the CUDA SDK ConvolutionSeparable is not well adapted to the non-squared convolution

kernels of the demosaicking algorithms. There is still room for optimizations in order to increase the speed performances (e.g. better shared memory filling strategy).

	Edge-adaptive	Non edge-adaptive
CPU	40	35
IPP (8u)	215	N-A
IPP (16u)	84	N-A
CUDA (Texture)	490	483
CUDA (Tex. + Shared memory)	360	351

Table 1 Performances measured in frames per second for CPU, IPP, and CUDA implementation of demosaicking algorithms

If we now consider the communication between GPU and CPU, the bandwidth is limited by the connection through PCI Express. Still considering the GeoAutomation sequence resolution, we measure the communication speed based on the time needed to transfer one bayered frame to the GPU and outputting a YUV frame (1.5 times larger than the bayered image) without processing. Measures of speed taking into account the communication then also add the processing time on GPU.

In this context, the measured speed of the communication without any processing is around 240 fps. Edge-adaptive and non-edge-adaptive demosaicking speeds then drop to 152 and 163 fps respectively. This clearly shows that the bottleneck is related to the communication between CPU and GPU.

However, this measure is not very relevant. Indeed, it would be nonsense to use a GPU for the demosaicking alone. In our context, the following kernel to be executed would be the video encoder. This means the output data would be significantly reduced by the compression algorithm, thus reducing the communication overhead.

If we measure the CUDA performances by only taking into account the input data communication time, the new figures are 711 fps for the communication alone, 285 fps for the non-edge adaptive demosaicking and 288 fps for the edge-adaptive demosaicking.

Finally, it is also possible to hide this communication bottleneck by overlapping streaming and processing on the GPU and CPU with the new CUDA 2.0 beta version. The communication can then be pipelined and scheduled by message passing.

The final observations then are that CUDA enables better performances than IPP optimized implementations. However, profiling simulation tests made on random data show that if multi-threading on 4 CPU is enabled, IPP should reach 400 fps. We could however not observe this on real data.

In conclusion, in the case of the processing on the van, CUDA is very promising. Further research is needed to optimize shared memory exploration as well as dealing with the bandwidth bottleneck with pipeline optimizations.

3.2. Compression

The considerations around compression algorithmic tools are detailed in the URBAN deliverable D1.1.1. As shown in this report, available multi-core implementations are not fast enough to reach the real-time requirements of the processing on the van. A possible outcome would be to apply similar parallelization strategies on the IMEC AVC encoder for Intra Baseline or High profiles. The other possibility is to choose between PGF and AVC for a mapping on GPU. Since wavelet-based encoders such as JPEG 2000 or PGF find their bottleneck in the entropy coding block which has poor parallelization opportunities, they seem less suitable for a mapping on GPU [WLHW07]. Mapping AVC on GPU could on the contrary significantly benefit from the huge potential of this platform regarding task and data parallelism. Some publications pave the way towards real-time encoders for high-definition video sequences such as [PRNW07] (for the decoder functional blocks), [KAWL07] (intra prediction only), [LLWCTC07] (motion estimation).

3.3. Interactions between De-bayering and compression

Some papers have analyzed whether it is more interesting to de-bayer first and encode later or *vice versa*. [MACE05] shows that compressing raw data and demosaicking afterwards at the decoder side can lead to a better performance since this data-flow reduces by a factor of three the amount of data to encode (the RGB bitstream size is exactly three times the size of the raw data bitstream). Fisher et al. [FKK07] also propose the same analysis but show that the raw data should then be separated in three streams according to the available color for each pixel. These gains are obtained at the cost of a higher complexity at the decoding side. Since the processing on the servers requires fast decoding time performances, the best trade-off should be found through experiments.

4. Processing on the servers

This section is now dedicated to the processing on the servers. Here the requirements and available platforms are very different when compared to the processing on the van. Here the tasks to be achieved are related to 3D reconstruction.

4.1. Overview

The overview of the processing on the server is illustrated on Figure 6. The first step consists in performing an automatic and robust feature extraction through corner detection for each image. Then the 3D reconstruction is achieved by matching the features across images. This step yields a large set of 3D points and the camera poses in their own coordinate system. Another step is the measure of reference points in the field and clicked in the images. Finally, from all data (3d points, camera

calibration and referenced points) all cameras are georeferenced in a large optimization process.

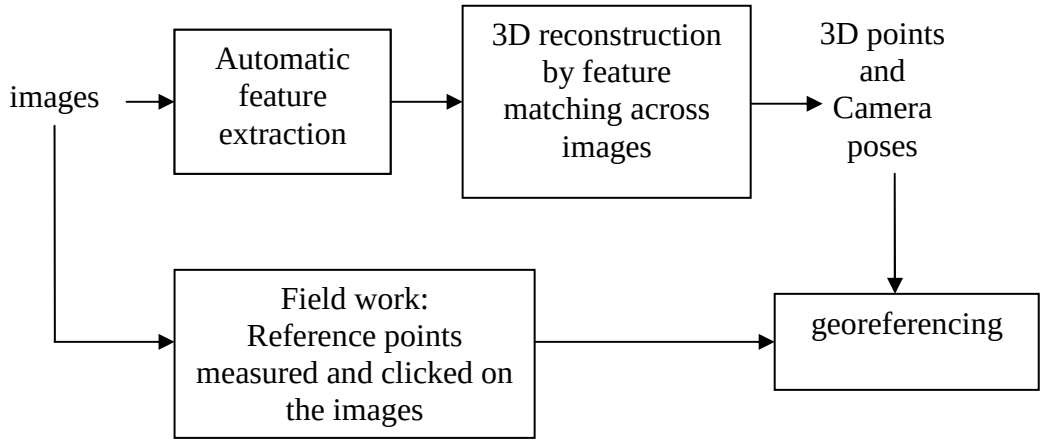

Figure 6 Processing on the servers: 3D reconstruction and georeferencing

4.2. Image decoding

In this framework, the decoding of frames has to be as fast as possible. The algorithm used will of course be based on the choice of the algorithm following the research presented in deliverable D.1.1.1.

If the algorithm used is AVC, then the Intel IPP encoder achieves decoding times between 100 to 300 frames per second. On the other hand, PGF and KAKADU decoders only perform in 10 to 15 fps with parallelization optimizations. CUDA implementations can also be envisaged to further speed-up the image decoding if necessary.

4.3. Feature detection and matching

The feature detection algorithm is based on corner detection [A06b]. As explained before, this detection can be affected by the kind of debayering filter that is chosen. A careful analysis should then also be run in order to quantify this influence in terms of false positive and false negative probabilities.

Correspondences between these image points need to be established through a matching procedure. Matches are determined through normalized cross-correlation (NCC) of the intensity values of the local neighborhood. NCC can be approximated through FFT, since an optimized package of CUDA contains FFT hardware optimizations, GPU could maybe be envisaged.

Next to image matching, many geometrical computations have to be performed in the pipeline. Because we have to deal with possible outliers, robust computation schemes (like RANSAC) are employed. These methods typically

have many if-then-else paths which makes them unsuitable for parallelization.

It would be interesting to investigate how would other feature points such as the ones detected by scale invariant feature transform (SIFT) [L04] compare with the corner detection in terms of speed and reliability.

Another possible research direction relates to the links between the feature point matching and the compression algorithm. If motion estimation is allowed with AVC or MVC, then these motion vectors would help to predict and reduce the size of the search area. Since AVC decoding can be performed in more than hundred frames per second, the end-to-end performance from the processing of the van to the processing on the servers could be increased.

In conclusion, many functional blocks of the processing on the van and on the servers can improve the performance of the end-to-end application. A global performance optimization methodology would certainly lead to an improved system.

4.4. Plane-Sweeping

For surveying purposes, the camera poses are the most important output of the GeoAutomation pipeline. However, the 3D point cloud that is generated during the processing can be interesting as well. The shear amount of reconstructed 3D points makes these points unsuited for direct practical applications. However, many parts of the scenery that is visible in the images can be well approximated by planar surfaces, like the ground plane or building facades.

In order to compute these planes, the points are locally clustered and planes are fitted to them. Then, points in the images must be searched that are located in these images. For this a plane sweep algorithm can be used that maps a point in one image via the plane to another image. If the corresponding pixel matches the color of the original pixel, this is an indication this pixel belongs to the plane.
Fast implementations on the GPU for planesweep algorithms exist [GFMYP07], [ZKMS06]. Research could be done on how to optimize them for GeoAutomation's huge amounts of data.

5. Conclusion

In this deliverable, we have presented our first analysis of the processing on the van and on the servers of GeoAutomation. We have shown that GPU and multi-core platforms can deliver significant speed-ups. However, we also point out that these platforms are better suited for different types of algorithms and parallel programming models. Data transfers can be prohibitive between GPPs and GPUs, and a trade-off has to be found based on a global view of the processing pipeline on the van and on the servers.

6. Bibliography

[A06] 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

[HP07] J. Hennesy and D. Patterson, “Computer Architecture: A Quantitative Approach”, 4th edition, Morgan Kauffman, San Francisco, 2007.

[M04] S. A. McKee, “Reflections on the memory wall”, CF ’04, April 14-16, 2004, Italy

[WLHW07] T.-T. Wong, C.-S. Leung, P.-A. Heng and J. Wang, “Discrete Wavelet Transform on Consumer-Level Graphics Hardware”, Trans. on Multimedia, Vol. 9, No. 3, pp. 668-673, April 2007

[KAWL07] M. C. Kung, O. C. Au, P. Wong, C.-H. Liu: “Intra Frame Encoding Using Programmable Graphics Hardware”. PCM 2007, pp. 609-618, 2007

[PRNW07] B. Pieters, D. Van Rijsselbergen, W. De Neve, and R. Van de Walle, Motion Compensation and Reconstruction of H.264/AVC Video Bitstreams using the GPU,Eight International Workshop on Image Analysis for Multimedia Interactive Services(WIAMIS'07), 2007.

[LLWCTC07] C.-Y. Lee, Y.-C. Lin, C.-L. Wu, C.-H. Chang, Y.-M. Tsao, and S.-Y. Chien Multipass and Frame Parallel Algorithms of Motion Estimation in H.264/AVC for Generic GPU, ICME07, pp. 1603-1606, 2007.

[B76] B. E. Bayer, "Color imaging array," U.S. Patent 3,971,065, Jul. 1976.

[APS98] J. Adams, K. Parulski, and K. Spaulding, "Color processing in digital cameras," IEEE Micro, vol. 18, no. 6, pp.20-30, Nov.-Dec. 1998.

[U99] M. Unser, "Spline: A perfect fit for signal and image processing", IEEE Signal Process. Mag., vol.16, issue.6, pp. 22-38, Nov. 1999.

[LCAS04] C. Lee, S. Cho, W. Ahn, and K. Sohn, "Rapid hybrid interpolation methods," Optical Engineering 43(05), pp. 1183-1194, May 2004.

[GSL00] J. Go, K. Sohn, and C. Lee, "Interpolation using neural networks for digital still cameras," IEEE Trans. Consum. Electron., vol.46, no.3, pp. 610-616. Aug. 2000.

[PT03] S. C. Pei and I. K. Tam, "Effective color interpolation in CCD color filter array using signal correlation," IEEE Trans. Circuits Syst. Video Technol., vol. 13, no. 6, pp. 503-513, Jun. 2003.

[C87] D. R. Cok, "Signal processing method and apparatus for producing interpolated chrominance values in a sampled color image signal," U.S. Patent 4,642,678, Feb. 1987.

[H97] Hamilton, Jr., "Adaptive color plane interpolation in single sensor color electronic camera," U.S. Patent 5,629,734, May 1997.

[K99] R. Kimmel, "Demosaicing: image reconstruction from color CCD samples," IEEE Trans. Image Process., vol. 8, pp. 1221-1228, Sep. 1999.

[HP05] K. Hirakawa, and T.W. Parks, "Adaptive homogeneity-directed demosaicking algorithm," IEEE Trans. Image Process., vol. 14, no. 3, pp. 360-369, Mar. 2005.

[MMKMS06] S. Moriya, J. Makita, T. Kuno, N. Matoba, and H. Sugiura, "Advanced demosaicking methods based on the changes of colors in a local region," IEEE Trans. Consumer Electron., vol. 52, no. 1, pp. 206-214, Feb. 2006.

[MP05] D.D. Muresan, and T.W. Parks, "Demosaicing using optimal recovery," IEEE Trans. Image Process., vol. 14, no. 2, pp. 267-278, Feb. 2005.

[ZW05] L. Zhang, and X. Wu, "Color demosaicking via directional linear minimum mean square-error estimation," IEEE Trans. Image Process., vol. 14, no. 12, pp. 2167-2178, Dec. 2005.

[LJL07] J. Lee, T. Jeong, and C. Lee, "Edge-adaptive Demosaicking for Artifact Suppression Along Line Edges", IEEE Transactions on Consumer Electronics, Vol. 53, No. 3, pp. 1076-1083, 2007.

[N05] S. Naffziger et al, When Processors Hit the Power Wall (or "When the CPU hits the fan", Evening discussion event on ISSCC 2005, summary available at: http://ieeexplore.ieee.org/iel5/9995/32118/01493852.pdf.

[WS07] T. Wiegand and G.Sullivan, The H.264/AVC Video Coding Standard, IEEE Signal Processing Magazine, March 2007

[C08] CUDA, http://developer.nvidia.com/object/cuda.html

[MACE05] D. Menon, S. Andriani, G. Calvagno, T. Erseghe, "On the dependency between compression and demosaicing in digital cinema", Proc. of European Conference on Visual Media Production (CVMP), Nov. 2005.

[FKK07] G. Fischer, K. Köhler, D. Kunz, "A Survey on Lossy Compression of DSC Raw Data", submitted to the Spectrum Conference at Cologne University of Applied Sciences, Nov. 2007

[I08] Intel Integrated Performance Primitives 5.3, http://www.intel.com/cd/software/products/asmo-na/eng/302910.htm

[A06b] A. Akbarzadeh et al. Towards urban 3d reconstruction from video. In Int. Symp. on 3D Data, Processing, Visualization and Transmission, 2006.

[PKVV00] M. Pollefeys, R. Koch, M. Vergauwen, L. Van Gool, "Automated reconstruction of 3D scenes from sequences of images", ISPRS Journal Of Photogrammetry And Remote Sensing (55)4, pp. 251-267, 2000.

[L04] D. Lowe, "Distinctive image features from scale-invariant keypoints," Int. J. of Computer Vision, vol. 60, no. 2, pp. 91–110, 2004.

[GFMYP07] D. Gallup, J.-M Frahm, P. Mordohai, Q. Yang and M. Pollefeys, Real-Time Plane-Sweeping Stereo with Multiple Sweeping Directions, IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2007), pp. 1-8, 2007.

[ZKMS06] X. Zabulis, G. Kordelas, K.Mueller, and A. Smolic. Increasing the accuracy of the space-sweeping approach to stereo reconstruction, using spherical backprojection surfaces. In Proc. Of Int. Conf. on Image Processing, 2006.