Computer Vision on the GPU – Tools, Algorithms and Frameworks

Hannes Fassold
JOANNEUM RESEARCH, DIGITAL – Institute for Information and Communication Technologies
Steyrergasse 17, 8010 Graz, Austria
hannes.fassold@joanneum.at

***Abstract*—In recent years, graphic processing units (GPUs) have emerged as an attractive alternative to CPUs for implementing algorithms in a wide range of applications. The focus of this work is to give an overview about the current state on using GPUs for computer vision. We describe briefly tools like CUDA, OpenCL and OpenACC used for GPU programming and their respective advantages / disadvantages. We give information about the current state of the art for implementing important computer vision algorithms like optical flow, KLT feature point tracking and SIFT descriptor extraction efficiently on the GPU. Finally, we describe open source frameworks which either provide GPU-accelerated computer vision algorithms or which are helpful for porting algorithms to the GPU.**

I. Introduction

Computer vision is a scientific discipline heavily used in a variety of application areas, ranging from established areas like remote sensing, surveillance, medical imaging, machine vision, quality inspection, multimedia & broadcast to emerging application areas like autonomous cars and advanced driver assistance systems. The combination of potentially large datasets of image or video to be processed, real-time requirements in some application area (e.g. for autonomous cars) and high computational complexity of the employed algorithms makes computer vision research challenging.

A major trend in recent years is to employ GPUs (Graphic Processing Units) for all sorts of computer vision algorithms, as they provide significantly more computational capabilities than CPUs. For example, a high end GPU like the NVIDIA Geforce Titan X is specified at 6.6 Teraflops and 336 GB/sec memory bandwidth, whereas a high end CPU like the Intel Xeon E5-2699 v3 achieves roughly 0.9 Teraflops and 80 GB/sec memory bandwidth. Many important computer vision algorithms like feature point detection and tracking [7], optical flow calculation [20] or SIFT descriptor extraction [6] have consequently been ported successfully to the GPU, reporting significant speedup factors typically in the range of five to ten when compared to a multi-threaded CPU implementation. Recognizing the impact GPUs have made for computer vision, in this work we give some information about tools commonly employed for GPU programming, about the state of the art for several important computer vision algorithms and summarize available open source frameworks which are helpful for implementing computer vision algorithms on the GPU.

II. Tools

There is a wealth of tools for programming GPUs in a massively parallel way. In the early days of GPU programming, graphics-related shader language like Cg, GLSL or HLSL have been used, forcing developers to reformulate an algorithm so that it fits into the standard rendering pipeline of OpenGL or DirectX. Shader languages have been quickly deprecated with the advent of general-purpose GPU programming languages like CUDA, OpenCL, C++ AMP, OpenACC, OpenMP 4.0 and DirectCompute. In the following, we focus on the most commonly used tools which are CUDA, OpenCL and OpenACC. A more detailed comparison of high-level parallel programming models can be found in [3].

A. CUDA

CUDA (Compute Unified Device Architecture)[1] is a programming environment developed by NVIDIA, targeting NVIDIA GPUs only. It provides a small set of extensions to C / C++ that allow to develop and execute a function (which is called a *CUDA kernel* in CUDA terminology) on the GPU in a massively parallel way and to access special GPU functionality like atomic operators and the different GPU memory hierarchies like global memory, local memory, shared memory, constant memory and texture memory. Furthermore, an API (*CUDA runtime API*) is provided for doing tasks like allocating / deallocating GPU memory, transferring memory between GPU memory and CPU memory space, launching kernels and managing multiple GPUs. More information regarding the CUDA GPU Architecture and the CUDA Programming model can be found in [7]. It is available for multiple operating systems (Windows, Linux, Mac OS X) and different CPU platforms (Intel, Power, ARM) and supports the programming languages C, C++ and Fortran. In the following, the most important advantages and disadvantages of CUDA with respect to OpenCL and OpenACC are given.

+ Frequent releases of the CUDA toolkit allow to exploit new features of NVIDIA GPUs as soon as they are available.
+ High-quality tools available for comfortable debugging and profiling (NVIDIA NSIGHT[2]).
+ Optimized NVIDIA libraries available providing functionality for image processing, FFT, linear algebra, random numbers etc. (see section IV-A).
- Low-level API – Management and transfer of buffers between CPU / GPU memory space must be done explicitly, all functions which shall be executed on

[1] http://www.nvidia.com/object/cuda_home_new.html
[2] https://developer.nvidia.com/nvidia-nsight-visual-studio-edition

the GPU in a massively parallel way have to be programmed as a CUDA kernel.

- Vendor lock-in, as CUDA code runs only on NVIDIA GPUs.

B. OpenCL

OpenCL[3] is an open standard for GPU programming maintained by the non-profit technology consortium Khronos Group. Its programming model and API shares many similarities with CUDA, like providing a set of extensions (differently named, but conceptually the same as in CUDA) to C which allow to access GPU-specific functionality. Functions which are to be executed on the GPU have to be implemented as a kernel. It enjoys broad support of the industry with around 20 GPU / CPU / FPGA vendors implementing OpenCL support in their products[4]. The major advantages and disadvantages of OpenCL can be summarized as follows:

+ OpenCL code which does not use vendor-specific extensions is highly portable, works for GPUs of different vendors (NVIDIA, AMD, Intel Integrated GPU, Smartphone/Tablet GPUs), accelerators like Intel Xeon Phi and even for FPGAs.
+ OpenCL code can be run also on CPUs, taking automatically advantage of all CPU cores.
- API even more low-level than CUDA runtime API, e.g. for launching a kernel significantly more code has to be written.
- Performance portability between GPUs of different vendors not guaranteed, potentially requiring to write performance-critical kernels for every vendor separately.
- Lacks (in a portable way) support of C++ templates, which are quite helpful to write highly performant kernels.

C. OpenACC

OpenACC[5] is a proramming standard for parallel computing developed by Cray, CAPS, NVIDIA and PGI. In the same manner as in OpenMP, the programmer annotates C, C++ or Fortran source code with simple directives to identify the sections of the code which shall run in a massively parallel way on a GPU / accelerator or on a Multi-Core CPU. The standard is supported by several commercial compilers from Cray, PGI and PathScale targeted for the HPC market as well as by various open-source compilers (GCC in version 6.0, OpenARC, OpenUH and RoseACC).

+ Highly portable, code runs on GPUs of various vendors, accelerators as well as Multi-Core CPUs.
+ High-level abstraction via directives hides the low-level details of GPU programming (like the data movement between CPU / GPU memory and kernel launching) from developers.
- No explicit access to advanced GPU features like shared memory, texture memory and streams, one has to rely on the compiler that well optimized GPU code is generated which exploits these features.
- Performance portability between GPUs (or accelerators) of different vendors not guaranteed.

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

[4]https://www.khronos.org/conformance/adopters/conformant-products#opencl

[5]http://www.openacc.org

III. Algorithms

There is a huge amount of literature for porting computer vision algorithms to GPUs, and various internet resources exist where GPU-related publications and source code are collected [6,7]. In the following, we report the state of the art of GPU implementations for several computer vision algorithms which are important in many application scenarios, namely optical flow, KLT feature point extraction and tracking and SIFT descriptor extraction.

A. Optical Flow

The optical flow problem deals with the calculation of a dense pixel-wise motionfield (see Figure 1 for a visualization) between two images. Optical flow calculation is often used in video processing, e.g. for image registration, video stabilization and restoration. We consider only optical flow methods based on variational calculus or neural networks, as they give high quality results (smooth motionfield with sharp motion boundaries) and are easy to port to the GPU. The GPU implementation in [9] is based on the variational optical flow method from [4], with several adaptions to the solver part. They replace the Gauss-Seidel solver with a damped Jacobi method and employ a multigrid algorithm with V-cycles. The resulting numerical scheme is implemented on the GPU with Shader language, leading to a near-realtime runtime of 60 milliseconds for a 512 x 512 image. In [17] a CUDA implementation of a complex variational optical flow method which works well even for large displacements is described. Alternative solver strategies (red-black relaxation, conjugate gradients) which are better suited for the GPU are investigated, and a runtime of 1.1 seconds for an image with 640 x 480 pixel is reported. In [5] an OpenCL implemention of the TV-L1 optical flow algorithm [22] is presented. They employ a different numerical scheme (FISTA) with better convergence properties and are able to process 8 frames per second on a low-end mobile GPU. The work of [20] is also based on the TV-L1 algorithm, but the isotropic TV regularization is replaced by anisotropic Huber regularization. The GPU implementation is done with CUDA, and a runtime of 1.1 seconds is reported for an image with 640 x 480 pixel. In [2] a comparison of different solver schemes (Fixed-Point, Duality-based, Split-Bregman) for the TV-L1 optical flow algorithm is given, indicating that the duality-based solver has the fastest convergence and also the lowest runtime. In [8] a deep learning based approach for optical flow calculation is presented, using convolutional neural networks (CNNs) with an additional correlation layer. They report competitive performance of the resulting algorithm in terms of quality and runtime when compared to variational methods.

[6]http://hgpu.org

[7]http://gpgpu.org

Fig. 1. Image and visualized optical flow field.

B. KLT feature point selection & tracking

The automatic detection and tracking of corner-like feature points (see Figure 2 for a visualization) throughout an image sequence is a key step for many algorithms in computer vision. It is used for example for pose estimation, camera self-calibration and for person or vehicle tracking. A very popular method for feature point tracking is the KLT algorithm [18] as it is robust, works reasonably fast and operates fully automatic. The KLT algorithm automatically detects a sparse set of feature points which have sufficient texture to track them reliable. Afterwards, detected points are tracked by estimating for each point the translation which minimizes the SSD dissimilarity of patches centered at the points. In [15] OpenGL and the Cg shader language are used for the GPU implementation of the KLT algorithm. A fixed number of iterations is done for each feature point to avoid conditional statements. Detection of new feature points is done only every fifth frame to save computation time. In [21] the KLT tracker is also implemented using Cg. Their KLT tracker compensates for varying camera gain by estimating it as a global multiplicative constant. Another Cg KLT implementation is described in [13]. They propose a modified variant of the feature detection process to circumvent some hard parallelizable parts of it. In [7] a CUDA implementation of the KLT algorithm is given where all steps of the detection (except for the minimum-distance enforcement step) and tracking are implemented on the GPU. The CUDA implementation achieves real-time performance ($>$ 25 frames per second) for video with Full HD resolution (1920 x 1080 pixel) and 10.000 feature points, with a speedup factor of 5 - 10 compared to the multi-threaded CPU implementation. In [11] the implementation described in [7] is optimized for the *Fermi* GPU architecture by employing several modifications, leading to a significant additional speedup of the tracking step by a factor of 2 - 3.

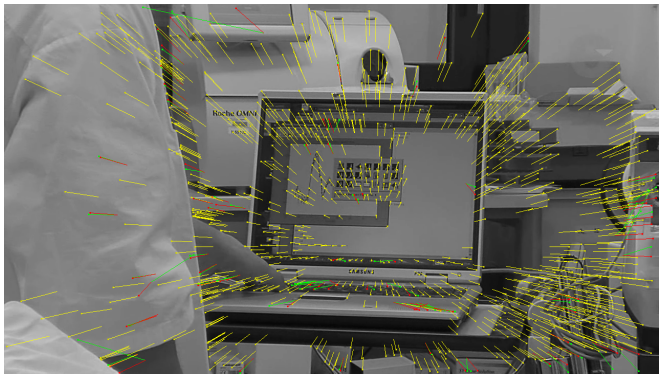

Fig. 2. Visualized result of KLT tracking (image courtesy of [7]).

C. SIFT descriptor extraction

The automatic extraction of a set of features (with a certain degree of invariance to scale, rotation, illumination) from an image is an essential component all sort of video analysis tasks like instance search, duplicate detection and automatic organizing / clustering diverse video content by similarity (e.g., professionally produced content and user-generated content captured with consumer devices[8]). The Scale-Invariant Feature Transform (SIFT) algorithm [12] is one of the most popular methods for feature extraction due to its robustness and good matching performance (see Figure 3 for a visualization). It extracts a set of features $\{f_k\}$ from an image I which serves as a compact high-level representation of the image. For each feature, its (x, y) position, scale s, rotation ϕ and a 128-bin descriptor d (which is sort of a gradient histogram) is calculated. Despite the practical importance of the SIFT algorithm, not many publications are available about a GPU implementation of the algorithm. This may be because the SIFT algorithm is a quite complex algorithm composed of several steps, some of them being not easy to map efficiently onto a GPU. In [16] OpenGL and the Cg shader language are used for the GPU implementation and several of the latter steps of the algorithm are calculated on the CPU. In [10] all steps of the algorithm have been ported to the GPU, also using shader language for the GPU implementation. In [19] an implementation is described where the first step of the SIFT algorithm, the construction of the 3D DoG scale space, is ported onto the GPU using CUDA and all latter steps are kept on the CPU. This is suboptimal because the 3D DoG scale space (an 3D image volume) has to be transferred from GPU memory to CPU memory which is very costly. Furthermore, the latter steps of the SIFT algorithm (like the calculation of the descriptor) take also a significant fraction of the total time and should therefore be also ported to the GPU. A speedup factor of 1.9 is reported in this work. The work [14] reports an implementation where also only the construction of the 3D DoG scale space is done on the GPU and all latter steps on the CPU, which shares the disadvantages already mentioned for [19]. Finally, in [6] all steps of the SIFT algorithm are ported to the GPU, using CUDA. A speedup factor of 4 - 6 (depending on resolution) is reported, which allows real-time processing on the GPU for SD (720 x 576) video.

[8] http://icosole.eu/project

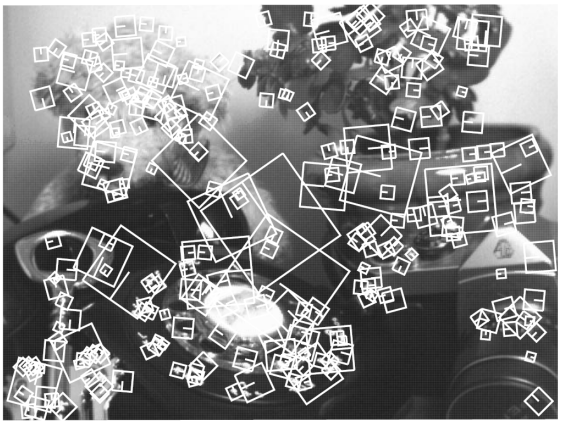

Fig. 3. Visualized result of SIFT feature extraction (image courtesy of [12]).

IV. Frameworks

Within the last few years, several frameworks and libraries have emerged for supporting a developer in the process of implementing an algorithm on the GPU. In the following we describe briefly several frameworks, either open source or freely available from a vendor, which provide full GPU-accelerated computer vision algorithms or are helpful for porting algorithms to the GPU.

A. NVIDIA CUDA Libraries

Several NVIDIA CUDA Libraries (NPP, CUFFT, CURAND, CUBLAS and others) are an integral part of the CUDA Toolkit and therefore immediately available upon installation of the toolkit. The CUDNN and CUSP libraries are freely available, but have to be downloaded separately [9,10].

The NVIDIA Performance Primitives library (NPP) is a collection of GPU-accelerated image and signal processing functions, containing many basic image and signal processing routines. It contains routines for image management, arithmetic and logical routines, convolutional and morphological operators, geometric and statistical routines. Its interface is very similar to the Intel Performance Primitives (IPP) library[11] for CPUs, therefore making the porting of CPU algorithms employing IPP internally very easy.

The CURAND library contains routines for generating pseudo-random sequences with different generators (MRG32k3a, Mersenne-Twister, XORWOW) and quasi-random sequences with the Sobol (unscrambled or scrambled) algorithm. Several probability distribution types (uniform, normal, log-normal, poisson) are supported.

CUFFT provides functionality for the Fast Fourier Transform (FFT) in single precision and double precision. The 1-D, 2-D and 3-D transform (and inverse transform) is supported, either in-place or out-of-place. Batched operations are supported in order to do multiple transforms at once.

CUBLAS is a GPU-accelerated version of the well known BLAS package[12], which provides building blocks for basic vector and matrix operations (vector-vector, matrix-vector and matrix-matrix). All routines are implemented for single, double, complex, and double complex data types and the usage of multiple GPUS is supported.

CUSPARSE and CUSP provide a set of routines for sparse linear algebra. Different datatypes and sparse matrix formats (COO, CSR, ELL, HYB) are supported. CUSP contains various sparse matrix operations, preconditioners (AMG, AINV), eigensolver (Lanczos, LOBPCG) and relaxations methods (Gauss-Seidel, SOR, Jacobi).

The CUDNN library is a library of primitives for deep neural networks. It includes routines for forward and backward convolution and implements common layer types like pooling, ReLU, Simoid, Tanh and softmax. Several popular deep learning frameworks like Caffe[13], Theano[14] and MatConvNet[15] employ CUDNN internally for best performance in neural network training.

B. Parallel programming primitives

In GPU programming, it is important to have high-performance implementations of parallel primitives for array transforms, sorting, reduction (e.g., generate the sum of all array elements) and compaction (filter out items in an array), as these basic steps are often occurring in many algorithms. Several open source libraries are available for both CUDA and OpenCL for these tasks. For CUDA, both CUDPP[16] and Thrust[17] provide GPU implementations of these routines. Furthermore, NCCL[18] contains several primitives which are able to take usage of multiple GPUs. For OpenCL, Bolt[19] and Boost.Compute[20] can be used. Finally, the *Cub* library[21] for CUDA is even more flexible as it provides these primitives not only globally, but also on different levels (device / thread block / warp), furthermore it contains additional functionality for histogram generation and for transferring data from global to shared memory an in an optimal *coalesced* way.

C. OpenCV

OpenCV[22] is a comprehensive open source computer vision package and very popular in the computer vision community. Although originally developed for CPUs, several parts of the package have been ported to the GPU (CUDA / OpenCL). Most basic image processing operations (arithmetic, convolution, morphology, geometric routines etc.) have been ported to the GPU. Furthermore, several high-level algorithms[23] are

[9] https://developer.nvidia.com/cudnn
[10] https://github.com/cusplibrary
[11] https://software.intel.com/en-us/intel-ipp
[12] http://www.netlib.org/blas
[13] http://caffe.berkeleyvision.org
[14] http://deeplearning.net/software/theano
[15] http://www.vlfeat.org/matconvnet
[16] http://cudpp.github.io
[17] http://docs.nvidia.com/cuda/thrust
[18] https://github.com/NVIDIA/nccl
[19] http://developer.amd.com/tools-and-sdks/opencl-zone/bolt-c-template-library
[20] https://github.com/boostorg/compute
[21] https://nvlabs.github.io/cub
[22] http://http://opencv.org/
[23] http://on-demand.gputechconf.com/gtc/2013/webinar/opencv-gtc-express-shalini-gupta.pdf

also available on the GPU, including methods for denoising, segmentation, hough transformation, feature extraction and matching (LBP, HOG, SURF, FAST, ORB), optical flow and stereo matching.

D. ArrayFire

ArrayFire[24] is a highly optimized open source framework for signal processing, image processing and computer vision. It supports 1-dimensional, 2-dimensional (image) and 3-dimensional (volume) arrays and several integer and floating point datatypes. In the same manner as the OpenCV library, ArrayFire contains a set of basic image processing routines (image management, arithmetic, convolution, ...) and several high-level routines for corner detection (FAST, Harris, SUSAN) and descriptor extraction with GLOH, ORB and SIFT. Furthermore, it provides random number generators, parallel programming primitives (sort, reduction, ...) and a few linear algebra routines. It is very portable as it has backends for CUDA, OpenCL and the CPU. Furthermore, it has a special mechanism for just-in-time compilation and automatic kernel fusion which helps to write GPU functions in an easy way without sacrificing performance.

E. MAGMA

MAGMA[25] is an open source library for dense and sparse linear algebra on the GPU with backends for CUDA and OpenCL. It supports multiple datatypes (single, double), out-of-core computation for matrices which do not fit into GPU memory and multiple GPUs. It contains several standard matrix factorization methods (LU, Cholesky, QR) for a matrix or a batch of small matrices, SVD decomposition and solvers for eigenvalues / eigenvectors. For sparse matrices, solvers for symmetric and nonsymmetric matrices are implemented (CG, GMRES, BICGSTAB, QMR, IDR) and preconditioners like ILU and IC. In [1] these sparse solvers are evaluated on a set of standard test matrices, concluding that IDR is a good default choice when no problem-specific information is available.

V. Conclusion

The last years have shown that computer vision algorithms can benefit greatly from porting them to the GPU. So an overview about commonly used tools for GPU programming has been given and the state of the art for porting key computer vision algorithms like optical flow, feature point detection and tracking and SIFT descriptor extraction to the GPU has been described. Furthermore, several frameworks providing support functionality for implementing computer vision algorithms on the GPU have been described briefly.

Acknowledgement

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 610370. ICoSOLE ("Immersive Coverage of Spatially Outspread Live Events", http://www.icosole.eu).

[24] https://github.com/arrayfire/arrayfire

[25] http://icl.cs.utk.edu/magma

References

[1] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Kohler. Efficiency of general Krylov methods on GPUs An experimental study. In *Proc. International Workshop on Accelerators and Hybrid Exascale Systems*, 2016.

[2] L. Bao, H. Jin, B. Kim, and Q. Yang. A Comparison of TV-L1 Optical Flow Solvers on GPU. In *GPU Technology Conference*, March 2014.

[3] E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A Survey of High-Level Parallel Programming Models. Technical report, Department of Computer Science, Heriot-Watt University, 2013.

[4] A. Bruhn and J. Weickert. Towards ultimate motion estimation: combining highest accuracy with real-time performance. In *Proc. Conference on Computer Vision (ICCV)*, volume 1, pages 749–755 Vol. 1, Oct 2005.

[5] E. d'Angelo, J. Paratte, G. Puy, and P. Vandergheynst. Fast TV-L1 optical flow for interactivity. In B. Macq and P. Schelkens, editors, *ICIP*, pages 1885–1888. IEEE, 2011.

[6] H. Fassold and J. Rosner. A real-time GPU implementation of the SIFT algorithm for large-scale video analysis tasks. In *Proc. Real-time Image and Video Processing*, 2015.

[7] H. Fassold, J. Rosner, P. Schallauer, and W. Bailer. Realtime KLT Feature Point Tracking for High Definition Video. In *Proc. GravisMa workshop*, 2009.

[8] P. Fischer, A. Dosovitskiy, E. Ilg, P. Häusser, C. Hazirbas, V. Golkov, P. van der Smagt, D. Cremers, and T. Brox. Flownet: Learning optical flow with convolutional networks. *CoRR*, 2015.

[9] H. Grossauer and P. Thoman. GPU-Based Multigrid: Real-Time Performance in High Resolution Nonlinear Image Processing. In *Computer Vision Systems, 6th International Conference, ICVS 2008, Santorini, Greece, May 12-15, 2008, Proceedings*, pages 141–150, 2008.

[10] S. Heymann, B. Fröhlich, F. Medien, K. Müller, and T. Wiegand. SIFT implementation and optimization for general-purpose GPU. In *WSCG*, 2007.

[11] R. Kaiser, M. Thaler, A. Kriechbaum, H. Fassold, W. Bailer, and J. Rosner. Real-time Person Tracking in High-resolution Panoramic Video for Automated Broadcast Production. In *Visual Media Production (CVMP), 2011 Conference for*, pages 21–29, Nov 2011.

[12] D. G. Lowe. Object recognition from local scale-invariant features. In *Proc. ICCV*, volume 2, pages 1150–1157 vol.2, 1999.

[13] J. F. Ohmer and N. J. Redding. GPU-Accelerated KLT Tracking with Monte-Carlo-Based Feature Reselection. In *Digital Image Computing: Techniques and Applications (DICTA), 2008*, pages 234–241, Dec 2008.

[14] B. Rister, G. Wang, M. Wu, and J. Cavallaro. A fast and efficient SIFT detector using the mobile GPU. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2674–2678, May 2013.

[15] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc. GPU-Based Video Feature Tracking and Matching. Technical report, Department of Computer Science, UNC Chapel Hill, 2008.

[16] S. N. Sinha, J. michael Frahm, M. Pollefeys, and Y. Genc. GPU-based Video Feature Tracking and Matching. Technical report, In Workshop on Edge Computing Using New Commodity Architectures, 2006.

[17] N. Sundaram, T. Brox, and K. Keutzer. Dense point trajectories by GPU-accelerated large displacement optical flow. In *IEEE ECCV*, 2010.

[18] C. Tomasi and T. Kanade. Detection and tracking of point features. Technical report, International Journal of Computer Vision, 1991.

[19] S. Warn, W. Emeneker, J. Cothren, and A. Apon. Accelerating SIFT on parallel architectures. In *Cluster Computing and Workshops, 2009. CLUSTER '09*, pages 1–4, Aug 2009.

[20] M. Werlberger, W. Trobin, T. Pock, A. Wedel, D. Cremers, and H. Bischof. Anisotropic Huber-L1 Optical Flow. In *Proc. of the British Machine Vision Conference (BMVC)*, London, UK, September 2009.

[21] C. Zach, D. Gallup, and J. M. Frahm. Fast gain-adaptive KLT tracking on the GPU. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08*, pages 1–7, June 2008.

[22] C. Zach, T. Pock, and H. Bischof. A Duality Based Approach for Realtime TV-L1 Optical Flow. In *Proceedings of the 29th DAGM Conference on Pattern Recognition*, pages 214–223, Berlin, Heidelberg, 2007. Springer-Verlag.