A Real-time GPU Implementation of the SIFT Algorithm for Large-Scale Video Analysis Tasks

Hannes Fassold\textsuperscript{a} and Jakub Rosner\textsuperscript{b}

\textsuperscript{a}JOANNEUM RESEARCH, DIGITAL – Institute for Information and Communication Technologies, Steyrergasse 17, 8010 Graz, Austria
\textsuperscript{b}Silesian University of Technology, PhD faculty of Data Mining, Ulica Academicka 16, 44-100 Gliwice, Poland

ABSTRACT
The SIFT algorithm is one of the most popular feature extraction methods and therefore widely used in all sort of video analysis tasks like instance search and duplicate/near-duplicate detection. We present an efficient GPU implementation of the SIFT descriptor extraction algorithm using CUDA. The major steps of the algorithm are presented and for each step we describe how to efficiently parallelize it massively, how to take advantage of the unique capabilities of the GPU like shared memory/texture memory and how to avoid or minimize common GPU performance pitfalls. We compare the GPU implementation with the reference CPU implementation in terms of runtime and quality and achieve a speedup factor of approximately 3 - 5 for SD and 5 - 6 for Full HD video with respect to a multi-threaded CPU implementation, allowing us to run the SIFT descriptor extraction algorithm in real-time on SD video. Furthermore, quality tests show that the GPU implementation gives the same quality as the reference CPU implementation from the HessSIFT library. We further describe the benefits of GPU-accelerated SIFT descriptor calculation for video analysis applications such as near-duplicate video detection.

Keywords: SIFT descriptor, real-time processing, GPU, CUDA

1. INTRODUCTION
The automatic extraction of a set of features from an image is an essential component of many computer vision tasks such as image registration, 3D reconstruction, object recognition and all sort of video analysis tasks like instance search and duplicate detection. A variety of approaches have been proposed for this task, like SIFT\textsuperscript{1}, SURF\textsuperscript{2}, ORB\textsuperscript{3} or BRISK\textsuperscript{4}. All of these approaches generate features which are invariant to scale, rotation and illumination changes. The Scale-Invariant Feature Transform (SIFT)\textsuperscript{1} algorithm is one of the most popular methods due to its robustness and good matching performance. It extracts a set of features \{f_k\} from an image I which serves a compact high-level representation of the image. For each feature, its \((x, y)\) position, scale \(s\), rotation \(\phi\) and a 128-bin descriptor \(d\) (which is sort of a gradient histogram) is calculated.

For video analysis tasks, it is desired that the analysis is at least real-time capable or faster in order to process large amounts of content (e.g. a broadcaster archive with hundreds of thousands hours of video) in a reasonable amount of time. The same applies for the processing of live events, where it is obvious that the analysis has to be done in real-time. Even with a highly optimized and multi-threaded CPU implementation, this is not possible to achieve due to the high computational complexity of the SIFT algorithm. In the last years, a major trend was to employ GPUs (Graphic Processing Units) for all sort of computer vision algorithms. Many important algorithms like feature point detection and tracking,\textsuperscript{5} optical flow calculation\textsuperscript{6} or object detection\textsuperscript{7} have already been ported successfully on the GPU. They report impressive speedup factors typically in the range of 5 to 10 compared to a multi-threaded CPU implementation. In this work, we describe a highly optimized GPU implementation of the SIFT algorithm which is able to process video in Standard Definition resolution (720 x 576) in real-time and video in Full HD resolution (1920 x 1080) in nearly real-time. We use CUDA (Compute Unified Device Architecture)* by NVIDIA because it is currently the most stable programming environment for GPU programming and provides useful tools like a debugger and profiler.

*https://developer.nvidia.com/about-cuda
The rest of the paper is organized as follows: In section 2 we mention some related work. In Section sec:algorithm an overview of the SIFT algorithm by Lowe et al.\textsuperscript{1} is given. In section 4 the CUDA implementation of the SIFT algorithm is described in detail and in section 5 we give a comparison regarding. In section 6 an application example is given where the GPU-accelerated SIFT algorithm is used and section 7 concludes the paper.

2. RELATED WORK

Despite the practical importance of the SIFT algorithm, not many publications are available regarding 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. Furthermore, in most publications certain steps which are difficult to port to the GPU are either skipped or kept on the CPU. In the work of Sinha et al.\textsuperscript{8} 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. Note that the usage of 3D graphic-related programming languages like OpenGL and Cg is considered as an outdated practice as nowadays general-purpose language for GPU programming like CUDA and OpenCL are available. In the work of Heymann et al.\textsuperscript{9} all steps of the algorithm have been ported to the GPU, also using shader language for the GPU implementation. Warn et al.\textsuperscript{10} describe an implementation 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 of Rister et al.\textsuperscript{11} 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 the approach of Warn et al.\textsuperscript{10}

3. SIFT ALGORITHM

The SIFT algorithm\textsuperscript{1} can be roughly divided into three major steps. In the first step (3D DoG Scale-space extrema detection, see subsection 3.1), a 3D DoG scale-space space is constructed and local extrema in this space are detected yield a first set of keypoint candidates. In the second step (keypoint candidate refinement and filtering, see subsection 3.2), the position of these keypoint candidates are refined and unstable candidates are filtered out. In the final step (keypoint descriptor calculation, see subsection 3.2), for all remaining keypoint candidates their SIFT descriptor is calculated. In the following, each step is described more in detail.
3.1 3D DoG scale-space extrema detection

Multiple octaves are build in a recursive way, where the input for the first octave is the input image itself, whereas the input of the octave with index \( n \) is the last (gaussian-blurred) image in the octave with index \( n - 1 \). Each octave constitutes a 3D volume of DoG (difference-of-gaussian) images which is calculated in the following way: Firstly, a set a gaussian-blurred images \( L_k \) is build where each image \( L_k \) is the result of blurring the input image (for the current octave) with a gaussian blur kernel with progressively increasing sigma \( \sigma_k \). From these, a set of DoG images \( D_k \) which build up the octave is calculated as the difference of two consecutive gaussian-blurred images via \( D_k = L_{k+1} - L_k \). Within a \((x, y, s)\) octave, where \( s \) denotes scale and \((x, y)\) the spatial position, local extrema are detected by comparing a voxel value with it 26 neighbor voxels in \((x, y, s)\) space. These are added as initial keypoint candidates to a list. See figure 1 for an illustration of the process.

3.2 Keypoint candidate refinement and filtering

In the first substep, the initial \((x_0, y_0, s_0)\) positions of the initial keypoint candidates are refined to sub-voxel precision \((x', y', s')\). As described in Lowe et al., this improves the performance of SIFT features for matching significantly and improves their robustness. For a keypoint candidate, a quadratic function \( D(x, y, s) \) is constructed which interpolates the nearby data points in \((x, y, s)\) space. Then, the maximum of the quadratic function \( D \) is determined by taking by derivative of this function and setting it to zero. This yields a \(3 \times 3\) linear equation system which can be solved e.g. with singular value decomposition. If the precise location \((x', y', s')\) is lying nearer to a integer position different from the initial \((x_0, y_0, s_0)\) position, then the refinement process is repeated from this position.

In the second substep, a filtering is done where unstable keypoint candidates are discarded. Unstable keypoint candidates are ones which either have low contrast are which are located on image edges. Low-contrast candidates are identified by having a absolute function value \(|D(x', y', s')|\) less than a certain threshold. Keypoint candidates located on image edges are detected in the following way. First, an approximation \( H \) to the \(2 \times 2\) Hessian matrix of the function \( D(x, y, s) \) is calculated at the refined position. The matrix has size \(2 \times 2\) because the (approximation of the) second derivatives are taken only with respect to \(x\) and \(y\), not for \(s\). From the matrix \(H\), a cornerness measure is calculated in a computationally efficient way and keypoint candidates having a too low cornerness are discarded.

In the third substep, for all remaining keypoint candidates their orientation \(\phi\) is determined as follows. For a certain keypoint \((x', y', s')\), first the gaussian-blurred image \(L\) which is nearest in scale is determined. From the image \(L\), the gradient is calculated and the gradient magnitude and orientation is calculated pixel-wise. A weighted orientation histogram with 36 bins is constructed from the gradient orientations of sample points within a region around the keypoint, where the weight is determined from the gradient magnitude. The keypoint orientation \(\phi\) is now determined from the peak in the orientation histogram, and the orientation value is refined by parabolic interpolation. If there are other significant peaks in the histogram (within 80% of the highest peaks), these are also taken to create a keypoint with that specific orientation. So it may happen that from one keypoint candidate multiple keypoints are generated with the same spatial position and scale \((x, y, s)\), but different orientation \(\phi\).

3.3 Keypoint descriptor calculation

For each remaining keypoint, the task is now to calculate its descriptor. First the image gradient magnitudes and orientation are sampled in a \(16 \times 16\) region around the refined keypoint location. In order to achieve orientation invariance, the coordinates of the descriptor and the gradient orientations are rotated relative to the keypoint orientation \(\phi\). The sampling process is illustrated at the left side of Figure 2. After applying a gaussian weighting function, the keypoint descriptor is formed by creating weighted orientation histograms over \(4 \times 4\) sub-regions. For robustness, within construction of the orientation histogram trilinear interpolation is employed to distribute a gradient value into adjacent histogram bins. At the right side of Figure 2, four of these orientation histograms are shown where each orientation histogram has eight bins. The 128-bin \((16 \times 8\) bins) descriptor is now aggregated from the 8-bin orientation histograms for all 16 sub-regions. Finally, the descriptor is normalized to unit length with makes it more robust with respect to illumination changes.
4. GPU IMPLEMENTATION

This section describes the optimized implementation of the SIFT algorithm for execution on many-core parallel processors. While the implementation presented in this paper is designed for GPUs using CUDA environment it should be noted that most, if not all, of the principles presented here can be successfully used for other processor architectures and in different environments. In many cases porting as many parts of a given algorithm to GPU as possible is desirable however one should remember that there are exceptions where leaving parts of the code to be executed on CPU side (heterogeneous programming) can greatly increase performance. Given that the source image has 3 channels and initially resides in CPU memory a typical operation that should be left on the CPU is the conversion to a one channel gray image. This is due to the fact that the operation is very simple so performing it on the GPU would bring only minor speedup and it leaves only one (instead of three) channel to be transferred from CPU to GPU memory since such transfer relatively very slow in comparison to conversion.

4.1 3D DoG scale-space extrema detection

4.1.1 Calculation of Gaussian and DoG octaves

This task, while being fairly simple, is however challenging to implement efficiently on the GPU due to its low arithmetic intensity combined with an extensive amount of data to process. For this reason our aim is to minimize the number of off-chip global memory accesses as much as possible, optimize those that are left by taking advantage of GPU’s global memory coalescing and exploiting fast, on-chip shared memory. A separable convolution algorithm (vertical then horizontal) is employed that minimizes both memory and arithmetic operations and makes them increase linearly with the radius of convolution kernel, as opposed to quadratic for nonseparable two-dimensional convolution. Portions of data, required by many threads of a thread block, are being loaded to shared memory once and then used for fast execution by those threads. For that we implemented a macro (templatized by the block size) for convenient and optimal gathering of spatially close data from global memory to shared memory. This macro is also used by many other GPU kernels as well.

The calculation of different scales in any given octave in a DoG pyramid can be done simultaneously which in some cases might increase performance, however it is much more important to minimize the number of global memory accesses. This can be achieved by progressively computing scales of any DoG octave with a single GPU kernel and re-using already to shared memory loaded portions of every scale in the Gaussian pyramid (except the first and the last scale of each octave) to obtain respective portions of two scales of DoG pyramid. For example to compute the first scale of a DoG octave, the first and second gaussian scales have to be loaded to shared memory, however to compute the second scale only the third gaussian scale has to be loaded as the second one is already there. This approach significantly reduces the number of global memory accesses required to obtain the DoG pyramid. In the example from figure 1a, with four DoG scales per octave, it reduces this number by 37.5%.
4.1.2 3D scale-space extrema detection

In order to detect the local extrema in a 3D DoG scale-space one GPU kernel iterates through all the scales of a single octave and generates a keypoint candidate list. Following the previously introduced concept each kernel first loads necessary data from the first three DoG scales to shared memory, finds local extrema, then moves one scale up by loading data from the fourth scale, once again searches for extrema and repeats in that fashion until the last scale of the given DoG octave. This approach however requires a lot of shared memory as in the final stage data from all the scales has to fit in it. A solution to that problem is each time replacing the data from the oldest scale which won’t be required anymore with the new scale data. This way the amount of shared memory required is reduced to the data from only three scales which enables more thread blocks to be computed simultaneously on a single multiprocessor for better memory latency hiding. The generation of keypoint candidate list is done in three consecutive steps to minimize the number of simultaneous atomic operations on the same data. When a local extrema is found, its position is saved in shared memory and the number of extrema found by this block is incremented atomically. After the searching is complete a single thread in each thread block atomically saves the current offset for the global candidate list as its own (the offset for the very first block is zero) and increases this offset with the number of extrema found by this block for the next blocks to use. Finally each thread block writes all its extrema from shared memory to global candidate list, starting at the offset position saved in the previous step.

4.2 Keypoint candidate refinement and filtering

While the singular value decomposition is a very complex operation, it is computed only for the previously created local extrema list and can be very efficiently implemented on GPUs using ideas presented by Adams et al. This technical report strives to minimize branches (which can significantly reduce the performance of GPU kernels) and uses only elementary floating point operations. The ideas were designed for CPUs with SSE and/or AVX instructions but most of them turned out to be also efficient on GPUs. The efficient calculation of the orientation histogram is not easy to implement efficiently on the GPU. The keypoint candidates are scattered throughout \((x, y, s)\) space so the standard approach to compute one keypoint per thread would lead to an extremely inefficient global memory access pattern, where each thread would aim at a different global memory cache line which in turn would cause the reads to be completely serialized. Another problem is that different threads, depending on their \(s\) position, have different neighborhood sizes, from which the orientation histograms are supposed to be obtained. This in turn leads to a very unbalanced workload between threads. A solution to both problems is to use a whole thread block to compute an orientation histogram for a single keypoint candidate. Each thread computes a magnitude and an orientation of a single pixel in the given neighborhood around current keypoint. This also evens out the work-load between threads belonging to the same thread block. Threads belonging to different blocks and thus different warps do not need to have balanced workload as they are executed independently. The orientation histogram is kept entirely in shared memory due to its relatively small size which speeds-up the computations.

4.3 Keypoint descriptor calculation

The computation of the histogram required for descriptor calculation faces similar issues with regard to global memory accesses as in case of orientation histogram. There are however many more arithmetic operations mainly due to the trilinear interpolation computed for each pixel of the keypoint neighborhood. Additionally the creation of histogram requires many more memory accesses to the histogram, as for a single pixel multiple bins need to be modified. The same strategies as mentioned in section 4.2 are used to overcome these issues. Additionally, due to extensive register usage of the original kernel, an effort has been made to minimize the number of variables and reuse them in multiple places in code to preserve GPU registers. Also a whole thread block normalizes a single histogram at the end however this block actually consists of a single warp (32 threads). To obtain the maximum value from all histogram bins an intra-warp reduction is applied to parallelize the operation as much as possible.
5. EXPERIMENTS AND RESULTS

This section describes the results from comparing the GPU implementation against an optimized CPU reference from HessSIFT library. The CPU implementation uses openCV library internally which provide heavily optimized image processing routines but only a single processor core is used for computations therefore the runtime has been divided by four in order to simulate results from multicore implementation on quad-core CPU. It should be noted however that due to many data dependencies in the algorithm the actual speedup obtained from multicore implementation will be much lower. The experiments were done on a 3.0 GHz Intel Xeon Quad-Core machine with 8 GB RAM, equipped with a NVIDIA GeForce GTX 480 GPU. It should be noted that the initial image resampling is enabled for both CPU and GPU thus doubling its size.

![Figure 3: Runtime for an image with 2500 keypoints, for different resolutions.](image)

Two sets of runtime tests have been performed. The first one for different video resolutions ranging from Standard Definition (SD) through HD (HD720p) to Full HD (HD1080p) and with a constant number of 2,500 keypoints. Note there is no parameter in the SIFT algorithm that explicitly limits the number of final keypoints, however with a careful modification of the minimum quality parameter this can be achieved. The number of 2,500 keypoints has been chosen as it is roughly the maximum number that can be obtained from a SD image. The second test has been performed for different numbers of keypoints (1,000 - 10,000) and constant Full HD

![Figure 4: Runtime for a Full HD image, for different numbers of keypoints.](image)
Figure 5: Example of takes of the same scene: The upper row shows keyframes of the first take, the lower of the second take of one scene. Images taken from BBC 2007 rushes video data set.

(HD1080p) resolution. Results of those tests are shown in Figures 3 and 4 respectively. A significant speedup factor ranging from 4.3 (SD with 2,500 keypoints) to 6 (Full HD with 10,000 keypoints) can be achieved for all examined image resolutions and keypoint numbers. All GPU timings include the preprocessing stage and all the required transfers between CPU and GPU memory. It is important to notice in the above figures that the speedup achieved by GPU implementation increases with both resolution and number of keypoints. For SD resolution and 1,000 keypoints the GPU implementation needs approximately 40 milliseconds to complete and therefore meets real-time performance requirements. Nearly identical results are obtained from both implementation (CPU/GPU) in terms of quality. In fact the only differences are caused by the initial bicubic resampling which is done slightly differently in the CPU implementation, but they occur for less than 0.01% of the SIFT keypoints.

6. APPLICATIONS

As already mentioned, the SIFT algorithm has found many applications in tasks such as image registration, 3D reconstruction, object recognition or video analysis. We will focus here on large-scale clustering and duplicate/near-duplicate detection of video content. This is an important but computationally expensive task which has many potential applications. E.g. for broadcast productions of live events like festivals, it can be used to automatically organize (e.g., cluster by shooting location) and integrated professionally generated content from broadcasters and user-generated content captured with consumer device like smartphones. Another important application scenario is post-production, where after shooting a large amount of raw material (rushes) must be organized in a meaningful way by clustering multiple takes of a scene automatically. This is illustrated in Figure 5 where two slightly different takes of a scene are shown. A typical algorithmic workflow for these tasks is described very coarsely. Firstly, SIFT features are calculated for keyframes (e.g., every 5th frame) of all videos. Then a (typically huge) affinity matrix is constructed where each entry $(i,j)$ is the similarity score between the SIFT feature sets (or between compact descriptors derived from the SIFT feature set like VLAD$^{14}$) of two individual keyframes $i$ and $j$. Finally, a clustering within the affinity matrix is carried out by searching for diagonally or block-wise oriented regions with high similarity values as these regions indicate two very similar video sections. Typically, the runtime of the whole process is dominated by the first step where the SIFT features are extracted from the keyframes. Therefore, the significant speedup we gain in this step (for standard definition content real-time processing is achieved on the GPU) by employing the GPU-accelerated SIFT algorithm speeds up also the whole process considerably.

7. CONCLUSION

An efficient GPU implementation of the SIFT descriptor extraction algorithm for automatic feature extraction was presented. For each step of the algorithm, we describe how to efficiently parallelize it massively, how to take
advantage of the unique capabilities of the GPU like shared memory / texture memory and how to avoid or minimize common GPU performance pitfalls. A comparison of the GPU implementation with the reference CPU implementation in terms of runtime and quality shows that we achieve a speedup factor of approximately 4 - 5 for SD and 5 - 6 for Full HD video. This allows us to run the SIFT descriptor extraction algorithm in real-time on SD video. The benefits of GPU-accelerated SIFT descriptor calculation for video analysis applications such as near-duplicate video detection are presented.

ACKNOWLEDGMENTS

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”. Furthermore, the work of Jakub Rosner was partially supported by the the European Social Fund within project UDA POKL-04.-01.01-00-106/09. BBC 2007 Rushes video is copyrighted. The BBC 2007 Rushes video used in this work is provided for research purposes by the BBC through the TREC Information Retrieval Research Collection.

REFERENCES