Scalable SIFT with Scala on NUMA

Frank Feinbube, Lena Herscheid, Christoph Neijenhuis, and Peter Tröger

Hasso Plattner Institute
University of Potsdam, Germany
{frank.feinbube,lena.herscheid,peter.troeger}@hpi.uni-potsdam.de,
christoph.neijenhuis@appetico.com

Abstract. *Scale-invariant feature transform* (SIFT) is an algorithm to identify and track objects in a series of digital images. The algorithm can handle objects that change their location, scale, rotation or illumination in subsequent images. This makes SIFT an ideal candidate for *object tracking* – typically denoted as *feature detection* – problems in computer imaging applications.

The complexity of the SIFT approach often forces developers and system architects to rely on less efficient heuristic approaches for object detection when streaming video data. This makes the algorithm a promising candidate for new parallelization strategies in heterogeneous parallel environments.

With this article, we describe our thorough performance analysis of various SIFT implementation strategies in the Scala programming language. Scala supports the development of mixed-paradigm parallel code that targets shared memory systems as well as distributed environments. Our proposed SIFT implementation strategy takes both caching and *non-uniform memory architecture* (NUMA) into account, and therefore achieves a higher speedup factor than existing work. We also discuss how scalability for larger video workloads can be achieved by leveraging the actor programming model as part of a distributed SIFT implementation in Scala.

1 Introduction

The research field of *computer vision* focusses on the automated extraction of numerical or semantic information from real world image data. It steadily gains importance due to the rising amount of image and video data in daily processing tasks. Application domains include robotics, medical systems, autonomous vehicles, satellite picture processing and semi-automated manufacturing.

The problem domain of visual data processing can be separated into the batch processing of large very images, and the online processing of a steady stream of small images. Most of this data is not recorded in a controlled studio environment, but comes from an *uncontrolled environment*. The camera, or target objects in the video data, may constantly change their position and lightning conditions.

One of the most challenging tasks in image data processing is the identification and tracking of interesting objects, typically denoted as *feature detection*.

Application examples are robots or vehicles automatically following given paths or objects, the continous tracking of players on a soccer field, the alignment of machine activities to human hands in medical environments, the detection of logos in TV streams, or any kind of face recognition approach. The algorithm known to be most accurate in this field is SIFT, originally proposed by Lowe [6]. The multi-stage algorithm is known to be computationally intense, which makes it hardly applicable in soft-realtime applications such as online video analysis.

With the given hard computational problem, the question arises if and how modern parallel execution environments can be leveraged efficiently for a SIFT implementation. Modern many-core systems not only provide parallel processing capabilities, but also implement a *non-uniform memory architecture* (NUMA), were data locality becomes a crucial aspect of performance optimization. When the amount of data to be processed becomes to large for a single system, an implementation must also be able to scale out the computation to multiple machines.

Discussions of an efficient SIFT implementation mostly either solely focus on single systems, single GPU accelerators, or distributed systems only. For this reason, we discuss how SIFT can be realized in heterogeneous parallel environments with NUMA characteristics. We rely on the *Scala programming language* in our approach, which has proven to be suitable for large-scale data processing tasks in both shared memory and shared-nothing environments. Productivity-focused programming languages have proven suitable even for some areas of high performance computing [2]. Scala is designed to allow for a high degree of parallelization and scalability.

Our article contributes a thorough performance analysis of various strategies for the Scala programming language. Our approach is cache- and NUMA-aware and therefore achieves a higher speedup factor than existing work. It allows supports the scale-out of the processing to multiple compute nodes, which is to the best of our knowledge not considered for SIFT so far.

2 Basics and Related Work

SIFT was first invented and described by Lowe in 1999 [6]. He refined and patented the algorithm in 2004 [7]. Since then, it has been widely used for automatic features detection in images.

SIFT determines a set of characteristic *features* or *keypoints* for an input image and represents it as a set of descriptors. An outline of the SIFT algorithm is depicted in Figure 1. The algorithm works on monochrome images, extracted from the video stream, in order to keep the impact of lighting changes low. Different *Gaussian blur filters* are applied to the monochrome image at different scales, in order to blur away all but the most characteristic structures. Subsequently, extrema of the so-called *difference of gaussians* (DoG) are detected as features. The DoG is the difference between two versions of the image which have been blurred with different Gaussian filters.

Figure 2 illustrates how the DoG maintains the most characteristic structures in the image, while abstracting away from small, potentially instable, details.

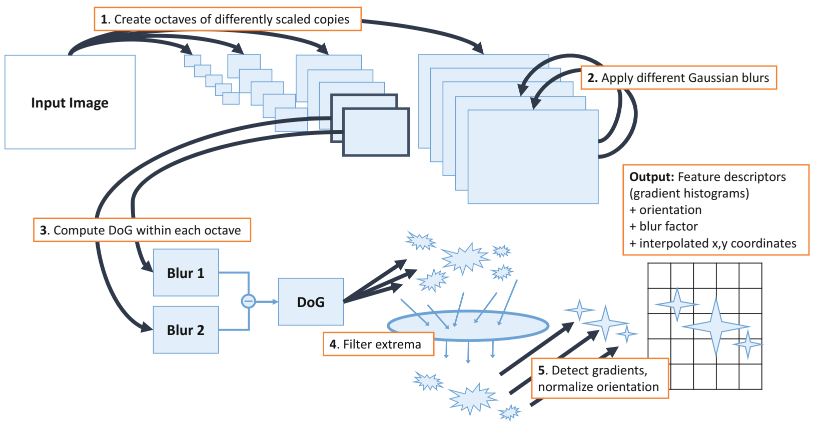
1. Create octaves of differently scaled copies
Input Image
2. Apply different Gaussian blurs
Output: Feature descriptors (gradient histograms)
+ orientation
+ blur factor
+ interpolated x,y coordinates
3. Compute DoG within each octave
Blur 1
Blur 2
DoG
4. Filter extrema
5. Detect gradients, normalize orientation

Fig. 1. Illustration of the SIFT algorithm: **1.** A *scale space*, consisting of differently scaled images is constructed. For each scale, a group of identical images is created: an *octave*. In our example, there are four scales of octaves with a size of five. **2.** For each octave, a differently configured Gaussian blur filter is applied to each of the images. **3.** The DoG of all two subsequent images within an octave is computed. This yields extreme values in areas with characteristic image features. **4.** Only extrema that lie above certain thresholds and exist across different scales are filtered out. **5.** Finally, the resulting *feature descriptors* contain information about the brightness gradient, the orientation and position of features. These are the outputs of the SIFT algorithm.

Formally, a *feature* describes the relative change of brightness in a certain area. Figure 3 depicts a feature descriptor. In the final phase of SIFT, the dominant *orientation* found in these histograms is saved as a descriptor property.

Features are invariant to scaling and rotation. Due to their flexible nature, SIFT behaves robustly against noise, changes in the lighting conditions and affine distortions. This stability makes it an attractive algorithm in all sorts of uncontrolled computer vision scenarios.

Table 1 summarizes related work on parallelizations of the SIFT algorithms for CPUs. Since our focus lies on NUMA scalability rather than single image performance, we describe the related work with a *speedup factor*, in order to express how efficiently additional compute resources are used. Because the speedup factor takes the number of used physical cores into consideration, it allows us to compare different implementations running on different test systems from related work.

Feng et al. [3] studied a number of performance optimizations and their impact on parallel SIFT on a 4-socket quadcore HP ProLiant DL580 G5 server. They observe that the use of SIMD optimizations can halve the runtime. In their experiments, a combination of thread affinity, false sharing removal and synchronization reduction yields a 25% performance improvement. The OpenMP-based implementation achieves speedup factors of 9.7 for large pictures and 11 for small

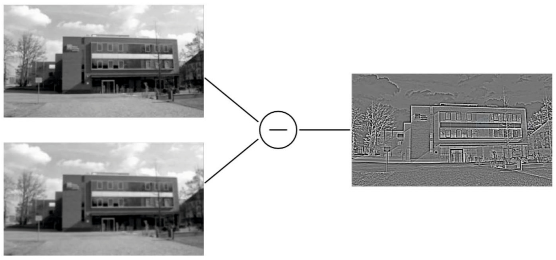
Fig. 2. Computing the DoG: Two different gaussian blur filters are applied to the same image. The difference between the two resulting images highlights the main image characteristics.

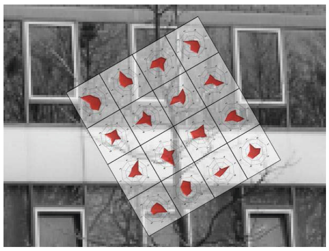
Fig. 3. Visualization of a descriptor: A descriptor comprises 4x4 gradient histograms describing the relative change of brightness in the area of a feature.

pictures. Further, Feng et al. investigate the scalability of their implementation in a CMP simulator with 64 cores (each equipped with its own L1 cache, but sharing the L2 cache). In this environment, a speedup of 52 for large pictures and 39 for small pictures is achieved.

Zhang et al. [13] presented another OpenMP-based SIFT implementation. On an 2-socket quadcore HP ProLiant DL380 G5 server test system, they achieve a speedup of 5.9-6.7 depending on the feature density of the test images. For 640x480 images, their speedup factor is slightly higher than that of Feng et al.'s implementation.

Warn et al. [11] proposed a straight-forward parallelization approach: The three most expensive loops of the serial SIFT++ implementation [10] were parallelized with OpenMP. This implementation works best with large satellite pictures and achieves a speedup of factor 2 on the eight core test system.

Several related SIFT implementations improve the processing performance by utilizing GPU acceleration hardware [5, 9, 11, 12]. Since the analyzed data first must be moved to accelerator card memory, the above named approaches suffer from significant data copying overhead. Warn et al. [11] show that it is feasible to execute only a part of SIFT, namely the Gaussian blurring algorithm, on the GPU. In this case, even with a copying overhead that eats up 90% of the execution time, the GPU version is 13 times faster than the CPU version.

Since absolute performance over various hardware architectures is not well comparable, we benchmarked our implementation against the state-of-the-art OpenCV implementation [4] on our test system. We executed the Scala code in a serial version, in order to be comparable to the serial OpenCV implementation. Our implementation is 1.29 times faster for a resolution of 1980x1080 and 1.18 times faster for 800x600.

Table 1. Overview of related work on SIFT parallelization. Speedups are expressed as factors relative to the execution on one core. The speedup factor denotes the ratio *speedup/physical cores*. Our implementation achieves superlinear speedup, since it is optimized for cache and NUMA locality on our target server system.

	Image Size	Features	Processors	Cores / Processor	Cores	Speedup	Speedup Factor
Feng et al. [3]	HDTV	1038	4	4	16	9.7	0.61
	720x576	700	4	4	16	11	0.69
	HDTV	1038	1	64	64	52	0.81
	720x576	700	1	64	64	39	0.61
Zhang et al. [13]	640x480	200-1000	2	4	8	5.9-6.7	0.73-0.84
Warn et al. [11]	4136x1424	40000	2	4	8	2-3	0.25-0.38
Our Approach (150 video frames)	854x480	1400-2700	1	6	6	7.15	**1.19**
	854x480	1400-2700	2	6	12	13.21	**1.10**
	854x480	1400-2700	4	6	24	24.23	**1.01**

3 Approach

Our example application of SIFT is the detection of company logos in high definition video streams, mainly for the purpose of advertising management and copyright control. In such a scenario, the feature detection needs to provide high accuracy even if the tracked image part is partially occluded or distorted. Since the processing works on large data amounts, it must be also fast and scalable at the same time. We focus on single-image performance and on overall scalability by combining single-node optimization (see Section 4) with multi-node optimizations (see Section 5).

To achieve good single-node performance on modern architectures, the implementation strategy must be NUMA-aware. This includes the consideration of memory hierarchies and the parallel utilization of multiple cores / processors in the system. Scalability is realized by supporting a distributed execution style, where parallel parts of the program run on different cluster / cloud machines. The requirement of scalability implies that the inter-node communication in the SIFT implementation must be limited to a neccessary minimum.

Our choice for Scala allows us to rely on the *actor programming paradigm* for the concurrent operation, as implemented in the *Akka* [1] framework. Each concurrent actor is parallelized in itself, in order to achieve the best possible utilization of node-local parallelism. We follow the common *farmer-worker* paradigm, were a central “master” actor is responsible for distributing image workload to other actors. Each of these "worker" actors receives an image, performs SIFT on it and sends the resulting descriptors and features back to the master. The functional programming model supported in Scala is well suited for parallelization, since it outlines algorithmic dependencies and allows the runtime to manage the actual execution. Scala includes both imperative and functional features.

Since the computation of the Gaussian pyramid, descriptor orientation, and filtering steps in SIFT can overlap, we defined additionally dedicated actors for these stages. They may be distributed across different cores or processors, accessing the image (or intermediate) data on this node. Due to the transparency of distribution in the Scala actor model, these computational units can be placed on resources in a flexible manner. This allows to switch between local and distributed execution without additional coding effort. The programming paradigm for all steps and sub-steps in the SIFT algorithm remains the same. Any mentioning of the term *node* in the following explanations therefore relates both to NUMA nodes in a single system (meaning processor sockets), or machine nodes in a parallel distributed system.

3.1 Test System and SIFT Configuration

All measurements mentioned in the subsequent sections were carried out on a Fujitsu RX600 S5 server. It contains four Intel Xeon E7530 CPUs with 1.86 Ghz and 6 physical cores and runs a Debian operating system (Version 6.0). Our *Java virtual machine*s (JVMs) has the version 1.6.0_24. We used the Scala programming language (version 2.9.1) and the Akka framework (version 2.0).

SIFT was configured as described in [7]. The initial image size was doubled. We computed five octaves of six scaled images each. During filtering, a threshold of 5% of the maximum contrast was used.

We used two test datasets. The first is a single HDTV image, which contains 9697 features (an average number for photographs). The image contains regions with both high feature densities (e.g., trees) and low feature densities (e.g., sky).

The second test dataset is a movie with a resolution of 854x480, showing a bicycling robot[1]. Images in the test film contain an average of 1400 to 2700 features.

Since *just-in-time compilation* (JIT) optimizations occur adaptively during runtime, all of our measurements are preceded by a warm-up phase of several seconds.

4 Single-Node Performance Optimizations

Before we discuss distribution techniques for SIFT stages across NUMA nodes, we show how to maximize the performance on single processors with multiple cores.

[1] `https://www.youtube.com/watch?v=wH8KzseCW58`, November 25, 2014.

4.1 Two-dimensional Data Structure for Images

The pixels of a two-dimensional image can either be saved as a two-dimensional array (represented as an array of arrays), or as a one-dimensional array. The two-dimensional array uses slightly more memory to store the references from the outer array to the inner arrays.

Allocation has a large influence on the overall runtime of the SIFT algorithm, since it creates many intermediate results. The HotSpot JVM, being the execution environment for Scala code, is optimized for the allocation of small objects; they can be allocated out of a buffer within ten native CPU cycles [8]. However, for large image sizes (e.g. HDTV), the resulting one-dimensional array is bigger than the buffer allows. We therefore used a two-dimensional array, where each inner array is created by the thread that is going to fill it with values. This leads to a more fine-grained allocation scheme that fits to the execution environment optimizations. The HotSpot JVM now creates one allocation buffer per thread, and each of them can use it without further synchronization. The two-dimensional array with thread-local allocation of the inner arrays is faster than the one-dimensional array when both are parallelized. Without thread-local allocation or parallelization, it is slower. For HDTV images, the JVM allocates the one-dimensional array slowly because of it size. The two-dimensional array uses up to 60% less time than the one-dimensional array when parallelized. With smaller image sizes, the parallelization overhead grows: For image sizes of $240x135$ pixels, parallel and serial execution take roughly the same amount time.

We conclude that using a nested two-dimensional array for image representation increases performance, because it leads to more JVM-friendly allocation patterns.

4.2 Optimization of the Gaussian blur

For a two-dimensional image, the Gaussian blur filter is applied in two phases: First, the image is blurred in the horizontal dimension. This intermediate result is then blurred in the vertical dimension. This is visualized in the left column in Figure 4.

A widely used optimization technique is to save the intermediate result with flipped dimensions. This approach is visualized in the second column. For the final result, we flip the dimensions again, bringing it back to the original layout.

Flipping the dimensions ensures that the pixels are read along the cache line. It is worthwhile to optimize for reading, because more pixels are read than written[2]. However, writing is not cache line aware. When used with the proposed two-dimensional data structure, the inner array cannot be tied to a single thread - multiple threads need to write into it. It is therefore not possible to use the thread local allocation buffer. Due to false sharing, additional synchronization overhead may occur.

[2] Depending on the scale, the blurring needs to read between 11 and 27 pixels for one written pixel.

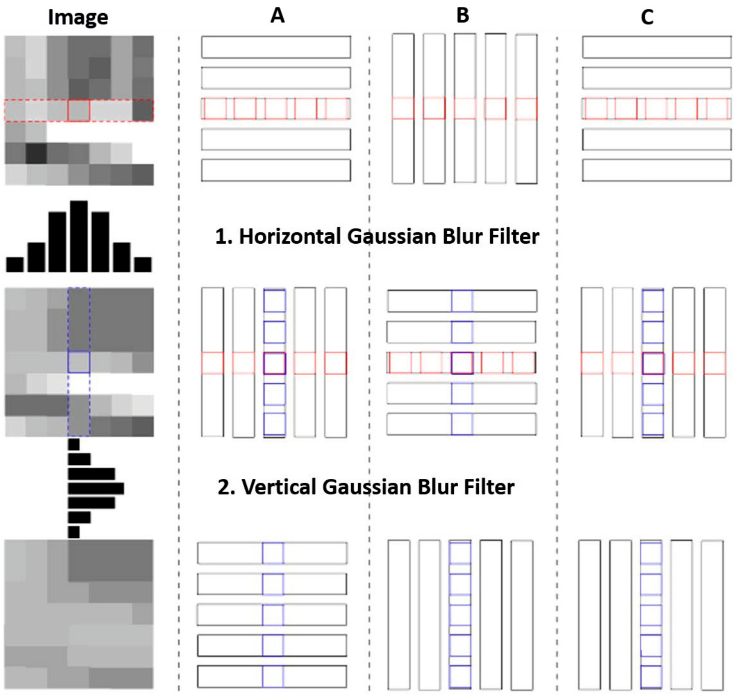
Image
A
B
C
1. Horizontal Gaussian Blur Filter
2. Vertical Gaussian Blur Filter

Fig. 4. Illustration of the two phases of the Gaussian blur: the first phase (red) blurs horizontally, the second phase (blue) vertically. *A (read-optimized)*: Pixels are read along the cache line. *B (write-optimized)*: Pixels are written into one inner array, but read from multiple arrays. *C (combined approach)*: The second phase does not flip the image, allowing both read- and write-optimization. The resulting image is mirrored.

Neither cache misses during writing, nor false sharing impose a problem when we optimize for writes. Threads read pixels from multiple arrays, and not along cache lines, but write into a single one. This write-optimized approach is visualized in the third column. It yields better scalability, but worse serial performance. For multi-core processors, it performs worse than the read-optimization.

We applied an optimization that exploits the fact that the blurred images are further processed in subsequent SIFT stages. The intermediate result of the blurring remains flipped. Thus, during the second phase, the algorithm can read and write into inner arrays simultaneously, accessing the cache lines when reading and the thread local allocation buffers when writing. This approach is visualized in the fourth column. Subsequent stages of SIFT have to be aware of the flipped images. Features, coordinates and orientation have to be adjusted before exporting the result image. The options for Gaussian blur are read-optimized for using cache lines (option 1), write-optimized for using thread-local allocation (option 2), or option 1 with a second phase that does not flip the image and is both optimized for using cache lines and thread-local allocation (option 3).

Our experiment show that option 1 and 3 perform similarly when executed serially, but option 2 is 67% slower. With six threads, option 2 is still 35% slower than option 1, while option 3 is 16% faster and turns out to be the best approach.

4.3 Optimization of the Order of SIFT Stages

Each octave of the Gaussian pyramid is created by blurring five images and subtracting them from their predecessor. The blurring creates one intermediate image. For the original image to still be inside the last-level cache after blurring, the cache needs to hold at least three images. The subtraction reads two images and writes a new one. For the next subtraction, one of the two images is read again, which is still in the cache if it holds more than three images.

The subtraction can be done directly after one image has been blurred, which will be present in the last-level cache because it has just been written. The other image will only be present if the cache holds more than six images. In this case, the image used for the next blurring is also still in the cache. Another possibility is to first blur all five images, then subtract them. If the subtraction is done backwards, for the first subtraction the last-level cache will hold the image that has just been blurred, and, if the cache holds more than four images, the other image as well. If the cache can hold at least 16 images the order does not matter, since all final and intermediate images fit into it. Similarly, if the cache can hold less than three images the order does not matter, because images have to be constantly loaded from main memory.

In our evaluated configuration with an L3 cache, it can hold about four images in the largest octave, and more than 16 images in the next octave (because the image size is quartered). We therefore first blur all five images of one octave and then subtract them, starting in reverse order with the image last blurred.

After the Gaussian pyramid has been created, extrema need to be detected and interpolated before the orientations and descriptors can be computed. Lowe [7] originally proposed extrema detection and interpolation as two separate steps. However, the interpolation stage accesses the same data as the extrema detection. To exploit the L2 cache more efficiently, we propose to interpolate immediately after each extremum has been detected.

The computation of the orientation and the descriptor works on the same image area and the steps share some intermediary data. We propose to execute both steps immediately after each other to optimize the use of the L2 cache.

The number of features per image is not predictable, but generally decreases with increasing blur. Images may even contain fewer features than cores available. If feature computation is done per image, significant parallelization overhead may arise. We propose to collect extremas from all octaves first and compute the features for all octaves together.

In our experimental evaluation of the described approach, we first create the Gaussian pyramid for all octaves, detect extrema, and then compute the features for all octaves together in a second step, which includes orientation and descriptor computation. As described in [3], the creation of features scales better. As Table 2 shows, the runtime of Gaussian pyramid creation rises from 43% of the combined runtime to 48% with 6 threads and to 50% with activated hyperthreading. For both stages, activated hyperthreading, and using 12 threads instead of 6, increases the speedup significantly. During feature computation, hyperthreading raises the speedup from 4.86 to 7.23. Because the pyramid creation constantly loads image data from the main memory into the cache, the main memory connection becomes a bottleneck sooner than during feature creation, which works on the same image area for a longer time.

Table 2. Scalability of the main SIFT stages

	1 Core		6 Cores (6 Threads)			6 Cores with HT (12 Threads)		
	Runtime	%	Runtime	%	Speedup	Runtime	%	Speedup
Pyramid	10258 ms ± 126	43%	2635 ms ± 96	48%	3.89	1890 ms ± 36	50%	5.43
Features	13814 ms ± 193	57%	840 ms ± 399	52%	4.86	1909 ms ± 159	50%	7.23

5 Scaling with Multiple Nodes

Our Scala implementation achieves scalability by distributing the workload across different nodes following the actor paradigm. As explained before, these nodes may either be part of a single machine (NUMA processor nodes) or a distributed environment.

While modern JVMs are NUMA-aware in themselves, the official Java API does not allow NUMA-aware application programming and provides no interface to control the distribution of threads on processors. If a single JVM is used in such a NUMA environment, uncontrollable latency can occur when accessing memory allocated by a remote thread. This problem even arises when the threads work on seperate data items, since runtime and object management is still centralized in one JVM instance. To avoid such problems, we apply a typical approach were each NUMA node in the system runs a dedicated JVM instance. We used the `numactl` program to bind the JVM instances to specific processors, and launch a single actor per NUMA node. The actors themselves are parallelized and cache aware (see Section 4).

As discussed in Section 3, our architecture comprises a master who decodes the video stream and distributes frames to worker actors. The workers perform stages of SIFT and reply with messages containing the features they found.

Varying numbers of features in different video frames can lead to an unbalanced distribution of workload. Therefore, we initially fill the messaging mailbox of each SIFT-actor with two images. When the actor finishes the computation on one image, the mailbox is filled up again by the master. This simple approach leaves enough work left per actor to hide potential memory latency effects.

The performance penalty for using one JVM for the whole system instead of using a seperate JVM per NUMA node is severe. In our test system, using two

JVMs on two NUMA nodes instead of one leads to a performance improvement of 54%. With four JVMs on four NUMA nodes the improvement is 79% as compared to using a single JVM.

For simple codecs, the image decoding speed on the master node is dominated by the speed of reading from disk. Thus, the maximum achievable throughput in batch processing scenarios is determined by disk access speed. With faster disk access, it is possible to scale out to more worker actors.

5.1 Execution on Multiple Machines

An expected, but still somehow impressive, outcome of the experimental evaluation is the fact that no additional modifications were needed to get a fully distributed SIFT implementation. The actor-based programming model makes the transition from a single NUMA system to a distributed system painless and straightforward – it basically boils down to the coordinated startup of multiple JVM's on different machines, running the same code as described above.

The additional amount of machines induces the typical increase in communication and synchronization overhead, but adds scalability to the given solution. Distributing SIFT over multiple machines increases the image throughput, which makes the overall analysis truly scalable. Widely known I/O optimizations for cluster computing, such as fast interconnects or parallel file systems, can help to decrease the latency impact from the networking part of distribution. From our point of view, the main advantage is that the proposed actor-based implementation can remain unchanged. Performance optimization now becomes an operational aspect, and no longer is a dedicated task for the developer. This allows for effortless re-use of such an implementation in different execution environments. When distributed across 5 machines, we achieved a speedup 3.74.

6 Conclusion

In this article, we presented several insights into how a scalable implementation of SIFT can be realized using the actor paradigm. As shown in Table 1, our implementation achieves superlinear speedup due to its cache-aware and NUMA-aware optimizations. We optimize for single node performance by fine-tuning the different SIFT stages for optimal exploitation of the L2 and L3 caches. As discussed in Section 4, this includes flipping the image twice during Gaussian blurring, saving it in a allocator-friendly two-dimensional data structure, and re-ordering the extrema detection and interpolation for better L2 cache efficiency.

Our Scala code is furthermore optimized for scalability in NUMA environments. Each NUMA node runs a dedicated actor, ensuring data locality and allowing distributed processing.

Since Gaussian blurring accounts for two thirds of the runtime, future work should invest additional effort into parallelizing this part of the algorithm. GPUs, due to their massively data parallel processing powers, are well suited for image filtering tasks, when the data transfer overhead problem is finally solved.

To allow fast and scalable support for live video recognition, the current video frame needs to be subdivided into multiple image parts and processed in parallel. Due to the nature of the SIFT algorithm, this leads to significant overhead, since the additional image margins – needed by both the Gaussian blurring and the extrema detection – have to be copied. This overhead grows with the number of actors and hinders seamless scalability. We are aware of this problem already and plan to further investigate optimizations for the live processing scenario.

References

1. Akka, `http://akka.io/` (Online; accessed May 28 2014)
2. Amedro, B., Bodnartchouk, V., Caromel, D., Delbe, C., Huet, F., Guillermo, L.T.: Current State of Java for HPC. Technical Report RT-0353, INRIA (2008)
3. Feng, H., Li, E., Chen, Y., Zhang, Y.: Parallelization and characterization of SIFT on multi-core systems. In: IEEE International Symposium on Workload Characterization, IISWC 2008, pp. 14–23. IEEE (2008)
4. Hess, R.: An open-source siftlibrary. In: Proceedings of the international conference on Multimedia, pp. 1493–1496. ACM (2010)
5. Heymann, S., Muller, K., Smolic, A., Frohlich, B., Wiegand, T.: SIFT implementation and optimization for general-purpose GPU. In: Proceedings of the international conference in Central Europe on computer graphics, visualization and computer vision, vol. 144 (2007)
6. Lowe, D.G.: Object recognition from local scale-invariant features. In: The Proceedings of the Seventh IEEE International Conference on Computer Vision, vol. 2, pp. 1150–1157. IEEE (1999)
7. Lowe, D.G.: Distinctive image features from scale-invariant keypoints. International Journal of Computer Vision 60(2), 91–110 (2004)
8. Sun Microsystems. Memory management in the Java HotSpot virtual machine (2006)
9. Sinha, S.N., Frahm, J.-M., Pollefeys, M., Genc, Y.: GPU-based video feature tracking and matching. In: EDGE, Workshop on Edge Computing Using New Commodity Architectures, vol. 278, p. 4321 (2006)
10. Andrea Vedaldi Sift++, `http://www.robots.ox.ac.uk/~vedaldi/code/siftpp.html` (Online; accessed May 28, 2014)
11. Warn, S., Emeneker, W., Cothren, J., Apon, A.W.: Accelerating SIFT on parallel architectures. In: CLUSTER, pp. 1–4 (2009)
12. Wu, C.: SiftGPU, `http://cs.unc.edu/~ccwu/siftgpu/` (Online; accessed May 28, 2014)
13. Zhang, Q., Chen, Y., Zhang, Y., Xu, Y.: SIFT implementation and optimization for multi-core systems. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2008, pp. 1–8. IEEE (2008)