Accelerating Local Feature Extraction using OpenCL on Heterogeneous Platforms

Konrad Moren, Thomas Perschke
Fraunhofer IOSB, Ettlingen, Germany
konrad.moren,thomas.perschke@iosb.fraunhofer.de

Diana Göhringer
Ruhr-University Bochum, Germany
diana.goehringer@rub.de

***Abstract*—Local feature extraction is one of the most important steps in image processing applications such as image matching and object recognition. The Scale Invariant Feature Transformation (SIFT) algorithm is one of the most robust as well as one of the most computation intensive algorithms to extract local features. Recent implementations of the algorithm focus on homogeneous processors like multi-core CPUs or many-core GPUs. In this paper, we introduce an OpenCL-based implementation, which can be used in homogeneous and heterogeneous CPU/GPU environments. We analyze possible coarse-grained and fine-grained parallelization solutions of the SIFT algorithm. Using a set of optimizations we implement a high-performance SIFT implementations for very different CPU/GPU architectures. The scalable implementation allows for a fast processing, more than 40 FPS for Full-HD images.**

***Keywords*—OpenCL, SIFT, Many-core GPU, Multi-core CPU, Heterogeneous computing, Platform specific optimizations**

I. Introduction

Recent developments in the area of programming models have made it possible to easily access the computational power of computing devices such as Graphics Processing Units(GPUs). With programming frameworks like OpenCL (Open Computing Language), developers are able to easily use different computational resources available on a platform. The use of heterogeneous devices provides an opportunity to significantly increase the overall run-time performance of an application and to achieve higher energy efficiency. However, it is unlikely to achieve the best performance without architecture specific optimizations and a proper distribution of the application across heterogeneous devices. There is a lot of ongoing research in the field of improving the run-time of algorithms for local feature detection in images on multi-core CPUs [1], [2] and many-core GPUs [3]–[5]. But many publications are focused on a single class of devices and the associated optimization techniques. We would like to present an alternative implementation, which is able to work in a heterogeneous environment and utilize all supported and available computing resources in parallel. Additionally, we present architecture specific optimization steps and describe the main differences between CPU and GPU devices in the context of the OpenCL programming model.

The paper is organized as follows. Section II introduces the SIFT algorithm. In Section III, we briefly introduce OpenCL and describe the mapping of the OpenCL programming model to generic CPU and GPU devices. Section IV describes the parallelization process and various optimization steps for specific hardware platforms. Section V introduces a multi-device implementation that utilizes many devices concurrently. Section VI describes the conducted experiments and discusses the results. Section VII concludes the paper and presents the future work.

II. Algorithm description

The SIFT algorithm is used to extract distinctive rotation- and scale-invariant features in images [6]. In this section, we briefly describe the algorithm and the inherent parallelization opportunities.

A. Gaussian scale space construction

The Gaussian scale space of an 2D-image $I(x, y)$ is divided into octaves. Each octave consists of scale images $L(x, y, \sigma)$

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{1}$$

defined as a convolution of the image with a normalized Gaussian function

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \mathrm{e}^{-\frac{(x^2+y^2)}{2\sigma^2}} \tag{2}$$

Adjacent scale images are subtracted to produce the Difference-of-Gaussian (DoG) images $D(x, y, \sigma)$

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{3}$$

with some multiplication factor k.

B. Feature detection and localization

In the detection phase, the DoG images are searched for local extrema and unstable features are filtered out. The single steps are

- Local extremum detection
  To detect a local extremum, each scale space point with position $\boldsymbol{x} = (x, y, \sigma)^T$ is compared to its 8 pixel neighbors at the image $D(x, y, \sigma)$ and to the 9 pixel neighbors at the adjacent DoG images. If the intensity is lower or higher than the intensities of all neighbors, the point is declared keypoint.
- Localization
  To localize a keypoint with sub-pixel precision, $D(\boldsymbol{x})$ is expanded into a Taylor series to second order and the offset $\hat{\boldsymbol{x}}$ is calculated by setting the derivative of Equation (4) to zero.

$$D(\boldsymbol{x}) = D + (\frac{\partial D}{\partial x})^T \boldsymbol{x} + \frac{1}{2} \boldsymbol{x}^T \frac{\partial^2 D}{\partial x^2} \boldsymbol{x} \tag{4}$$

$$\hat{\boldsymbol{x}} = -(\frac{\partial^2 D}{\partial x^2})^{-1} \frac{\partial D}{\partial x} \tag{5}$$

The offset is used to get an interpolated keypoint position.

- Low contrast rejection
  If the contrast value $D(\hat{\boldsymbol{x}})$ is lower than some threshold value Th, the keypoint is rejected.

- Edge response rejection
  In the last step, all keypoints along edges are rejected by calculating the eigenvalues of the Hessian matrix

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix} \tag{6}$$

C. Orientation-assignment

For every keypoint $\eta(x, y, \sigma)$ found, the scale image with the closest σ value is used to assign an orientation to the keypoint by calculating the gradient magnitude $m(x, y)$ and the orientation $\theta(x, y)$, as defined by equations (9) and (10), in a local region around η. An orientation histogram is formed from the gradient orientations of the sample points. Peaks in the orientation histogram correspond to dominant directions of the local gradients. The highest peak in the histogram defines the orientation of the keypoint.

$$\nabla_x = L(x+1, y) - L(x-1, y) \tag{7}$$

$$\nabla_y = L(x, y+1) - L(x, y-1) \tag{8}$$

$$m(x, y) = \sqrt{\nabla_x^2 + \nabla_y^2} \tag{9}$$

$$\theta(x, y) = \arctan \frac{\nabla_y}{\nabla_x} \tag{10}$$

D. Feature-descriptor

For the feature descriptor, gradient and magnitude of a local 2D area of L around the keypoint position are calculated. The area is divided into $M \times N$ cells. For every cell, the orientations weighted with the magnitudes are used to fill a N_{Hist} bin histogram. To suppress the influence of far pixels, the entries are additionally weighted with a Gaussian function. The feature descriptor is defined as the combination of all histograms. For the typical parameters $M = N = 4$ and $N_{Hist} = 8$ this gives a 128 element feature vector. The feature vector undergoes a normalization procedure to reduce the effects of illumination changes. The vector is normalized, large values are clipped to some threshold value Th_{norm} and the clipped feature vector is normalized again.

III. OpenCL Architecture

OpenCL is a framework for writing programs for different hardware such as CPUs, GPUs, Field-Programmable Gate Arrays (FPGAs) and other supported processors [8], [9]. OpenCL provides a common programming language based on $C99$ and a programming interface (API). The OpenCL hardware abstractions enable developers to accelerate applications with task-parallel or data-parallel computations in a heterogeneous computing environment. In this section, we briefly introduce the OpenCL platform model and its interpretation for CPU and GPU devices.

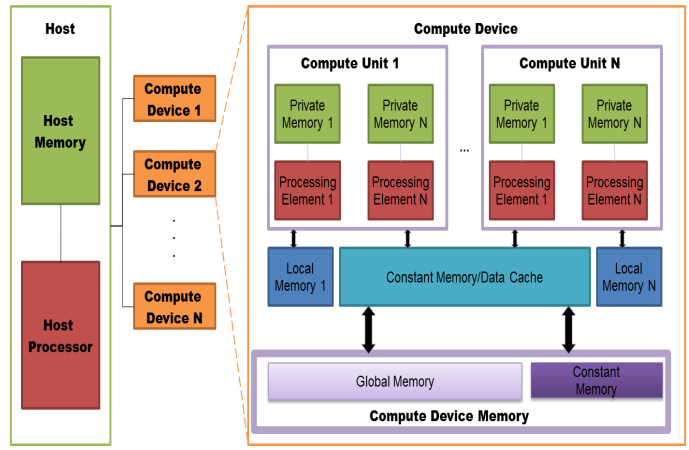


Fig. 1. OpenCL Platform Model

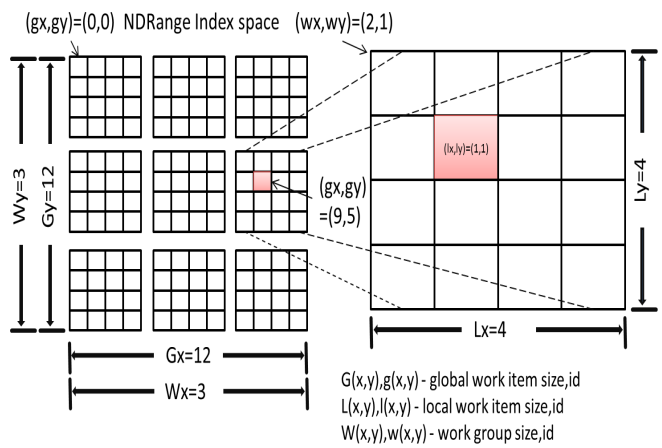


Fig. 2. OpenCL Execution Model - Index space [7]

TABLE I. Mapping of OpenCL platform model to the multi-core CPU processors and many-core GPUs

OpenCL model	multi-core CPU	many-core GPU
Host processor	CPU log. core	CPU log. core
Host memory	CPU main mem.	CPU main mem.
Compute device (CD)	CPU log. core	GPU streaming cores
Compute Unit (CU)	CPU log. core	Streaming core
Processing element (PE)	log. core resources	Streaming processor
Global memory	CPU main mem.	GPU global mem.
Constant memory	CPU main mem.	GPU constant mem.
Local memory	CPU main mem.	GPU shared mem.
Private memory	CPU main mem.	GPU registers

A. OpenCL platform model

Figure 1 shows the generic OpenCL platform model that consists of a host connected to one or more compute devices. Each compute device is divided into one or more compute units (CU). A CU is further divided into one or more processing elements (PE).

An OpenCL program has two parts: one or more kernels written in the OpenCL programming language, implementing an application, and a host program that manages the execution of the kernels. When a kernel is submitted for execution by the host, it works on an index space as defined in Figure 2. A kernel instance is called a work-item (WI). Work-items are organized into work-groups (WG). The work-items in a given

TABLE II. COMMON DESCRIPTION OF AMD/NVIDIA GPU HARDWARE ARCHITECTURES

AMD GPUs	Common description	NVIDIA GPUs
SIMD engine	Streaming core	SM multiprocessor
Thread processor	Streaming processor	Scalar-CUDA core
VRAM-global memory	GPU global memory	Global memory
Constant memory	GPU constant memory	Constant memory
LDS-local data share	GPU shared memory	Shared memory
Registers memory	GPU registers	Registers memory

work-group execute concurrently on the processing elements of a single compute unit.
OpenCL defines four distinct memory regions on a compute device: global, constant, local, and private memory. As shown in Figure 1, the compute device can have global and constant memory regions. Global or constant memory may be cached in global/constant memory data caches. These regions are shared by all CUs in a compute device (i.e., all work-items in all work-groups). The local memory is shared by all PEs in a CU (i.e., all work-items in a single work-group), and the private memory is assigned to and only accessible by the individual PEs (i.e., private to each work-item).

B. OpenCL mapping to multi-core CPU devices

Table I shows the mapping between the OpenCL platform model and multi-core CPUs. In this mapping, a set of logical cores is a compute device and a logical core is the host processor. The host processor core may be shared with the compute device. Each logical core in the compute device is a CU. The host memory and the device memory are allocated in the main memory of the multi-core system.

C. OpenCL mapping to many-core GPU devices

Table I shows the mapping between the OpenCL platform model and a typical GPU system that consists of a CPU and a GPU. A CPU logical core is the host processor and the GPU is the compute device. A CU in the GPU concurrently executes thousands of work-items. To manage such a large number of work-items, the GPU employs a SIMT(Single-Instruction, Multiple-Thread) architecture, where work-items execute one common instruction at a time. The CU creates, manages, schedules, and executes work-items in a group called a warp for NVIDIA GPUs and wavefront for AMD GPUs.
In our experiments we tested GPUs from NVIDIA and AMD. Both hardware vendors support OpenCL, but use a different terminology to describe GPU hardware architectures. To describe the mapping between GPU hardware and the OpenCL model, we propose our own terminology provided in Table II.

From the perspective of a performance oriented programmer, the most important hardware feature of modern GPUs is the memory architecture. Modern GPUs have very efficient memory units, optimized for streaming workloads. The OpenCL programming model maps perfectly to the memory architecture of modern GPUs.
We distinguish between two different memory levels, on-chip and on-board memory. The fastest memory, as can be seen in Table III, is the on-chip memory, defined as registers and shared memory [10]. This memory has a typical size of 16-48 kilobyte(KB) and is directly accessible by the streaming processors. The on-board global memory is shared among all streaming cores of the GPU. It is the largest, typically about 2-8 gigabyte(GB) large, and the most commonly used memory, but also the slowest memory on the GPU. Managing the significant performance difference between on-board and on-chip memory is the primary concern of a GPU programmer.

TABLE III. GPU MEMORY BANDWIDTH AND MAPPING TO OPENCL MEMORY MODEL [10]

OpenCL memory	GPU memory	Bandwidth
Private	Register on-chip	<10000 GB/s
Local	Shared on-chip	<2000 GB/s
Global	Global on-board	<300 GB/s

IV. OPENCL BASED SIFT IMPLEMENTATION

We start with an analysis of the SIFT algorithm. The goal of this analysis is the extraction of data and task dependencies to explore the parallelization opportunities. The parallelism of the SIFT algorithms can be defined at different levels.

- Octave level
  Different algorithm parts can be pipelined and concurrently processed. The initial image is incrementally convoluted with Gaussian functions to generate the scale space images. Adjacent scale space images are subtracted to get the DoG images. When all scale images of octave n have been processed, we start in parallel the computation of octave $n+1$. Except for the first octave, the input of the next octave is the resampled last scale space image of the preceding octave. The computational flow is visualized in Figure 3;
- Image level
  Each scale space and DoG image can be decomposed into smaller sub-images and processed in parallel.
- Feature level
  The calculation of the feature orientation and the descriptor uses read-only memory operations. Read-only memory operations are independent and can be easily parallelized.

During the analysis process, we identified an additional algorithmic change that can be applied in case of the SIFT algorithm. The Gaussian scale space is based on the Gaussian smoothing Equation (1).
A 2D Gaussian smoothing can be implemented as a separable two-pass Gaussian smoothing method. This method enables reduction in number of executed instructions.
In computation terms, the non-spearable filtering of $M \times N$ image with a $P \times Q$ filter kernel requires $MNPQ$ multiply-accumulate(MAC) operations. The horizontal convolution pass requires MNP MAC operations, the vertical pass MNQ MAC operations, for a total of $MN(P+Q)$ MAC operations. The gain in saved instructions is given by $PQ/(P+Q)$.
Another aspect is the handling of pixels at the image boundary. For all boundary pixels of an image, it is necessary to load additional pixels. We have implemented a boundary pixel repetition method for the CPU and GPU implementation which is based on [3]. In the following we will describe a baseline algorithm implementation for both CPU and GPU and the applied architecture specific optimizations to speed up the baseline implementation on different platforms.

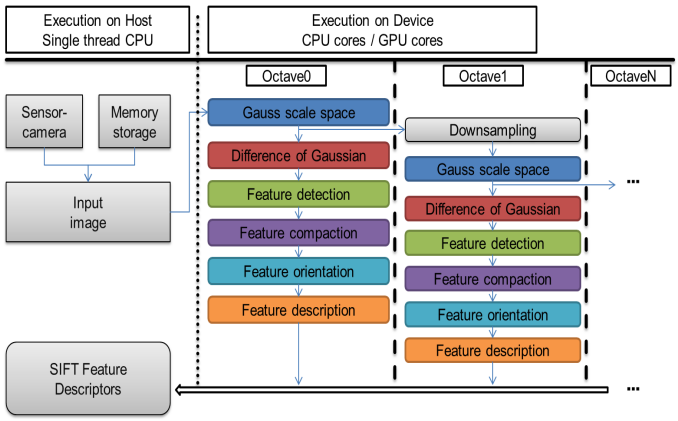


Fig. 3. High level view of SIFT-OpenCL implementation

TABLE IV. MAPPING TO OPENCL RESOURCES;W/H - IMAGE WIDTH/HEIGHT, WI - WORK-ITEMS, WG- WORK-GROUP

Kernel names	# WI	# WI/WG
Gauss Scale Space	$w \times h$	runtime control
DoG	$w \times h$	runtime control
Feature Detection	$w \times h$	runtime control
Feature Compactification	$w \times h$	runtime control
Feature Orientation	# features	runtime control
Feature Description	# features	runtime control

TABLE V. WORKLOAD DISTRIBUTION, SINGLE OCTAVE ON TESLA C2050, ECC=ON, IMAGE IL2

Kernel names	baseline kernels	# feat.
Gaussian scale space	20.3%	-
DoG	1.9%	-
Feat. Detection	5.3%	4539
Feat. Compactification	5.7%	4539
Feat. Orientation	28.1%	4539
Feat. Descriptor	38.4%	4539
Transfer from/to host	0.05%	4539

A. Baseline implementation of SIFT and mapping to OpenCL programming model

Our baseline implementation of the SIFT algorithm is a parallelized implementation without platform specific optimizations. The implementation has, as can be seen in Figure 3, two main parts: the host part and the device part. The workload for each octave is send through a separated command queue for each avaliable device. Our implementation with independent command queues enables pipelined octave execution.
The host part is a C++ control program that is responsible for controlling and communicating with the found OpenCL-capable compute devices. It is also responsible for the distribution of the OpenCL-kernel functions and the synchronization of their execution. The host program reads image frames and sends them to the compute devices for processing. When the calculation is completed, the results are transferred back from the compute devices. On the device part, the different kernels are processed.
To keep our host program unified for CPU and GPU, we decided to use only generally available OpenCL memory spaces for GPUs and CPUs and not to use special memories like the image2D buffers available for many GPUs. The workloads are differently mapped to the OpenCL resources.The execution setup is implicit controlled by the OpenCL runtime. The OpenCL runtime decide how many work items should be allocated for a single work group. Table IV shows the execution setup and mapping of the work-items to the individual workloads.
Our solution contains an additional stage, the feature compaction. After the detection phase, the keypoints form a sparse matrix in the scale space with an unknown number of entries. The number of entries, the keypoints, is calculated in parallel with an compaction algorithm based on prefix-sums [11]. The use of the compaction algorithm allows for an efficient memory allocation for the descriptor buffers and avoids the creation of redundant work-items during the descriptor building phase.

B. Optimization of OpenCL-kernels for many-core GPU devices

An analysis of the execution time of the baseline implementation and a comparison with the computational capabilities of GPUs shows a significant inefficiency. Therefore it is necessary to explore optimization techniques. The applied GPU-specific optimization techniques are well described in [10], [12], [13].
To identify the bottlenecks, we begin with an analysis of the baseline implementation using NVIDIA's and AMD's profiler tools [7], [14]. The output information from the profilers gives a full view about the relative runtime of each workload and helps to identify performance bottlenecks. As performance metric we choose the number of registers used, the amount of used shared memory and the number of work-items inside a single work-group. These parameters are highly correlated with the run-time. The results in Table V show the relative execution time for a single octave. The experimental setup is described in Section VI. We decided to optimize the following kernels: Gaussian scale space, feature orientation and feature descriptor. Those kernels consume more than 80% of the total execution time and are a proper target for further analysis and optimization.

We focus on optimization techniques that target the memory hierarchy of GPUs. The memory hierarchy of GPUs is generic and common for all tested GPUs in our experiments. There are also special optimization techniques related to specific GPU architectures like the AMD Graphic Core Next(GCN) [15]. They are not applied in our implementation. The applied optimization techniques are based on the following three observations:

1) Global memory access
   Modern Dynamic Random Access Memory(DRAM) is designed to transfer large lines of data in bursts. Poor usage of those bursts means wasted memory bandwidth. In these situations, the most effective optimization strategy is to transform the data layout in the memory or to change the memory access pattern. The memory coalescing technique makes off-chip accesses efficient by combining load/store requests from consecutive work-items to reduce the number of requested words.
2) Tiling
   Manual Tiling improves data locality and on-chip memory usage. By tiling, one uses smaller sets of data, so that the sets fit into the faster on-chip memory

TABLE VI. UNOPTIMIZED KERNELS MAPPING TO OPENCL RESOURCES;REG-REGISTERS,SM-SHARED MEMORY

Kernels	# reg./WI	SM/WG [bytes]	# WI	# WI/WG
Conv. hor.	5	0	$w \times h$	runtime cont.
Conv. vert.	6	0	$w \times h$	runtime cont.
Feat. orient.	33	0	# feat.	runtime cont.
Feat. desc.	17	0	# feat.	runtime cont.

TABLE VII. OPTIMIZED KERNELS MAPPING TO OPENCL RESOURCES

Kernels	# reg./WI	SM/WG [byte]	# WI	# WI / WG
Conv. hor.	5	640	$w \times h$	128×1
Conv. vert.	7	7296	$w \times h$	16×16
Feat. orient.	18	5376	# feat. ×36	36
Feat. descr.	12	4608	# feat. ×128	128

while the system processes them.

3) Occupancy
The occupancy is defined as the number of active warps/wavefronts per streaming core divided by the maximum possible number of warps/wavefronts per streaming core. When some work-items from a work-group are accessing global memory, these items are effectively stalled for hundreds of instruction cycles, while the other work-group items continue working. Internally, the GPU has a warps/wavefronts scheduler. This scheduler enables some warps/wavefronts to perform memory accesses, while others perform calculations. This is effectively hiding the memory accesses.

This leads to the following individual kernel optimizations.

1) Gaussian scale space - image blur kernel: The Gaussian blurring is based on a separable convolution operator. The major problem of the baseline implementation, is the access pattern to the GPU global memory, where all scale space images are stored. To optimize this, we use the GPU shared memory. The on-chip memory access is faster than a global memory access and has a higher throughput on a GPU as shown in Table III. Table VII shows how the global work-items are divided into different work-groups compared to the baseline mapping shown in Table VI.
Each work group fetches a set of data into the faster on-chip memory and then does the convolution. The global memory access pattern is transformed, so that work-items request load/store operations in a coalesced manner. After completion, the results are transferred back to the global memory. Table VIII shows the speedup. The low speedup is explained by the implicit use of the cache memory on the streaming cores of the used GPU. Due to the low arithmetic density of this kernel, other optimizations than memory access optimizations will not give a significant speedup.

2) Feature orientation kernel: For each keypoint, the gradient histogram with 36 bins is created from pixels around its position. In the baseline case, we use one work-item to calculate the histograms and to find the dominant orientation as shown in Table VI. Each work item inside a single wavefront/warp reads data which is spread in memory space. The amount of used registers, together with the uncached access into memory prohibits a high throughput on a streaming core. If the amount of memory requests is high and not properly mapped to consecutive memory addresses, than the memory unit of GPU is unable to coalesce memory accesses on a streaming core [13].

TABLE VIII. EXECUTION TIME OF SINGLE OCTAVE ON TESLA C2050, ECC=ON, IMAGE IL2 [MS]

Kernels	Baseline	Optimized	# feat.	Speedup
Conv. hor.	1.56	1.00	-	1.56
Conv. vert.	2.04	1.24	-	1.64
Feat. orient.	26.8	3.8	4539	7.05
Feat. descr.	36.7	10.6	4539	3.4

TABLE IX. WORKLOAD DISTRIBUTION, SINGLE OCTAVE ON XEONE5, IMAGE IL2

Kernel names	un-optimized kernels	# feat.
Gauss Scale Space	55.43%	-
DoG	2.2%	-
Feat. Detection	5%	4539
Feat. Compaction	8.91%	4539
Feat. Orientation	8.76%	4539
Feat. Description	20.5%	4539
Transfer from/to host	0.001%	4539

To achieve a higher throughput, we use shared memory and create more work-items. For each feature, a work-group of 36 work-items is started as shown in Table VII. The set of pixels around a keypoint position is divided in 36 sub-regions. The division of the local area in smaller sub-regions enables parallel computation on them and to start additional work-items. Each work-item concurrently calculates the histogram on a sub-region and saves the result in the shared memory. When all sub-histogram calculations are finished, a merge step is performed to obtain a single histogram. In the next step, the dominant orientation is obtained. The use of additional work-items combined with shared memory usage reduced the amount of non-coalesced memory transactions, resulting in a speedup of 7x as shown in Table VIII. A further speedup is limited due to the high amount of shared memory, which decreases the number of active wavefronts/warps.

3) Feature descriptor kernel: As described in Section II, the descriptor kernel also works on a local area around a keypoint. For each keypoint, 16 gradient histograms are created from the pixels around its position and combined to a descriptor. In the baseline implementation, we use one work-item of the work-group to calculate the histograms. Each created histogram contains 9 bins. The histogram is stored in registers as shown in Table VI. Similar to the orientation kernel, for each feature, a work-group with 128 additional work-items is started as shown in Table VII. The work-group is divided into 16 sub-groups with 8 work items each. Each work-item independently does its calculations on a sub-region around the keypoint. Each computed histogram is temporarily saved in the shared memory and all 16 histograms are concurrently calculated on the streaming processors. This results in a speedup of 3.4 as shown in Table VIII.

C. OpenCL-kernels optimization for multi-core CPU devices

For our CPU optimization process, we choose a hardware feature that is available on many modern CPUs, the SIMD-vector instructions. Table IX shows the relative run-time of the baseline kernels on a multi-core CPU. The Gaussian convolution is the most time consuming task.

There are important differences between typical CPUs and GPUs in the context of OpenCL. The most important difference between CPU and GPU is the memory architecture. A

TABLE X. EXECUTION OF GAUSS SEPARABLE CONVOLUTION ON XEONE5, IMAGE IL2 [MS]

Workloads	baseline	optimized	Speedup
Conv. horizontal	11.5	3.2	3.59
Conv. vertical	12.8	3.7	3.45

typical GPU has different memory levels that the programmer can manually address. In contrast, the CPU contains only one memory region, the global memory with implicit cache levels. Another difference is the mapping of vector data types into hardware resources. The OpenCL-C programming language provides vector data types and corresponding operators and functions. In the case of the CPU, vector data types are mapped by the OpenCL compiler to Single Instruction Multiple Data (SIMD) functions and Streaming SIMD Extensions (SSE) registers. Utilizing SIMD units has been one of the key performance optimization techniques for CPUs [16], [17].

1) Gaussian Scale space - Image blur kernel: We have implemented the Gaussian blurring using vector data types and vector functions. Due to the high spatial locality of data accesses, a separable convolution is an ideal candidate to use SIMD instructions. The main difference between the optimized and the baseline version lies in the data access granularity. The optimized version with vector types uses a coarse-granular access to data items while the baseline version reads/writes single data items.
Due to the high spatial SIMD-locality of the data, the memory management unit of the CPU is able to efficiently fetch the data to the SSE registers which have, similar to GPU registers, a very low memory access time. The scalar float variables in the baseline implementation were replaced by float4 vector data types. Table X shows a speedup very close to the maximum theoretical value of 4. This value could not be reached as the calculation on the boundary pixels is done without SIMD instructions.

V. SIFT IMPLEMENTATION ON COMBINED CPU/GPU DEVICES

In a second step, the optimized implementations are used to run them in parallel on a system with a CPU and several GPUs. The first step is to implement a partition strategy. The host code identifies all available OpenCL-capable devices. The input image is divided into sub-images, corresponding to the number of found devices. If the size of the image is not evenly dividable, the last device will get the remaining parts of the image, while the other devices get the same amount of data. The sub-images are distributed to the compute devices and processed. After completion, all sub-results are merged together and the whole process is repeated with the next frame. The partition strategy is named static scheduling.
We evaluated the static scheduling with the setup described in Section VI. The results in Figure 4 show, that the static scheduling does not optimally utilize the combined performance of the individual devices. The reason of the inefficiency is the unbalanced work distribution, as the slowest (CPU0) and fastest device (GPU1) get the same amount of work. As a consequence, the processing on CPU0 constrains the whole execution time. Another major difficulty, inherent in every algorithm for feature detection, is the variation of work in sub-images due to the unpredictable feature distribution. The static

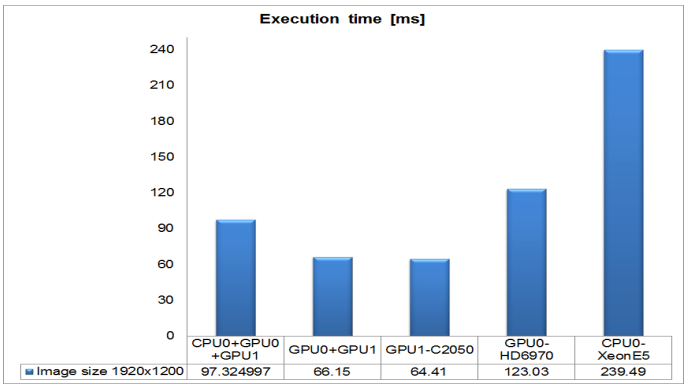


Fig. 4. Execution of SIFT on multiple devices

scheduler cannot predict variations in the number of features and properly distribute the work based on such a prediction. A more efficient approach would be a dynamic scheduling based on a fine-grained partition of the images to increase the performance. Each device will get a new small sub-image after completing the calculations on another sub-image. So devices well suited for some kernel execution will get more work than less suited devices.

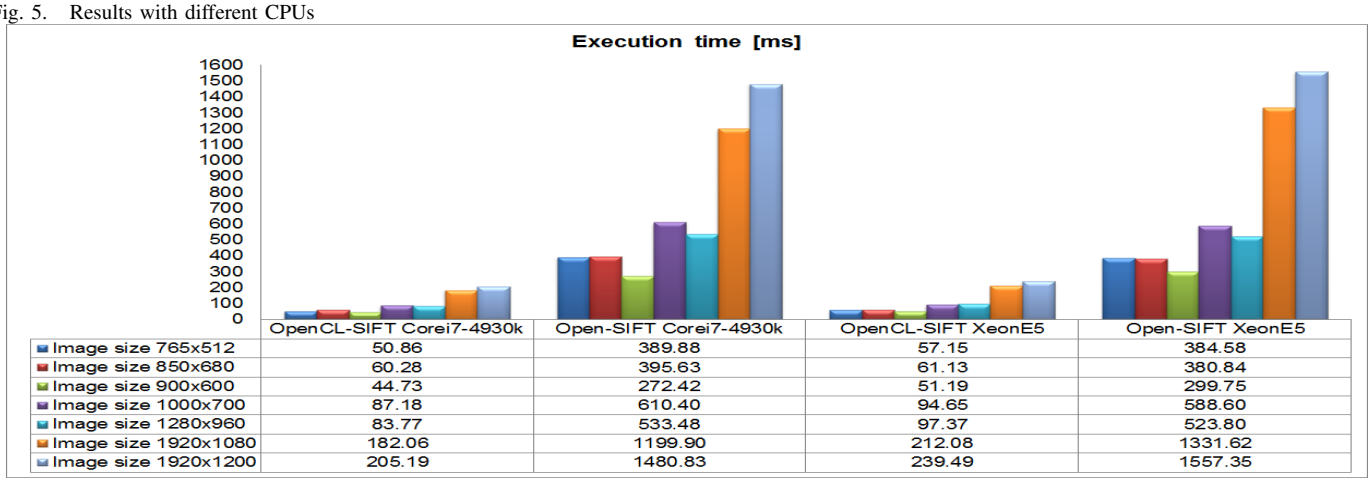
VI. EXPERIMENTAL METHODOLOGY AND RESULTS

To evaluate the scalability and portability of our implementations, different tests were performed on various CPUs and GPUs. The GPUs used in our experiments have different computational capabilities and are based on different architectures, like AMD Very Long Instruction Word (VLIW), Graphic Core Next-GCN or NVIDIA Fermi, Kepler architectures [15], [18], [19]. All of them have a memory hierarchy as described in Section III. Additionally, the CPUs support SIMD-vector instructions.

A. Experimental setup

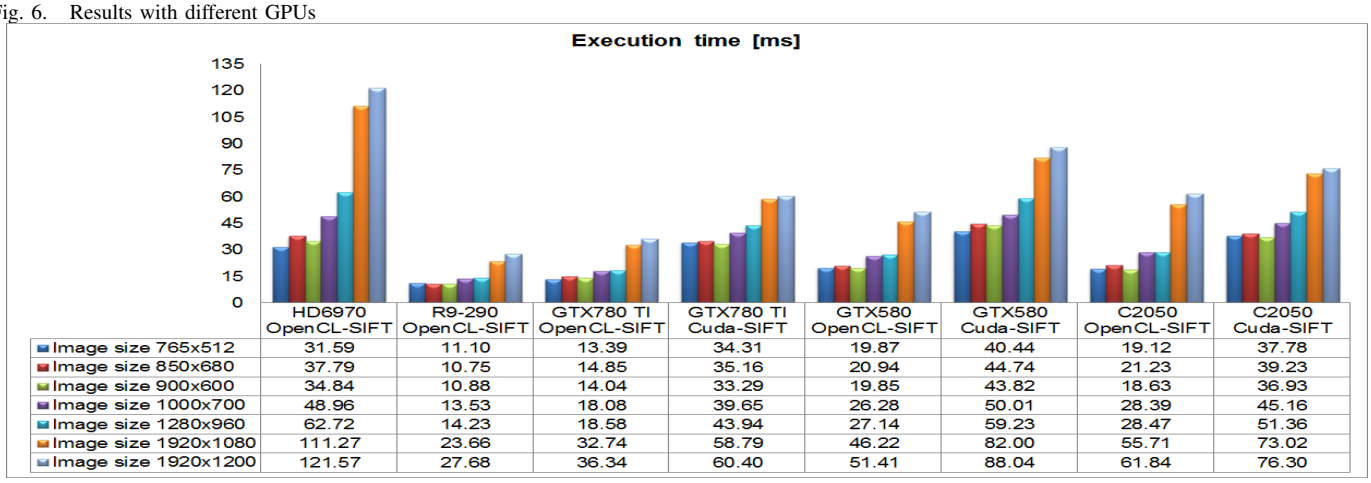
Results showed along this paper are performed on two heterogeneous computing systems. The first one consists of an Intel Xeon E5-2667 CPU, a NVIDIA Tesla C2050 and a AMD Radeon HD 6970 GPU. The second computing system consits of an Intel CoreI7-4930k CPU and AMD R9-290 or Nvidia GTX780 TI GPUs. The Intel CPUs have 6 physical cores each and support hyper-threading technology. Both desktops have 32GB of main memory and run Windows7 Ultimate OS.
The first and second OpenCL-CPU runtime is configured by the INTEL OpenCL-XE SDK R3 [20] and uses the entire 6 physical cores as the compute device. It uses one of the logical cores as the host processor. This logical core is shared by the host and the device. The OpenCL-GPU runtimes are configured by the AMD-APP SDK 2.9 [21] and CUDA Toolkit 5.0 [22], use a logical CPU core as the host processor and the GPU streaming cores as the compute device. The kernel

Fig. 5. Results with different CPUs



	OpenCL-SIFT Corei7-4930k	Open-SIFT Corei7-4930k	OpenCL-SIFT XeonE5	Open-SIFT XeonE5
Image size 765x512	50.86	389.88	57.15	384.58
Image size 850x680	60.28	395.63	61.13	380.84
Image size 900x600	44.73	272.42	51.19	299.75
Image size 1000x700	87.18	610.40	94.65	588.60
Image size 1280x960	83.77	533.48	97.37	523.80
Image size 1920x1080	182.06	1199.90	212.08	1331.62
Image size 1920x1200	205.19	1480.83	239.49	1557.35

Fig. 6. Results with different GPUs



	HD6970 OpenCL-SIFT	R9-290 OpenCL-SIFT	GTX780 TI OpenCL-SIFT	GTX780 TI Cuda-SIFT	GTX580 OpenCL-SIFT	GTX580 Cuda-SIFT	C2050 OpenCL-SIFT	C2050 Cuda-SIFT
Image size 765x512	31.59	11.10	13.39	34.31	19.87	40.44	19.12	37.78
Image size 850x680	37.79	10.75	14.85	35.16	20.94	44.74	21.23	39.23
Image size 900x600	34.84	10.88	14.04	33.29	19.85	43.82	18.63	36.93
Image size 1000x700	48.96	13.53	18.08	39.65	26.28	50.01	28.39	45.16
Image size 1280x960	62.72	14.23	18.58	43.94	27.14	59.23	28.47	51.36
Image size 1920x1080	111.27	23.66	32.74	58.79	46.22	82.00	55.71	73.02
Image size 1920x1200	121.57	27.68	36.34	60.40	51.41	88.04	61.84	76.30

codes are compiled with the device compiler provided by each OpenCL SDK.

B. Methodology

We performed tests with different hardware platforms and use additional open-source implementations of the SIFT algorithm as references. For the CPU, the Open-SIFT implementation was selected. Open-SIFT is a C99 implementation without parallelization [23]. For the GPU, we choose Cuda-SIFT [3]. Cuda-SIFT is implemented with the CUDA-API provided by NVIDIA. CUDA is a vendor specific framework which only supports NVIDIA GPUs.

We use two sets of images with different sizes as shown in Table XI. The small size image set is a data-set collected by K.Mikolajczyk [24] and is a standard image set used in many computer vision research activities. Images in the large size image set are downloaded from the Internet. With the different image sizes we want to test the scalability of our solution. The SIFT algorithm has a group of adjustable parameters which control the number of keypoints and the computational load. All implementations, Open-SIFT, Cuda-SIFT and our OpenCL-SIFT, work on the image set shown in Table XI and have a common set of input parameters: 6 scale images per octave, 5 octaves, 1 orientation pro feature, a contrast threshold of 0.0066, a curvature threshold of 10.0 and a Gaussian filter σ of 1.6. The execution time is defined as the averaged execution time of 10 consecutive test runs.

TABLE XI. SMALL IMAGE SET [24] / LARGE IMAGE SET

Image name	Image size
bark	765x512
boat	850x680
leuven	900x600
trees	1000x700

Image name	Image size
IL0	1280x960
IL1	1920x1080
IL2	1920x1200

C. Results

Figure 5 shows the execution time on different CPUs for the Open-SIFT and OpenCL-SIFT implementations. Both CPUs, a XeonE5-2667 and a Core i7-4930k processor, have 6 physical cores with enabled Hyper-Threading. The OpenCL run-time recognizes these cores as 12 CU. In Figure 5 we can observe an average-linear speedup on XeonE5 and on Corei7-4930k of factors 6.1 and 6.7 respectively. In comparsion to the Open-SIFT implementation our solution finds more stable features as shown in Table XII. The reason for this is a different algorithm for the sub-pixel interpolation used in the detector kernel. The CPUs are on average 4.11 times slower than the GPUs. This difference is mainly due to lower memory bandwidth of the CPU as one sees a strong correlation between the CPU performance and the data size. For small size images,

TABLE XII. COMPARE OPEN-SIFT AND OPENCL-SIFT

Implementation	Open-SIFT Corei7-4930k		OpenCL-SIFT Corei7-4930k	
Image name	bark1	bark2	bark1	bark2
Count features	1071	1100	1902	2132
Count matches	277		366	
Image name	boat1	boat2	boat1	boat2
Count features	1170	1111	1739	1697
Count matches	376		459	
Image name	leuven1	leuven2	leuven1	leuven2
Count features	603	494	1168	1143
Count matches	294		638	
Image name	trees1	trees2	trees1	trees2
Count features	2299	2193	3064	2910
Count matches	394		620	

TABLE XIII. COMPARE CUDA-SIFT AND OPENCL-SIFT

Implementation	Cuda-SIFT GTX780 TI		OpenCL-SIFT GTX780 TI	
Image name	bark1	bark2	bark1	bark2
Count features	1928	2129	1988	2230
Count matches	243		420	
Image name	boat1	boat2	boat1	boat2
Count features	1591	1479	1802	1761
Count matches	321		503	
Image name	leuven1	leuven2	leuven1	leuven2
Count features	1033	860	1190	1162
Count matches	401		658	
Image name	trees1	trees2	trees1	trees2
Count features	2903	2735	3121	2963
Count matches	273		641	

the run-time of CPU and GPU are comparable (Figures 5 and 6). On larger images, the GPUs outperform CPUs due to the higher memory bandwidth and the higher peak arithmetic performance. The quality of the features was also compared. For the comparison we matched SIFT features between images using kd-trees and Best Bin First search method. Additionally, the geometrical image transformation from feature matches using RANSAC is performed [23].

Figure 6 shows the execution times, including communication times on different GPUs with the Cuda-SIFT and the OpenCL-SIFT implementation. We tested low-end and high-end GPUs with different computation capabilities. Our tests showed that the proposed optimization techniques fits different GPUs well. The expected scaling between low-end HD 6970 results and high-end R9 290 results can be observed. We also observed a difference of the run-time related to the variable number of features found in the different implementations and different algorithms used to implement the kernels. In comparison to Cuda-SIFT, our solution finds again more stable features as shown in Table XIII. Additionally, our implementation is portable across GPUs and is in avarage 1.84 times faster than Cuda-SIFT on Nvidia GPUs.

The experimental results shows that OpenCL-SIFT finds more stable features and still is faster than reference implementations. The maximum achieved framerate, 43 FPS on AMD R9-290, enables a real time local feature extraction.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented OpenCL based SIFT implementations that transparently run on both CPUs and GPUs. The proposed optimizations for both devices have proved efficient and demonstrate scalability across various devices. Additionally, our approach enables the concurrent processing on multiple devices. The results show that the proposed optimizations increase the performance of the solution despite significant differences of the internal GPU architectures.

We plan to investigate more efficient scheduling methods to further speed up the application on systems with multiple devices. Additionally, we want to explore the mapping of the OpenCL programming model and associated optimization techniques to specialized coprocessors such as the Intel Xeon-Phi.

REFERENCES

[1] Q. Zhang, Y. Chen, Y. Zhang, and Y. Xu, "Sift implementation and optimization for multi-core systems," in *IEEE International Parallel and Distributed Processing Symposium(IPDPS 2008))*, 2008, pp. 1–8.

[2] N. Zhang, "Sift implementation and optimization for multi-core systems," in *Proc. of the 5th international conference on Emerging intelligent computing technology and applications*, 2009, pp. 287–296.

[3] C.Wu, "SIFTGPU: A GPU implemenatation of scale invariant feature transform," 2011. [Online]. Available: http://www.cs.unc.edu/~ccwu/siftgpu/\#lowesift

[4] M. Lu, "Fast implementation of scale invariant feature transform based on cuda," *Appl. Math. Inf. Sci.*, vol. 7, no. 2L, pp. 717–722, 2013.

[5] S. Heymann, B. Froehlich, F. Medien, K. Mueller, and T. Wiegand, "Sift implementation and optimization for general-gurpose gpu," presented at the Winter School of Computer Graphics, 2007.

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

[7] AMD CodeXL 1.4. [Online]. Available: http://www.amd.com

[8] "The OpenCL specification, v1.2, rev. 19," Khronos OpenCL Working Group, 2012. [Online]. Available: http://www.khronos.org/opencl

[9] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," in *ICS '12 Proc. of the 26th ACM international conference on Supercomputing*, 2012, pp. 341–352.

[10] "AMD Accelerated Parallel Processing - OpenCL Programming Guide, v2.7," 2013. [Online]. Available: http://www.amd.com/

[11] M. Harris, "Parallel prefix sum with CUDA.GPU Gems 3-Chapter 39," April 2007. [Online]. Available: http://www.nvidia.com/

[12] "OpenCL programming guide for the cuda architecture, v3.2," 2010. [Online]. Available: http://www.nvidia.com/

[13] "NVIDIA.OpenCL best practices guide," May 2010. [Online]. Available: http://www.nvidia.com/

[14] NVIDIA Visual Profiler. CUDA Toolkit 4.2. [Online]. Available: http://www.nvidia.com

[15] "Amd graphics cores next (gcn) architecture," White Paper, AMD, June 2012. [Online]. Available: http://www.amd.com

[16] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W. m. W, "Efficient compilation of fine grained spmd-threaded programs for multicore cpus," in *CGO '10 Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 111–119.

[17] S. Maleki, Y. Gao, M. J. Garzaran, T. Wong, and D. A. Padua, "An evaluation of vectorizing compilers," in *PACT '11 Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 372–382.

[18] "Nvidia's next generation cuda compute architecture: Kepler, v1.0," White Paper, NVIDIA, 2012. [Online]. Available: http://www.nvidia.com

[19] "Nvidia's next generation cuda compute architecture: Fermi, v1.1," White Paper, NVIDIA, 2009. [Online]. Available: http://www.nvidia.com

[20] Intel SDK for OpenCL Applications XE 2013 R3. [Online]. Available: http://www.software.intel.com

[21] AMD APP SDK v2.9. [Online]. Available: http://www.amd.com

[22] CUDA Toolkit 5.0. [Online]. Available: http://www.nvidia.com

[23] R. Hess, "An open source sift library," in *Proc. ACM Multimedia (MM)*, 2010. [Online]. Available: http://robwhess.github.io/opensift/

[24] K.Mikolajczyk. Local feature evaluation dataset. [Online]. Available: http://www.robots.ox.ac.uk/~vgg/research/affine/