A Speed-Up Study for a Parallelized White Light Interferometry Preprocessing Algorithm on a Virtual Embedded Multiprocessor System

Dominik Schoenwetter, Max Schneider and Dietmar Fey

Chair of Computer Science 3 (Computer Architecture)
Friedrich-Alexander-University Erlangen-Nuremberg
Martensstr. 3, 91058 Erlangen
{dominik.schoenwetter, max.schneider, dietmar.fey}@informatik.uni-erlangen.de

Abstract: Parallel computing has been a niche for scientific research in academia for decades. However, as common industrial applications become more and more performance demanding and raising the clock frequency of conventional single-core systems is hardly an option due to reaching technological limitations, efficient use of (embedded) multi-core CPUs and many-core platforms has become imperative. 3D surface analysis of objects using the white light interferometry presents one of such challenging applications. The goal in this article is to get an impression which speed-up for an established and parallelized white light interferometry preprocessing algorithm, called *Contrast Method*, is possible on an embedded system that works without any operating system. Therefore, we decided to use a virtual environment that is able to simulate embedded multi-core as well as many-core systems and that enables running real application code on the designed system. The results show, that a significant speed-up is possible when using a many-core platform, instead of a design that only implements one single core, if the algorithm is parallelized for getting full advantage of the many-core design. Furthermore, an acceptable absolute run time is achievable.

1 Introduction

The white light interferometry scanning is a versatile technology which provides a reliable non-contact, 3D optical measurement of surface roughness in the nanometer range [KM07]. In the scanning device, which is usually a Michelson interferometer equipped with a broadband light source, the emitted white light beam is split into two separate beams. One of the beams is projected onto the object to be measured, while the other beam follows a well defined and constant path to a reference mirror. Both beams are reflected and superimposed, resulting in an interference pattern of light and dark fringes. This fringe pattern is captured on a CCD camera chip and processed in software. By moving the object closer to the scanning device in discrete time steps, the path difference between the two beams and, thus, the fringe pattern changes. During a common measurement process, the whole interference range (the region where the path length difference of reflected beams is less than the coherence length of the wight light) is covered [Lar00].

Thereby, individual time series of interference intensities are recorded by the pixels of a CCD sensor. Figure 1 shows such a two dimensional series, called *correlogram* or *interferogram*, for one pixel. At each pixel, where the optical path length difference of both beams is zero, the occurred constructive interferences have reached their maximum value. In the case that the object is not a flat plane, the maximum interference of each pixel point is obtained in different time steps for each pixel. A 3D map can be derived from the positions of the translation arm, where maximum intensities are observed, and the distance to the start position [His05]. Thus, the aim of the white light interferometry analysis process is to find the corresponding maximum interference value for each pixel of the CCD sensor.

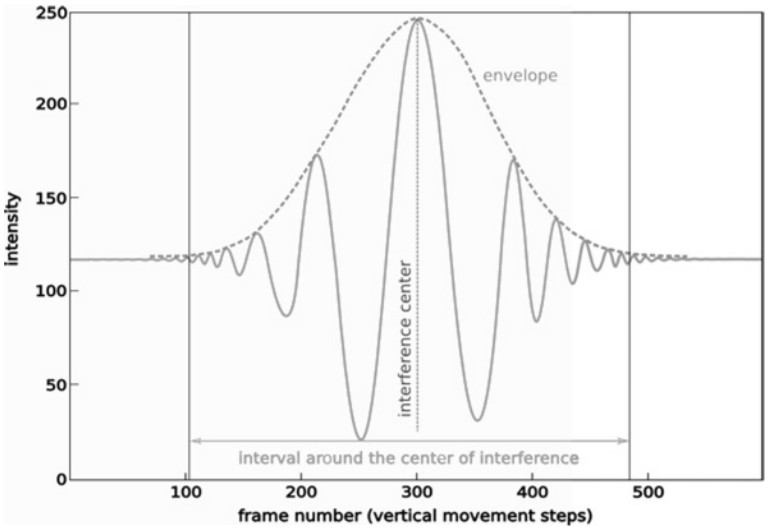

Figure 1: A synthetic interferogram or correlogram signal of a pixel

The measurement of the whole test object is done using the so-called stitching process. This means, that the scanning area is subdivided in 1×1 mm^2 regions that are scanned subsequently. For example a 25×25 mm^2 measuring area must be subdivided in 625 partitions. Usually more than 200 frames are necessary for a complete scan of each subarea. The scan of the whole area takes approximately 70 minutes with the camera that is used at the moment. To increase performance, a high speed camera is used to accelerate the scanning procedure. The necessary computations on the data, which are captured for such a subarea, are performed while the scan of the next subarea is already in execution. Because the scanning procedure is accelerated, the elaboration procedure has to be accelerated, too. This can be achieved by the parallelization of the elaboration procedure.

The *interference range* (*IR*) span at most few hundred intensity values and all other intensities captured by the camera are not relevant for the height map calculation. Thus, a preprocessing step in the white light interferometry analysis is used, to reduce the required image data to significant regions for the height map calculation. In the postprocessing

stage, the maximum interference for each pixel in the corresponding extracted region is calculated [His05].

Because each pixel's correlogram is elaborated independently from those of the other pixels, the analysis process is well suited for parallel processing.

As can be seen in Section 2 there are studies about performance and speed up using white light interferometry with GPUs as well as with conventional multi-core systems from Intel and AMD. According to our present knowledge there are no studies which performance and which speed up could be achieved on single or multi-core and many-core, respectively, embedded processors. As a consequence this is an unexplored niche which is, according to our opinion, a very interesting region because the usage of embedded multi-core systems is growing more and more and embedded computing gives a substantial added value to many products in fields such as automotive, industrial automation, telecommunications, consumer electronics or entertainment [COMS11]. Using an embedded hardware layout for the white light interferometry has the advantage of more power efficiency (less power consumption). When using a NVIDIA Tesla C2050 GPU, there is a maximum power consumption of 238 W TDP (Thermal Design Power) [Cor10]. For the Xeon X5650 Hexacore (2.66 GHz), a conventional multi-core processor developed by Intel, the power consumption amounts to 95 W TDP [Int11]. In comparison to that, a single ARM CortexA9 processor, that we used for our investigations, has a power consumption of 0.4 W (soft macro implementation) [ARM11]. As a consequence, the power consumption of an embedded multi-core system, consisting of single CortexA9 processors and up to ten cores, is only a fraction compared to a NVIDIA Tesla C2050 GPU or a Xeon X5650 Hexacore. In addition to that, a more compact construction of the measurement system is possible when using an embedded system instead of a GPU or a conventional multi-core system.

Our final goal is to integrate such a compact embedded system in a smart thrilling head, which includes micro optical devices and fast electronic evaluation systems, too. A GPU board with cooling equipment, for example, could not be integrated in the even mentioned thrilling head, because there is plain and simple not enough space available.

The development of a real embedded hardware layout can be very expensive because, as it is often the case, the hardware layout has to be changed during the development phase. To avoid this it will be of great advantage to use an environment that emulates the embedded hardware and could be modified whenever necessary. Such a virtual environment for embedded hardware and software development is *Open Virtual Platforms*^{TM} (*OVP*^{TM}) provided by *Imperas*^{TM}. With the aid of OVP it is possible to build single-core as well as multi-core and many-core hardware platforms, add desired peripherals and simulate real application code [OVP11].

Because of the ability to establish multi-core and many-core platforms running real application code, it is possible to develop parallel applications that can be simulated, verified and evaluated. That is a very important feature for our work and the reason why we chose OVP as the virtual environment for our study. The chosen preprocessing algorithm is the so-called Contrast Method (see Section 4). This method uses only one arithmetic instruction to calculate the central fringe in each interferogram and achieves a high performance as a consequence [SFKM11].

2 Related Work

The usage of multi-core and many-core technology in industrial white light interferometry is gaining more and more attention. The first attempt to use a graphical processing unit in the surface metrology has been approached by Purde et al. in 2004 [PMS+04]. They implemented analysis algorithms of the so called electronic speckle pattern interferometry on GPUs, allowing the measurement of surface contours using the High Level Shading Language (HLSL).

Gao presented in September 2010 a solution for imaging processing using GPUs for the wavelength scanning interferometry [GJMM10]. Thereby a GeForce GTX 280 was used for the parallelization of the computational intensive data analysis procedure and a approximately 30x speed-up was achieved.

Also in September 2010 Sylwestrzak et al. presented the application of massively parallel processing of Spectral Optical Coherence Tomography (SOCT) data using GPUs [SSST10]. By utilizing NVIDIA's GeForce GTX 285 with 2 GB device memory Sylwestrzak achieved overall imaging speed of over 100 fps for 2D tomograms consisting of 1024 A-scans.

In 2011 Schneider et al. published an article about three designated preprocessing white light interferometry algorithms on emerging multi- and many-core architectures [SFKM11]. He figured out that the best performing algorithm is the Contrast Method and that conventional multi-core systems are the best suited architectures for this algorithm.

3 The simulator

The simulator included in OVP is an instruction accurate simulator because the provided processor models are instruction accurate, too. This means, that the functionality of a processor's instruction execution is represented without regard to artifacts like pipelines.

OVP processors in multiprocessor platforms are not working simultaneously. For efficiency, each processor advances a certain number of instructions in turn. So in multiprocessor simulations a processor cannot respond until the processor has signaled, that he has finished its quantum. The quantum is defined as the time period in which each processor in turn simulates a certain number of instructions [Lim11a]. Simulated time is moved forward only at the end of a quantum. This can create simulation artifacts, where a processor spends time in a wait loop, while waiting for the quantum to finish. To avoid this the quantum has to be set very low (perhaps even to one, which will have a significant impact on simulation performance) so that the measurements will not be affected by this simulation artifacts. This can be adjusted in the simulator settings [Lim11b]. The simulation can only figure out how many instructions were executed. Assuming a perfect pipeline, where one instruction is executed per cycle, the instruction count divided by the mips rate (millions of instructions per second) would give the amount of time the program runs. Instruction accurate simulation cannot make a clear statement about time spent during pipeline stalls, due to cache misses and other things that are not modeled, so any conversion to time will have limited accuracy compared to actual hardware. But it is still

useful for comparing the relative performance of different algorithms, assuming that they have similar pipeline stall effects. Furthermore, the OVP-simulator provides the possibility for measuring instruction counts within a program. As a consequence, the instruction counts for specific code snippets can be recorded and if the quantum is set to one, like in this study, the registered instruction count divided by the mips rate is the amount of time the processor requires for executing the code snippet. In single-processor platforms there is no need to set the quantum to one because the multiprocessor scheduling algorithm does not affect and intervene, respectively, the simulation.

4 The preprocessing algorithm

In the preprocessing analysis step, each pixel's correlogram is demodulated, separating the carrier wave from the envelope. The demodulation process can be done by a simple approximation of envelope values. For this purpose, depending on the surface characteristics and the signal-to-noise ratio of the generated signals, different approaches have to be considered [Rob93]. Envelope demodulation serves also as a data reduction step, because only the interval around the center of the interference is relevant for the postprocessing [KM07], as can be seen in Figure 1. The center of interference signal itself can not be seen as the envelope's peak. Due to noise and the discrete measurement along the z-axis, the actual maximum could be between two captured intensities or it could be shifted in some direction away from the measured interference center. Thus, in the demodulation process the center of the interference is determined and a predefined number of intensities left and right from this point is extracted. This is done parallel to the scanning procedure, so that not all interference images must be stored, but only the relevant regions in corresponding correlograms. These regions are used in the postprocessing stage to get an approximation of the actual maximum, as accurate as possible. The approximation can be achieved by fitting a model envelope function to the detected envelope [Lar00].

For our investigations a algorithm called Contrast Method was chosen. For each pixel p it uses the maximum absolute difference of successive sampling points I from the input signal as an estimator for the envelope, see (1). Variable i represents the number of the current translation step. This filter becomes maximum where the interferogram oscillations have a maximum gradient, which is approximately around the maximum of the envelope [His05].

$$\hat{z}_0(p) = \arg\max_i |I_{i-1}(p) - I_i(p)| \tag{1}$$

5 Implementation details

5.1 Implementation of the virtual test environment

For generating the virtual hardware design with OVP, an application, that defines the necessary components at runtime, was written. The application accepts several parameters, e.g. if intermediate results shall be displayed or not, and of which type the used virtual processor is, e.g. a *CortexA9* made by *ARM* that we used for our investigations. One important parameter defines the number of CortexA9 processors that should be implemented for the design. Depending on that parameter, the corresponding application to run by the respective core is automatically loaded into the corresponding processor memory. There are two different programs that have to be loaded into the memories, if more than one core shall be used. These core-applications are called *MasterApp* and *SlaveApp* (see section 5.2). For each processor two local memories are generated. The first one contains the heap, the stack and the memory area for the application program to run. The second local memories (one belonging to each processor) contain the data, the Contrast Method works with, as well as necessary synchronization variables. These second local memories are implemented as a distributed shared memory system. As a consequence, each core has the possibility to access the variables that are required for the synchronization of every processor. After all local memories are instantiated, they are linked to the associated cores. The resulting hardware design, depending on the number of cores to implement, is graphically illustrated in Figure 2.

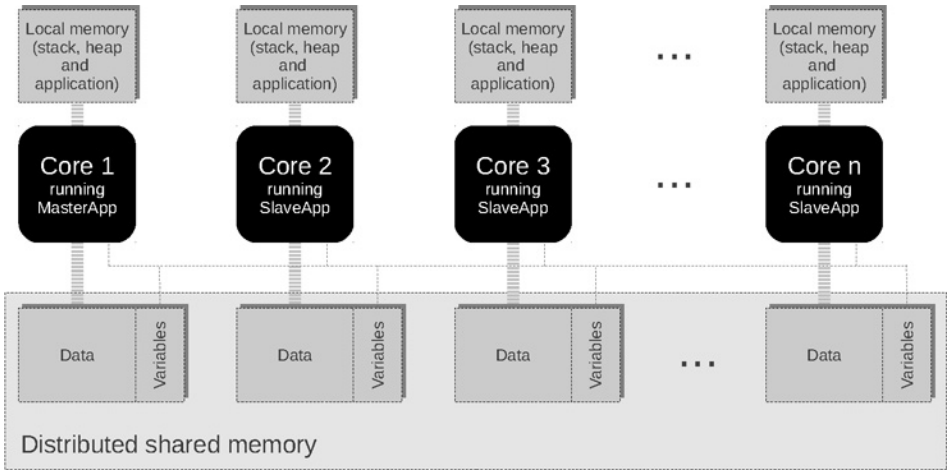

Figure 2: Generated design depending on the number of cores to use

5.2 Parallel implementation of the algorithm

Employed cameras in industrial applications use different color depths (8 bit / 16 bit). However, the data analysis has to be accomplished in single precision or in double precision mode. Therefore, we decided to employ template classes, so that all the functionality needed to execute the described algorithm with corresponding parameter combinations, can be accessed through the same class. The user only has to provide the template parameters during instantiation. The methods of this class are compiled and linked into an application named *MasterApp*, that is executed by only one core, independent of the number of cores used in the design, called *master* in this paper. This processor defines all necessary shared (synchronization) variables and their contents in the initial phase. All other cores (the *slaves*) run another application that waits continuously for instructions of the master with the exception of the initial phase where all address assignments of required shared variables are performed. The application running on the slaves is called *SlaveApp* (refer to Figure 2). The interaction of both applications is illustrated in the following. As mentioned in Section 1, each pixel's correlogram is elaborated independently from those of the other pixels. So the first stage of the preprocessing algorithm (computing the maximum of each pixel's interferogram) and the second stage (calculating the relevant intensities around the maximums) may also be regarded as independent. Hence, the parallelization of the algorithm takes places here. The number of pixels of an image can be seen equal to the runs of several for-loops that have to be executed during the algorithm and whose instructions are the same for every pixel's correlogram. The number of pixels each core has to process shall be evenly distributed, i.e. n_P/n_C. Thereby, n_P is the number of pixels per frame and n_C is the number of cores used in the design. No core may process more than one pixel than any other core to achieve an almost equal load balancing. If the reminder rem of the division n_P/n_C is unequal to zero, then one pixel more is assigned to the first rem cores.

For a clearer understanding, Listing 1 illustrates this circumstance in pseudo code.

```
// Total number of cores used in the design
unsigned int number_of_cores;
// Total number of pixels per frame
unsigned int number_of_pixels;
// Array that holds the runs each core has to perform
// runs_per_core[0] = number of loop passes for core 1
// runs_per_core[1] = number of loop passes for core 2
// ...
unsigned int runs_per_core[number_of_cores];
// The number of cores that have to perform one loop pass more
// than the other cores
unsigned int rem = number_of_pixels % number_of_cores;
// Assign loop passes
for(unsigned int index = 0; index < number_of_cores; index++)
{
    // Integer division
    runs_per_core[index] = number_of_pixels / number_of_cores;
    if(i < rem)
        runs_per_core[index] += 1;
}
```

Listing 1: Pseudo code for equal load balancing

Because all relevant data is located in the distributed shared memory, and each core is working on a different address space within the memory area, there are no simultaneous memory accesses that could trigger additional delays. Everytime such a parallelized region is accessed by the application running on the master, he instructs all slaves to perform the necessary (and the same as the master) computations on their assigned data regions in the distributed shared memory. After all cores (including the master) have finished their work, the application on the master is continued and the slaves wait for processing (sleep mode) until the next parallelized region is entered. The even mentioned procedure is realized using the spin lock principle and without any operating system. The synchronization variables are located in the shared memory area which is accessible for each processor.

The whole number of input frames to analyze by the preprocessing algorithm, as well as the number of intensities to store around each pixel's interferogram for postprocessing, are depending on the user input. For test purposes the input frames are generated in software.

6 Results

A detailed and precise overview about the execution times, the Contrast Method requires for the chosen test cases on the respective virtual hardware, is shown in Table 1.

	256 x 256 pixels			**512 x 512 pixels**			**1024 x 1024 pixels**		
	p/sec	m/sec	c/sec	p/sec	m/sec	c/sec	p/sec	m/sec	c/sec
1 core	10.93	9.55	1.28	43.70	38.20	5.14	174.71	152.80	20.55
2 cores	5.59	4.78	0.72	22.32	19.10	2.86	89.19	76.40	11.43
3 cores	3.76	3.18	0.47	14.00	12.73	1.91	59.91	50.93	7.62
4 cores	2.85	2.39	0.36	11.34	9.55	1.43	45.27	38.20	5.71
5 cores	2.30	1.91	0.27	9.14	7.64	1.14	36.49	30.56	4.57
6 cores	1.93	1.59	0.24	7.68	6.37	0.95	30.64	25.47	3.81
7 cores	1.67	1.37	0.20	6.63	5.46	0.82	26.45	21.83	3.27
8 cores	1.47	1.19	0.18	5.85	4.78	0.71	23.32	19.10	2.86
9 cores	1.32	1.06	0.16	5.24	4.25	0.64	20.88	16.98	2.54
10 cores	1.20	0.96	0.14	4.75	3.82	0.57	18.93	15.28	2.29

Table 1: Detailed overview of obtaining times depending on the number of cores and the resolution

Thereby, variable p stands for the required time of the whole preprocessing algorithm, variable m for the time that is required for calculating the maximum of each pixel's interferogram and variable c is the representative for the time required for computing the intensities around the maximums. Each measurement was executed with a setup where a color depth of 16 bits and the double precision processing mode were used. Furthermore, the number of frames for the scanning procedure was set to 100 and the number of frames

containing relevant data (intensity values) around the maximum of each pixel's interferogram was placed to 30. The resolution by contrast changed during the measurements from 256×256 to 512×512 and finally to 1024×1024 pixels.

As expected, the amount of the required time for calculating the maximums and for computing the intensities is not equal to the time the whole preprocessing algorithm requires (serial proportion). For a resolution of 256×256 pixels this proportion amounts to 100 ms, for 512×512 pixels to 359 ms and for 1024×1024 pixels to 1.357 seconds of the whole algorithm. As a consequence, the percentage of this serial proportion rises (with reference to the time the whole algorithm requires), the more cores are used.

Based on the results shown in Table 1, the speed-up, depending on the number of cores used and in comparison to one core, for each resolution can be calculated. Figure 3 illustrates the resulting speed-ups of the whole preprocessing algorithm for different resolutions graphically.

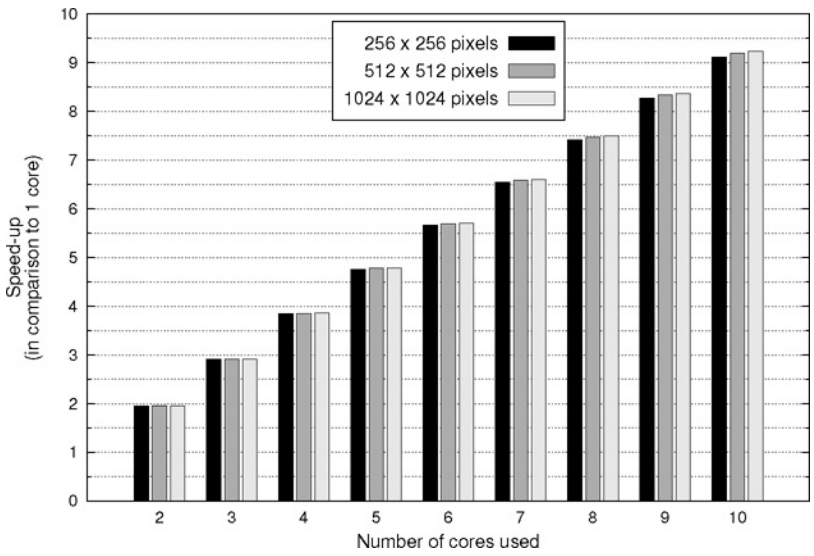

Figure 3: Speed-up of the whole preprocessing algorithm for different resolutions

The speed-up for calculating the maximum of each pixel's correlogram, in comparison to one core, is numerically equivalent to the number of cores used and as a consequence constant. As opposed to this, the speed-up for the computation of the intensity interval around the maximum of each pixel's interferogram is numerically not equal to the number of cores used, but constant, too. For that reason, the speed-up of the whole algorithm and the Contrast Method, respectively, is not numerically equivalent to the number of cores used. Despite this, the speed-up of the whole algorithm is linear. This results from the distributed shared memory architecture. Each core operates on the data in its own local memory and thus independent of the other cores. As a consequence, there are no

simultaneous memory accesses, that could slow down the application, even in practice. Only simultaneous access to the shared synchronization variables could slow down the application. This is the case when the master polls for the synchronization variable of a slave (read), while the slave wants to signal that he has finished its computations (write).

In multiprocessor simulations, and as mentioned in Section 3, each core advances a certain number of instructions in turn. This number of instructions (the quantum) can be set very low to increase simulation accuracy and to avoid simulation artifacts. As a consequence, setting the quantum very low has a significant impact on simulation performance. We investigated this circumstance for a resolution of 256×256 pixels and for quanta of 1, 100 and 100000. The results have shown, that the *simulated time* of the whole system, i.e. the time the simulator determines for the execution in the corresponding real system, is not influenced by the quantum and amounts to 55.67 seconds for a system consisting of two cores. As opposed to this, the *wall-clock time* changes significantly. For the simulation of two cores and a quantum of 1, the wall-clock time amounts to 4 hours and 27 minutes. Setting the quantum to 100 evokes a wall-clock time of 33 minutes and 36 seconds. The fastest simulation is achievable when setting the quantum to 100000. The wall-clock time than amounts to 42 seconds only. When investigating the instruction counts for the whole preprocessing algorithm, an amendment depending on the quantum is ascertainable, too. For a quantum of 1 the instruction count of the whole preprocessing algorithm results in 558,969,518 instructions, for a quantum of 100 in 558,969,742 instructions and for a quantum of 100000 in 559,090,510 instructions. When dividing that instruction counts by the mips rate, which was set to 100, preprocessing times of 5.58970 seconds for a quantum of 1, 5.58970 seconds for a quantum of 100 and 5.59091 seconds for a quantum of 100000 are emerging. Thus, the simulation accuracy changes at the second decimal place and at the tenth millisecond place, respectively. This circumstance was noticed independent of the number of cores used. Because the results of our measurements for the whole preprocessing algorithm are in a range of seconds (see Table 1), a quantum of 100000 is rather adequate to provide enough accuracy for our simulations. As a consequence, the wall-clock time of the simulation is reduced heavily.

7 Conclusion and future work

The speed-up for the parallelized preprocessing algorithm, called Contrast Method, rises sharply the more cores are used. Furthermore, the speed-up is approximately independent of the chosen resolution for a constant number of frames to evaluate. When more than ten cores are used, we expect the speed-up to remain linear until the saturation is reached, also independent of the resolution. In this context, saturation is reached, when no more notable speed-up can be achieved by raising the number of cores. This depends on the number of frames to evaluate and on the chosen resolution. In comparison to the times that are achievable on a GPU (range of milliseconds for a resolution of 1024×1024 pixels), the times we achieved (see Table 1) are slower, but sufficient enough for the timings required for an industrial inspection process. Concerning to power consumption we can clearly make the statement that an embedded solution will reduce power consumption in compar-

ison to a GPU or a conventional multi-core architecture (see Section 1). As a result, the full advantages, regarding to compactness and less power consumption can be occupied. The advantage of a virtual environment is the possibility to simulate any desired number of cores, even if there is no corresponding real hardware. Our final goal to integrate such a compact embedded system in a smart thrilling head is therefore achievable.

In future work we will extend our approach by parallelization of the remaining steps from the white light interferometry data analysis process, which includes the analysis of each interferograms phase information [His05] and the postprocessing step, where a Gauss-Newton fitting is applied to each correlogram of a pixel. Furthermore, we will investigate the difference between the measured times and instruction counts, respectively, when using other available virtual cores than the CortexA9. It would be very interesting to see what total power consumption appears on the simulated hardware. OVP provides statistics of power consumption for the simulated hardware not directly. However, with some software effort it is possible to get these statistics. This is another point we will treat in future. Last but not least, one important thing is the comparison of the times figured out with OVP and the times appearing on real hardware. Therefore, we want to recreate the virtual hardware design, introduced in this paper, with one to four cores in real. Afterwards, we will port our software to the real hardware and compare the times we determined with OVP to the times arising on the real hardware.

References

[ARM11] http://www.arm.com/products/processors/cortex-a/cortex-a9.php?tab=Performance, 2011. Last visit on 14.12.2011.

[COMS11] M. Conti, S. Orcioni, N. Martinez Madrid, and R. E.D. Seepold. *Solutions on Embedded Systems*. Springer Dordrecht Heidelberg London New York, 2011. doi: 10.1007/978-94-007-0638-5.

[Cor10] NVIDIA Corporation. *Datasheet TESLA C2050/C2070 GPU computing processor*, July 2010.

[GJMM10] F. Gao, X. Jiang, H. Muhamedsalih, and H. Martin. Wavelength scanning interferometry for thin film analysis of fusion target. In *3rd European Target Fabrication Workshop, Science & Technology Facilities Council*, 2010.

[His05] M. Hissmann. *Bayesian estimation for white light interferometry*. PhD thesis, Combined Faculties for the Natural Sciences and for Mathematics of the Ruperto-Carola University of Heidelberg, 2005.

[Int11] http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-(12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI), 2011. Last visit on 14.12.2011.

[KM07] D. Kapusi and T. Machleidt. White Light Interferometry in Combination with a Nanopositioning- and Nanomeasuring Machine (NPMM). In *Proceedings of the International Society for Optical Engineering*, 2007.

[Lar00] K. G. Larkin. *Topics in multi-dimensional signal demodulation*. PhD thesis, The Faculty of Science in the University of Sydney, 2000.

[Lim11a] Imperas Software Limited. *OVP Guide to Using Processor Models*. Imperas Buildings, North Weston, Thame, Oxfordshire, OX9 2HA, UK, July 2011. Version 0.4, docs@imperas.com.

[Lim11b] Imperas Software Limited. *OVPsim and Imperas CpuManager User Guide*. Imperas Buildings, North Weston, Thame, Oxfordshire, OX9 2HA, UK, September 2011. Version 2.0.40, docs@imperas.com.

[OVP11] www.ovpworld.org, 2011. Last visit on 10.11.2011.

[PMS+04] A. Purde, A. Meixner, H. Schweizer, T. Zeh, and A. Koch. Pixel shader based real-time image processing for surface metrology. In *Instrumentation and Measurement Technology Conference, 2004. IMTC 04. Proceedings of the 21st IEEE*, volume 2, pages 1116 – 1119 Vol.2, 2004. doi: 10.1109/IMTC.2004.1351259.

[Rob93] D. W. Robinson. *Interferogram Analysis: Digital Fringe Pattern Measurement Techniques*. Institute of Physics, Bristol, Philadelphia, 1993.

[SFKM11] Max Schneider, Dietmar Fey, Daniel Kapusi, and Torsten Machleidt. Performance comparison of designated preprocessing white light interferometry algorithms on emerging multi- and many-core architectures. In *Procedia CS*, volume 4, pages 2037–2046, 2011.

[SSST10] M. Sylwestrzak, M. Szkulmowski, D. Szlag, and P. Targowski. Real-time imaging for spectral optical coherence tomography with massively parallel data processing. In *Photonics Letters of Poland*, volume 2, 2010.