59th ILMENAU SCIENTIFIC COLLOQUIUM
Technische Universität Ilmenau, 11 – 15 September 2017
URN: urn:nbn:de:gbv:ilm1-2017iwk-013:8

USING FPGA BLOCK-RAM FOR FAST WHITE LIGHT INTERFEROMETRY

T. Scholz, M. Rosenberger and G. Notni

Research fellow, Group for Quality Assurance and Industrial Image Processing,
TU-Ilmenau, Newtonbau, Gustav-Kirchhoff-Platz 2, 98693 Ilmenau

ABSTRACT

White light interferometry is a time consuming operation even on modern architectures. To overcome the high power consumption and size of traditional desktop computers an embedded approach containing the hybrid architecture Zynq will be presented. This architecture contains a dual core ARM and programmable logic provided by an FPGA. FPGAs offer massively parallel logic gates and DSP-slices to parallelise certain tasks. Another important part is the internal memory BRAM. The presented approach aims to speedup calculation time of the ARM processor by utilization of this BRAM. It is well known that memory transfers consume a lot of time. To speed the transfers up, the bottlenecks have to be identified. In this paper it will be illustrated how to easily access an FPGA BRAM from a running operating system and the possible speedup will be analysed and estimated.

Index Terms - Zynq, FPGA, white light interferometry, BRAM, Xilinx, Linux

1. INTRODUCTION

White light interferometry is a key technology in measurement for surface reconstruction. It is a very precise method with accuracy in nanometre range. A stack of images is recorded and interpreted pixel-wise. By the use of piezo positioning systems and scanning characteristic interferometry fringes are visible as shown in Figure 1.

A huge amount of data must be processed. Depending on the measurement range the image acquisition takes up to minutes, which can lead to problems because the environmental conditions (e.g. temperature, vibration) for the unit under test are likely to change in that time. In the second stage this stored data must be reconstructed into a 3d surface. Every pixel is independent from their neighbours, which allows for a high degree of parallelisation.

Today, this family of tasks is solved by using massive parallel processors. Typical representatives of that are computer clusters or general purpose programmable graphics processing units (GPUs). Another part of this family are field-programmable gate arrays (FPGAs). Combined with general purpose processors of the ARM-family, they offer an easy to use programming architecture, using high-level languages and well-known operating systems.

On the other hand, this requires additional effort for the implementation due to the shared use of RAM. Typically, the RAM in embedded systems is smaller compared to desktop computers or consumer-class GPUs. Physically near distant memory is a lot faster but lacking in size. Common examples are cache in CPUs or block-ram (BRAM) in FPGAs. While faster memory is preferable for computation speed, algorithms tend to require more than what is available.

These challenges must be faced when using any kind of processing unit for white light interferometry. While the processing system is easy to program the programmable logic needs a much higher implementation time. To avoid bottlenecks, it is necessary to source out the heavy load operations to the PL and also keep the time for memory operations in mind.

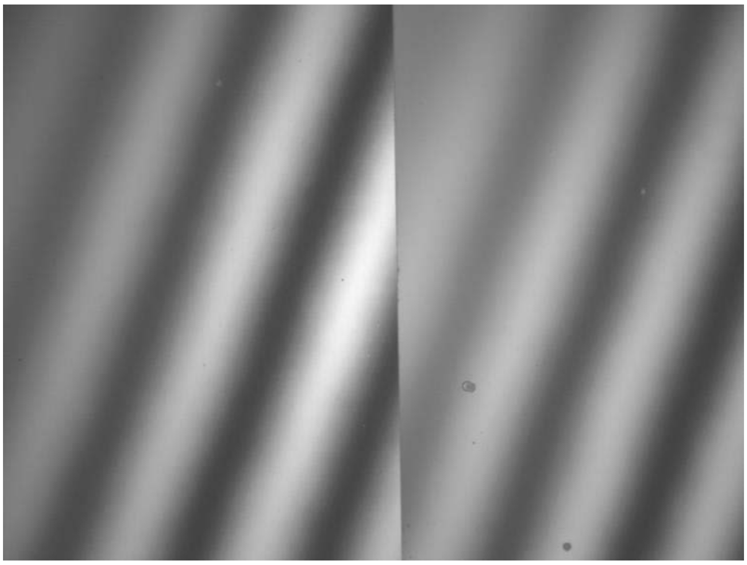

Figure 1 characteristic white light fringes of a micro step surface acquired by 8 bit grey value camera

2. WHITE LIGHT INTERFEROMETRY SYSTEM WORKING CONDITIONS

In practical use there are several reconstruction algorithms for accurate white light interferometry. Most of them rely in some way on Fourier transform implementations like FFT. Even if it is possible to calculate the algorithm over the complete raw values it is more straight-forward to use a fast algorithm preselecting the area of interest and use an accurate algorithm only there.

In Figure 2 the grey values for a single pixel acquired by scanning different positions by the use of a piezo positioning system are shown. It is easy to see the position of the maximum which is located near the real maximum of the white light fringe. Nevertheless, finding this maximum is a time consuming operation, since the memory in modern computers is too limited. This is even more emphasised in the embedded world. The data acquired by an image sensor or read from a mass storage are images. To process data it is advisable to reorder the raw data in a format suiting the needs of the algorithms. Doing so in general purpose processors is easy to implement but slow. Every operation has to be read in the processor cache and written back to the new position in the memory. This can be done in a multithreaded environment and relates directly to the size of the processor cache as well as the number of processors. During this time it is not wise to use the processor for other operations to not interfere the reorganization process. This leads directly to the idea of outsourcing the memory reorganization process to a different architecture, preferably to a massively parallel one – in this case FPGA-based programmable logic.

Profiling is a very important part of this task. Optimizing this class of memory operations is highly platform specific and needs exact knowledge of the throughput and bandwidth. At first it is necessary to compare the time to find the maximum directly. This must be done for single pixels and for a several ones to find the bottlenecks and the maximum speed to reach. This will be compared in the two use cases: sorted data and raw values. Additionally it is necessary

to know the user space bandwidth of the memory operations. Therefor the copy from memory to memory needs to be clarified.
As a first step to reach a speedup, internal FPGA BRAM has to be made accessible to read and write. To achieve this in the programmable logic the memory must be initialized and made accessible from the user space of the processing system. Afterwards reading and writing between the two levels have to be measured in the aspects bandwidth and throughput.
As a last step it is necessary to synthesise the programmable logic and upload the bitstream to use the FPGA to optimize the overall performance. Both work independently in different clock domains. For that reason some kind of access control needs to be used for a productive environment. Otherwise unreproducible errors may appear as the memory is not exclusively accessed by either side. In the processing system this is usually covered by the use of mutual exclusion (MUTEX) or semaphores. These can be used in Vivado too, but add some overhead and need to be implemented carefully weighing the pros and cons.

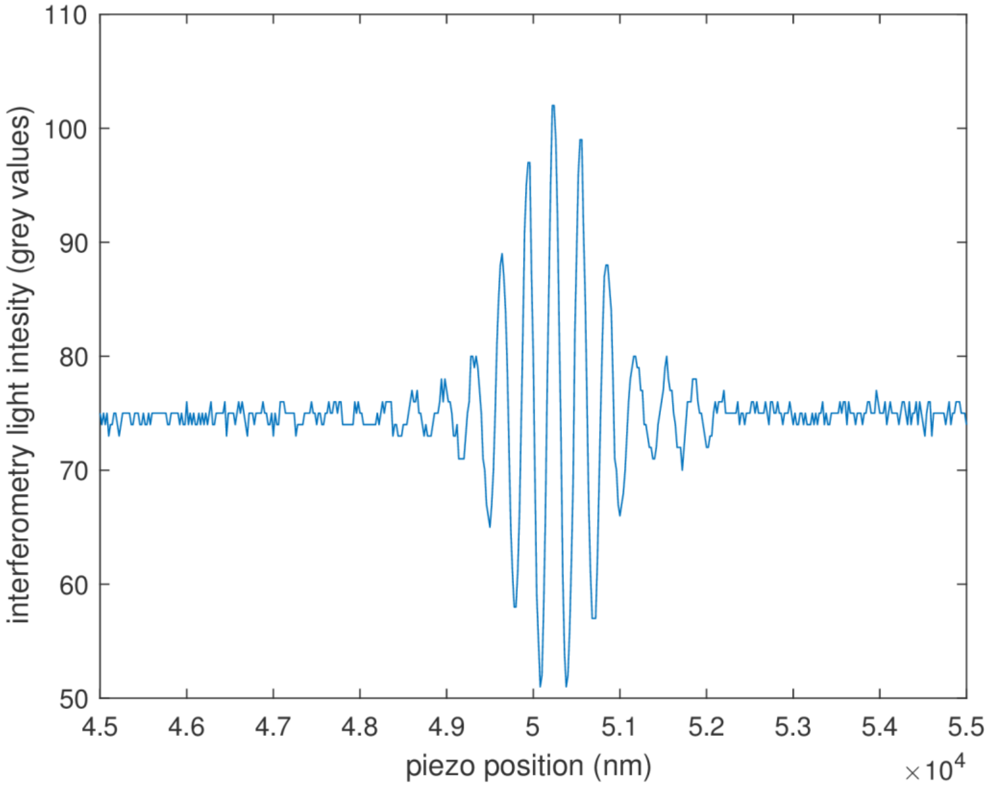

Figure 2 raw values for a single pixel

2.1 Summary

In the Zynq architecture the FPGA-based programmable logic is independent from the ARM processing. Additionally the FPGA is a massive parallel system which can lead to heavy speedups in concurrent algorithms. In white light interferometry the pixels are independent from their neighbours so this method is concurrent by design, which makes FPGAs especially suited. While there are other massively parallel architectures, FPGAs are arguably more versatile and at a lower power consumption. From this point of view it is a logical step to investigate the abilities offered by this architecture in addition to conventional easy to program systems to support their abilities.

3. COMPUTING SYSTEM SETUP

The primary development is done on a Zynq-7000 device of the vendor Xilinx. This is a hybrid architecture with dual core ARM Cortex-A9 and programmable logic. Detailed specifications are given in Table 1. As reference architecture for speed comparisons serves an i7-6700k [1].

Table 1 Z-turn specification

CPU frequency	L1 cache	L2 cache	RAM	#36 Kb BRAM	Programmable logic cells
2x667 MHz	2x32 KB	512 KB	2x512 MB, 32 bit	140 (4.9 Mb)	85K

[2], [3]

To ensure the fastest, most versatile and comparable software possible, a Gentoo Linux drives both ARM and x86 systems. The applied compiler optimizations can be found in Table 2. CFLAGS and CPU_FLAGS are used to compile the whole operating system while the CXXFLAGS are used in a “Makefile” to compile the test programs. Additionally the library “pthread” was linked to add multithreading abilities.

Table 2 compiler optimizations used for the architectures

Architecture	CFLAGS	CPU_FLAGS	CXXFLAGS	Compiler
x86_64	-O2 -pipe -march=broadwell	aes avx fma3 mmx mmxext popcnt sse sse2 sse3 sse4_1 sse4_2 ssse3	-std=c++14 -O2	g++-5.4.0-r3
ARM	-O2 -march=armv7-a -mtune=cortex-a9 -mfpu=vfpv3-d16 -mfloat-abi=hard -pipe -fomit-frame-pointer		-std=c++14 -O2	g++-5.4.0-r3

The first stage boot loader was created using the Xilinx tools Vivado and SDK in the version 2016.4. Same applies to the creation of the bit file loaded to initialize the programmable logic. Currently only a single clock domain is used to access the BRAM from the processing system and also to reorder the raw data. For stability purpose this value is set to 50 MHz.

The above-mentioned compiler options are fundamental. Modern processing units offer great optimisation potential. Most efficient is the use of single instruction multiple data (SIMD) extensions like Neon or SSE. While this offers a lot of potential, it also can lead to code not usable in cross-platform projects because they heavily depend on the architecture. Others can be used with less effort by instructing the compiler to run optimisations. Using the gcc as compiler the -O flag is an important optimisation option. The higher the number the more effort is put in the optimisation. From experience a too aggressive option can induce instability and for some operations even slower executables. For that reason the stable but fast optimization flag -O2 was used. This procedure is important cause modern compilers offer huge improvements to optimize high level language code. Especially in heavy load situations to help the branch predictor in the processing unit to run the executable.

4. IMPLEMENTATION

In order to be able to communicate between the processing system and the programmable logic the new Xilinx design tool Vivado helps a lot for the Zynq architecture. Necessary parts are the IP blocks *Zynq Processing System*, *BRAM Controller* and *Block Memory Generator*. The auto configuration scripts provided by the vendor simplify the expenditure to create a synthesizable design to read and write from and into BRAM. Additionally in this use-case a custom block was created with the ability to access the BRAM.

The physical addresses of each BRAM can be configured in the Vivado address editor. It is important to set the addresses in a range not already covered by the RAM. Also overlapping addresses have to be avoided.

Linux is a versatile and powerful operating system for embedded systems. The majority of distributions offer numerous programs and libraries to allow fast development in this area. For this purpose a C compiler and the Linux headers allow data exchange between both systems. To reach this the system call mmap is used. Originally this was implemented in 4.2 BSD in the year 1983 [4]. For this implementation the Linux version was used [5]. To make use of the more versatile and easy to use changes in modern C++ a slightly modified version to access this call was created.

This is used as a basis to benchmark the throughput between the programmable logic and the processing system in user space running the Linux operating system. To ensure comparability much more data is written than the BRAM can handle at a time.

5. RESULTS

The self-written VHDL IP reads the data from a BRAM and writes to other BRAMs in the right order. The read and write commands have a latency of 1 clock cycle. To reorder a single pixel already existent in BRAM to another one needs 501 clock cycles in the case of 500 raw pictures. This can be optimized by reading 4 pixels at the same time and save them in 4 different BRAMs and change the address simultaneously. Expecting 1024x1024 pixel images this approach needs 500 KB to store all values for the first line and nearly the same as temporary data cache. To simplify it only reading half the line at a time leads to an expected copy time of 2048x501x128 clock cycles. Using 50 MHz to drive the AXI clock the FPGA time to read and write in the reorganised memory needs round about 2.6 s. In addition there is an overhead to synchronize the start and stop signals as well as the time to copy the data between the processing system and the programmable logic. This was done in a pipeline.

Table 3 measured time to get the maximum per pixel

	find maximum unsorted	sort	find maximum sorted	overall time	BRAM operations	time using BRAM	speedup
Zynq	211 s	195 s	9.4 s	204.4 s	10.7 s	20.1 s	9.7
i7	1.9 s	0.8 s	0.05 s	0.85 s			2.2

Using the processing system on the Zynq to find the maximum per pixel is a demanding operation. There is much difference between the sorted raw data and the not sorted raw data. Nevertheless the time to sort the pixel data in the right order is time and memory consuming. For further operations this is a good approach but to find the maximum it is an overrated approach. Copying and reading the results using the internal FPGA memory is much faster and the time to order is orders faster. So using the BRAM to sort the memory leads to a speedup factor of nearly 10.

Compared to the above-mentioned reference system the overall time to sort and find the maximum is much higher. The most demanding operation are the read and writes between both architectures. The measured times for the different processors and the detailed operations are shown in Table 3.

6. CONCLUSIONS AND FUTURE WORK

FPGAs are versatile and fast devices to solve multithreaded tasks. While there are other architectures like CPU clusters or GPUs, this kind of hardware is superior in power consumption. They are powerful and versatile architectures as a coprocessor to accelerate demanding tasks and also standalone in real-time situations. To improve the overall calculation time needed for white light interferometry they offer great potential. Disadvantageous is the expenditure of time to implement the algorithms in the programmable logic. As expected the data operations to read and write between processing system and programmable logic have a huge impact.

To lower this, more logic to preprocess the data has to be implemented in future work. Another big improvement could be to switch from user space to kernel space and use direct memory access. By this approach the data should be more independent from the cache of the processor.

7. ACKNOWLEDGEMENT

The approaches presented in this paper are sponsored by the federal ministry of education and research (BMBF, FKZ: 03ZZ0427G).

SPONSORED BY THE

Federal Ministry
of Education
and Research

REFERENCES

[1] Intel, "Core i7-6700K Processor," Intel, [Online]. Available: https://ark.intel.com/products/88195/Intel-Core-i7-6700K-Processor-8M-Cache-up-to-4_20-GHz. [Accessed 11 07 2017].

[2] Xilinx, "Xilinx Support Data Sheets," 07 06 2017. [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf. [Accessed 13 07 2017].

[3] M. T. Limited, "Z-Turn Board Overview," 12 02 2014. [Online]. Available: http://www.myirtech.com/download/Zynq7000/Z-turnBoard.pdf. [Accessed 14 07 2017].

[4] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick and D. Mosher, "4.2 BSD System Manual," 07 1983. [Online]. Available: https://cdn.preterhuman.net/texts/manuals/4.2BSD%20UNIX%20System%20Manual.pdf. [Accessed 11 07 2017].

[5] M. Kerrisk, "Linux Programmer's Manual," 03 05 2017. [Online]. Available: http://man7.org/linux/man-pages/man2/mmap.2.html. [Accessed 11 07 2017].

CONTACTS

T. Scholz	tobias.scholz@tu-ilmenau.de
Dr.-Ing. M. Rosenberger	maik.rosenberger@tu-ilmenau.de
Univ.-Prof. Dr. rer. nat. G. Notni	gunther.notni@tu-ilmenau.de