embeddedworld2018
Exhibition&Conference
... it's a smarter world

Partitioning of computationally intensive tasks between FPGA and CPUs

Tobias Welti, MSc (*Author*)
Institute of Embedded Systems
Zurich University of Applied Sciences
Winterthur, Switzerland
tobias.welti@zhaw.ch

Matthias Rosenthal, PhD (*Author*)
Institute of Embedded Systems
Zurich University of Applied Sciences
Winterthur, Switzerland
matthias.rosenthal@zhaw.ch

***Abstract*—With the recent development of faster and more complex Multiprocessor System-on-Chips (MPSoCs), a large number of different resources have become available on a single chip. For example, Xilinx's Zynq UltraScale+ is a powerful MPSoC with four ARM Cortex-A53 CPUs, two Cortex-R5 real-time cores, an FPGA fabric and a Mali-400 GPU. Optimal process partitioning between CPUs, real-time cores, GPU and FPGA is therefore a challenge.**

For many scientific applications with high sampling rates and real-time signal analysis, an FFT needs to be calculated and analyzed directly in the measuring device. The goal of partitioning such an FFT in an MPSoC is to make best use of the available resources, to minimize latency and to optimize performance. The paper compares different partitioning designs and discusses their advantages and disadvantages. Measurement results with up to 250 MSamples per second are shown.

Keywords—FPGA; UltraScale+ MPSoC; partitioning; ARM NEON; SIMD; asymmetric multi-processing; high performance FFT; low latency processing

I. INTRODUCTION

The transition from field-programmable gate arrays (FPGAs) to System-on-Chips (SoCs) in 2011 was the unavoidable development when FPGAs needed to execute ever more complex software programs. The soft-core processors available for inclusion in the programmable logic were either not powerful enough or took up too many logic resources. The combination of hardware processors with the FPGA, interconnected through a high-performance bus showed the potential of this architecture.

With the recent development of faster and more complex Multiprocessor System-on-Chips (MPSoCs), many different resources are available on one chip. For example, Xilinx's Zynq UltraScale+ MPSoC combines up to four ARM Cortex-A53 application processor cores, two ARM Cortex-R5 real-time cores, an ARM Mali-400 GPU as well as an FPGA fabric with programmable logic, on-chip memory, hardware multipliers (DSP slices) and many high-throughput I/Os. The challenge for the system architect has now become finding the optimal execution environment for your design's processes: the partitioning. The goal is to make best use of the available resources, minimizing latency and optimizing performance.

In this paper, we use the Fast Fourier Transform as a computationally expensive algorithm that can be accelerated through several means:

- multiprocessing on several cores of the same type (Symmetric Multiprocessing)
- vector processing using a special instruction set for Single Instruction, Multiple Data (SIMD), available on most current processors
- using additional, different processors than the ones the main software is run on (Asymmetric Multiprocessing)
- generating accelerator functions that run in the FPGA fabric and using them as external functions
- running the whole algorithm in the FPGA core, controlled by a CPU core
- running the algorithm standalone in the FPGA

For each method, we present the communication paths and software architecture, along with performance data.

The FFT is a well-studied algorithm and many papers have been published on methods for efficient execution on specific multiprocessor architectures in [1], [2], [3], [4] and [5]. It is not the goal of this paper to improve on these methods, but to provide an overview and an understanding of the possibilities available in today's devices.

The paper is organized as follows:

Section II introduces the FFT algorithm and how it can be calculated on multiple processing devices. In Section III, we discuss the partitioning methods based on software, executing on processor cores. The FPGA-based methods are explored in Section IV. Section V elaborates on ways of collaboration between the FPGA and processors. Finally, in Section VI we sum up the advantages of the presented methods.

II. FFT PARTITIONING

The discrete Fourier Transform (DFT) is used to transform a sequence of samples from the time domain into the frequency domain to analyze the frequency components of the sampled signal. Spectral analysis, measuring and controlling, signal processing and quantum computing are but a few applications of the DFT. The DFT has a very high computational cost of $O(N^2)$. The Fast Fourier Transform (FFT) improves efficiency of the transform by reducing the number of redundant calculations. This is achieved by splitting the sequence into smaller parts and performing the Fourier Transform on these as shown in Fig. 1. In doing so, the computational cost can be reduced to $O(N \log N)$. Note that the splitting includes a reordering of the input values, effectively selecting every other input value for each subset.

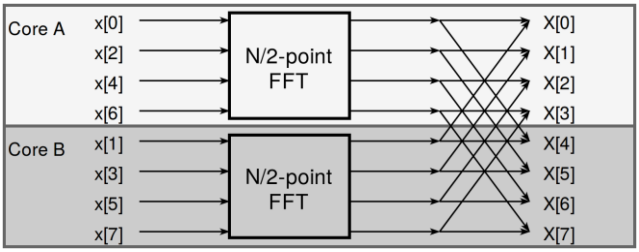

Fig. 1. Principle of the FFT algorithm.

Since the FFT algorithm is a divide-and-conquer approach, it is well suited for parallel processing on multiple processors. Each core can perform the smaller FFT on its part of the data, independent of the remaining data. However, as shown in Fig. 1, there will be at least one step requiring data from other cores when combining the smaller FFTs into the complete spectrum. This all-to-all communication is a critical step because it requires synchronization of the cores. The optimization of this step has been the subject of several publications, i.e. [2] and [4].

The technique of calculating smaller FFTs and combining them into larger spectra allows to efficiently process FFTs larger than the memory of your processor, given that the currently unused data is stored in an efficient way. Refs. [2], [4] and [6] have published possible implementations.

III. MULTICORE PROCESSING FOR FFT

A. Available resources

The Xilinx Zynq UltraScale+ MPSoC portfolio offers multiple ranges of SoCs with varying numbers of processor cores and FPGA fabric resources. In this paper, we use the XCZU9EG device, an SoC with the following resources:

- four ARM Cortex-A53 application processing cores, running at 1100 MHz and featuring the NEON instruction set
- two ARM Cortex-R5 real-time processing cores with tightly-coupled memory (TCM) for low-latency access, running at 500 MHz
- an ARM Mali-400 GPU
- an FPGA fabric with 600'000 System Logic Cells, 32 Mbit of FPGA memory and 2520 hardware multipliers (DSP slices)

Fig. 2 shows the block diagram of the available resources.

The ARM Cortex-A53 core is a mid-range application processing core that balances power usage vs. performance. It is equipped with the ARMv8 instruction set, including NEONv2 SIMD instructions for vectorized execution on multiple data (up to 128 bit wide). Four A53 cores make up the Application Processing Unit (APU), in the bottom right of Fig. 2. The ARM Cortex-R5 core is a real-time processor with a focus on fast reaction to events. Its 128 kB of TCM allow very fast memory accesses, but in turn limit the amount of data that can be worked on. Two R5 cores form the Real-time Processing Unit (RPU) in the top right of Fig. 2. The Level 3 interconnect enables fast data transfers between the APU, the RPU and the FPGA fabric with on-chip memory, DSP slices and programmable logic.

B. Executing the FFT in Software

When executing an FFT in software, you have the choice of several FFT libraries, many of them capable of exploiting both multiprocessing and vector processing.

ARM Ne10 [7] provides highly optimized ARM NEON intrinsics written in Assembler and its FFT algorithm makes use of these. However, it does not support multiprocessing. kissFFT [8] is a very lightweight library with the goal of being easy to use and moderately efficient while supporting multiprocessing. However, it makes use of NEON instructions only to execute four separate FFTs in parallel instead of accelerating one FFT transform.

The fastest and most versatile FFT library that was tested in our work is FFTW3 [9], exploiting both multi-processing and NEON instructions and including a mechanism to optimize the algorithm for the available hardware. This mechanism will test many possible FFT optimization algorithms, measuring performance and selecting the fastest one as described in [10]. This is done in order to make the best possible use of first and second level caches, memory access speeds and other hardware characteristics. For our implementations, we used FFTW3 on the A53 and Ne10 on the R5.

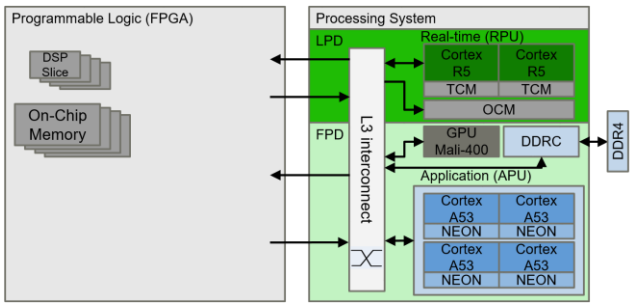

Fig. 2. Block diagram of MPSoC.

C. Implementations

The software-only implementations were run on the Xilinx PetaLinux operating system, using the FFTW3 library to calculate double precision floating-point complex FFTs. Using double precision limits the speed improvement for the NEON instruction set to a factor of two, because the NEON registers are 128 bit wide and can therefore accommodate only two double precision floating-point values.

The following five scenarios were tested:

a. Single-core A53

b. Single-core A53 with NEON instructions

c. Symmetric Multi-Processing (SMP) with four A53

d. Symmetric Multi-Processing (SMP) with four A53 with NEON instructions

e. Asymmetric Multi-Processing (AMP) with the R5 as coprocessor

Scenarios *a-d* require no special software stack except the pthread library for SMP.

Scenario *e* requires additional frameworks and drivers for communication between the Master A53 core and the remote R5 core to enable AMP. Fig. 3 shows the software architecture. Two operating systems are required: Linux on the APU master CPU and FreeRTOS on the RPU slave CPU. First, the APU boots Linux and uses the OpenAMP framework to load the RPU firmware into the TCM via a DMA transfer. The RPU is then booted out of the TCM. The remoteproc driver handles the life cycle management, allocates the required resources and creates a virtual I/O (virtIO) device for each remote processor. RPMsg is the Remote Processor Messaging API to provide inter-process communication between processes on independent cores.

The flow of OpenAMP booting and software execution is as follows (as in [11]):

1. The remote processor is configured as a virtual IO device, shared memory for message passing is reserved.
2. The Master loads the firmware for the remote processor into its memory, then boots the remote processor.
3. After booting, the remote processor creates the virtIO and RPMsg channels and informs the master.
4. The Master invokes the callback channel and acknowledges the remote processor and application.
5. The remote processor invokes the RPMsg channel.
6. The RPMsg channel is established, both Master and Slave can initiate communication via RPMsg calls.
7. During operation, communication buffers in reserved shared DDR memory are used to pass messages between the Master and the Slave. Usually, these buffers are small. To load larger amounts of data, such as the FFT input and output data, the data is written to or read from on-chip memory (OCM) of the R5, and the pointers are passed via message buffer.

Shutdown proceeds in the reverse sequence of the booting and initialization process.

D. Performance

To provide an overview of the performance, FFTs of three sizes (4’096, 16’384 and 65’536 data points) were calculated using the implementation scenarios *a-d*. The Cortex-R5 can only perform 4’096 point FFTs due to its limited amount of OCM

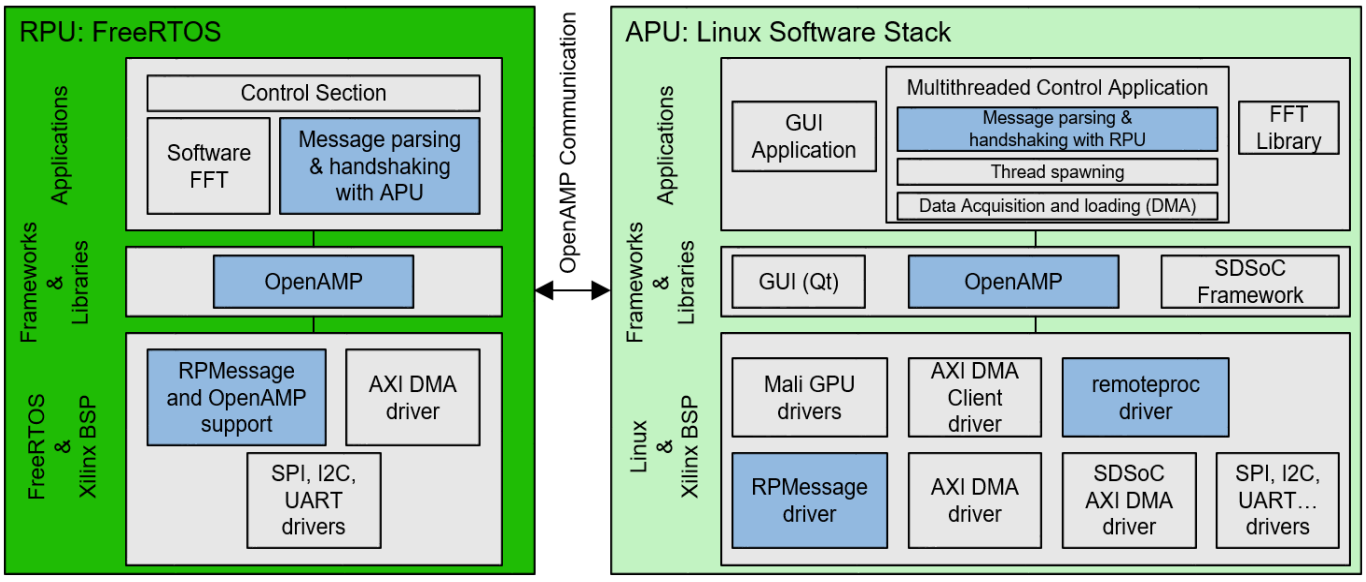

Fig. 3. Software stack for Asymmetric Multi-Processing (as in [11])

Table I shows the achieved calculation times in microseconds, comprised of the times for loading the input data, executing the FFT and storing the result in memory. We also show the feasible sampling rates that would allow the CPUs to keep up a seamless processing of the input data.

TABLE I. SOFTWARE FFT PERFORMANCE

Scenario		4'096		16'384		65'536	
		time (µs)	*max. rate (MSa/s)*	*time (µs)*	*max. rate (MSa/s)*	*time (µs)*	*max. rate (MSa/s)*
a	A53	320	12	1600	10	8670	7
b	A53NEON	290	14	1460	11	7760	8
c	4xA53	120	33	511	32	2770	23
d	4xA53NEON	114	35	434	37	2290	28
e	R5[a]	1455	3	--	--	--	--

[a] R5 time includes OpenAMP communication overhead (approx.100 µs)

It is evident from the data that the FFT scales well for multiprocessing. When using four A53 cores, a speed-up of up to factor 3.3 is observed. Scaling is better for large FFTs, because there are more calculation steps that don't require all-to-all communication.

More detailed tests have been run with one to four cores, but for the sake of brevity, the data is not shown here. In summary, the speedup corresponds almost directly to the number of cores as long as there remains at least one core for execution of the processes of the operating system. When all cores are used for multi-processing, the speedup is capped. A reasonable explanation is that the FFT is competing with the other processes, resulting in many context switches.

Using the NEON instruction set, a speedup of roughly 10% is observed for single-processing. For multi-processing, enabling NEON yields a speed gain of 5-20%. This is nowhere near a factor of two that could be expected since two values can be processed at the same time. Among the possible reasons, we suspect the fact that the NEON instructions have their own execution pipeline and registers. If the algorithm can not be optimized to 100% NEON instructions, there will be data transfers between NEON and standard registers.

The R5 is clearly not designed for computationally heavy tasks, its target assignment is to react to events in real-time. It has to be noted that the communication overhead of the OpenAMP framework contributes approximately 100 µs to the execution time, but even if this overhead could be avoided, the R5 would be no competition for the A53s.

IV. ACCELERATORS IN FPGA

Traditionally, bringing an algorithm to programmable logic means writing HDL code or using an existing IP core. Today, there are tools to generate HDL code from software code. Xilinx provides the SDSoC (Software Defined System on Chip) toolchain that generates logic blocks from your C-code along with the required data transfer logic. To allow your software to interface with this computation block, SDSoC compiles a software library with the required function for configuring the FFT core, loading and storing data as well as the necessary interrupt service functions.

We have compared the performance of the following scenarios:

f. SDSoC-Accelerator controlled by A53

g. FFT IP-core controlled by A53

h. FFT IP-core working standalone

Scenario f: An SDSoC accelerator core can only be implemented for a fixed FFT size. Therefore, three accelerators were implemented in the programmable logic, clocked at 300 MHz. The big advantage of performing the FFT in programmable logic is that the processor core can perform other tasks in the meantime. The processor will still be required for loading and storing the data. Fig. 4 shows the block diagram of this setup.

Scenario g: The Xilinx FFT IP-core can be configured for different FFT sizes at runtime. One instance of the FFT core is therefore sufficient for our tests. The processor will load the input data into on-chip memory in the FPGA fabric, then start the FFT core and finally transfer the processed data back to DDR memory. These transfers can be done by DMA, leaving the CPU free for other tasks. This setup is shown in Fig. 5.

Scenario h: If the input data is acquired in the FPGA fabric, there is no sense in transferring the data to DDR memory first, then loading it to FPGA on-chip memory for the FFT. Instead, the FFT core is configured for constant, standalone operation on the input data stream and a DMA stream is set up for transfer of the output data to a range of reserved DDR memory, where the APU can retrieve the processed data for analysis (See Fig. 6).

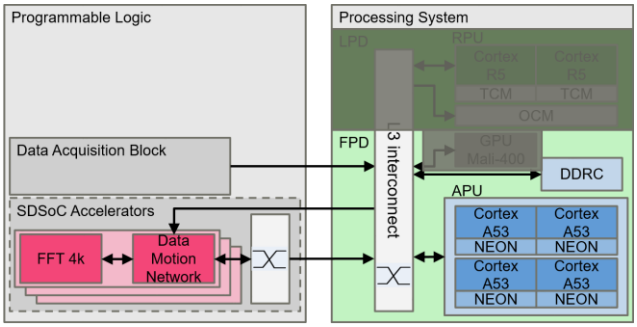

Fig. 4. SDSoC accelerators in FPGA.

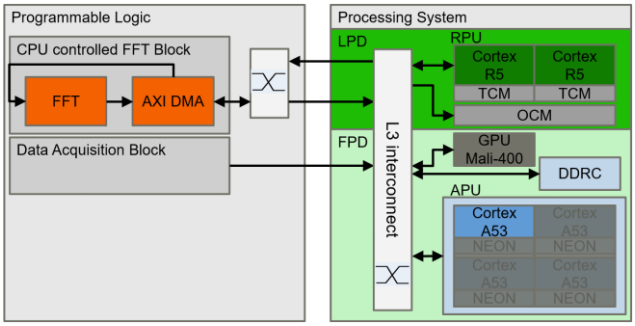

Fig. 5. FFT IP block, controlled by A53

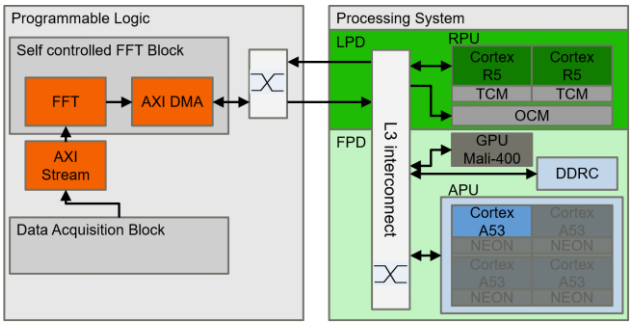

Fig. 6. FFT IP block, self-controlled

Table II shows the achieved execution times for FPGA-accelerated FFT on the Zynq UltraScale+ MPSoC. Scenarios *f* and *g* have similar performance, showing that the SDSoC-generated HDL code is efficient and compares well to manually optimized HDL code of the FFT IP.

TABLE II. FPGA FFT PERFORMANCE

Scenario		4'096		16'384		65'536	
		time (µs)	*max. rate (MSa/s)*	*time (µs)*		*time (µs)*	*max. rate (MSa/s)*
f	SDSoC	108	38	410	38	1720	38
g	IP-A53	101	41	400	41	1680	39
h	IP-standalone	51	250	202	250	807	250

The more efficient data path in scenario *h* can easily explain the difference in performance between scenarios *g* and *h*, omitting the transfer of the input data from DDR to on-chip memory. In fact, the limiting factor for the sampling rate is the DMA transfer rate from FPGA fabric into DDR memory.

V. COMBINING FPGA AND PROCESSING SYSTEM

The results in Sections III and IV show clearly that the FPGA easily outperforms the pure software implementations. Nevertheless, there are limits to the size of FFT than can be executed in the FPGA. The FFT IP core can process up to 65'536 points for one FFT. The SDSoC toolchain would allow to create accelerator functions for larger FFT sizes, but the available FPGA resources (DSP slices and on-chip memory) would be exhausted quickly.

We have explored ways for the FPGA and processing system to collaborate in processing the FFT. The goal was a 65'536 point FFT, using fewer resources in the FPGA while still maintaining good performance.

As shown in Fig. 1, the FFT algorithm is divided in clearly defined steps that can be processed in separate units. Our idea was to process the first steps of the FFT in the FPGA grid, then transfer the data into processor memory and do the remaining steps in software, as shown in Fig. 7. The FFT would be split into four 16'384 point FFTs in the FPGA. These smaller FFTs can be processed either in parallel or in series.

Parallel processing requires four FFT cores with four times the resource usage. For serial processing (Fig. 8), only one FFT core is implemented, but the data of the three remaining FFTs must be stored until the core is ready for processing. For efficiency reasons, this is best done in on-chip memory (BRAM).

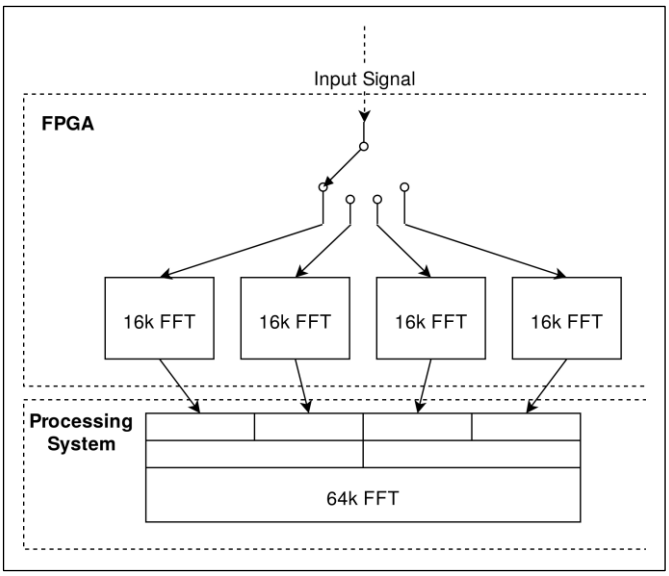

Fig. 7. Partitioning FFT, parallel processing in FPGA

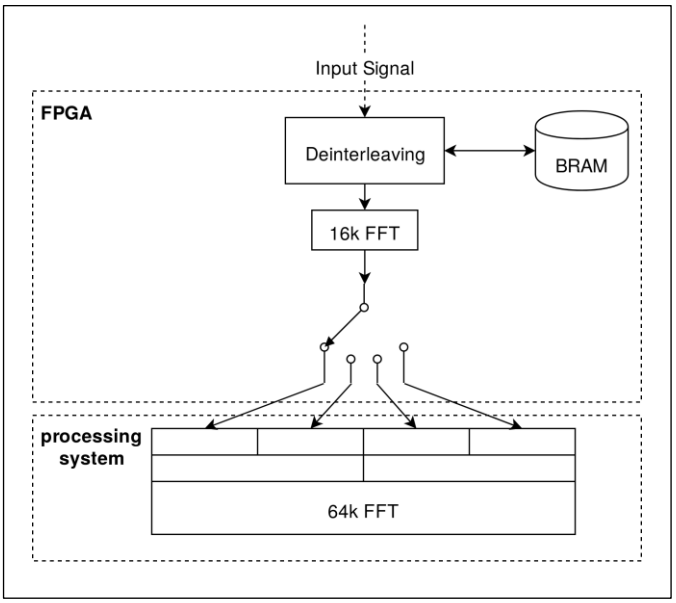

Fig. 8. Partitioning FFT, serial processing in FPGA

We found that the amount of BRAM resources used is similar for both the parallel and the serial approach. Table III shows the number of BRAM blocks and DSP slices used. Values in parentheses show the percentage of all available resources. Because the amount of data to be stored or processed is the same as for a 65'536 point FFT, our approach even uses roughly the same amount of BRAM as the full 65'536 point FFT core. With BRAM being the most limited FPGA resource for this application, there is no gain from partitioning the FFT between FPGA and processing system.

Furthermore, the FFT calculation needs to be finished in the processing system, adding more latency and resource usage to the bill.

TABLE III. RESOURCE REQUIREMENTS OF PARTITIONED FFT

Scenario		BRAM	DSP
Parallel FFT	4x 16k FFT	232 (27%)	180 (7%)
Serial FFT	1x 16k FFT & BRAM	244 (25%)	45 (2%)
Full FFT	1x 64k FFT	238 (27%)	54 (2%)

VI. DISCUSSION

For the FFT, we have shown that the FPGA fabric is able to perform several times faster than the complete processing system of the Zynq UltraScale+ MPSoC. This power can be harvested in several ways, be it as stand-alone FFT processor or as an external accelerator function.

Depending on the amount of processing to be done apart from the FFT, doing the whole transform in the processing system can also be an option, leaving more room in your FPGA.

The decision where to execute an algorithm depends on many factors, such as:

- Where does your data originate? Try to keep it local, reducing the amount of data transfer.
- What are the required data rates? Can the amount of data be transferred over the L3 interconnect without interfering with the remaining processes?
- How well can your algorithm be split up and be processed in parallel? The more an algorithm can be parallelized, the better the FPGA will perform in comparison to the processing system.
- How many FPGA resources can you spare for your algorithm?

In the end, it remains the challenge of the system architect to choose where and how the data is to be processed. A deep understanding of the algorithm and both processing system and FPGA hardware is required.

REFERENCES

[1] P. N. Swarztrauber, "Multiprocessor FFTs," Parallel Computing, Vol. 5, Issues 1–2, pp. 197-210, 1987.

[2] J. Sánchez-Curto, P. Chamorro-Posada, "On a faster parallel implementation of the split-step Fourier method," Parallel Computing, Vol. 34, Issue 9, pp. 539-549, 2008.

[3] T. H. Cormen, D. M. Nicol, "Performing out-of-core FFTs on parallel disk systems," Parallel Computing, Vol. 24, Issue 1, pp. 5-20, 1998.

[4] E. Chu, A. George, "FFT algorithms and their adaptation to parallel processing," Linear Algebra and its Applications, Vol. 284, Issues 1–3, pp. 95-124, 1998.

[5] S. Xue, J. Wang, Y. Li and Q. Peng, "Parallel FFT implementation based on multi-core DSPs," 2011 International Conference on Computational Problem-Solving (ICCP), Chengdu, pp. 426-430, 2011.

[6] R. Lyons, "Computing large DFTs using small FFTs", [Online]: https://www.dsprelated.com/showarticle/63.php

[7] ARM Ne10 Project [Online]: https://projectne10.github.io/Ne10/

[8] kissFFT [Online]: https://sourceforge.net/projects/kissfft/

[9] FFTW3 [Online]: http://www.fftw.org/

[10] M. Frigo, S. G. Johnson, "The Design and Implementation of FFTW3," Proc. IEEE, vol. 93, no. 2, pp. 216-231, 2005.

[11] Xilinx User Guide UG1211, "Zynq UltraScale+ MPSoC Software Acceleration Targeted Reference Design", [Online]: https://www.xilinx.com/support/documentation/boards_and_kits/zcu102/2017_2/ug1211-zcu102-swaccel-trd.pdf. Xilinx, Inc. 2017