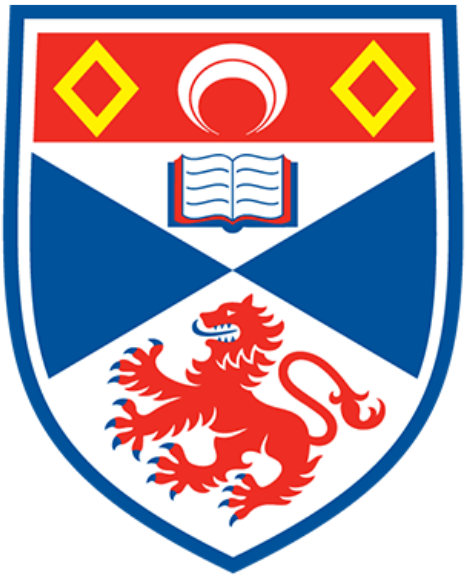


University of
St Andrews

MASTERS PROJECT

Task Farm Parallel Pattern for FPGA-Based MPSoC Environments

Author:
Martynas Noreika

Supervisor:
Dr Vladimir Janjic

Co-Supervisor:
Prof Kevin Hammond

18th January 2019

Declaration of Authorship

I, Martynas Noreika, declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 12,381 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Abstract

The current direction of computer architecture is moving towards a design of heterogeneous systems. Modern devices use integrated circuits to combine multiple computing units, such as CPUs, GPUs and Field Programmable Gate Arrays (FPGAs) into a single chip. Programmable fabric in the FPGAs provides flexible logic blocks that can be used to implement efficient hardware accelerators. However, the flexibility of FPGAs does come at a cost. Developers need domain-specific knowledge to design hardware accelerators and the implementation process is both time consuming and difficult due to the lack of high-level abstractions available. We investigated the use of a task farm parallel pattern as a high-level abstraction for accelerating sequential computations in FPGA-based Multi-Processor System on Chip (MPSoC) environments. Our implementation achieved a speedup of up to 20.16 times when used to accelerate Discrete Fourier Transforms algorithm and up to 8.8 times with the Black Scholes model calculations under certain experimental conditions. Furthermore, we analysed the scalability of the task farm pattern and showed that it can be successfully applied to generate hardware accelerators and utilise them to accelerate computations in a variety of applications.

Contents

Chapter 1

Introduction

The current direction of computer architecture is moving towards a design of heterogeneous systems with a focus on power utilisation and efficient execution of specialised tasks. We are reaching a limit of how much performance can be achieved from multiprocessor systems due to increasing design complexity and power requirements. Computer architects are now focusing on domain specific architectures (DSAs), in a similar fashion when we moved from uniprocessors to multiprocessors in the past [9]. Current computing systems are devised of general purpose processor cores to run large, varied tasks and domain-specific processors that are used to accelerate a narrow range of specific tasks [9].

Modern devices use integrated circuits called System on Chip (SoC), to combine multiple computing units, such as CPUs, GPUs and Field Programmable Gate Arrays (FPGAs) into a single chip. SoCs provide an efficient way to manage power consumption and improve performance by tightly coupling all components together. Furthermore, with economical and industrial drive for performant, energy efficient and economically feasible computing systems there has been an increased interest in development of domain-specific hardware devices.

Programmable fabric in the FPGAs provides flexible logic blocks that can be reconfigured to act as arbitrary logic circuits. This way designers can have custom logic without the need of Application Specific Integrated Circuits (ASICs), that usually are impractical due to high non-recurring engineering (NRE) costs [23]. However, the flexibility of FPGAs does come at a cost. In order to accelerate their programs, developers have to identify parts of the program that could be offloaded to specialised circuits. Furthermore, hardware design requires domain-specific knowledge and it is usually time-consuming and difficult due to lack of high-level abstractions available. Moreover, in order to integrate new hardware designs into the existing application execution model, novel runtime and operating systems are required.

Parallel patterns have been successfully used to simplify parallel programming and code generation in a variety of architectures including multi-core chips, GPUs and FPGAs [23]. They hide the underlying implementation details and allow the developer to express the parallelism according to the data and control flow of the program. Multiple approaches have been made to utilise parallel patterns in generating efficient hardware designs and increasing the developer productivity. However, there seems to be a lack of research that would utilise the parallel patterns to both simplify the accelerator development process and provide infrastructure to integrate the custom hardware designs into a usual software application's execution flow.

This project investigates the potential of using a task farm parallel pattern combined with high-level synthesis tools to accelerate sequential computations on heterogeneous MPSoC (Multi-Processor System on Chip) platforms. This report is structured into five sections: the rest of Chapter 1 gives the context survey and provides the necessary background knowledge, Chapter 2 discusses the SoC development process, Chapter 3 describes the design and implementation of the pattern, Chapter 4 performs a use case evaluation, Chapter 5 analyses the scalability and potential implementation approaches and Chapter 6 concludes the results.

1.1 Programmable Logic in SoCs

Many modern devices, especially mobile ones, have an integrated circuit called SoC (System on Chip). It contains a range of components from computing units such as CPUs, GPUs, ASICs and FPGAs, to system memory (RAM and ROM), peripherals and voltage regulators [5]. The purpose of an SoC is to reduce the physical space needed for an electronic system and contain all the required components in a single chip [5]. This more tightly integrated computer design improves performance, allows computer architects to more easily manage the overall power consumption [31] and, as a consequence, is commonly used in embedded systems and Internet of Things [31] devices. An SoC with multiple processing cores is also called a Multi-Processor System on Chip (MPSoC).

1.1.1 Field Programmable Gate Arrays (FPGAs)

FPGAs are highly heterogeneous computing devices that are capable of implementing any function that can run on a processor [26]. They have a range of resources that allow programmers to define arbitrary circuits and map them to the programmable fabric for execution. This enables developers to cost effectively prototype novel computing architectures and define production level accelerators that can be used in conjunction with a CPU and other processing units. The programmable fabric in the FPGA is composed of Look up Tables (LUTs) that perform logic operations, Flip-Flop registers that store the results from the LUTs, I/O pads which are physical ports that get data in and out of the FPGA and wires to connect elements together [25]. These primitives are encapsulated into Configurable Logic Blocks (CLBs), which are the main logic resources for implementing sequential and combination circuits [1]. Each CLB element is connected to a switch matrix for access to the general routing matrix that interconnects CLBs into the programmable fabric [1]. A basic FPGA architecture can be seen in Figure 1.1.

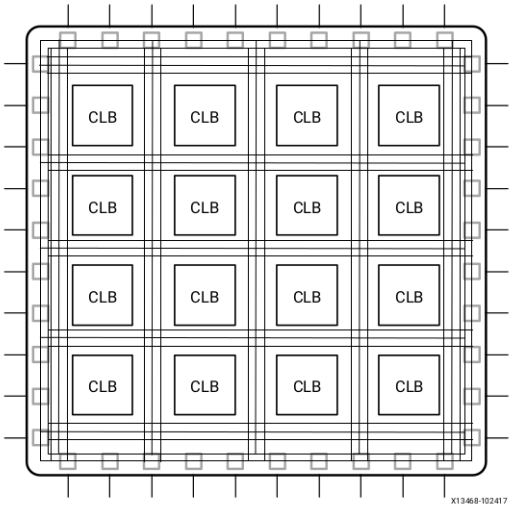


FIGURE 1.1: Basic FPGA Architecture
The CLBs are interconnected and make up the programming fabric. Inputs and ouput ports are accessible from multiple sides. The FPGA compiler tries to map logic circuits to most effectively utilise the available CLBs.
Image source: Xilinx [25]

Modern FPGAs introduce additional computational and data storage blocks that increase the efficiency of the device [25]. These additional elements include: embedded memories for distributed data storage, phase-locked loops (PLLs) for driving the FPGA fabric at different clock rates, high-speed serial transceivers and others [25]. A more in depth explanation of the core FPGA components can be found below:

- Look-up table (LUT)

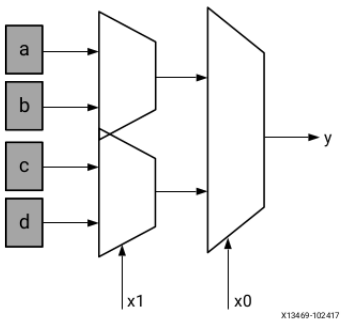


FIGURE 1.2: *Image source: Xilinx [25]*

LUTs are the basic building blocks of the programmable fabric, that are able to implement any logic function of N boolean variables [25]. They represent a truth table where different combinations of inputs produce functions that map to the desired outputs [25].

They are implemented as a collection of memory cells connected to a set of multiplexers (Figure 1.2), enabling them to be both a function compute engine and a data storage element [25].

- Flip-Flop (FF)

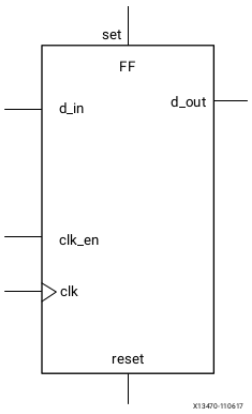


FIGURE 1.3: *Image source: Xilinx [25]*

Flip flops are the basic units of storage in an FPGA system. They are made of data input, clock input, clock enable, reset and data output components [25], as portrayed in Figure 1.3.

During normal operation, the data value is passed from the input port to the output on every pulse of the clock [25]. However, the flip flop is also capable of holding a specific value for more than one clock cycle [25]. This is achieved using the clock enable pin [25].

- DSP48 Block

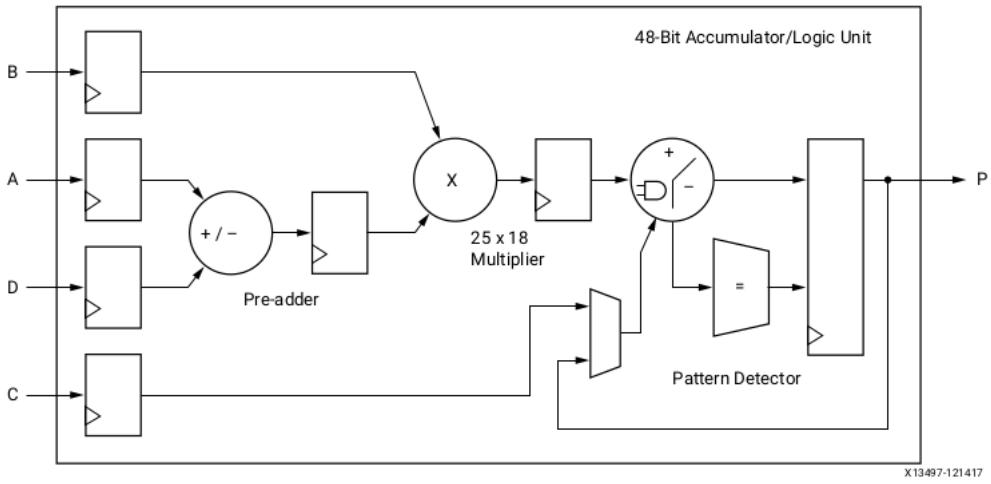


FIGURE 1.4: *Image source: Xilinx [25]*

The DSP48 block is the most complex computation block available in the programmable logic made by Xilinx [25]. It is an arithmetic logic unit (ALU) that is capable of performing chained arithmetic operations in an efficient manner. It acts as a specialised component that performs a lot of the computational load within a synthesised hardware function [25].

- Embedded memories

  The FPGA contains a range of elements that can be used to store data. Three types of memory can be instantiated in the programmable logic: random-access memory (RAM) - implemented using block RAM, read-only memory (ROM) - expressed using LUTs and shift registers - usually defined as a sequence of connected flip-flops [25].

Programmable logic has a very different model of execution compared to a classical CPU [25]. A processor executes a program as a sequence of instructions. Each instruction defines a particular operation such as load/store data or add two operands.

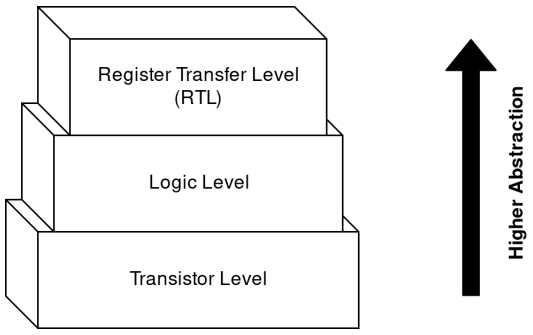
The CPU has a fixed amount of execution units which are shared between the instructions. Furthermore, instructions might take different amount of time to complete [25]. For example, if the desired data item is currently stored in the local cache, then accessing it might only take a couple of clock cycles to complete. However, if the data is stored in DDR RAM, the access time will be multiple magnitudes higher [25].

On the other hand, FPGA is an inherently parallel processing device [25]. It does not have a fixed execution structure and is not hindered by restrictions imposed by a cache or a unified memory space [25]. The FPGA compiler is able to arrange memories as close as possible to the point of use and thus significantly increases the computational performance [25]. Furthermore, each execution unit is instantiated separately using LUTs for each computation in the algorithm rather than reusing a fixed set of shared units [25]. This flexibility of the programmable fabric enables development of optimised hardware functions that can significantly reduce the required computation time. However, at the same time, it introduces an additional complexity that developers have to work with.

The requirements for power efficient, performant and reconfigurable systems have led chip designers to integrate FPGAs into SoCs and develop tools to facilitate rapid application development in such environments. FPGA architectures such as UltraScale [38], developed by Xilinx, and Hyperflex [10], developed by Intel, provide the newest advancements in the programmable logic technology. Combined with ARM Cortex processor cores, they deliver a truly heterogeneous SoCs that are used in a variety of industries including AI, automotive, enterprise networking, aerospace, defence, industrial automation and others [18].

1.1.2 FPGA Development

FPGA development is a complicated process. Engineers have to understand and know not just the software algorithm, but also the low-level details and implementation strategies of the hardware design. Such endeavour requires significant time commitment, even for the experienced developers, due to the need to specify many low-level details of the implementation. Nonetheless, layers of abstraction have been develop to make the development process more scalable. The three lowest levels of digital circuitry design are [36]:



Transistor and logic levels are rarely used for development due to the complexity of the design process and lack of almost any abstractions. While, the Register-Transfer Level (RTL) enables developers to describe their circuit by capturing the desired behaviour of a circuit in terms of a high-level state machine and is commonly used

in practise [36]. The RTL description is defined using Hardware Description Languages (HDL), such *Verilog* and *VHDL*. The FPGA compiler translate the RTL description into a gate level equivalent and perform multiple component placement optimisations. The final circuit is mapped to the available resources on the programmable fabric and is expressed as a bitstream. Before the execution, the bitstream is loaded to the FPGA and configures the available resources on the device to produce the desired behaviour.

This level of reasoning still requires specialised knowledge and is time consuming. In order to simplify hardware development and allow developers with more software based knowledge to be able to define their own circuit designs, high-level synthesis (HLS) tools were developed. They take a high-level description of a hardware function expressed in one of the common programming languages such as C, C++ [16], Haskell [8] and generate an RTL description from it. Such approach enables more rapid hardware prototyping and requires less knowledge from the developer regarding the underling intricacies of the hardware design.

1.2 Heterogeneous Parallel Systems

Heterogeneous designs combine a variety of computing devices including CPUs, GPUs, FPGAs and ASICs. Programmable fabric in the FPGAs is especially dynamic and useful, as it provides flexible logic blocks that can be reconfigured to act as an arbitrary logic circuits. Furthermore, programmable logic provides massive parallel processing capabilities while being power efficient [7]. A performance analysis performed by Berten [7] indicates that FPGAs (e.g. Artix-7 200T) can be 10 times plus more power efficient (72 GFPLOS/W vs 7 GFPLOS/W) than GPUs (e.g. GeForce GT 730), while providing a similar computing power (0.65 TFLOPS vs 0.69 TFLOPS). Moreover, newer FPGA designs allow the logic to be reconfigured dynamically, which enables designs of complex runtime systems that can change the hardware logic during execution.

Since the appearance of the first programmable logic devices, researches and developers have been utilising them to develop complex systems that provide hardware optimised solutions. Recently, with the economical and industrial drive for performant, energy efficient and economically feasible computing systems there has been an increased interest in development of domain-specific-hardware devices, such as Tensor Processing Unit designed by Google [13], used to accelerate domain-specific computations. Multi-core CPUs and GPUs have their limitations and developers are looking for ways to reduce the energy consumption while increasing the performance. However, designing ASICs is time consuming and expensive due to NRE costs [23]. Therefore, FPGAs are being considered as an economically feasible solution for developing hardware accelerators, as they provide an inherently parallel, flexible and energy efficient accelerator design model.

However, this flexibility of programmable logic does come at a cost. A wide range of possible hardware designs brings a new complexity to manage. Programmers have to change their programming models to extract parallelism in the algorithms for acceleration. Furthermore, design of hardware-based accelerators requires domain-specific knowledge and it usually time-consuming and difficult due to lack of high-level abstractions available. Moreover, in order to integrate these new computing units into the existing execution flow and enable sharing of the programmable logic

resources between programs and users, novel operating and runtime systems are needed.

1.2.1 Runtime Systems

Many solutions have been proposed that aim to mitigate both of these issues. In terms of systems, with advancements in the programmable fabric, researches have managed to utilise the partial reconfiguration process to develop runtime systems that are capable of dynamically updating the available accelerators on the chip. One of the major blockers of dynamic reconfiguration used to be a significant reconfiguration time. Kadi et al. [14] managed to reduce this time to an acceptable margin and showed that it is possible to achieve significant speedups using dynamically reconfigurable modules. His team developed a Linux based solution on a Zynq 7000 architecture powered by a dual core ARM 8 processor [14]. The researchers used a novel reconfiguration interface PCAP (Process Configuration Port), which allowed to dynamically reconfigure the FPGA using an ARM processor [14]. The reconfiguration was managed using specifically designed kernel module and a hardware driver *xdevcfg* provided by Digilent [14].

Based on ideas proposed by Kadi et al. [14], Rettkowski et al. [24] designed a runtime system (LinROS), that dynamically schedules and configures various processing elements (PEs), such as processors and accelerators on the programmable fabric [24]. LinROS automatically manages the software and hardware of the reconfigurable MPSoC during runtime [24]. It achieves this using a novel Linux device driver with combination of an IP core developed to facilitate easy hardware integration of PEs using a High-Level-Synthesis (HLS) tools such as VivadoHLS [24]. With this system the authors hope to simplify runtime management of reconfigurable MPSoCs and the integration of hardware accelerators generated by HLS tools [24].

Furthermore, work has been done to integrate programmable fabric accelerators into the cloud computing infrastructure. Traditionally, FPGAs have been used as static accelerators, designed with a single use case in mind [6]. However, given the multi-user cloud environment, an approach that allows sharing of the FPGA resources between users is essential [6]. Fahmy et al. [6] proposes a framework to provide this feature by splitting the programmable fabric into partially reconfigurable regions and adding a management layer to enable deployment and execution of virtual accelerators. Using this approach, in a case-study the researches have demonstrated a significant improvement in the computational efficiency of FPGAs, when compared to software, even with the virtualisation overhead taken into account [6].

1.2.2 High-Level Synthesis

In terms of programming abstractions, many high-level synthesis tools have been proposed over the years that improve the productivity of programming FPGAs and enable design of efficient accelerators. Li et al. [17] using Vivado HLS showed that technological advances by Xilinx provide programming models for FPGAs which are similar to CPU program design. The Vivado HLS compiler transforms a C specification into a register-transfer level description that can be loaded onto an FPGA [17]. Furthermore, it enables designers to exploit various strategies of parallelism, such as loop unrolling or pipelining operations, without the need to generate new

RTL descriptions every time [17]. Finally, the functional behaviour of the produced solutions can be understood and verified by engineers from various abstraction levels [17]. Figure 1.5 compares the Vivado HLS compiler with other programming solutions available to the developer in terms of performance gained versus time spent in development [11]. According to Xilinx, Vivado HLS manages to achieve a significant performance while maintaining development productivity [11].

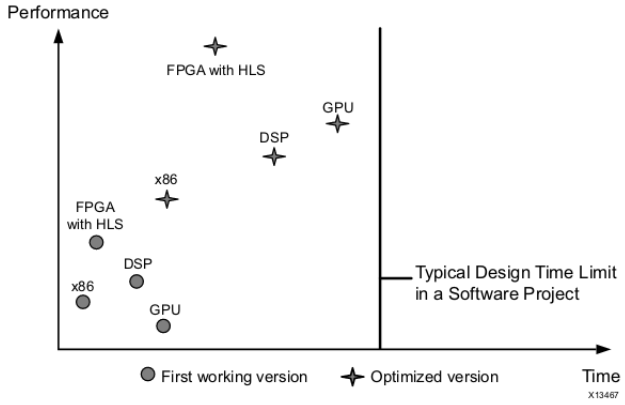


FIGURE 1.5: The diagram illustrates the design time required for initial working version and optimised solution of an application on a variety of architectures vs the performance achieved by the application [11]
Image source: Xilinx [11]

A survey performed by Nane et al. [19] analyses existing the state-of-art high-level synthesis tools available in the market, both academic and commercial, on performance and resource usage. Their findings show that HLS tool can significantly improve the performance with benchmark-specific optimisations and constraints [19]. Furthermore, academic tools seemed to be of similar quality to their commercial equivalents, although proprietary tools had more features and were more robust in general [19]. On the other hand, in order to achieve high performance in hardware, engineers have to embrace new optimisation strategies (e.g. loop pipelining) which differ significantly from software-oriented ones (e.g. data restructure for improved cache locality) [19].

1.2.3 Parallel Patterns

Parallel patterns have been an effective tool in simplifying parallel programming for a variety of heterogeneous systems [23]. They have emerged from parallel programming research as useful high-level abstractions that can elegantly capture data locality, memory access patterns, and parallelism across a wide range of applications [23]. Furthermore, parallel patterns have been successfully used to aid hardware circuit design. Prabhakar et al. [22] showed how parallel patterns expressed with functional languages can be used to generate efficient hardware. He and his team presented two important optimisations techniques: tiling and metapipelining, to automatically tile patterns and generate optimal hardware designs [22]. Their experimental results showed performance speedups up to 39.4 times on a set of benchmarks from the data analytics domain when using the proposed techniques [22] .

Moreover, parallel patterns have been also used to improve the high-level synthesis process. Josipovic et al. [12] extended the standard C-based HLS tools to use computational patterns. They present a template-based hardware generation strategy which enables production of high-quality hardware modules [12]. In order to achieve the desires performance, the computations have to be expressed using a set of parallel patterns, such as *map*, *zip*, and *reduce* [12]. After evaluation, their approach achieved up to 2.8 times speed-up over a state-of-the-art commercial HLS tool and showed that parallel patterns might not only improve the productivity of developing FPGAs, but also lead to more efficient hardware designs [12].

1.3 Motivation

Many solutions have been proposed to increase the productivity of FPGA development and integrate the synthesised computing units into the existing execution flow. High-level synthesis significantly abstracts the low-level details of hardware implementation and improve the development productivity. Nonetheless, developers still need to have a good understanding of the underlying mechanisms to achieve significant performance gains. Moreover, with increasing heterogeneity of hardware, programmers not only have to worry how to design the accelerators, but also how to integrate them into a single program that can co-execute with a CPU. Parallel patterns have been proposed and used as an abstraction to generate efficient hardware designs and increase developer productivity. However, there seems to be a lack of research that would utilise parallel patterns to both simplify the accelerator development process and provide infrastructure to integrate the custom hardware designs into a usual software application's execution flow.

This project explores the potential solutions that would enable rapid development and parallelisation of sequential applications on MPSoC systems. It aims to investigate the potential benefits of applying parallel patterns to development and deployment of primarily sequential applications for highly heterogeneous MPSoC systems. In particular, it tries to evaluate how a task farm parallel pattern combined with high-level synthesis tools could be used to efficiently accelerate sequential code by generating hardware accelerators and effortlessly integrating them into an environment of heterogeneous computing units.

Chapter 2

SoC Development

For this project, we have chosen an SoC that contains multiple CPU cores and a programmable fabric provided by the FPGA, also called MPSoC. After researching the existing options in the market, we decided to use Zynq UltraScale+ EG MPSoC provided by Xilinx (Figure 2.1). This particular chip features a quad-core ARM Cortex-A53 platform running up to 1.5GHz. Furthermore, combined with dual-core Cortex-R5 real-time processors, a Mali-400 MP2 GPU and 16nm FinFET+ programming logic [35].

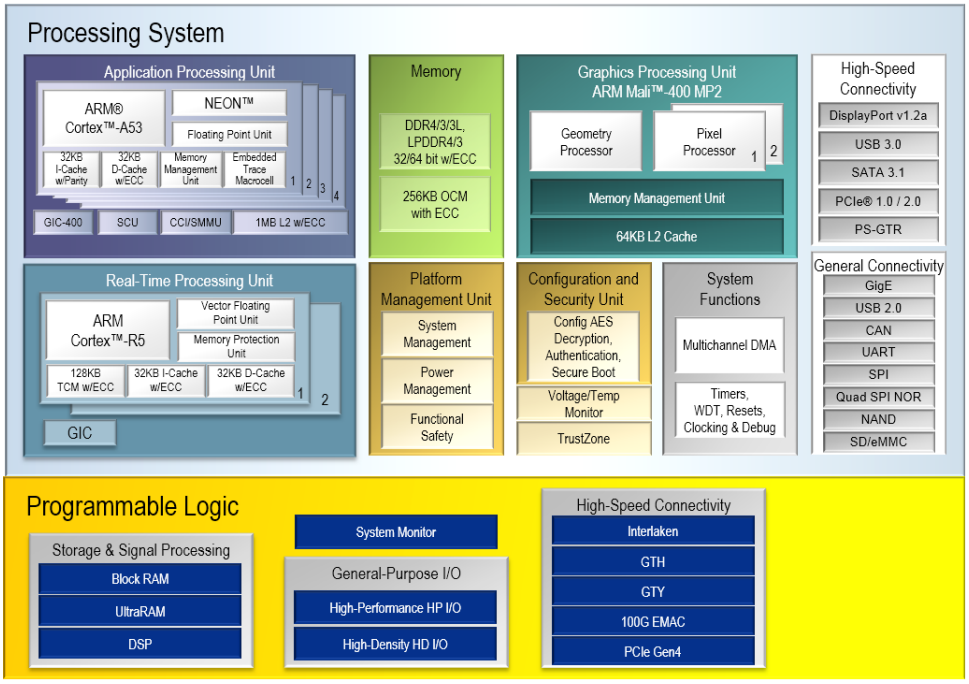


FIGURE 2.1: *Image source: xilinx.com*

Xilinx is an established producer of SoC chips and provides an extensive infrastructure for developing applications on heterogeneous systems. Their development environment called SDSoC (Software Defined System on Chip) provides a framework for designing hardware accelerated embedded processor applications using standard programming languages such as C and C++ [27]. Given our project aims of developing programming abstraction on top of high-level synthesis tools, the Xilinx infrastructure seemed like a perfect choice, especially with established community and mature enough tools available.

To use the Zynq UltraScale+ platform, we needed a development board that would embed the chip into a device that we could access using standard peripherals and perform experiments on. Our choice was Ultra96 development board based on the Linaro 96Boards specification [34] and manufactured by Avnet (Figure 2.2). It provides 2GB of Micron LPDDR4 memory, microSD card reader, Wi-Fi/Bluetooth and other I/O interfaces [34] and comes with a Xilinx SDSoC software license.

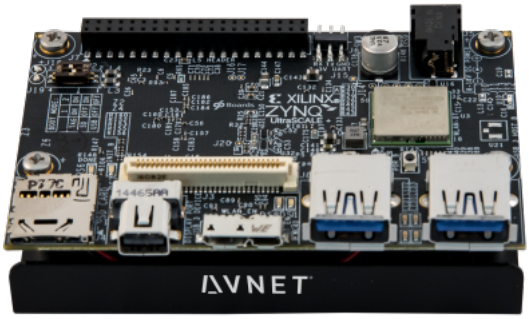


FIGURE 2.2: Ultra96 Development Board
Image source: zedboards.org

In the Xilinx ecosystem, the SDSoC (Software-Defined System on Chip) development environment provides the tools and functionality to develop embedded systems in the Xilinx family devices [30]. It allows design of a full system that combines the development of both software and hardware in one environment and enables developers to define accelerators using HLS to offload some of the computations for faster execution on the programmable logic [30]. The environment comes with an Eclipse-based IDE that provides compilers, debuggers, and profilers for Arm and MicroBlaze processors [30]. Furthermore, it has a hardware emulator and compiler which synthesizes C/C++ functions for optimised execution on the FPGA. Finally, it contains an sdscc/sds++ system compiler, which generates a complete software/hardware system and defines data movers between the two [30].

The SDSoC application can be best understood as a C++ program running on a target CPU after the platform has booted [30]. If it contains any hardware functions, then during the compilation process those function calls will be replaced by stub functions that will act as a synchronisation layer. Each hardware function invokes an accelerator as a task and passes the data between the CPU and accelerator using data movers [30]. A full architecture of a SDSoC system can be seen in Figure 2.3.

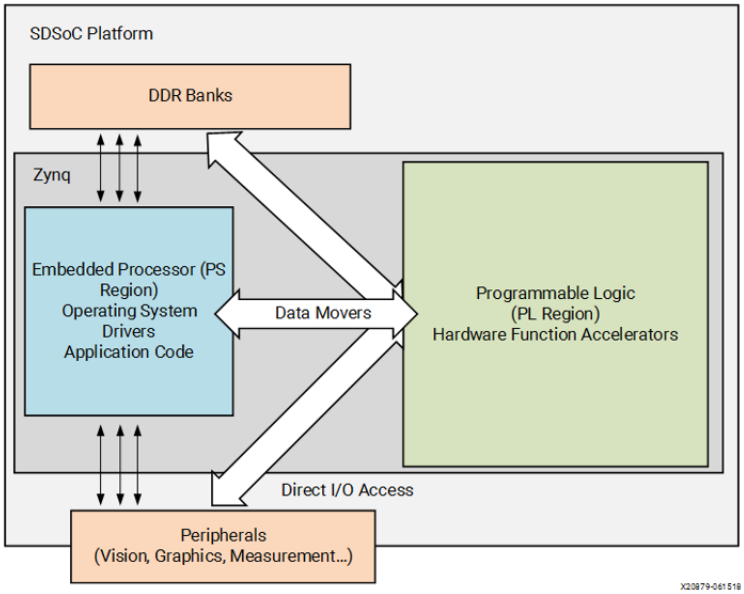


FIGURE 2.3: Architecture of an SDSoC system
Image Source: Xilinx [30]

The SDSoC environment uses a standard compilation process, where sub-processes are invoked to accomplish compilation and linking [30]. A diagram of the full build process can be seen in Figure 2.4. Compilation involves multiple steps during which the hardware functions are synthesised into accelerators using Vivado HLS tool, the application is compiled using standard GNU Arm build tools and the produced outputs are linked by analysing the data movement in the design and modifying the hardware platform to accept the accelerators [30]. The final result is a bootable image that contains application executable and the bitstream used to configure the programmable logic before the execution.

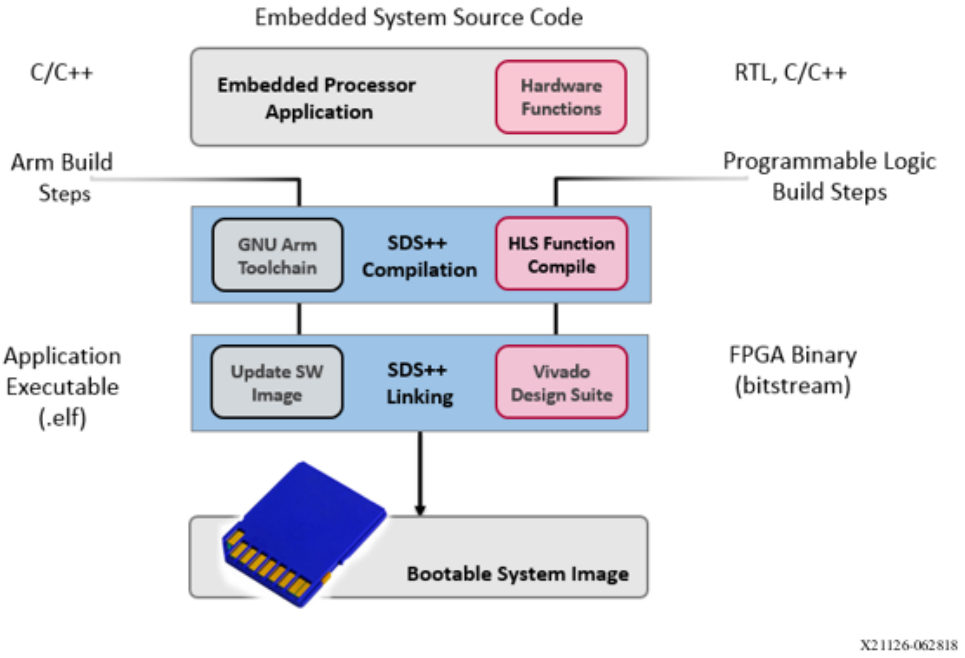


FIGURE 2.4: SDSoC Build Process
Image source: Xilinx [30]

To deploy the compiled solutions to the board and measure the produced results, we used a USB-to-JTAG/UART pod (Figure 2.5) which can be attached to the Ultra96 board to enable serial communication using JTAG and UART dongles available on the board [33]. Using this interface, we were able to control the board over the serial and run our experiments without having to physically restart the board with a new image on the SD-CARD every time. Furthermore, the serial port also enabled us to receive the experimental results live during the execution of our test programs.

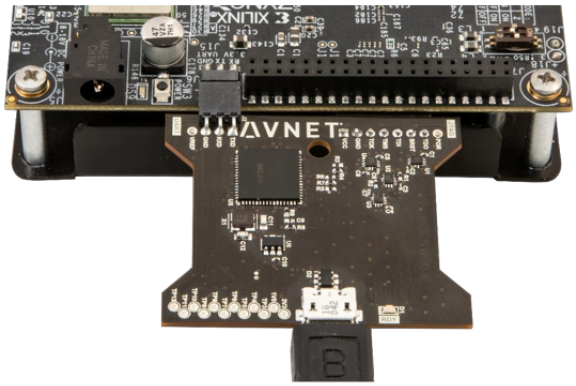


FIGURE 2.5: USB-to-JTAG/UART pod
Image source: Avnet [33]

2.1 SDSoC Programming Flow

The SDSoC environment provides a software-centric approach that enables developers to program software and hardware functions using high-level languages such

as C and C++ [30]. A typical programming flow in such an environment can be described as follows. Firstly, the programmer implements his application in software and determines its correctness by running it on the processing system, which includes the CPU, GPU and other computing devices available on the chip. Secondly, functions that should be executed on the hardware are marked for high-level synthesis and deployment to the programmable logic that contains the FPGA. Furthermore, the developer specifies compiler directives called pragmas, which drive the high-level synthesis process and direct the compiler how a particular hardware function should be converted to a collection of logic circuits. Finally, data movers are specified that indicate how the data will be moved between the processing system and programmable logic during the execution. Once all of these steps are performed, the solution can be then compiled to a targeted hardware platform for execution.

To maximise the effectiveness of hardware acceleration, a function should be computationally intensive and process a lot of data [30]. In an ideal scenario, to take the full advantage of the massively parallel architecture of the programmable logic, the data transfers would be limited to streaming the data into and from the accelerator and instruction-level and task-level parallelism techniques would be highly utilised [30]. The accelerator should aim to consume, process, and ouput the data as soon as possible [30]. The performance of the accelerator is usually measured by determining the Initiation Interval (II), which describes the number of clock cycles required before the function can accept new input data [30]. The ideal II would be one clock cycle, however that might not be possible to achieve due to limitations imposed by the available resources on the programmable logic [30].

2.2 Data Movers

The performance of a SDSoC application also highly depends on the selection of appropriate data movers between the processing system and programmable logic [30]. By default, a data mover is added for each argument in the hardware function by the sds++/sdscc compiler to move the data into and out of the accelerator [30]. The type of the data mover can be inferred by the compiler based on the volume of the data transferred and other characteristics or it can be explicitly specified by the programmer to force the compiler to use the specified data movement strategy [30]. Given a particular data mover, the compiler implements the appropriate Intellectual Property (IP) core in the programmable logic, including the automating control signals and interrupts [30] to facilitate the movement of data. A list of all available data movers and their specifications in the SDSoC development environment can be seen in Table 2.1.

SDSoC Data Mover	Vivado IP Data Mover	Accelerator IP Port Types	Transfer Size	Contiguous Memory Only
axi_lite	processing_system7	register, axilite		No
axi_dma_simple	axi_dma	bram, ap_fifo, axis	<= 32 MB	Yes
axi_dma_sg	axi_dma	bram, ap_fifo, axis		No (but recommended)
axi_fifo	axi_fifo_mm_s	bram, ap_fifo, axis	<= 300B	No
zero_copy	accelerator IP	aximm master		Yes

TABLE 2.1: SDSoC Data Movers Table [28]

Two possible approaches to data movement are available. The data can be either explicitly copied between the main memory and the hardware function, using a suitable data mover for the transfer [16], or the hardware function can access the data directly from the shared memory through an AXI master bus interface. The programmer can decide which data movement approach and data mover would work best for a particular argument in the function or can leave it to the compiler to infer the best combination.

2.3 High-Level Synthesis

The SDSoC environment uses a Vivado High-Level-Synthesis (HLS) tool which transforms a C/C++ specification into an RTL level implementation that then can be synthesised into a logical circuit on the FPGA [37]. The tool achieves this by automatically analysing and exploiting concurrency in the algorithms and mapping the computations to the programmable logic in the most efficient manner to achieve the targeted clock frequency [15].

The Vivado HLS tool comes with a lot of benefits: higher productivity, development and verification of hardware at the C-level, improved system performance for software engineers [37] and others. However, it requires developers to adopt a specific programming paradigm that introduces certain limitations, as the tool is not able to handle any arbitrary software code [15]. For example, no recursive calls, no dynamic memory allocation is allowed inside the hardware functions, the system calls are not supported and only a limited set of standard libraries are available for use [15].

The effective synthesis requires developers to utilise certain hardware optimisation strategies. As discussed earlier, ideally, the hardware accelerator would achieve II = 1. However, there are multiple factors that might limit the potential performance of a hardware circuit [15]. The most important one would be recurrence, which happens then a computation by a component depends on a previous computation by the same component [15]. Fundamentally, existence of recurrence limits the achievable throughput of a design and therefore, developers have to select algorithms for hardware acceleration that have limited recurrence present [15]. Another key factor is the amount of resources available [15]. There are a limited number of memory blocks and wires available on the FPGA and it might not be possible to fully parallelise the computations and achieve the optimal performance. Nonetheless, Vivado HLS tool provides many optimisation techniques and approaches, such as loop pipelining and unrolling, memory partitioning and others, to address these limiting factors and improve the performance of a synthesised circuit.

2.4 Debugging & Testing

The SDSoC design process abstracts a lot of low-level details of both hardware development and its integration into a software system. However, due to the underlying complexity and incorporation of multiple computing units, debugging and testing the produced solutions tend to be highly challenging. The advantages provided by the high-level abstractions can easily back-fire and significantly hinder the development flow in case a non-standard problem is encountered. To deal with these issues,

the SDSoC environment introduces multiple tools that help to debug and test the implementation at multiple stages of the design process.

First of all, hardware synthesis is a computationally heavy process and requires significant amount of time to perform using modern consumer targeted machines. During our design process, using Dell XPS 15 9550 with Intel Core I7 and 16GB of RAM, the compilation process took from ten minutes to multiple hours to complete. This meant that in order to test our implementation on the hardware, we had to on average wait for about half an our for the compiler to finish, even with an optimisation that uses multiple cores in parallel to execute the synthesis process. To improve the development productivity, we used performance estimation tool provided by the SDSoC environment. It takes a hardware function and compiles it normally with the rest of the software system, excluding the computationally heavy generation of the bitstream part. Then, it is able to estimate the hardware performance and compare it against actual measurements performed using a software version of the same function to see the potential achieved speedup. The estimations always assume the worst case cache and communication channel behaviour, thus if a predicted speedup is achieved, the developer can be quite certain that a similar result or usually a better one can be expected during the execution on the real hardware.

Furthermore, the SDSoC environment provides a possibility of emulating the implementation on a cycle-accurate development board emulator. The solution can be efficiently compiled to a specialised emulator form and the developers are able to check the runtime validity of their application before compiling it for execution on the hardware. On the other hand, emulators still take a significant amount to load and perform the virtual execution and they are unable to provide any meaningful potential performance characteristic details.

During our development process, we came up with a design procedure that utilised the available debugging and testing resources and minimised the deployment overhead. Firstly, we checked if our implementation works correctly on a software-only system. Then, we used a debug compilation mode and performance estimation to drive the hardware synthesis design process. Once, we achieved a desired predicted speedup, we would execute the application on the hardware emulator to check its validity while running on multiple computing units. Finally, we would compile the solution using both software and hardware optimisations and benchmark its performance on the actual hardware. This approach enabled us to detect errors early and allowed rapid prototyping of software-hardware solutions.

Chapter 3

Task Farm Parallel Pattern on an FPGA

Parallel patterns have been successfully used as a higher-level abstractions that help developers to express parallelism present in their code and improve execution performance on CPUs, GPUs and FPGAs [23]. Previous work done by Prabhakar et al. [22] and Josipovic et al. [12] show how high-level synthesis tools can be extended using parallel patterns, such as *map*, *zip* and *reduce* to efficiently generate hardware accelerators on the FPGA. In this work, we explore how a task farm pattern combined with high-level synthesis tools could be utilised to automatically offload sequential computations for acceleration on FPGA.

The task farm pattern, also referred to as work farm, is a task parallel algorithmic skeleton [21] that enables concurrent execution of a collection of tasks distributed among multiple computing units. It has been used in a variety of applications, including video compression and decompression, network processing, and graphics applications, where a stream of data needs to be processed in a homogeneous way [3]. The pattern consists of a *farmer* (master) and several *workers* (slaves) [21]. The farmer accepts a sequence of tasks from another process and distributes those tasks to the workers for execution [21]. Once all the workers are finished with their tasks, the farmer returns the results to the original or another process [21]. A diagram illustrating this simple approach can be seen in Figure 3.1.

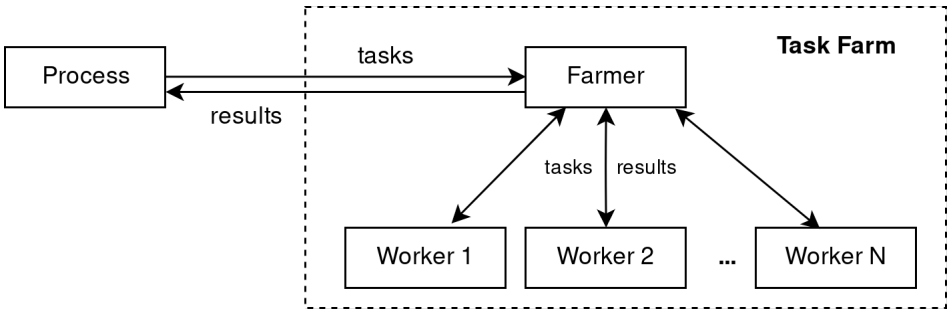


FIGURE 3.1: Task Farm Parallel Pattern

The farmer takes care of all the task scheduling and management level details from the user. The advantage of this is that the user can be completely oblivious to the underling details of the farm and still achieve a significant performance gain for his application. Given the complexity of accelerating computations on programmable

logic, for algorithms where the computations can be divided into tasks and distributed for parallel execution, a task farm pattern abstraction would be a compelling approach to increase computational performance without the extra development overhead.

Furthermore, it might be possible to combine the high-level synthesis and task-farm pattern to produce an algorithmic skeleton that is able to take sequential code, synthesise it to a logical circuit and replicate the circuit among multiple workers on the FPGA for parallel execution. Such an approach would resolve multiple challenges of using FPGAs as accelerators. First of all, it would automatically generate the accelerator for the programmer using high-level synthesis. Secondly, it would provide automatic parallelisation by replicating the accelerators as workers in the programmable fabric. Finally, it would handle the management and integration of the accelerators within a software system. Therefore, ideally, the developer would only have to structure the algorithm so that it fits the pattern expressed as a higher-order function and the compiler would do the rest. We decided to explore the design of such algorithmic skeleton using Xilinx SDSoC environment with the Ultra96 development board.

3.1 Design

We started the design process by deciding how the pattern will handle inputs, outputs and task distribution among the workers. To simplify the implementation, we assumed that all tasks are functionally equivalent and, in other words, perform the same procedure with a given set of data inputs. This makes it easier to synthesise the task functions, however still enables workers to perform different computations by, for example, using conditional branching in the task description. Furthermore, the tasks can be provided to the farmer in full or in chunks. The latter scenario could be useful in situations where the tasks are not all immediately available for execution and are being generated on the go. For example, an external thread might generate a set of tasks on every user I/O operation and pass it to the farm for quick execution. Moreover, we introduced a worker capacity parameter to describe how many tasks a worker can do in one iteration of the farm cycle. This property could be useful if there is a cost associated with the size of work that each worker can execute.

Given these requirements and assumptions, the task farm pattern can be characterised by $task\_farm(I, T, W, C, O)$ function signature where:

- I - a set of inputs
- T - task description expressed as a function that takes a set of inputs and produces a set of outputs
- W - number of workers in the farm
- C - work capacity of each worker
- O - a set of produced outputs

3.2 Implementation

To implement the pattern in an SoC environment we needed tools that enable development of applications which utilise multiple resources, such as CPU and FPGA, available on the SoC system. For this reason, we chose the SDSoC development environment. It comes with a variety of tools that enable rapid development of complete hardware/software systems for SoC devices [27]. As discussed in Chapter 2, the sds++ compiler handles the high-level synthesis of the target functions, generates stub functions to control and communicate with the hardware accelerators on the FPGA and integrates it all with the software that runs on the processing system to produce a single executable. Furthermore, the sds++ compiler, through the use of pragmas, allows the developer to specify not just how the hardware should be synthesised, but also which communication and data movement interfaces should be used between the processing system and programming logic in the SoC environment.

Our first goal was to implement workers on the programmable logic. There are multiple ways how this could be achieved. We started our implementation by utilising the *SDS resource* pragma, which directs the compiler to create multiple instances of a hardware function and generates a data motion network realising the hardware functions in the programmable logic [29]. The pragma has to be used as portrayed in Figure 3.2. It is specified immediately preceding a call to a hardware function and binds the caller to a specified accelerator instance that is identified by the given resource number [29].

```
#pragma SDS resource(1)
worker(A, B, C);
#pragma SDS resource(2)
worker(D, E, F);
```

FIGURE 3.2

In this approach, the CPU would act as the farmer and distribute the tasks to the workers by calling them as portrayed in Figure 3.2. However, this would lead to CPU executing the calls sequentially and blocking on each call until the worker is finished. To avoid this, SDSoC provides two pragmas for explicit control of the *hardware threads* called *SDS async* and *SDS wait* [29]. We can combine them with the *SDS resource* pragma as shown in Figure 3.3, to achieve the desired parallel execution.

```
#pragma SDS async(1)
#pragma SDS resource(1)
worker(A, B, C);

#pragma SDS async(2)
#pragma SDS resource(2)
worker(D, E, F);

#pragma SDS wait(1)
#pragma SDS wait(2)
```

FIGURE 3.3

Furthermore, to distribute the tasks among the workers, the farmer needs to take into account the capacity of each worker and divide the tasks accordingly for execution. In this implementation, we execute the tasks in discrete iterations called work cycles. During each work cycle, the tasks are divided into chunks that are of size equivalent to the worker capacity and are equally distributed to each worker. The farmer tries to utilise as many workers as possible, however it will only use a worker if it can fill its full capacity, due to the latency penalty associated with transferring the data to the programmable logic. The only exception to this, is when the total amount of tasks is lower than the capacity of a single worker. In that case, all the tasks are assigned to a single worker for execution.

To simulate multiple work cycles with varying number of workers, worker capacities and total number of tasks, we designed a procedure portrayed in Algorithm 1. This task farm simulator takes the total number of tasks, number of workers and worker capacity, and executes the tasks in work cycles. The first while loop uses all the workers available until the work left is lower than the maximum total capacity of all workers. The second one, executes the remaining tasks by determining how many workers can still be fully utilised during each work cycle and if there are only a couple of tasks left, a single worker is used to finish the rest.

The worker function is implemented as a hardware accelerator which takes a list of inputs and produces the outputs according to the task definition (Figure 3.4). Each task has a unique id or number that is used by the worker to determine which data elements to access from the memory. Before the start of a work cycle, the worker receives his initial task number and the number of tasks he has to execute expressed as a batch size. The tasks are mapped to the workers in batches and are executed in a sequential manner. The implementation uses a *for* loop which executes the tasks according to the size of the batch and the initial task specified. The *HLS PIPELINE* pragma used inside the loop, directs the compiler to pipeline the loop and execute each operation as soon as the data is available. This reduces the II to 1 and increases the throughput achieved during the computations [29]. With II = 1, the worker is able to execute a single task during every clock tick.

Algorithm 1 Farm Simulator

procedure TASK_FARM_SIM($tasks, W_{number}, W_{capacity}$)
    $work\_left \leftarrow len(tasks)$
    $work\_per\_cycle_{max} \leftarrow W_{number} * W_{capacity}$

    while $work\_left \geq work\_per\_cycle_{max}$ **do**
        work_cycle($tasks, work\_per\_cycle_{max}, Worker_1$ ... $Worker_{W_{number}}$)
        $work\_left -= work\_per\_cycle_{max}$

    while $work\_left > 0$ **do**
        $n \leftarrow work\_left / W_{capacity}$

        if $n \neq 0$ **then**
            $work\_per\_cycle \leftarrow n * W_{capacity}$
            work_cycle($tasks, work\_per\_cycle, Worker_1$ ... $Worker_n$)
            $work\_left -= work\_per\_cycle$
        else
            work_cycle($tasks, work\_left, Worker1$)
            $work\_left = 0$

```
#pragma SDS data zero_copy(...)
void worker(int batch_size, int init_task, ...) {
    for (int task_number = init_task; task_number < batch_size;
         task_number++) {
        #pragma HLS PIPELINE

        /* Task definition */
    }
}
```

FIGURE 3.4: Task Farm worker implementation

The inputs and outputs are specified as a hardware function arguments and need to have pragmas which indicate how the data will be moved from the processing system to the accelerators. The SDSoC development environment provides multiple ways to generate the data motion network. Multiple data movers are available, as discussed in Chapter 2 and can be observed from the Table 2.1.

For our initial implementation, we used the shared memory data transfer strategy which is defined using the *zero_copy* pragma. It generates an AXI Master bus interface which enables workers to directly access the data from the shared memory [29]. We chose the this approach, as it seems to be the simplest data access strategy to start with. It has fewest limitations in terms of the memory access patterns allowed and the amount of data that can be transmitted using it. Furthermore, during our initial performance estimates we noticed that the predicted speedup per accelerator was much more stable using the shared memory approach rather than coping the data, as we increased the number of workers in the farm. It is possible that copying the data to the accelerators at the beginning of the work cycle introduces a bottleneck,

while in case of a shared memory model, the workers accesses the data on demand and the data access is potentially more spread over time.

The *zero copy* data transfer interface is used for parameters that are changing between each task. For constant values, the data is simply copied to each accelerator by the processing system before the start using the *axi_lite* data mover. Futhermore, to use the shared memory model, the memory is alloced using an *sds_alloc* function provided by the SDSoC environment. It ensures that the data block allocated is contiguous in memory. The contiguity of data is a pre-requirement to using the *zero copy* approach. Moreover, the allocated data is marked as cacheable and the clock frequency of 100 MHz is used for the data motion network and execution on the programmable logic.

The implemented solution, synthesises and deploys workers on the programmable logic that can execute a stream of tasks allocated to them during every work cycle. We can scale the execution capacity of the farm by increasing the number of workers on the FPGA. Each worker is implemented as an independent accelerator on the programmable fabric, therefore, knowing the resources required to implement a single worker and the total amount of resources available, we can estimate the maximum capacity of the farm on a given FPGA device. Furthermore, this simple scaling approach should enable us to predict the best data transfer strategy and the achievable performance.

During the implementation process, we faced many challenges. The SDSoC environment provides a lot of tools and abstraction that enable rapid development of software and hardware combined applications. However, at the same time, it acts as a black box and is hard to determine the exact configuration of the synthesised hardware and its location on the programmable logic. For our analysis we used the measurement results and reports produced by the sds++ compiler. However, even with all of that data it was still hard to build the full picture of the produced application. Therefore, we performed multiple experiments and tests to ensure that the produced solution behaves as intended.

Chapter 4

Use Case Evaluation

To evaluate how our task farm pattern design would perform when used in a real word scenario, we implemented two widely used algorithms: Discrete Fourier Transforms and Black Scholes model calculations. Both algorithms were restructured to fit the task farm execution model and were synthesised as a hardware accelerators/workers for use on the FPGA. Numerous experiments and test were then performed to determine the potential of using task farm pattern as an algorithmic skeleton and the tradeoffs involved in offloading computations to the programmable logic in general.

4.1 Experimental Methodology

The experiments were performed on Xilinx UltraScale+ EG MPSoC using the Ultra96 development board [34]. To run applications on the Xilinx MPSoC we had to use a board platform specification that defines the firmware for the processing system. Also, includes basic software components such as the first stage boot loader, system libraries and the applications themselves. Rather than defining our own, we chose an SDSoC Baremetal platform provided by Avnet. Using it we developed a standalone executable that runs a target application on bare metal. Each experiment was performed using a single ARM Cortex-A 53 core (1.5GHz) and the 16nm FinFET+ programming logic (100MHz) available on the device.

The experiments focused on measuring the speedup gained by offloading some of the computations to the FPGA for execution. Experiments were first executed on a single CPU core only and then compared to the results achieved by using a single CPU core and FPGA in combination. The performance comparison was made using CPU clock cycles. Each measurement was made ten times and an average number of clock cycles was taken as the final result. We decided that no more trails were needed as the experiments were executed using a standalone application, that is not affected by multiprogramming and pre-emptive execution strategies found in full-size operating systems. The input data for the experiments was generated using a pseudo-random number generator.

4.2 Discrete Fourier Transforms

For our first use case example we chose the Discrete Fourier Transforms (DFT) algorithm. The DFT algorithm is widely used in many fields of engineering. Furthermore, it is relatively simple to implement, however still contains interesting properties and data access patterns. Our goal was to choose an application that is not inherently parallel and show how we could improve its execution by accelerating it on the programmable logic using the task farm pattern design. A software-based C++ implementation can be seen in Figure 4.1

4.2.1 Implementation

The DFT algorithm takes real and imaginary parts of the data samples and produces the transformed results (Figure 4.1). The transformation is performed by the outer and inner loops. The inner loop accesses each sample from the input and calculates the appropriate transformation given the current indexes of the inner and outer loops. The results are summed and stored as the transformed sample for each output value in the outer loop. Notice that the input data is accessed multiple times during each transformation.

```
void dft(int n, double[] in_real, double[] in_imag,
         double[] out_real, double[] out_imag) {
    for (int k = 0; k < n; k++) {
        double sum_real = 0;
        double sum_imag = 0;

        for (int t = 0; t < n; t++) {
            double angle = 2 * PI * t * k/n;

            sum_real += in_real[t] * cosf(angle)
                      + in_imag[t] * sinf(angle);
            sum_imag += -in_real[t] * sinf(angle)
                      + in_imag[t] * cosf(angle);
        }

        out_real[k] = sum_real;
        out_imag[k] = sum_imag;
    }
}
```

FIGURE 4.1: Discrete Fourier Transforms implementation
Adapted from: Project Nayuki [20]

As discussed in Chapter 2, the accelerators that perform best on the programmable logic have no recurrence, a short initiation interval and are in the bounds of the available resources on the programmable fabric. In an ideal scenario, the accelerator would be able to consume the input data and process it in a single clock cycle. This way the highest throughput would be achieved. When implementing the DFT, we tried to transform the algorithm into an accelerator function that would satisfy the

aforementioned criteria. After many design attempts, we came up with the following solution.

Given that we are able to use multiple computing devices on the same chip, we decided to split the algorithm into two parts and execute each part on a most suitable computing unit. In this case, the FPGA performs the most computationally heavy part and implements the inner loop. While, the CPU combines the results produced by the hardware accelerator and executes the outer loop.

The inner loop is recurrence-free and can be completely parallelised. Therefore, we decided to execute it on the programmable fabric. We defined an iteration of the inner loop as a task a worker would be able to perform and implemented a hardware accelerator to execute it (Figure 4.2). The accelerator is able to consume real and imaginary parts of the input data and produce a partial transformation result in a single clock cycle. As discussed in Chapter 3, this is achieved by pipelining the computations and reducing the II to 1. The worker is able to determine which input values to access by calculating the k and t values (indexes of the outer and inner loops respectively), from the task number and the *iner_index* array.

```
#pragma SDS data zero_copy(real[0:WORKER_CAPACITY], ...);
void worker(int batch_size, int init_task, int n, double* real,
            double* imag, double* real_res,
            double* imag_res, int* iner_index) {

       for (int task_number = init_task;
            task_number < init_task + batch_size; task_number++) {

           #pragma HLS PIPELINE

           int k = task_number / n;
           int t = iner_index[task_number];

           float angle = 2 * PI * t * k / n;
           double cos_angle = cosf(angle);
           double sin_angle = sinf(angle);


           real_res[task_number] = real[t] * cos_angle
                                 + imag[t] * sin_angle;
           imag_res[task_number] = -real[t] * sin_angle
                                 + imag[t] * cos_angle;
       }
 }
```

FIGURE 4.2: Task farm worker implementation of the DFT algorithm

In the original implementation, the inner loop also adds the partial results to the output sums. However, performing summation using the workers would potentially lead to synchronisation bottlenecks as multiple workers would have to access the same memory location and would slow down the achievable throughput. Therefore, instead, we decided to leave the summation of the partial results to the CPU,

where synchronisation is not an issue. Furthermore, due to limited amount of programmable logic resources available on our development board, we have reduced the precision of trigonometric functions used in the algorithm both in software-only and task farm implementations. This allowed us to fit more workers on the FPGA and perform a more in-depth analysis of the task farm pattern.

4.2.2 Evaluation

To determine the effectiveness of our task farm DFT implementation we performed multiple experiments and compared the results with the software-only solution. Figure 4.3 shows the performance results comparing a single core CPU implementation versus a single core CPU + FPGA task farm version using three workers with a worker capacity of 1000.

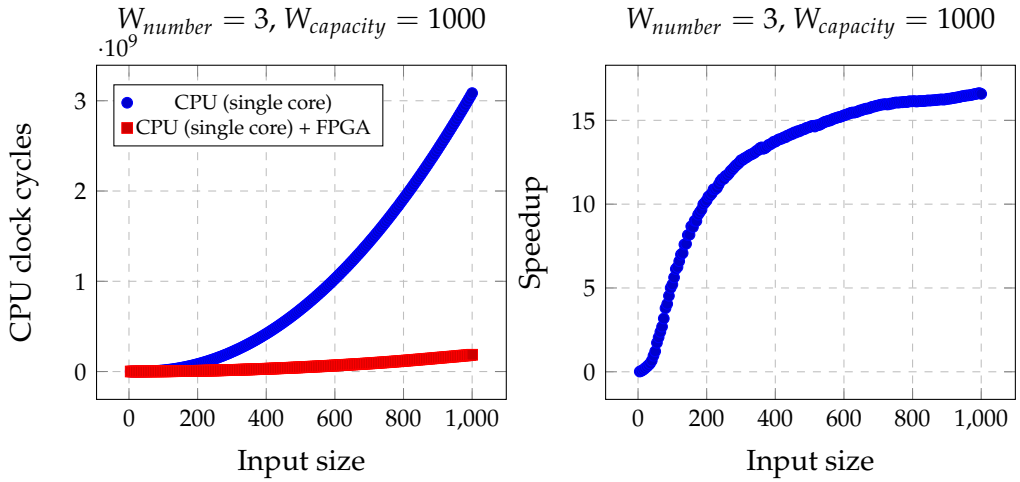


FIGURE 4.3: Discrete Fourier Transforms software only vs task farm implementation

We can see that task farm approach scales significantly better and achieves a speedup of almost 16.7 times with an input size of 1000, when compared to a the single core CPU implementation. It should be noted, that the input size is not directly linked to the number of tasks. For an input of size n, the workers need to execute n^2 number of tasks.

Furthermore, we performed experiments with a varying number of workers. The results can be seen in Figure 4.8. As expected, increasing the number of workers, enables more parallelism and better performance is achieved. At the same time, having more workers requires additional use of programmable logic resources, which leads to increased energy consumption. In this case, three workers were the maximum amount we could fit into the available programming logic.

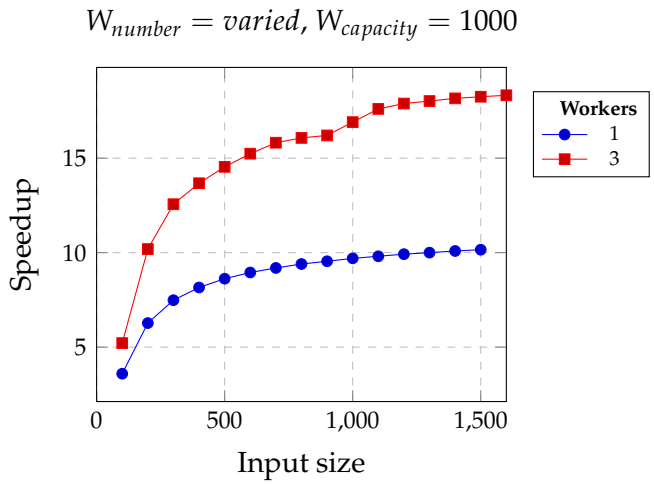


FIGURE 4.4: Discrete Fourier Transforms software only vs task farm implementation with varying worker number

Further experiments also showed the effects of changing the worker capacity. Having a higher worker capacity, usually means that a fewer amount of calls have to be made to the accelerators. Each call incurs a data transfer latency penalty, potentially reducing the performance. Therefore, to maximise the speedup achieved, we would want to provide the farmer with a high number of tasks and increase the worker's capacity to its theoretical limit, which is $W_{max\ capacity} = \frac{N}{W_{number}}$, given that the tasks can be equally divided between the workers, where N is the number of tasks to execute. However, in realistic applications, data might be acquired in certain intervals, especially if a user input is required. Therefore, a limited worker capacity might help to simulate the performance of the task farm in those scenarios. Figure 4.5 shows the impact of increasing the worker capacity on performance with an increasing input size.

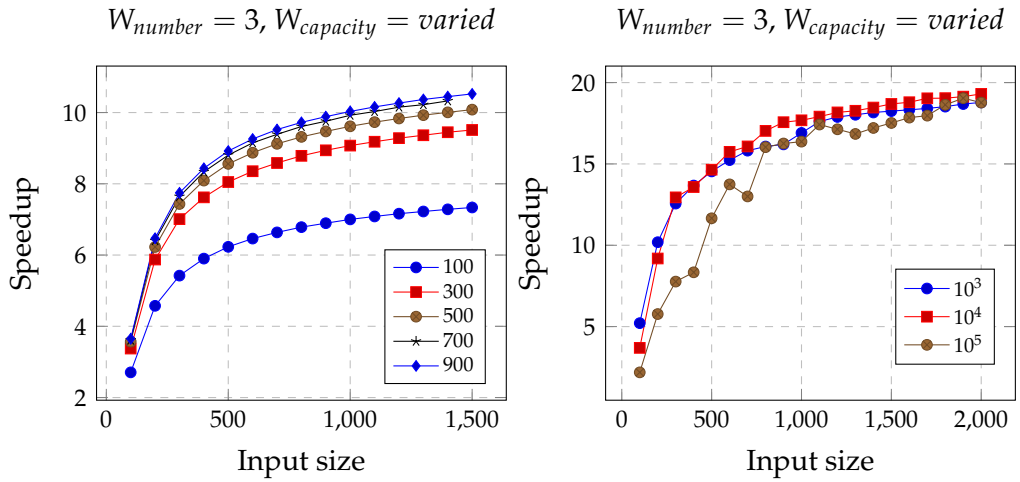


FIGURE 4.5: Discrete Fourier Transforms software only vs task farm implementation with varying worker capacity

We can see that increasing the worker capacity generally improves the achieved performance. However, up to a certain point. The graph on the right in Figure 4.5 indicates that after a certain threshold the speedup becomes less stable and performs

worse than compared to smaller capacities, as the input size increases. It is hard to determine the reason for such behaviour. Most likely, the increased worker capacity interferes with the caching and data transfer optimisation techniques used by the shared memory interface and produces such unstable performance results.

Moreover, further design efforts lead us to notice a bunch of areas in the current DFT implementation that could be improved. For example, the *iner_index* input array is not required and the solution can be simplified by calculating the outer and inner loop indexes as portrayed in Figure 4.6. Additionally, we noticed that it is possible to improve the performance by implementing a FIFO streaming interface as part of the accelerator. The performance gained from such design improvements can be found in Appendix A.

```
int k = task_number / n;
int t = task_number % n;
```

FIGURE 4.6: Given the bounds of nested loops, we can encode each iteration into a single number. The variables k and t here represent the indexes of the outer and inner loops of the DFT algorithm and are calculated from a unique task number

During the initial experimental measurements, we compared the results produced by the task farm implementation and the software-only equivalent to ensure the accuracy of the measurements. However, due to rounding errors present in the floating point calculations [4], it was hard to compare the results as they would sometimes differ by a marginal amount. In those cases, to check the correctness of our implementation we would reduce the execution scale and manually ensure that all the results are correct.

After the initial measurements discussed above, we performed a more thorough analysis and noticed that our hardware implementation of DFT would produce incorrect results after a certain input size. This lead us to use a more sophisticated method for determining the correctness of the results. We implemented a relative epsilon comparison approach described by Dawson [4]. This method finds the difference between two numbers and compares it to their magnitudes. The numbers are considered equal if the difference is smaller than the n procent of the largest of the two, where n is a constant selected to reflect the allowed error for each application [4].

Furthermore, to ensure the validity of our early DFT experiments, we reproduced some of our earlier measurements using the aforementioned result validation technique. We discovered that the outputs produced by the hardware implementation are incorrect if the input size is larger than the worker capacity. Nonetheless, the results were accurate with an allowed relative error of 0.01, while the input size was below the worker capacity, with some measurements occasionally exceeding the allowed error range.

The reason behind the incorrect results turned out to be an invalid use of the *zero copy* pragma. The *zero copy* data mover can be defined as shown in Figure 4.7. Each array variable which will utilise the shared memory interface has to be defined in the brackets by indicating the variables name, offset and the data transfer size [29]. According to the pragma manual, the offset is ignored and should be specified as

zero [29]. Given that this pragma enables shared memory access, we assumed that specifying the data transfer size was enough, no matter where in the array the data is accessed. However, it seems that the array size also indicates the allowed access dimensions of the array. In our scenario, we used the worker capacity to indicate the data transfer size (Figure 4.2), therefore as soon as the worker tried to access an element in an input array which index was higher than the worker capacity, the access was potentially blocked by the data mover interface (from our observations the returned value seems to default to zero) and incorrect results were produced.

```
#pragma SDS data zero_copy(A[<offset>:<length>])
```

FIGURE 4.7: Semantics of the zero_copy pragma [29]

To mitigate this problem, we investigated how the *zero copy* pragma could be used validly in our use case scenario. We found that, instead of passing a pointer variable to an input array starting from the beginning of the array, each worker could receive a pointer pointing to their first task in a batch allocated to them. That way the zero offset would always be valid and the workers could access elements from any interval in the array. Furthermore, it seems that the data transfer size can be specified dynamically by a variable passed to the accelerator function during the execution.

Unfortunately, due to the time constraints imposed on this project, we were unable to replicate all of our experiments. Instead, we decided to determine if the behaviour observed in the initial evaluation could have been influenced by the invalid use of the shared memory interface. Figure 4.8 shows the experiment results of a speedup achieved when comparing the DFT software-only solution versus a revised task farm hardware accelerated implementation with a varying worker number.

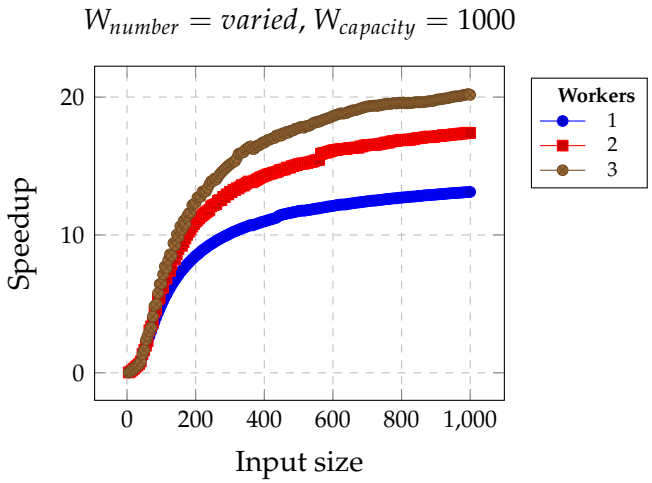


FIGURE 4.8: Discrete Fourier Transforms software only vs revised task farm implementation with varying worker number

The results seem to suggest a similar behaviour of the task farm pattern applied to the DFT algorithm as observed during our initial experiments. Therefore, we believe that the invalid use of the shared memory interface did not significantly influence the results of the initial implementation, although further measurements would have to be made to be sure. Furthermore, it seems that specifying the data

transfer size dynamically have improved the general performance of a worker. As a result, the revised task farm DFT implementation achieves a speedup up to 20.16 times with an input of size 1000.

4.3 Black Scholes Model

Our second use case example was the Black Scholes algorithm. Black Scholes is a widely used model in finance to determine the fair price or a theoretical value for a call or put option, based on six variables such as volatility, type of option, underlying stock price, time, strike price, and risk-free rate [32]. We implemented an analytical approach, that is used in PARSEC parallel benchmark, to estimate the effectiveness of parallel systems in solving partial differential equations [2].

The algorithm works by taking a set of input values and, according to the Black Scholes formula, calculating two output results: the option's call and put values. We implemented the algorithm following a similar design process as with the DFT example. However, compared to DFT, this algorithm was easier to synthesise as the computations naturally fitted the worker task's structure. No loops are present and the computations can be pipelined to process the data inputs in one clock cycle giving the II = 1. In this implementation, to minimise the FPGA resource usage, we have again used a reduced precision mathematical functions.

Furthermore, with insights made during the design of the DFT algorithm, we improved the farm worker implementation to better reflect the data access pattern exhibited by the Black Scholes model. Instead of using the task number to indicate which input data elements the user should access, we allocated each worker a pointer that points to the input interval that they should execute next. This way we can use the shared memory model with the *zero copy* pragma and ensure that the data access offset is always zero, as required by the pragma specification guidelines [29]. Moreover, to ensure the correctness of the results we used the same relative epsilon comparison method as described before.

Our initial measurements showed performance boost when the task farm solution was compared against a single core CPU implementation. However, due to the complexity of the algorithm, we did not have enough resources available on the FPGA to synthesize more than one worker. Nonetheless, successful speedups were still observed (Figure 4.9). Our task farm Black Scholes implementation achieved a speedup of 6.25 times using a single worker with capacity of 1000 and an input size of 10000, when compared to software-only solution.

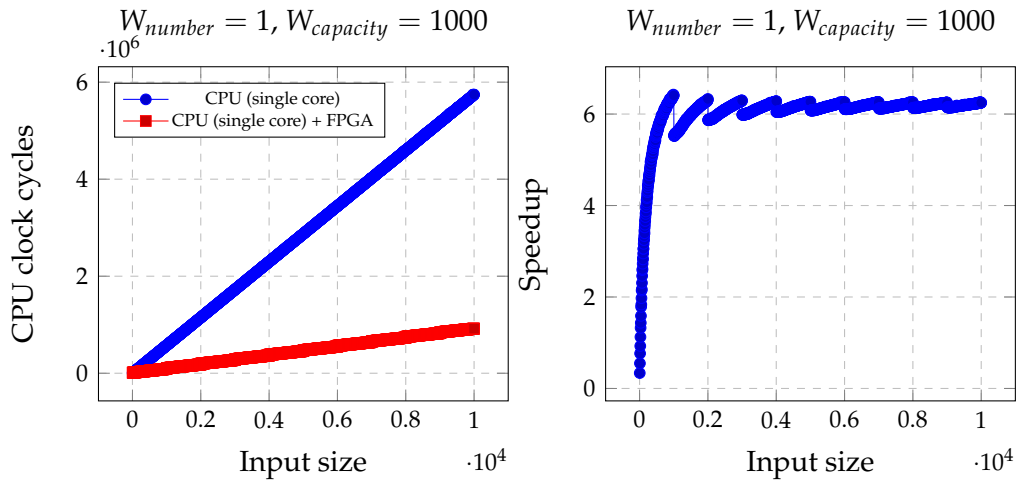


FIGURE 4.9: Black Scholes software only vs task farm implementation

In contrast to task farm applied to DFT, both software-only and task farm implementations of the Black Scholes algorithm scale in a linear manner. We can explain this by considering the asymptotic time complexities of the two algorithms. For a given input size n, the DFT has an asymptotic time complexity of $O(n^2)$, while, Black Scholes can perform computations in $O(n)$ time.

Furthermore, the speedup curve in Figure 4.9, seems to be fluctuating in *steps* with decreasing intensity. Most likely this phenomena happens due to the amortisation of the data transfer latency as the input size increases. Initially, after the worker capacity is exceeded the farmer has to call the worker twice to execute all the tasks. Therefore, an extra latency penalty has to be paid for the second call. However, as the input size increases, the latency cost becomes less and less significant, compared to the time it takes for the worker to execute the given tasks, and the achieved speedup becomes more stable.

Moreover, both with DFT and Black Scholes task farm pattern implementations we can see that the speedup achieved after a certain input size starts to converge to a limit. This limit is most likely imposed by the clock frequency of the programmable logic. The ARM Cortex A5 core used in our experiments operates up to 1.5 GHz, while the programmable logic can only reach a clock frequency of 100 MHz. Therefore, when we offload execution of the tasks onto the FPGA we see a performance increase due to the execution advantage gained by utilising an efficient hardware implementation of the algorithm. Furthermore, as the input increases, we see this performance increase rising as well. This behaviour most likely happens due to the data transfer latency penalty being amortised by the longer computation times required to process the ever increasing input size. However, at some point, the data processing limit dictated by the clock frequency is reached and the speedup settles. The exact speedup limit seems to depend on the maximum clock frequencies of the CPU and FPGA, the task definition, input size and the worker data access patterns.

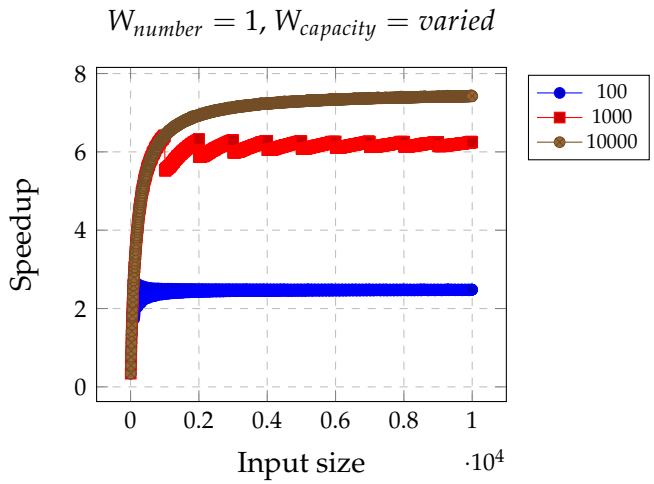


FIGURE 4.10: Black Scholes software only vs task farm implementation with varying worker capacity

We also investigated the effects that the worker capacity has on performance achieved (Figure 4.10). Similarly to the DFT example, the worker capacity seems to dictate the limits of the achievable speedup. A higher capacity requires a lower amount of work cycles to be used for execution and thus a smaller latency penalty has to be paid, leading to an increased overall performance.

Finally, up until now, we have been using the shared memory model. It enabled us to predictably scale the performance of the farm by introducing more workers and had fewest limitations in terms of the data access patterns allowed. However, it has an extra latency cost associated with it, as the memory model is shared and has to be coherent between multiple users. Given that the Black Scholes algorithm has a sequential data access pattern, we modified our implementation to use a *dma_simple* data mover which copies the data to the programmable logic and delivers a reduced transfer latency when compared to the *zero copy* approach. Using this approach, the experimental results showed an increased performance. With a worker capacity of 10000 we achieved a speedup of 8.8 times compared to 7.4 times previously observed using the shared memory interface (Appendix A).

Chapter 5

Scalability of Task Farm Parallel Pattern

After successfully accelerating DFT and Black Scholes algorithms, we decided to investigate the properties of the task farm pattern further. Previous examples utilised a significant amount of the available programmable logic resources and thus only a small amount of accelerators could be fitted into the available silicon. Furthermore, more complicated circuitry takes longer to synthesise and makes it harder to experiment with different settings and designs. Therefore, we decided to use a synthetic task, that might be rarely used in practise, but would nonetheless enable us to perform a more in-depth analysis.

5.1 Synthetic Application Example

We designed a simple accelerator that computes a *sin* function for a given array of inputs. We used the floating point approximation of the sin function and reduced the precision in order to minimise the silicon cost even more. The produced solution simply reads an array of float inputs, computes a *sin* value for each element in the array and stores the result in results array, as shown in Figure 5.1.

```
#pragma SDS data zero_copy(input[0:batch_size], result[0:batch_size])
void worker(int batch_size, float* input, float* result) {
    for (int i = 0; i < batch_size; i++) {
        #pragma HLS PIPELINE

        result[i] = sinf(input[i]);
    }
}
```

FIGURE 5.1: Synthetic example worker implementation

We then compared its execution on hardware and software and measured the observed number of CPU clock cycles taken to execute both functions. To verify the software and hardware results we used the relatively epsilon comparison method as described in Chapter 4. Figure 5.2 shows the observed results. We can see a significant speedup provided by the hardware accelerator that goes as high as 52.2 times

with the input size of 1000. Furthermore, it achieves this while only using 5.8% of DSP and 5.7% of LUT available, as can be observed from the resource utilisation estimates in Table 5.1.

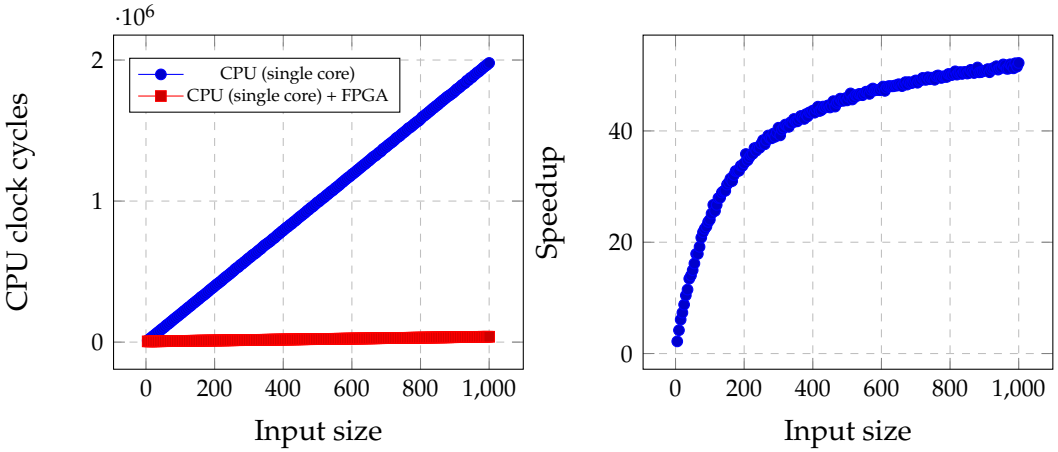


FIGURE 5.2: Synthetic Example software only vs task farm implementation

Resource	Used	Total	% Utilisation
DSP	21	360	5.8
BRAM	4	432	~ 0
LUT	4022	70560	5.7
FF	2453	141120	1.7

TABLE 5.1: Synthetic example resource utilisation estimates on the programmable logic

To test if the program would benefit from the task farm pattern, we have measured the performance with varying number of accelerators as shown in Figure 5.3. In these experiments, we have chosen a different task distribution strategy between the workers. Instead of having a fixed worker capacity as before, to simplify the analysis, the farmer dynamically splits the input equally between each worker, with an exception of the first worker who also executes the remainder, if the tasks cannot be divided equally.

In contrast to the results achieved while benchmarking the *DFT* algorithm (Figure 4.3), increasing the number of accelerators seems to have almost no effect on the observable speedups. In this case, a higher number of workers in the farm resulted in a marginally reduced rather than significantly increased performance. We can explain this by considering the nature of the algorithm and its memory access pattern. The *sinf* worker (Figure 5.1) accesses the data in a sequential manner, therefore the inputs and outputs can be efficiently burst read/written from and to the accelerator. Once we introduce multiple workers in the farm, the solution now has to split the input stream between the workers and synchronise them at the end. This approach most likely introduces an extra overhead cost that reduces the performance.

In comparison, the *DFT* algorithm has a more complicated memory access pattern and reuses the same input values between different tasks. Furthermore, it is possible that the shared memory model is able to utilise the memory hierarchy available in the processing system and retrieve recently accesses entries from the cache. Therefore, having multiple accelerators, in the DFT's case, allows to potentially retrieve the data quicker from the cache and increases the performance.

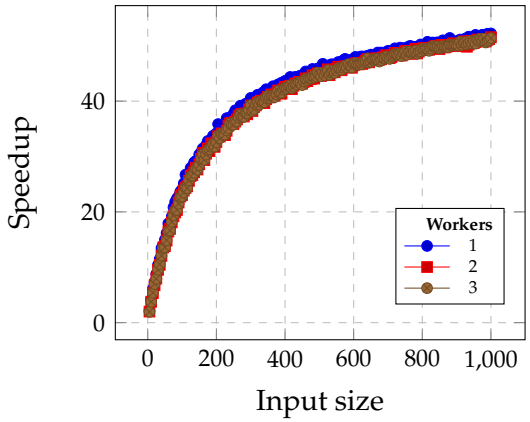


FIGURE 5.3: Synthetic Example software only vs task farm implementation with varying worker number

Given the lack of performance improvement of applying the task farm pattern to the synthetic task, we decided to investigate different pattern implementation approaches and measure their benefits and tradeoffs. Up until now, we have been using *SDS resource* pragma to define a new worker in the farm. However, this approach splits the programmable fabric into multiple accelerators, introduces a separate data port for each and requires manual handling of hardware threads in an asynchronous execution scenario. As observed in Figure 5.3, this design seems to add additional resource management overhead and does not lead to an increased performance.

A different approach would be to synthesise a single accelerator that defines the farm workers as internal modules. In such case, we would only need to manage one accelerator from the processing system's perspective and have a simpler memory access interface on the programming logic's side. Figure 5.4 illustrates the differences between the two farm design approaches.

To achieve this, we can simply tweak the code snippet from Figure 5.1 and add support for multiple workers inside the accelerator. Figure 5.5 shows the resultant code. We now have two calls to the worker function with different input and output pointers. Notice that we used *HLS DATAFLOW* pragma here instead of *HLS PIPELINE*. The former tells the compiler to execute operations in a concurrent manner, which in this case leads to a merge of the two workers into one. The latter directs the compiler to perform task-level parallelism and enables hardware functions to overlap and execute in parallel. We can confirm this by observing the diagram from the synthesis analysis report in Figure 5.6.

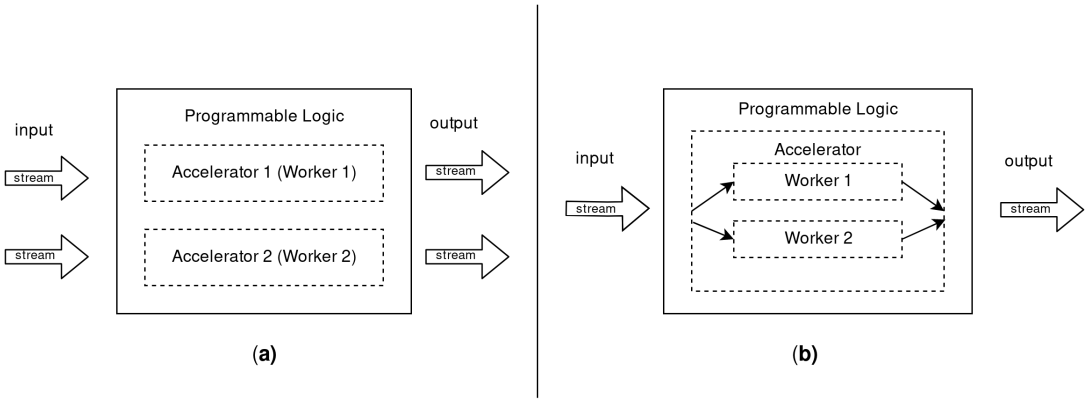


FIGURE 5.4: Diagram (a) illustrates the task farm implementation scenario where workers are implemented as independent accelerators. In that case, a separate shared memory access port is generated for each worker and thus the input stream has to be split between the workers. While, diagram (b) shows workers implemented as modules and being part of the same accelerator. Such approach requires only one shared memory port per input and performs the division of the data stream inside the accelerator.

```
void worker(int n, float* input, float* result) {
    for (int i = 0; i < n; i++) {
        #pragma HLS PIPELINE
        result[i] = sinf(input[i]);
    }
}

/* Shared Memory Interface (zero copy) */
void task_farm(int n1, int n2, float* in1, float* in2,
           float* res1, float* res2) {
    #pragma HLS DATAFLOW
    worker(n1, in1, res1);
    worker(n2, in2, res2);
}
```

FIGURE 5.5: Alternative task farm implementation using modules to implement workers in a single accelerator

The experimental results show that choosing this task farm implementation improves the performance, however only once a particular input size is reached (Figure 5.7). It seems that there is an overhead cost involved in splitting the input stream to multiple workers on the programmable logic as well. We can see that up until a certain point both two and three worker farms are performing worse than a single worker implementation. However, once a certain threshold is reached, the advantage of having multiple workers becomes more apparent and a higher speedup is achieved. It seems that the higher the number of workers, the longer it takes to reach the threshold point where the overhead cost of splitting the stream between the workers is outweighed by the advantages of processing it in parallel.

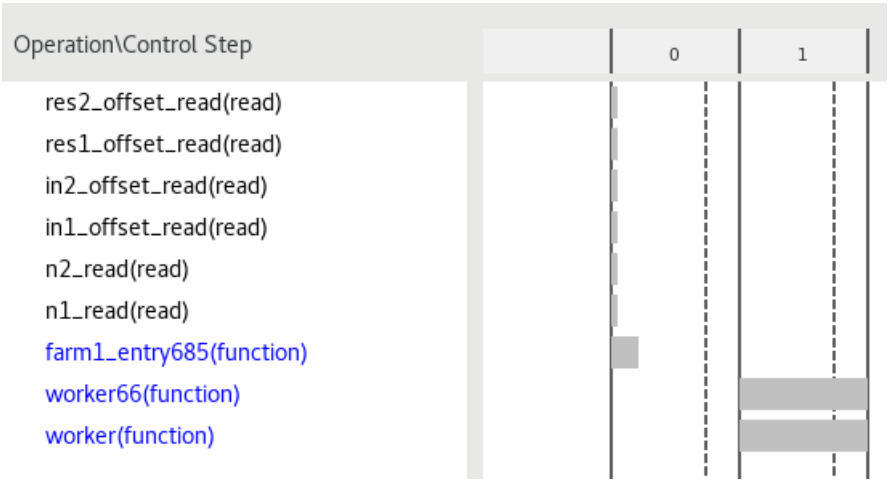


FIGURE 5.6: Synthesis report of the alternative task farm implementation: we can see the worker functions being synthesised as two separate modules that execute concurrently.

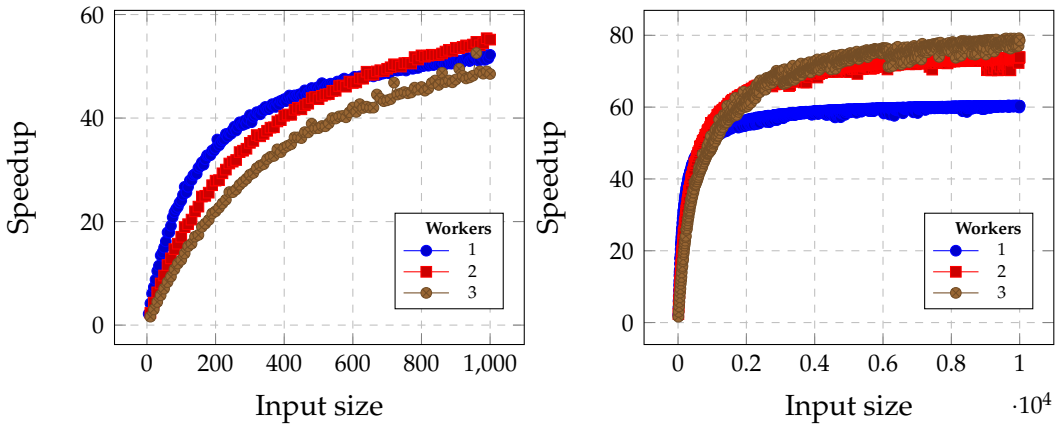


FIGURE 5.7: Synthetic Example software only vs alternative task farm implementation with varying worker number

Furthermore, our previous task farm pattern implementations have been using the shared memory model and it allowed us to stream the data, compute the result for each input in a single clock cycle and stream back the result. This approach seemed most favourable in the beginning, as we were able to compute an arbitrarily sized inputs and receive predictable and consistent speedups. However, it might be worthwhile to explore an alternative implementation strategy where the data is copied first in chucks to the accelerator and then is accessed by multiple workers to produce the results in parallel. A diagram illustrating the structure of such an accelerator can be seen in Figure 5.8.

In this design, we first copy the input data to the accelerator and store it in blocks of RAM. Then, the computations are performed on each data item using the farm workers in parallel and the results are stored in the result blocks of RAM. Finally, the output data is then copied to the processing system. The code describing such a design can be seen in Figure 5.10. In order to enable a truly parallel execution of the workers, we had to partition the input array into multiple blocks of RAM. Usually, a block of RAM has only one access port, however using the *HLS array_partition*

pragma, we are able to completely partition the input array and store each element in a flip-flop register that can be directly accessed by the worker. This way a truly parallel execution can be achieved.

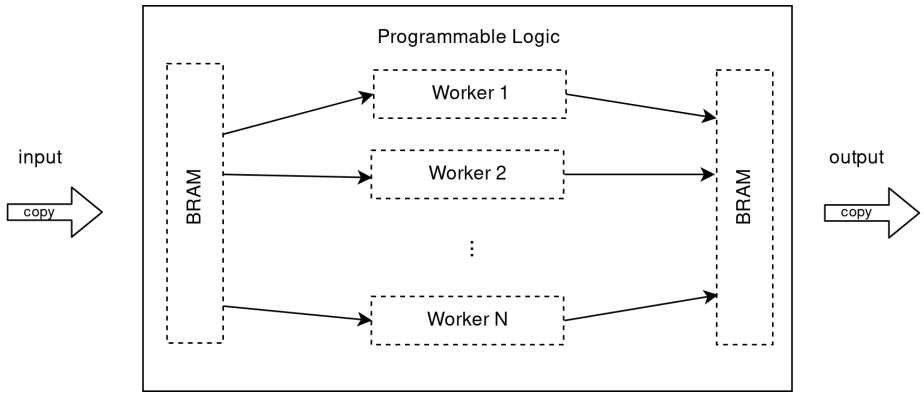


FIGURE 5.8: Task farm pattern implementation using memory blocks on the programmable logic to store inputs and outputs

In comparison, such task farm pattern design is similar to execution principles of vector processors. In both computing models, multiple data items are processed by the same compute function. In case of vector processors, the execution is performed using multiple lanes with arithmetic logic units (ALUs). While, in our implementation, the execution is performed by the synthesised farm workers. The main difference being the granularity of computations between the two. The ALUs can only execute the most basic arithmetic calculations, while the worker is able to execute any arbitrary circuit that can be defined as a task. Therefore, it is possible to consider this particular implementation of the task farm pattern as a generalised vector processor.

This task pattern implementation approach was harder to test experimentally, as the amount of input data that can be copied at a time must be specified during the compilation. In order to attain comparable results, for inputs larger than the defined storage capacity of the accelerator, we have resorted to splitting the input data into chunks and calling the accelerator multiple times. Experimental measurements showed promising results (Figure 5.9). The achieved speedups appear to be very stable as the input size increases.

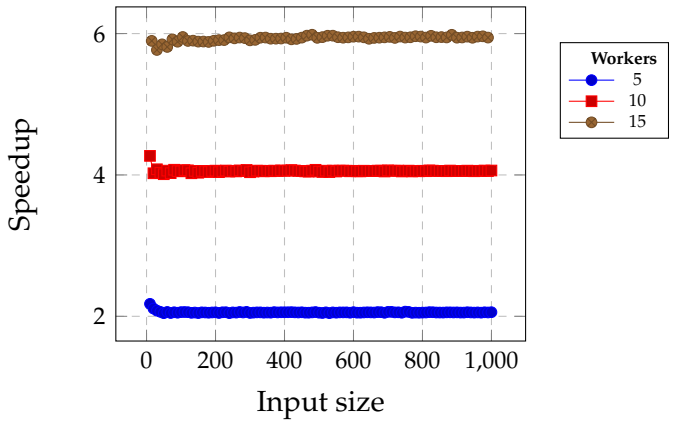


FIGURE 5.9: Synthetic Example software only vs alternative task farm implementation with varying worker number

```
void executor(float* input, float* result) {
     for (int i = 0; i < FARM_CAPACITY; i++) {
        #pragma HLS PIPELINE
        #pragma HLS UNROLL

        result[i] = sinf(input[i]);
    }
}

/* Data copy interface */
void farm(DTYPE* input, DTYPE* result) {
    #pragma HLS DATAFLOW

    float l_input[FARM_CAPACITY];
    float l_result[FARM_CAPACITY];

    #pragma HLS array_partition variable=l_input complete
    #pragma HLS array_partition variable=l_result complete

    data_loader(input, l_input);

    executor(l_input, l_result);

    data_writer(result, l_result);
}
```

FIGURE 5.10: Alternative data copy task farm implementation

Workers	Input size	Speedup (Data Copy)	Speedup (Shared Memory)
5	10	2.18	0.98
10	10	4.27	1.85
15	15	5.89	2.70

TABLE 5.2: Speedup comparison between different task farm implementation strategies

Furthermore, compared to the shared memory implementation, using the data copy approach, we achieved higher speedups with small input sizes. Especially, when the capacity of the farm matched the input size, as portrayed in Table 5.2. For example, we can see that with 15 workers in the farm and an input size of 15 we achieved an execution speedup of 5.89. While, previously with the shared memory model the achieved speedup was 2.70 using a single worker with same input size. This gives a 2.18 relative speedup increase between the two approaches. On the other hand, the increase in performance comes at a cost in terms of the hardware resources required. Table 5.3 shows the resource usage report for the data copy model of task farm pattern with 15 workers. Compared to the FPGA resource requirements of a single worker using the shared memory model, we see a significant increase in terms of the resources needed to implement the farm in the programmable fabric. An

increased resource utilisation leads to a higher energy consumption. However, for a small input size the speedup advantage gained by using the data copy approach might in fact lead to energy savings, due to a smaller total execution time.

Resource	Used	Total	% Utilisation
DSP	315	360	87.5
BRAM	30	432	6.9
LUT	45906	70560	65
FF	13531	141120	9.6

TABLE 5.3: Estimated resource utilisation using the data copy task farm implementation

There seems to be multiple approaches to implementing the task farm parallel pattern on the programmable fabric. Each approach has its own advantages and drawbacks and performs better in certain situations than others. Two key factors seem to influence the performance gains the most: the data transfer strategy and the worker implementation model on the programmable fabric. Therefore, a particular implementation should be chosen depending on application's execution context, its data excess patterns and the specific hardware used.

5.2 Discussion

Our analysis suggest that there are potential benefits for applying the task farm parallel pattern in an FPGA-based MPSoC environment to accelerate sequential computations. The use case evaluation results show that a task farm pattern model can be successfully used to improve DFT calculations. Furthermore, even in a scenario where multiple workers can not be fitted to the programmable logic due to lack of resources available, our task description model used with a single accelerator/worker can be useful in providing improved performance on the programmable fabric, as observed with the Black Scholes use case example.

The pattern is implemented using the SDSoC development environment and compiler directives to drive the high-level synthesis process. The infrastructure of the pattern is defined by a number of function calls and could be generated by a meta-programmer before the compilation process. Therefore, it seems that it would be possible to implement the pattern as a higher-order function. The user would only need to provide a task description, inputs and outputs and parameters for the execution of the pattern and the compiler would be able to generate the infrastructure an ensure a correct execution of the tasks. This would allow the user to be oblivious to the underlying hardware implementation details and still achieve an improved performance.

On the other hand, as with almost any abstraction, the advantages gained in one area will require compromise in another. In our case, hiding the underlying hardware implementation details from the user limits his ability to optimise the accelerator for an even better performance gain. Furthermore, the user has to transform his algorithm to fit our task description model and that might not be possible to do in all use

cases. Moreover, our current implementation of the pattern, simply replicates the synthesised circuit of a task and therefore uses a significant amount of resources on the FPGA, which could lead to an increased energy consumption. Nonetheless, the development productivity gained from using the pattern might outweigh the drawbacks, especially for a developer that does not have the required background. For such user, the task farm pattern would not only save a lot of development time and result in improved performance, but might also lead to energy savings, as alternatively such user would simply use a CPU-only based solution which would have to execute for a longer time and potentially consume more energy in total.

There seems to be multiple ways in which the task farm pattern can be implemented. Each approach has its own advantages and disadvantages. The key factor in achieving performance seems to lie in the selection of the data motion network. There is a transfer of data cost associated with offloading computations to the programmable fabric. Therefore, an efficient implementation should manage to amortise that cost. Another important factor and potential place for bottleneck seems to be the distribution of tasks between the workers. In our analysis, we used the farmer, implemented on the processing system's side, to launch the workers and divide the tasks. This approach requires synchronisation after each work cycle and a better alternative might be to use a decentralised farm. In such scenario, the workers would act independently and could *compete* with each other for tasks. The competition mechanism could be implemented using a lock-less queue to minimise the cost of multiple workers choosing the same task. Furthermore, it might be possible to have different types of workers. For example, a heterogeneous task farm could have CPU and FPGA workers. Each worker would be able to determine if a particular task is best suited for his type and would only chose tasks that he can handle best, this way removing the need of a centralised scheduler.

Moreover, with improving speed of dynamic reconfiguration, the workers could be reconfigured during runtime to perform a range of different tasks. The SDSoC development environment already provides such capabilities, therefore it would be possible to extend our implementation to allow the use of the task farm with multiple types of tasks in a single application. The reconfiguration time penalty could potentially be amortised by speculatively reconfiguring the FPGA before the start of computations or during software execution on the CPU.

Finally, instead of reconfiguring the whole programmable fabric, it might be possible to use a set of *architectural skeletons* that would act as the basic hardware infrastructure for transferring, distributing and storing the data on the FPGA, while the actual execution could be performed by a custom reconfigurable worker unit. Such skeletons could replicate the architectural structures of vector processors, GPUs or other parallel computing devices and gain the advantages provided by their execution models. At the same time, still maintaining the benefits of the programmable fabric.

Chapter 6

Conclusions

We introduced the advantages and drawbacks of developing applications for FPGA-based MPSoC enviroments and proposed the use of task farm parallel pattern as a solution to mitigate the current most prevalent challenges present. We implemented the task farm pattern using the SDSoC development environment combined with the high-level synthesis tools and tested its effectiveness as an abstraction over the accelerator development and deployment process using the Ultra96 development board with Xilinx UltraScale+ architecture based programmable fabric. Furthermore, we evaluated our solution using two use case examples: Discrete Fourier Transforms algorithm and Black Scholes model calculations. Our experimental results showed a significant performance improvement for both cases. Using the task farm pattern, we managed to achieve speedups up to 20.16 times with DFT algorithm and up to 8.8 times with the Black Scholes model under certain experimental conditions. Furthermore, we investigated the scalability of the pattern using a synthetic application example, introduced different task farm pattern's implementation strategies and their potential application scenarios and discussed key factors that determine successful acceleration of computations on the programmable logic. Finally, we argued that our task farm parallel pattern implementation could be turned into a high-order function and successfully used by developers to efficiently and productively accelerate sequential computations in a heterogeneous FPGA-based MPSoC environments.

In the future, we would like to deliver a more in depth analysis, using a range of use case examples with different data access patterns, to determine how does the task farm parallel pattern perform when being used as part of a varied scope of applications and which implementation strategies would be most suitable for a given situation. Furthermore, the current implementation uses hardware and development tools produced by a single manufacturer. Therefore, it would be useful to investigate the benefits and tradeoffs of using the task farm pattern with different tools, environments and hardware producers. Finally, as shown by our analysis, the data transfer strategy significantly affects the achievable performance of the task farm pattern. In our implementation, we used the data movers provided as part of the SDSoC development environment. However, given the importance of effective data transfer, we would like to research the development of custom hardware components that could transfer the data form the processing system to the programming logic and distribute the tasks among the workers in the most effective manner.

Appendix A

Additional Measurements

A.1 Discrete Fourier Transforms

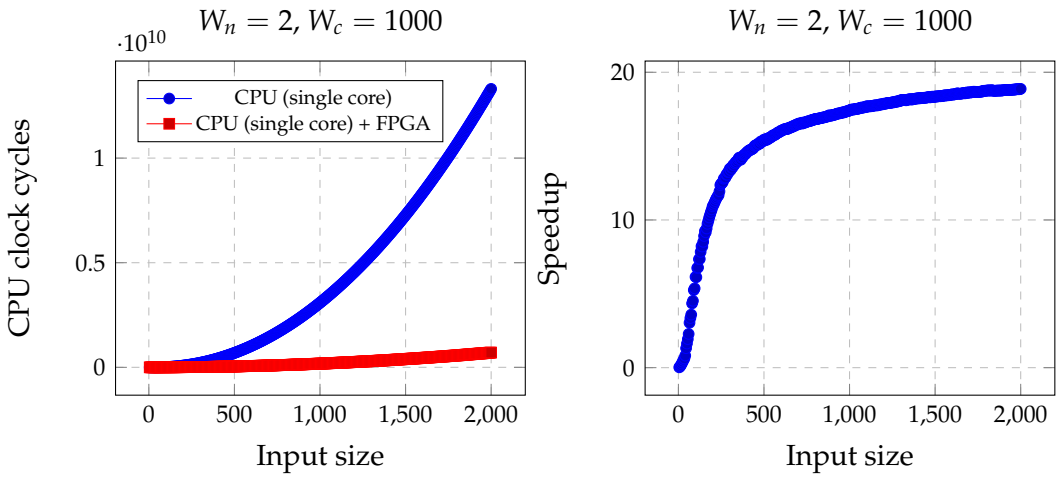


FIGURE A.1: Discrete Fourier Transforms software only vs improved stream-based task farm implementation

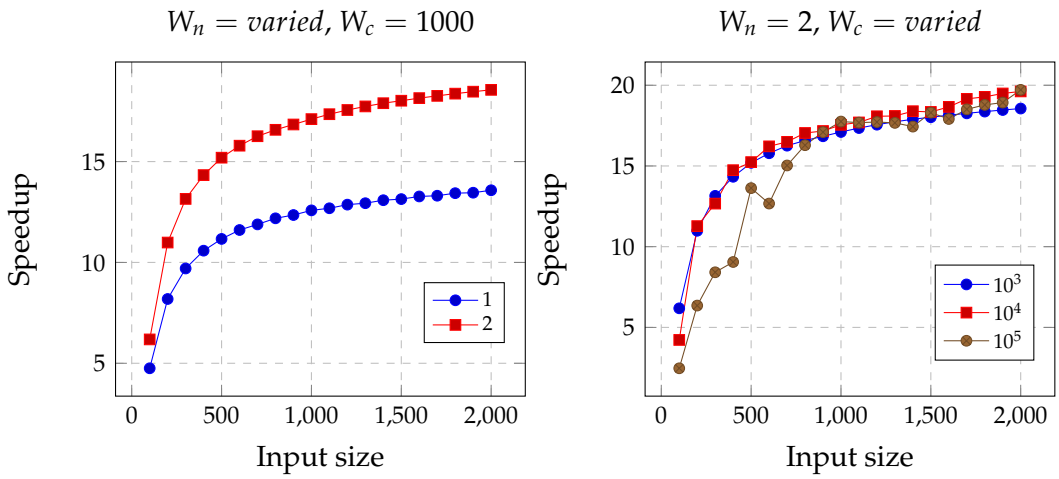


FIGURE A.2: Discrete Fourier Transforms software only vs improved stream-based task farm with varying worker number and capacity

A.2 Black Scholes Model

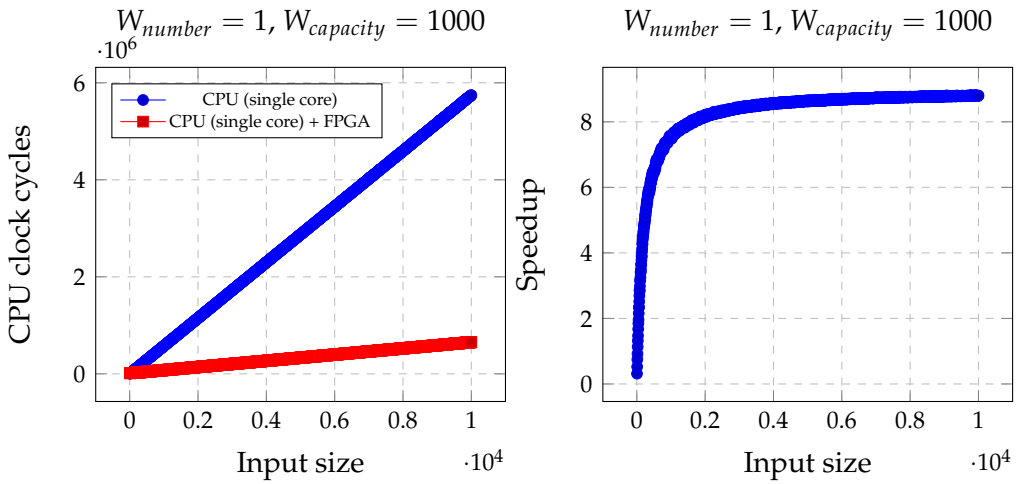


FIGURE A.3: Black Scholes software only vs improved data copy task farm implementation

Bibliography

[1] *7 Series FPGAs Configurable Logic Block User Guide*. UG474. v1.8. Xilinx. Sept. 2016.

[2] Christian Bienia et al. 'The PARSEC Benchmark Suite: Characterization and Architectural Implications'. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. PACT '08. Toronto, Ontario, Canada: ACM, 2008, pp. 72–81. ISBN: 978-1-60558-282-5. DOI: 10.1145/1454115.1454128. URL: http://doi.acm.org/10.1145/1454115.1454128.

[3] Michael Butts et al. 'Reconfigurable Work Farms on a Massively Parallel Processor Array'. In: *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines*. FCCM '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 206–215. ISBN: 978-0-7695-3307-0. DOI: 10.1109/FCCM.2008.6. URL: https://doi.org/10.1109/FCCM.2008.6.

[4] Bruce Dawson. *Comparing Floating Point Numbers, 2012 Edition*. URL: https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/ (visited on 06/01/2019).

[5] The Tech Terms Dictionary. *SoC Definition*. URL: https://techterms.com/definition/soc (visited on 26/12/2018).

[6] Suhaib A. Fahmy, Kizheppatt Vipin and Shanker Shreejith. 'Virtualized FPGA Accelerators for Efficient Cloud Computing'. In: *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*. CLOUDCOM '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 430–435. ISBN: 978-1-4673-9560-1. DOI: 10.1109/CloudCom.2015.60. URL: http://dx.doi.org/10.1109/CloudCom.2015.60.

[7] *GPU vs FPGA Performance Comparison*. Tech. rep. Berten, 2016.

[8] Ian Graves et al. 'Hardware Synthesis from Functional Embedded Domain-Specific Languages: A Case Study in Regular Expression Compilation'. In: *Applied Reconfigurable Computing*. Ed. by Kentaro Sano et al. Cham: Springer International Publishing, 2015, pp. 41–52. ISBN: 978-3-319-16214-0.

[9] John Hennessy and David Patterson. *Computer Architecture. A Quantitative Approach*. 6th Edition. Morgan Kaufmann, 2017.

[10] Mike Hutton. *Understanding How the New Intel® HyperFlex™ FPGA Architecture Enables Next- Generation High-Performance Systems*. Tech. rep.

[11] *Introduction to FPGA Design with Vivado HLS*. UG998. v1.0. Xilinx. July 2013.

[12] Lana Josipovic, Nithin George and Paolo Ienne. 'Enriching C-based High-Level Synthesis with parallel pattern templates'. In: *2016 International Conference on Field-Programmable Technology, FPT 2016, Xi'an, China, December 7-9, 2016*. 2016, pp. 177–180. DOI: 10.1109/FPT.2016.7929527. URL: https://doi.org/10.1109/FPT.2016.7929527.

[13] Norman P. Jouppi et al. 'In-Datacenter Performance Analysis of a Tensor Processing Unit'. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 1–12. ISBN:

978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. URL: http://doi.acm.org/10.1145/3079856.3080246.

[14] Muhammed Al Kadi et al. 'Dynamic and partial reconfiguration of Zynq 7000 under Linux'. In: *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)* (2013), pp. 1–5.

[15] R. Kastner, J. Matai and S. Neuendorffer. 'Parallel Programming for FPGAs'. In: *ArXiv e-prints* (May 2018). arXiv: 1805.03648.

[16] Vinod Kathail et al. 'SDSoC: A Higher-level Programming Environment for Zynq SoC and Ultrascale+ MPSoC'. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '16. Monterey, California, USA: ACM, 2016, pp. 4–4. ISBN: 978-1-4503-3856-1. DOI: 10.1145/2847263.2847284. URL: http://doi.acm.org/10.1145/2847263.2847284.

[17] Huan Li and Wenhua Ye. 'Efficient implementation of FPGA based on Vivado High Level Synthesis'. In: *2016 2nd IEEE International Conference on Computer and Communications (ICCC)* (2016). DOI: 10.1109/CompComm.2016.7925210.

[18] Ann Steffora Mutschler. *FPGAs Becoming More SoC-Like*. URL: https://semiengineering.com/fpgas-becoming-more-soc-like/ (visited on 16/01/2019).

[19] Razvan Nane et al. 'A Survey and Evaluation of FPGA High-Level Synthesis Tools'. In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 35.10 (Oct. 2016), pp. 1591–1604. ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2513673. URL: https://doi.org/10.1109/TCAD.2015.2513673.

[20] Project Nayuki. *How to implement the discrete Fourier transform*. URL: https://www.nayuki.io/page/how-to-implement-the-discrete-fourier-transform (visited on 17/01/2019).

[21] Michael Poldner and Herbert Kuchen. 'On Implementing the Farm Skeleton'. In: *Parallel Processing Letters* 18.1 (2008), pp. 117–131.

[22] Raghu Prabhakar et al. 'Generating Configurable Hardware from Parallel Patterns'. In: *SIGOPS Oper. Syst. Rev.* 50.2 (Mar. 2016), pp. 651–665. ISSN: 0163-5980. DOI: 10.1145/2954680.2872415. URL: http://doi.acm.org/10.1145/2954680.2872415.

[23] Raghu Prabhakar et al. 'Plasticine: A Reconfigurable Architecture For Parallel Paterns'. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 389–402. ISSN: 0163-5964. DOI: 10.1145/3140659.3080256. URL: http://doi.acm.org/10.1145/3140659.3080256.

[24] J. Rettkowski et al. 'LinROS: A Linux-Based Runtime System for Reconfigurable MPSoCs'. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Vol. 00. 2016, pp. 208–216. DOI: 10.1109/IPDPSW.2016.156. URL: doi.ieeecomputersociety.org/10.1109/IPDPSW.2016.156.

[25] *SDSoC Environment Profiling and Optimization Guide*. UG1235. v2018.1. Xilinx.

[26] *SDSoC Environment Profiling and Optimization Guide*. UG1235. v2018.2. Xilinx. July 2018.

[27] *SDSoC Environment User Guide*. UG1027. v2017.4. Xilinx. Jan. 2018.

[28] *SDSoC Profiling and Optimization Guide*. UG1235. v2018.2. Xilinx. July 2018.

[29] *SDx Pragma Reference Guide*. UG1253. v2018. Xilinx. July 2018.

[30] *SDx Programmers Guide*. UG1278. v2018.2. Xilinx. July 2018.

[31] Inc. Stephen M. Nolan Vidatronic. *Power Management for Internet of Things (IoT) System on a Chip (SoC) Development*. URL: https://www.design-reuse.com/articles/42705/power-management-for-iot-soc-development.html (visited on 26/12/2018).

[32] The Economic Times. *Definition of 'Black-scholes Model'*. URL: https://economictimes.indiatimes.com/definition/black-scholes-model (visited on 08/01/2019).
[33] *Ultra96: Hello World*. Avnet. Feb. 2018.
[34] *Ultra96 Product Brief*. 5354. v3b. Avnet.
[35] *UltraScale Architecture and Product Data Sheet: Overview*. DS890. v3.6. Xilinx. Nov. 2018.
[36] Frank Vahid. *Digital Design with RTL Design, Verilog and VHDL*. 2nd. Wiley Publishing, 2010. ISBN: 0470531088, 9780470531082.
[37] *Vivado Design Suite User Guide High-Level Synthesis*. UG902. v2017.1. Xilinx. Apr. 2017.
[38] *Xilinx UltraScale Architecture for High-Performance, Smarter Systems*. WP434. v1.2. Xilinx. Oct. 2015.