Technical University of Crete

School of Electronic & Computer Engineering

Live Video Stream Processing on a Maxeler Reconfigurable Computer

Author:

Argyrios – Alexandros Gardelakos

Thesis Committee:

Professor Apostolos Dollas (supervisor)

Professor Dionisios Pnevmatikatos

Professor Michalis Zervakis

Chania 2015

Στις δύο Α.

Acknowledgements

First of all, I would like to thank my supervisor, Professor Apostolos Dollas for believing in me and his invaluable help and guidance throughout my thesis. Also, I would like to thank my family for being so understanding and supportive. Last but not least, I would like to thank all of my friends at Chania for every moment we spent together.

Abstract

The dataflow architecture is a computer architecture that directly contrasts the traditional von Neumann architecture. It does not have a program counter, and the execution of the instructions is solely determined based on the availability of input arguments. It is not a newly emerged concept but with the recent development of integrated systems, like the Maxeler's Series, a new era at dataflow computing has begun.

Considering the massive growth of internet, the Live Stream or in general the Data Stream has never been more crucial for the scientific and the industrial field. Astronomy, weather forcasting and networks are just a small sample of examples. But, even for all of us, everyday people, the use of data stream grow exponentially in our daily life. Satellite TV, gaming and mobile phones are just the tip of the iceberg.

In this thesis, a complete system that can stream, edit and present to the final user video feed was developped, using the dataflow architecture. Also, a series of experiments were conducted in order to present the capabilities of the Maxeler MPC – C Series system and a library for network connection with the TCP/IP and UDP protocols was implemented, for future reference to all that may be interested.

List of Figures

List of Tables

Contents

1. Introduction

This Chapter begins with an introduction to FPGAs, the fundamental component of this thesis. Following, there is a brief analysis on what a data stream is and what it means for the system that was develloped. After that, I attempt to make a brief comparison between the conventional Von Neumann architecture and the Dataflow architecture. Finally, the purpose and the outline of this thesis are presented.

1.1 Field Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) are programmable semiconductor devices that are based around a matrix of Configurable Logic Blocks (CLBs) connected through programmable interconnects. As opposed to Application Specific Integrated Circuits (ASICs), where the device is custom built for the particular design, FPGAs can be programmed to the desired application or functionality requirements. Although One – Time Programmable (OTP) FPGAs are available, the dominant type are SRAM – based which can be reprogrammed as the design evolves.

FPGAs allow designers to change their designs very late in the design cycle – even after the end product has been manufactured and deployed in the field.

The FPGA configuration is generally specified using a hardware description language (HDL), such as Verilog and VHDL.

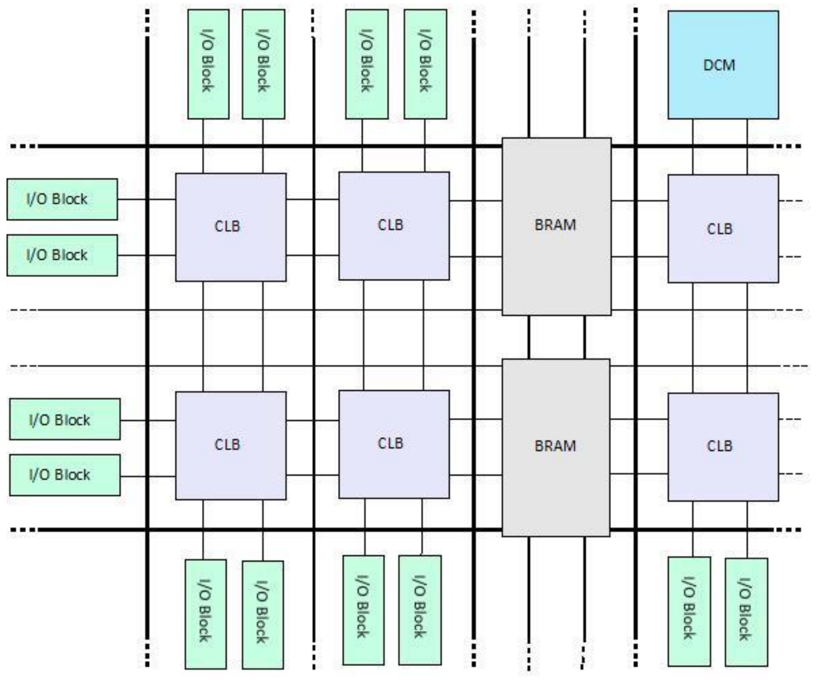

Figure 1: A typical FPGA block structure

FPGAs have evolved far beyond the basic capabilities present in their predecessors, and incorporate hard (ASIC type) blocks of commonly used functionality such as RAM (Random Access Memory), clock management, DSPs (Digital Signal Processors), CLBs (Configurable Logic Blocks) and hard peripherals, such as PCIe or Gigabit Ethernet. Common FPGA applications can be found in the fields of Aerospace [1], Medical Electronics [2], Datacenters [3] and High Performance Computing (HPC) [4].

1.2 Data Stream

A general definition of the data stream is, “a sequence of digitally encoded singals used to represent information in transmission”. As it is a rather vague definition, in this thesis I treated the data stream as input data that comes at a very high rate. High rate means it stresses communication and computing infrastructure, so it may be hard to

- Transmit (T) the entire input to the program,
- Compute (C) sophisticated functions on large pieces of input at the rate it is presented, and
- Store (S), capture temporarily or archive all of it long term.

When data is transmitted, if the links are slow or the communication is erroneous, we may have delays but correct data eventually gets to where it should go (at least for the TCP/IP protocol). If computing power is limited or the program has high complexity, it take quite a lot of time to get the desired respones, but in principle, we will get it. Also, we usually save in some form all the data that we need.

Recently, there have been two major developments that have led us to produce new challenges to the TCS infrastructure.

First, the ability to generate automatic, highly detailed data feeds comprising of continuous updates.

This ability has been built up over tha past few decades beginning with networks that were required for banking and credit transactions. But it has evolved well beyond that now, with some examples being satellite based high resolution measurements of earth geodetics [5], radar derived meteorological data [6] and continuous large scale astronomical surveys in optical, infrared [7] and radio [8] wavelengths. The Internet is a general purpose network system that has distributed both the data sources as well as the data consumers over millions of users. It has scaled up the rate of transactions tremendously, generating multiple streams: browser clicks, user queries, IP traffic logs, email data and traffic logs and web server and peer-to-peer downloads, to name a few.

Second, the need to be able to do sophisticated analyses of update streams in near-real time manner.

With traditional datafeeds, one modifies the underlying data to reflect the updates, and real time queries are fairly simple such as looking up a value. This is true for the banking and the credit transactions. More complex analyses, such as trend analysis [9] and forecasting [10] are typically performed offline in data warehouses. However, the automatic data feeds that generate modern data streams arise out of monitoring applications, for example, in astronomy [8], weather forecasting [6], networks [3], sensor – related [4] and surveillance [11]. They need to detect outliers, extreme events, intrusion and unusual or anomalous activity. These are time critical tasks and they need to be completed in near-real time to accurately keep pace with the rate of stream updates and accurately reflect rapidly changing trends in the data.

There are three major Data Stream Models [12]:

1. The Time Series Model. Each input stream arrive sequentially, item by item, in order and it is independent of previous values. This is a suitable model for time series data. For example, in observing the traffic at an IP link every 5 minutes.
2. Cash Register Model. Each input stream arrive sequentially, item by item, in order but it depends on previous values. This is perhaps the most popular data stream model. It fits applications such as monitoring IP addresses that accesses a web server. The same IP addresses may access the web server multiple times or send multiple packets on the link over time.
3. The Turnstile Model [13]. Each input stream arrive sequentially, item by item, in order and it is an update to a previous value. This is the most general model. It is the appropriate model to study fully dynamic situations where there are inserts as well deletes.

Many workable systems exist nowadays that deal with these TCS challenges. The common denominator of all these systems is the parallelism. A lot of data stream processing is highly parallelizable in Computing (C) and Storage (S) capacity, but it is harder to parallelize the Transmission (T) capacity on demand.

1.3 A brief comparison between the Von Neumann and the Dataflow Architecture

The Von Neumann architecture is a computer architecture based on that described in 1945 by John von Neumann. It is a design architecture for an electronic digital computer with parts consisting of a processing unit containing an arithmetic logic unit and processor registers, a control unit containing an instruction register and program counter, a memory to store both data and instructions, external mass storage, and input and output mechanisms.

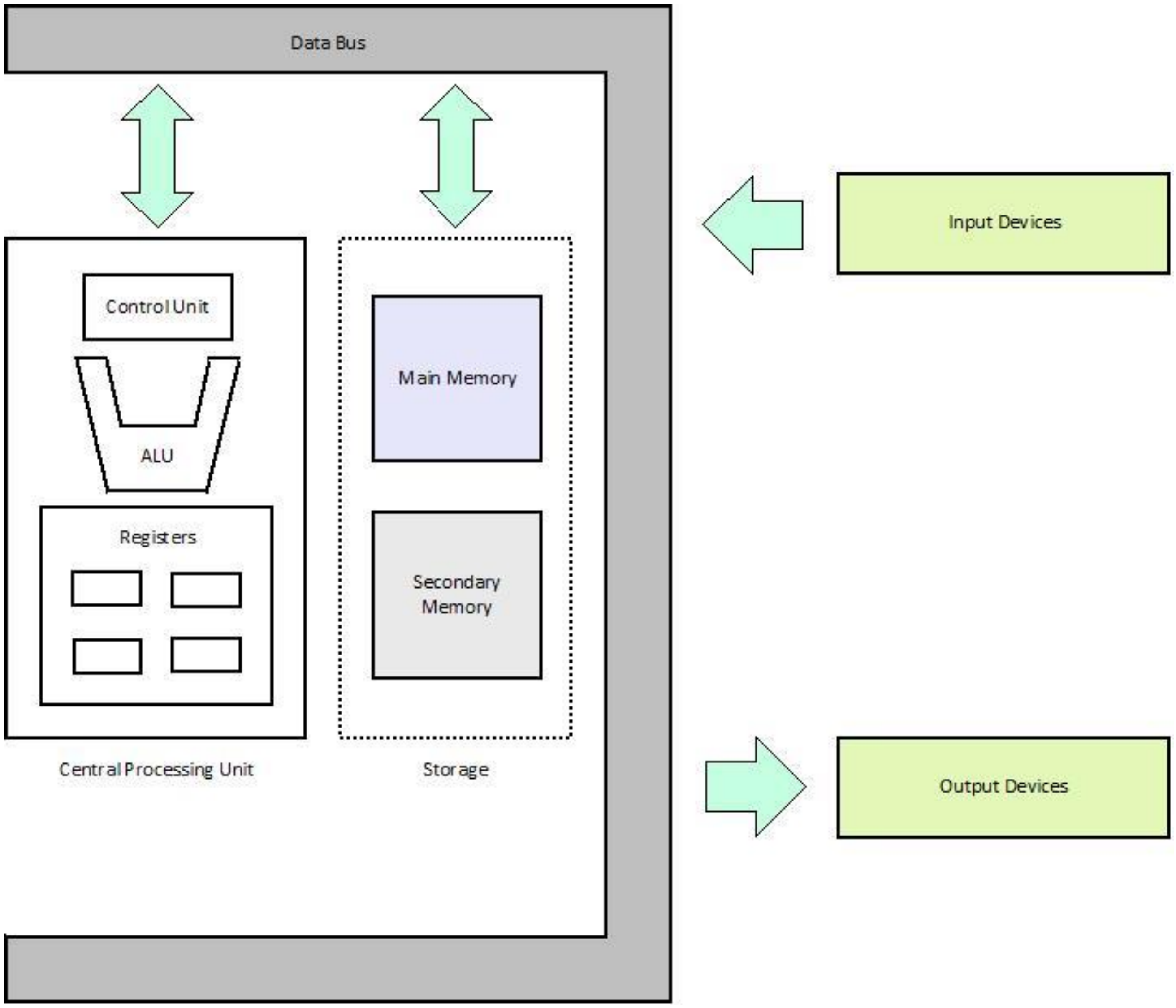

Figure 2: Von Neumann Architecture Scheme

Over the years it has evolved to be any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the Von Neumann bottleneck and often limits the performance of the system. For many, the answer to this bottleneck is parallelism [14]. Another problem with this architecture is the lack of useful mathematical properties which complicates tasks as program verification. Thus, some others architectures have emerged trying to face this problem. One of them is the Dataflow architecture.

The Dataflow architecture is a computer architecture that directly contrasts the traditional von Neumann architecture. Dataflow architectures do not have a program counter and the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable, meaning that the behavior is indetermenistic.

In the Dataflow model of computation, there are two major principles that make it different than the conventional Controlflow model. First, the enabling rule for an instruction to be executed is one; an instruction is executable if all input data are available, so it is an asynchronous model. Therefore, the synchronization of parallel activities is implicit in the Dataflow model. This contradicts the von Neumann model in which the instruction is executable only when it is pointed by the program counter. Second, the instructions in the dataflow model do not impose any constraints on sequencing except the data dependencies in the program. Hence, the Dataflow graph representation of a program exposes all forms of parallelism eliminating the need to explicity manage parallel execution of a program. Any two enabled instructions can be executed in either order or concurrently. This implicit parallelism is achieved by allowing side-effect free expressions and functions to be evaluated in parallel. In a Dataflow environment, conventional language concepts such as "variables" and "memory updating" are non-existent. Instead, objects (data structures or scalar values) are consumed by an instuction yielding a result object which is passed to the next instruction. The same object can be supplied to different functions at the same time without any possibility of side-effects.

The basis of dataflow computation is the dataflow graph. In dataflow computers, the machine level language is represented by dataflow graph. The dataflow graphs consist of nodes and arcs. The basic primitives of the dataflown graph are shown in the image below.

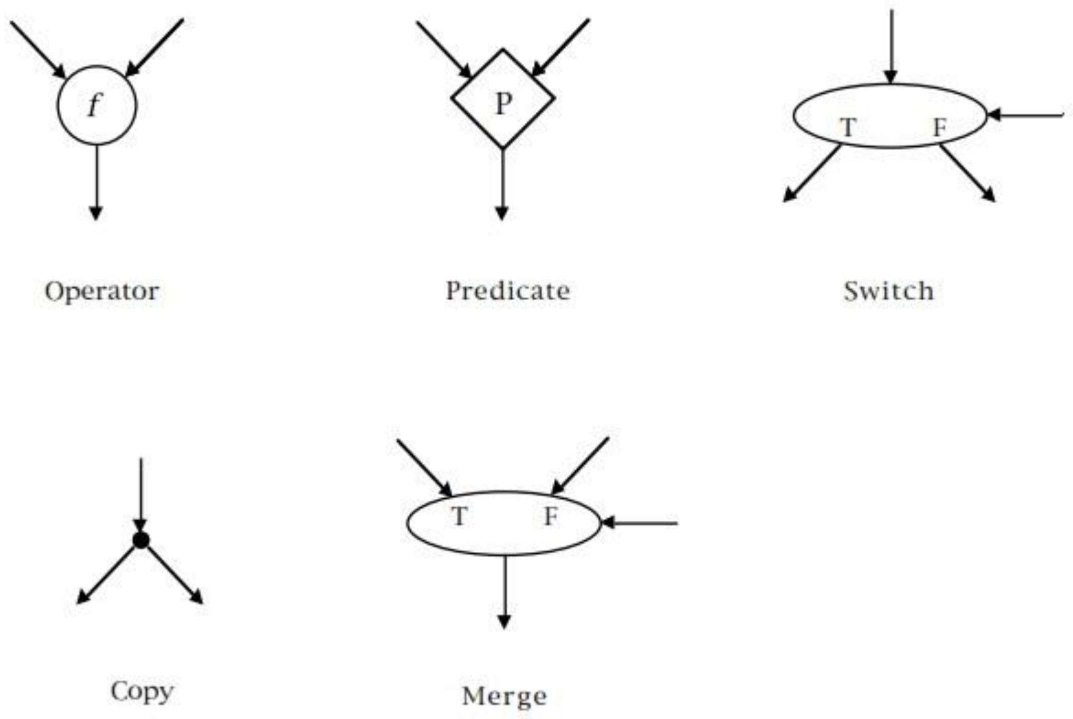

Figure 3: Primitives of a Dataflow Graph

A data value is produced by an Operator as a result of some operation, f. A true or false control value is generated by a Decider depending on its input tokens. Data values are directed by means of either a Switch or a Merge actor. For example, a Switch actor directs an input data token to one of its outputs depending on the control input. Similarly, a Merge actor passes one of its input tokens to the output depending on the input control token. Finally, a Copy is an idenity operator which duplicates input tokens.

There is a special need to provide a high-level language for dataflow computers since their machine language, the dataflow graph, is not an appropriate programming medium. They are error-prone and difficult to manipulate. There are basically two high-level language classes which have been considered by dataflow researchers [15]. The first is the imperative class. The aim is to map existing conventional languages to directed graphs using dataflow analysis often used in optimizing compilers. For example, the Texas Instruments group used a modified Advanced Scientific Computer (ASC) FORTRAN compiler for their dataflow machine [16]. But, programming dataflow computers in conventional imperative languages would tend to be inefficient since they are by their nature sequential. Therefore, a number of dataflow or applicative languages has been developed to provide a more efficient way of expressing parallelism in a program. The goal is to develop high-level dataflow languages which can express parallelism in a program more naturally and to facilitate close interaction between algorithm constructs and hardware structures. Examples of dataflow languages include the Value Algorithmic Language (VAL) proposed by the MIT [17], the Irvine Dataflow language (Id) proposed by the University of California [18], the Stream and Iteration in a Single-Assignment Language (SISAL) proposed by the Lawrence Livermore National Laboratories [19], and the MaxCompiler created and developped by Maxeler Technologies [20].

The image below demonstrates the differences in computing with the conventional control flow architecture against the Maxeler's dataflow architecture.

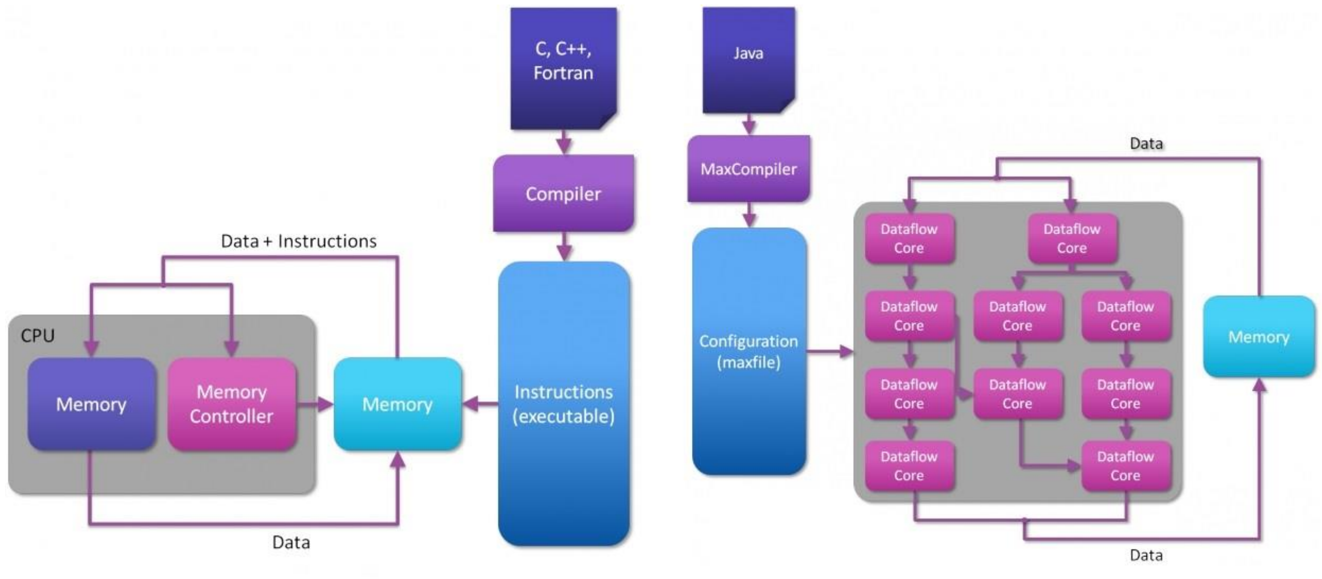

Figure 4: Computing with a Control Flow core and computing with Dataflow cores [21]

1.4 Thesis' Purpose and Outline

The purpose of this thesis is to present the tremendous advantages that the Maxeler MPC – C Series has as a real time coprocessor in certain computations. A series of experiments have been conducted in order to determine all the vital information that are required for such a system to be viable. Also, as a test case I have implemented a system which is capable of applying five filters to a live stream in near-real time. As it will be proven in the next chapters, the only real limiting factor is the bandwidth of the internet connection. Finally, a library for creating network sockets, TCP/IP and UDP protocols, was implemented in order to assist anyone interest on the topic.

This thesis is organized as follows. Chapter 2 demonstrates related work that has already been done in FPGAs as coprocessors. Chapter 3 sets all the issues that this thesis is about. Chapter 4 describes the entire functionality of the system that was develloped. Chapter 5 provides information about the performance of the Maxeler MPC – C Series system, the internet bandwidth of TUC, the system I developped and verification of the correct functionality of this system. Finally, Chapters 6 and 7 are the conclusion and the references respectively.

2. Related Work on FPGAs

In this Chapter first the definion of the Coprocessor is attempted and then various, small and large scale, examples of FPGA coprocessors are presented. Finally, a project using FPGAs, that attempts to remotely offload workload is presented.

2.1 Coprocessor

A coprocessor is a computer processor used to supplement the functions of the primary processor (the CPU). Operations performed by the coprocessor may be floating point arithmetic, graphics, signal processing, string processing, encryption or I/O Interfacing with peripheral devices. By offloading processor-intensive tasks from the main processor, coprocessors can accelerate system performance. Coprocessors allow a line of computers to be customized, so that customers who do not need the extra performance need not pay for it.

A coprocessor may not be a general-purpose processor in its own right. Coprocessors cannot fetch instructions from memory, execute program flow control instructions, do input/output operations, manage memory, and so on. The coprocessor requires the host (main) processor to fetch the coprocessor instructions and handle all other operations aside from the coprocessor functions. In some architectures, the coprocessor is a more general-purpose computer, but carries out only a limited range of functions under the close control of a supervisory processor.

2.1.1 FPGAs as Coprocessor

Virtually all applications that perform signal or video processing seek levels of performance beyond what a traditional single-core processor can deliver. A variety of devices are available to address this performance gap, including multicore DSPs, Graphics Processing Units(GPUs), application-specific Standard Products(ASSPs) and FPGA co-processing platforms that feature DSP-optimized programmable hardware.

The full extent of pros and cons of all the proposed solutions is a vast topic in academia and is well beyond the purpose of this thesis, so we should focus on FPGA as coprocessor.

2.1.1.1 Small scale examples

Modern FPGA families include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed I/O logic and embedded memories. Higher-end FPGAs can contain high speed multi-gigabit transceivers and hard IP cores such as processor cores, Ethernet MACs, PCI/PCI Express controllers, and external memory controllers. These cores exist alongside the programmable fabric, but they are built out of transistors instead of LUTs so they have ASIC level performance and power consumption while not consuming a significant amount of fabric resources, leaving more of the fabric free for the application-specific logic. The multi-gigabit transeivers also contain high performance analog input and output circuitry along with high-speed serializers and deserializers, components which cannot be built out of LUTs. Additionally, the FPGA cards provide the ability to support FPGA Mezzanine Cards (FMC). It is a flexible, modular I/O interface to an FPGA located on a host system baseboard or carrier card.

The below image is a standard example of a PCIe FPGA card

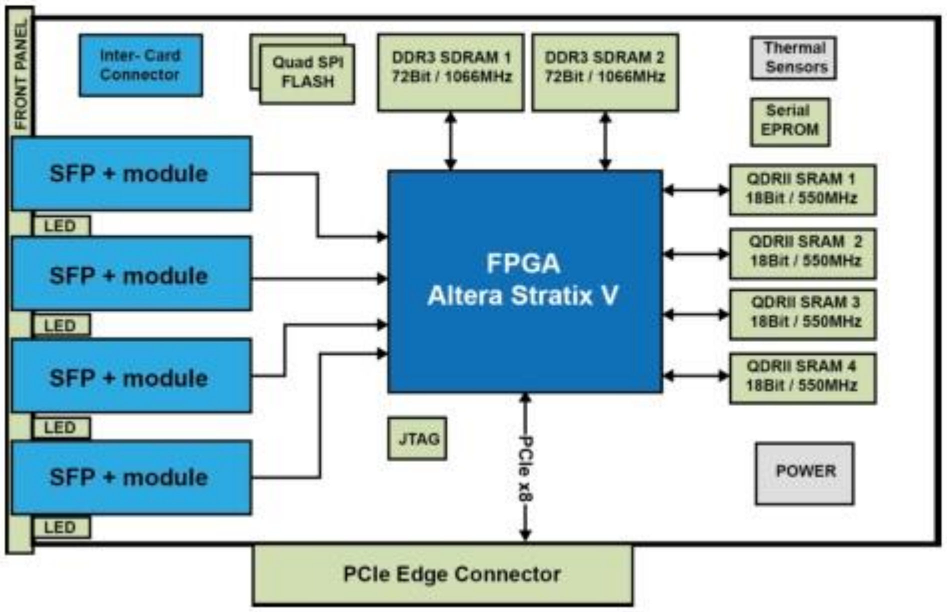

Figure 5: Example architecture of a FPGA card: V5031 Quad Channel 10GbE FPGA PCI Express Card Architecture [22]

2.1.1.2 Large Scale Example 1: Convey

Convey Hybrid Core systems utilize a commodity motherboard that includes an Intel 64 host processor and standard Intel I/O chipset, along with a reconfigurable coprocessor based on FPGA technology. This coprocessor can be reloaded dynamically while the system is running with instructions that are optimized for different workloads. It includes its own high bandwidth memory subsystems that is incorporated into the Intel global memory space, creating the Hybrid-Core Globally Shared Memory (HCGSM). Coprocessor instructions can be thought of as extensions to the Intel instruction set – an executable can contain both Intel and coprocessor instructions and those instructions exist in the same virtual and physical address space.

The coprocessor supports multiple instruction sets (referred to as "personalities"). Each personality includes a base set of instructions that are common to all personalities. This base set includes instructions that perform scalar operations on integers and floating point data, address computations, conditionals and branches, as well as miscellaneous control and status operations. A personality includes a set of extended instructions that are designed for a particular workload. The extended instructions for a personality designed for signal processing, for instance, may implement a SIMD model and include vector instructions for 32-bit complex arithmetic.

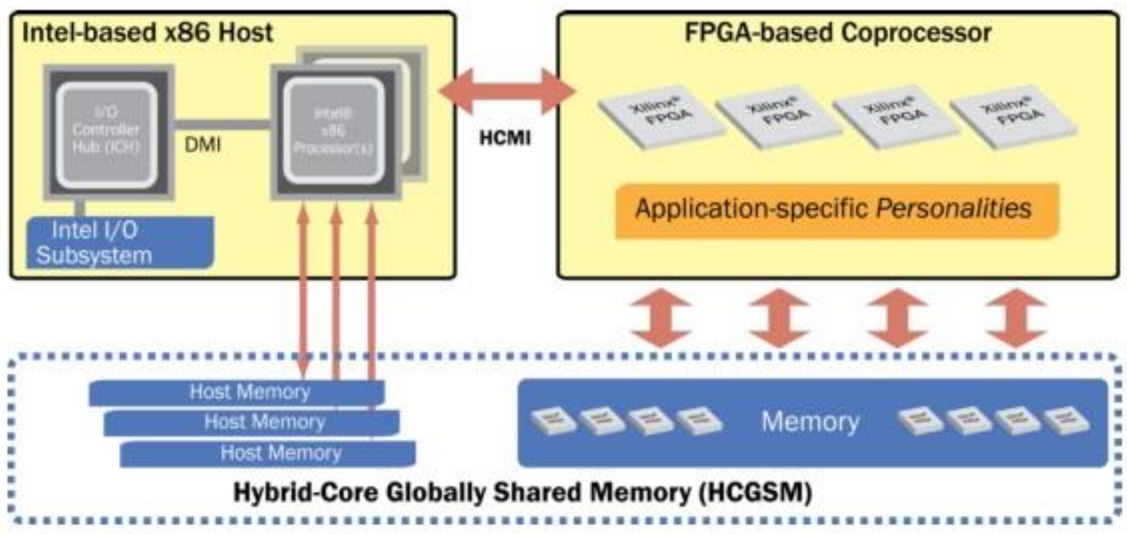

Figure 6: Convey Hybrid Core System Diagram[23]

In order to delve into the coprocessor a bit more, it is comprise of three major sets of components, the Application Engine Hub (AEH), the Memory Controllers (MCs) and the Application Engines (AEs).

The AEH is the central hub for the coprocessor. It implements the interface to the host processor and to the Intel I/O chipset, fetches and decodes instructions, and executes scalar instructions. It processes coherence and data requests from the host processor, routing requests for addresses in coprocessor memory to the MCs. Scalar instructions are executed in the AEH, while extended instructions are passed to the AEs for execution.

In order to support the bandwidth demands of the coprocessor, 8 Memory Controllers support a total of 16 DDR2 memory channels, providing an aggregate of over 80 GB/sec of bandwidth to ECC protected memory. The MCs translate virtual to physical addresses on behalf of the AEs, and include snoop filters to minimize snoop traffic to the host processor. The MCs support standard DIMMs as well as Convey designed Scatter-Gather DIMMs. The Scatter-Gather DIMMs are optimized for transfers of 8-byte bursts, and provide near peak bandwidth for non-sequential 8-byte accesses. The coprocessor therefore not only has a much higher peak bandwidth than is available to commodity processors, but also delivers a much higher percentage of the peak for non-sequential accesses.

Together the AEH and the MCs implement features that are present in all personalities. This ensures that important features such as memory protection, access to coprocessor memory, and communication with the host processor are always available.

The AEs implement the extended instructions that deliver performance for a personality. The AEs are connected to the AEH by a command bus that transfers opcodes and scalar operands, and to the memory controllers via a network of point-to-point links that provide very high sustained bandwidth. Each AE instruction is passed to all four AEs. The process of the instructions depends on the personality. For instance, a personality that implements a vector model might implement multiple arithmetic pipelines in each AE, and divide the elements of a vector across all the pipelines to the processed in parallel [24].

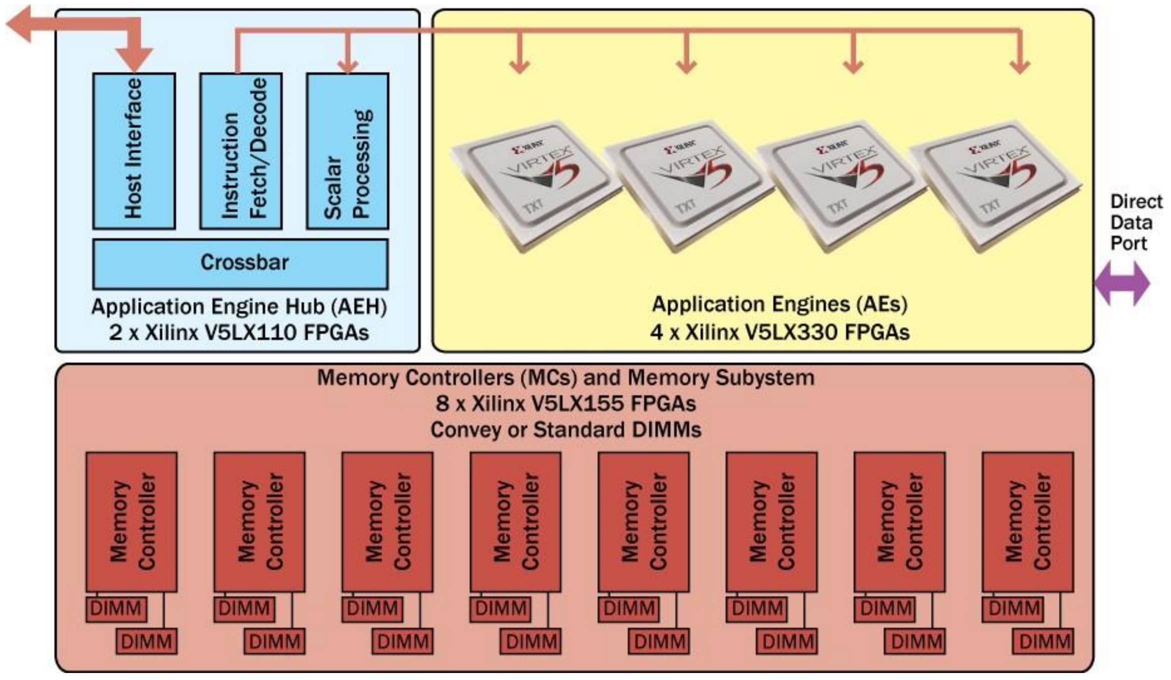

Figure 7: Convey Coprocessor Diagram[24]

2.1.1.3 Large Scale Example 2: Maxeler

The Maxeler dataflow engines (DFEs) map the algorithm onto dataflow cores, and the data is streamed from memory through the dataflow engine (DFE) where operations are performed.

Then, the data is forwarded directly from one dataflow core to another without being written to the off-chip memory. The dataflow structrure not only provides high compute performance, but also naturally optimizes the use of memory bandwidth, so even algorithms that are traditionally regarded as memory bound such as sparse matrix solvers can still be accelerated by orders of magnitude.

Maxeler technology has been succesfully utilized for large-scale complex Monte Carlo simulations, irregular financial tree-based partial differential equation (PDE) solvers, 3D finite difference (FD) and finite element (FE) solvers, as well as optimization and pattern matching problems.

The Maxeler's product that was used in this thesis is the MPC-C Series node. It is a server-class HPC system with 12 Intel Xeon CPU cores, up to 192GB of RAM for the CPUs and 4 vectis dataflow compute engines (DFEs) closely coupled to the CPUs. The DFEs are provided with up to 192GB RAM (48GB per DFE). Each node typically provides the compute performance of 20-50 standard x86 servers, while consuming around 500W, dramatically lowering the cost of computing and creates exciting new opportunities to deploy next generation algorithms and applications much earlier than would be possible with the standard servers. Each dataflow engine is connected to the CPUs via PCI Express gen2 x8, and DFEs within the same node are directly connected with the MaxRing interconnect. The node also supports optional Infiniband or 10GE interconnect for tight, cluster-level integration. It is fully programmable with the MaxCompiler.

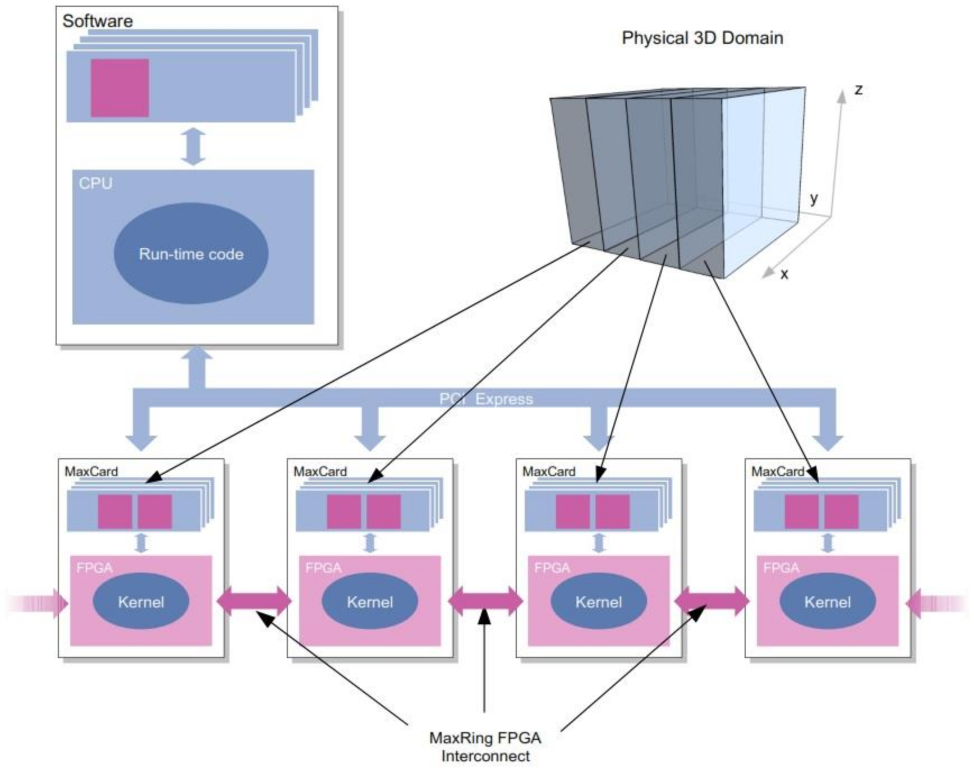

Figure 8: Maxeler node Architecture[25]

The MPC-C Series run production-standard Linux distributions, including Red Hat Enterprise Linux 5 and 6. This entire thesis was implemented with Centos 6.5. The dataflow engine management software is installed as a standard RPM package that includes all software necessary to utilize accelerated applications, including kernel device driver, status monitoring daemon and runtime libraries. All the management tools use standard methods for status and event logging. The nodes support the Intelligent Platform Management Interface (IPMI) for remote management. A status monitoring deamon logs all accelerator health information. User tools provide visibility into accelerator state and system check utilities can be easily integrated into cluster-level automatic monitoring systems.

Maxeler high-performance dataflow computing systems are fully programmable using the general-purpose MaxCompiler programming tool suite.

Any accelerated application runs on a Maxeler node as a standard Linux executable. Programmers can write new applications using existing dataflow engine configurations by linking the dataflow library file into their code and then calling simple function interfaces.

To create application exploiting new dataflow engine configurations, MaxCompiler allows an application to be split into three parts:

1. Kernel(s), which implement the computational components of the application in hardware.
2. Manager configuration, which connects Kernels to the CPU, engine RAM, other Kernels and other dataflow engines via MaxRing
3. CPU application, which interacts with the dataflow engines to read and write data to the Kernels and engine RAM.

MaxCompiler includes tools to support all three steps: the Kernel Compiler, the Manager Compiler and a software library (accesible from C or Fortran) for bridging between hardware and software.

Programmers develop kernels by writing programs in Java. However, using the tools requires only minimal familiarity with Java. Maxeler provides MaxIDE, an Eclipse-based development environment to maximize programmer productivity.

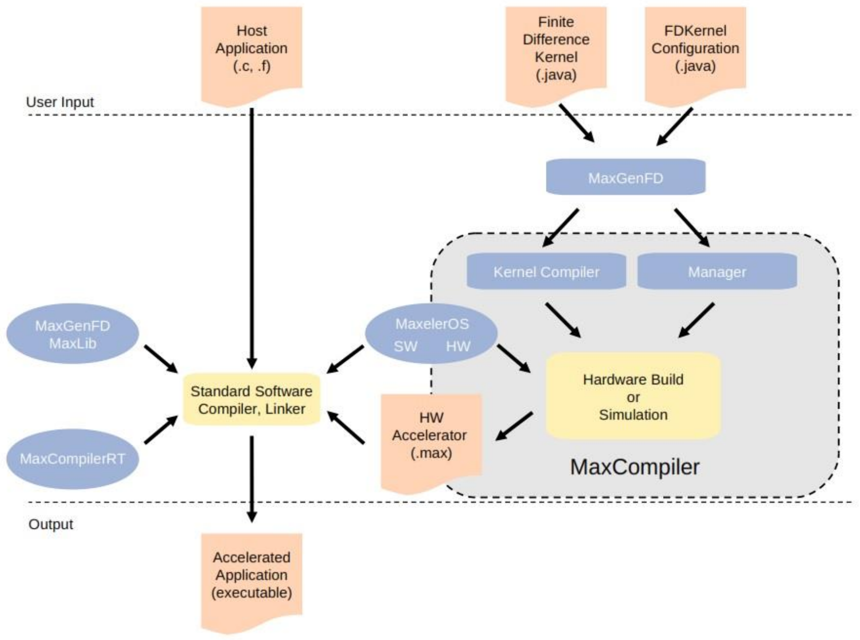
Host Application (.c, .f)
Finite Difference Kernel (.java)
FDKernel Configuration (.java)
User Input
MaxGenFD
Kernel Compiler
Manager
MaxGenFD MaxLib
MaxelerOS SW HW
Standard Software Compiler, Linker
Hardware Build or Simulation
HW Accelerator (.max)
MaxCompilerRT
MaxCompiler
Output
Accelerated Application (executable)

Figure 9: The cretion and implementation of a MaxCompiler Project[25]

Once written, MaxCompiler transforms user kernels into low-level hardware and generates a hardware dataflow implementation (the .max file) which the developer can link into their CPU application using the standard GNU development tool chain.

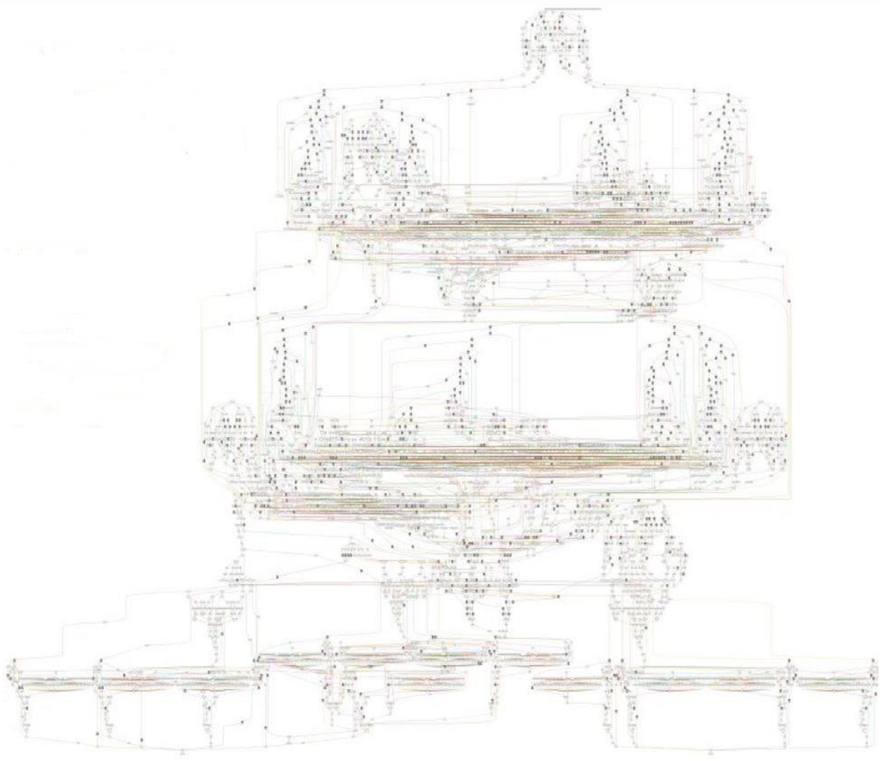

Figure 10: A MaxCompiler produced dataflow graph[26]

Finally, MaxCompiler provides complete support for debugging during the development cycle, including a high-speed simulator for verifying code correctness before generating a hardware implementation and the MaxDebug tool for examining the state of running chips.

2.2 Remote Offloading Workload

Any time that information is transferred from one processor to another, some communication is necessary. If this communication does not go through shared memory, it requires the serialization and deserialization of various data structures and objects. This translation overhead may be significant when considering the performance of large distributed or networked systems. The EVAGORAS project [27] is demonstrated as a relevant example.

2.2.1 EVAGORAS project(ΕΥΑΓΟΡΑΣ)

This project is about creating a supercomputer for Biological, Ecological and Medical data based on the re-programmable logic (FPGAs) and it will be used on fitting labs.

Today, the Medicine, the Biology and the Ecology are sciences that present increasing growth with a very broad field of research. The scientists-researchers have to deal with large data sets that are being produced daily at the labs and facing discouraging times for mining the final – useful information. The current approach for this problem is the usage of supercomputer or Grid computer. An alternative approach is the suggested system. It consists of two general purpose computers and a series of boards with reconfigurable logic. It can execute any given application developed for this system fast.

The general purpose computers will provide the internet connection and the user interface. The series of boards will be connected either with the general purpose computers through a very fast connection or between them. During this project will be implemented interfaces and libraries for the most common actions, like access to external memory, internal hidden cache, etc. After the creation of the libraries, qualified BioInformatics applications and Ecological models will be implemented. The selection of the applications to be implemented will be of the outmost importance (NCBI BLAST) in order to make the supercomputer useful for as many as possible end users. Every user will have a registered account which will be linked to certain local databases. The applications that will be implemented will be tested thoroughly through a series of experiments. The average expected reduction of time per application per reconfigurable board is 20. As such, 16 reconfigurable boards is equivalent to 320 conventional computers.

The proposed supercomputer will be considered an important tool for the end users and will aid them to be notably competitive on their respective fields of study. The labs that will design and implement this supercomputer will have a remarkable system for new applications and research. The maintenance and the energy consumption of this system is notably lower than the general purpose computers.

3. Problem Modeling

As mentioned before in Section 2.1.1.3, the Maxeler MPC – C Series consists of two parts, the Host Computer and the DFEs. For their communication each DataFlow Engine (DFE) is connected to the CPU via the PCIe Standard. Also, in many projects, the data are not created by the Host Computer but there is the need to be transmitted to the Server over an Internet connection. Thus, interesting topics arouse about the theoretical and practical values of these communications. In this Chapter first, an evaluation about the communication between the Host Computer and the DFEs is introduced. Then, an analysis on a connection between the Server and two workstations is presented.

3.1 Host CPU – DataFlow Engines Communication

One part of this thesis, is to evaluate the rate that data are streamed to the DFEs, processed and streamed back to the Host Computer.

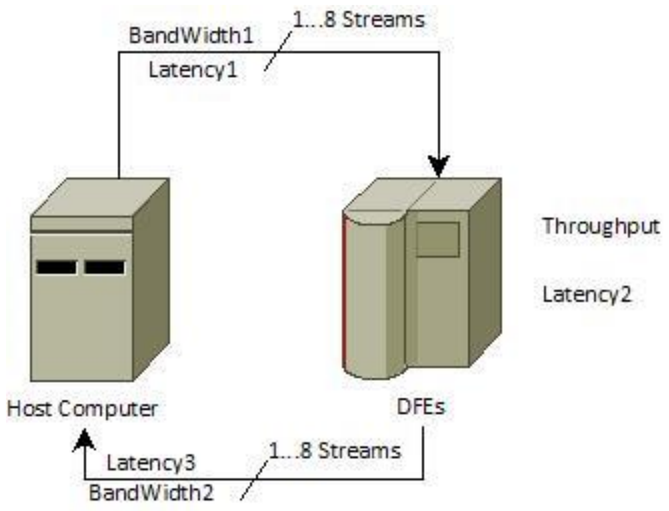

Figure 11: CPU - DFEs communication issues

As seen in Figure 11, we have two main aspects to consider, the latency and the bandwidth of this communication.

First, the overall latency of the system. The overall latency can be calculated as the summary of Latency1, Latency2 and Latency3 as seen in Figure 11. This latency will occur every time we make a call to the DFEs in order to send any amount of data. The profound experiment to measure the latency is to make calls to the DFEs, without any data and any commands for editing. But the

MaxCompiler has the limit that data must be sent to DFEs in every call. So, I used various small size of data (1024 bytes was the maximum size) and no commands for editing. Thus, the data were streamed to the DFEs and then back to the Host Computer. The result was that the whole process lasted approximately 27 ms. Also, after a series of experiments I conducted and are described in Section 5.2, I concluded that the latency increases as the number of streams increases.

Second, in order to determine the bandwidth of CPU-DFEs communication we have to take the minimum value between BandWidth1, Throughput and BandWidth2 as described in Figure 11. Each Dataflow Engine is connected to the CPUs via PCI Express gen 2 x8 [28]. Although in theory the bandwidth of this connection is 4 GB/s (for 8-lane PCIe connectors) (Source: http://en.wikipedia.org/wiki/PCI_Express), Maxeler Technologies set their optimal bandwidth at 2 GB/s. Thus, for each one of the eight lanes BandWidth1 equals BandWidth2 and they are 250 MB/s. In order to calculate the throughput of the DFEs, I conducted two experiments. In the first one I calculated the mean filter of a video chunk and in the second I found the square root of some numbers. These two experiments are described in depth at Section 5.2. I concluded that the throughput of each DFE for a dataset of 720 Mbytes is approximately 140 MB/s for each lane (stream)[Table 3] . Note that this is an average value that varies greatly depending on the size of the data for editing. So, considering that the minimum value is the throughput, I can conclude that the bandwidth of the system is about 140 MB/s.

3.2 WorkStations – Server Communication

In many applications, the source that produces data is not the Server itself. Thus, we need a way to send the data to the Server in order to edit them. One of the most common ways for this communication is to send them through the Internet. In order to have a complete picture of the workload that the Server can handle along with the rate that we can feed it data for editing, I have evaluated the bandwidth of the internet at the Technical University of Crete (TUC), Section 5.1.

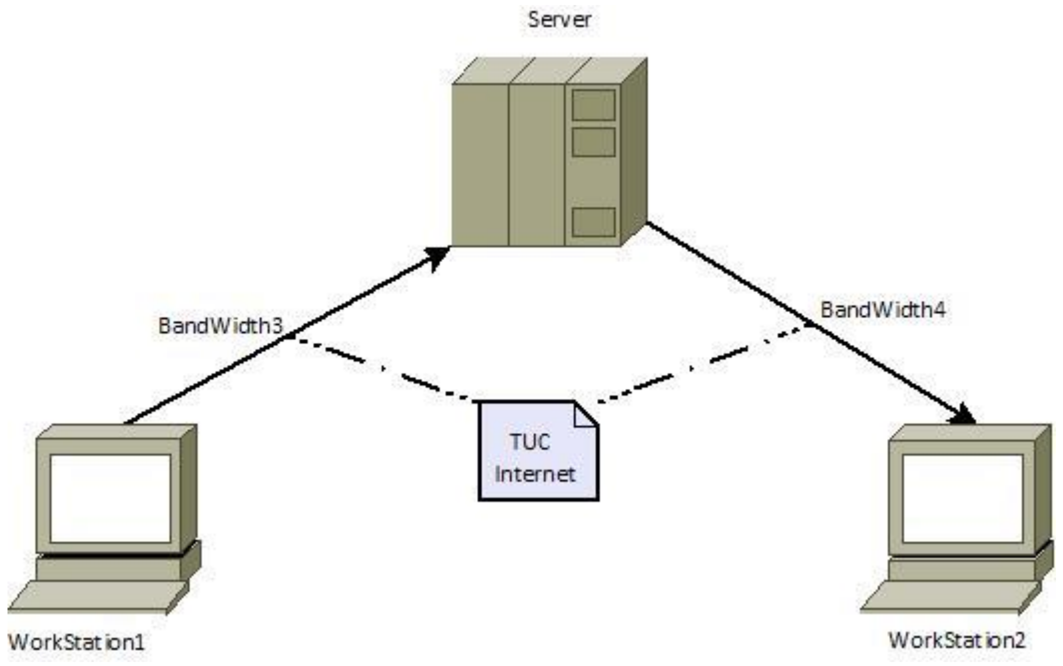

Figure 12: Server-WorkStations communication

As seen in the above figure, let us name the download speed of the Server as BandWidth3 and the upload speed as BandWidth4. I used the system described in the next Chapter to evaluate these values. The protocol that was used is the TCP/IP as for many applications the integrity of the data is more important from receiving them fast but without order and missing a potencial crucial information. It is described in Section 4.3. I performed experiments with various data sets and each measurement is the average of 10 iterations. It is described in depth in Section 5.1. The result was that the BandWidth3 is approximately 11.7 MB/s and the BandWidth4 is about 6.4 MB/s.

In conclusion, if we want a complete system that streams data to the Server, the Server processes them and then sends them to a workstation the limiting factor is the Internet bandwidth. The bandwidth of the CPU – DFEs communication at 140 MB/s enables us to run plenty of applicatios on the Maxeler MPC – C Series. But, if we want to stream data to the Server we will have to make sure that the data to be edited is small enough that the bandwidth of 11.7 MB/s and, more importantly, the upload speed of 6.4 MB/s is sufficient for our application.

4. System Architecture

At the beginning of this Chapter, a general diagram of the system that was implemented is shown. Following, there is an analysis on each of the subsystems. Closing, the library that was develloped for all the future users that will want to use sockets, for the TCP/IP or the UDP protocol, is demonstrated.

4.1 System Diagram

In order to test all the issues presented in Chapter 3, an implementation of a system was necessary. The role of producing data has a web camera that is connected to a workstation. The workstation receives the data and feed them on the Server through a TCP socket. The Server receives the data, edits them, creating 5 different filters, and streams them to a different workstation in order to be displayed to the final user. The diagram below illustrates the functionality of the entire system.

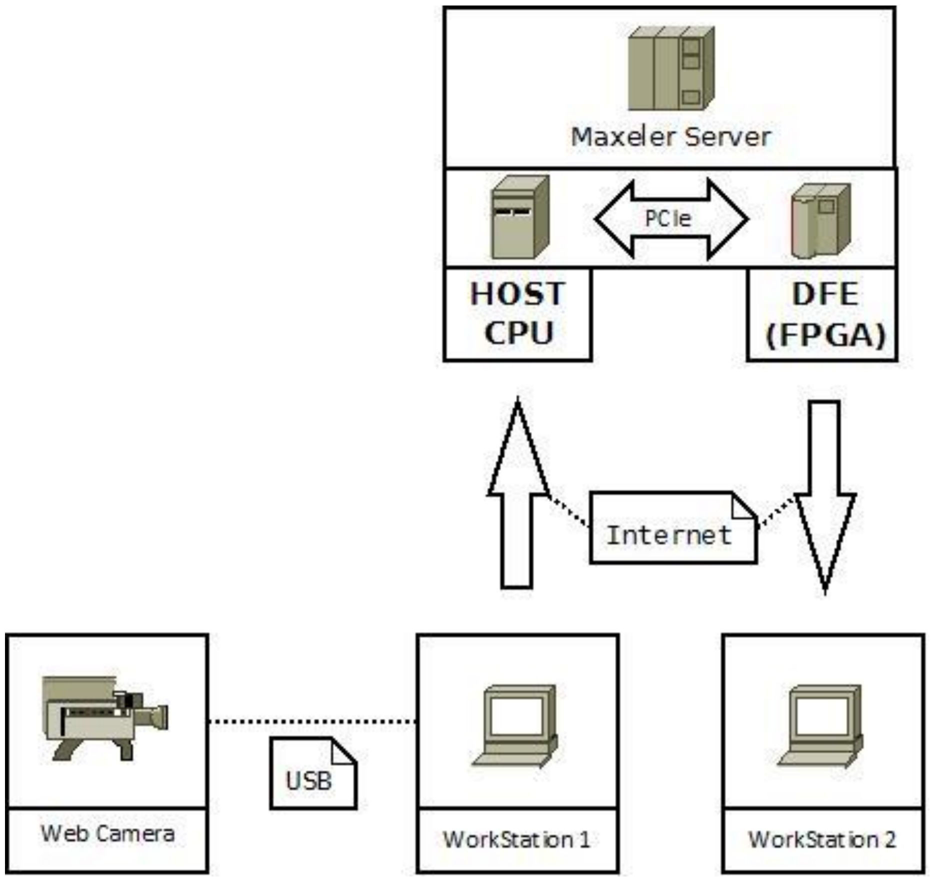

Figure 13: Datapath of the Live Video Stream application

4.2 System Functionality

Now, let us delve into each subsystem a bit more.

4.2.1 WorkStation 1

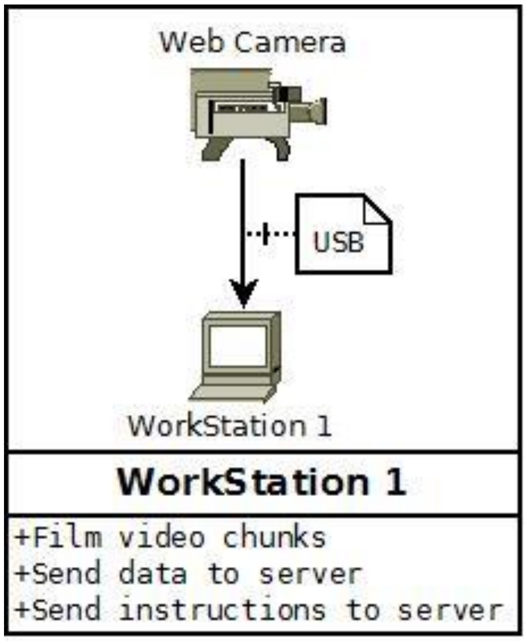

Figure 14: WorkStation 1 functionality

The WorkStation1 role is simple. Capture video and send it to the server for further editing.

In more detail, a web camera is connected via usb 2.0 to the WorkStation1. For my implementation, I used the Logitech V-UAV35 QuickCam Notebook Webcam Deluxe. It supports maximum 640x480 resolution (VGA) at 15 fps. The supported pixel format is yuvj420p.

After a thorough search for compatible drivers for this web camera I have not been able to find one for Centos 6.5. Although I have tried to implement my own driver, I was not able to fullfil this task and as it was beyond the scope of this thesis I preferred to implement the video capturing using a free software project called ffmpeg. The ffmpeg project is the leading multimedia framework, able to decode, encode, transcode, mux, demux, stream, filter and play pretty much anything has been invented. It supports the most obscure ancient formats up to the cutting edge. No matter if they were designed by some standards committee, the community or a corporation.

The codec used for the captured video chunks is the Motion JPEG (MJPEG) as it was the only one that the above mentioned web camera supported. The Motion JPEG is a video compression format in which each video frame or interlaced field of a digital video sequence is compressed separately as a JPEG image. It was originally developed for multimedia PC applications, but MJPEG is now used by video-capture devices such as digital cameras, IP cameras, webcams and by non-linear video editing systems.

MJPEG is an intraframe-only compression sheme. Whereas modern interframe video formats, such as MPEG1, MPEG2 and H.264/MPEG-4 AVC, achieve real-world compression-ratios of 1:50 or better, MJPEG's lack of interframe prediction limits its efficiency to 1:20 or lower, depending on the tolerance to spatial artifacting in the compressed output. Because frames are compressed independently of one another, MJPEG imposes lower processing and memory requirements on hardware devices. The image quality of MJPEG is directly a function of each video frame's static complexity. Frames with large smooth transitions or monotone surfaces compress well, and are more likely to hold their original detail with few visible compression artifacts. Frames exhibiting complex features, fine curves and lines are prone to exhibit DCT-artifacts such as ringing, smudging and macroblocking. MJPEG compressed video is insensitive to motion complexity, to highly random motion and to absence of motion.

```
            if (system("ffmpeg -loglevel panic -f v4l2 -framerate 15 -video_size 
640x480 -i /dev/video0 -y -q 2 -vcodec mjpeg -pix_fmt yuvj420p -t 10 
Process/Transmitter/chunks/output1.avi") == -1)
            {
                perror("Thread record: Capturing 1 failure!\n\n");
                exit(1);
            }
```

Example code of using the FFMPEG program in order to capture a video chunk with duration 10 seconds.

I have chosen not to have continuous feed from the web camera but video chunks with specific duration. There are two reasons for this choice. First, with the current version of the MaxCompiler, only finite size of data can be streamed to the DFEs for editing. Thus, even if I had continuous feed to the Server, I would had to implement some kind of cropping to the feed. Second, because of the use of FFMPEG in my system,for the reasons mentioned above, I cannot have continuous feed from the web cam. The FFMPEG needs to write the captured file to the hard disk.

Back to the WorkStation1 functionality analysis, when the camera captures a video chunk, an inside trigger is fired and the saved chunk is streamed to the server while the camera captures the next video chunk. After experimenting thoroughly, I discovered that the optimal video chunk duration is at 10 seconds. In that way, I get the minimum frame loss per chunk, which is about 2-3.

The streaming process is implemented with the libraries I developped, using a TCP socket and they are discussed in the Chapter Client-Server Library.

Finally, I developped a synchronous function that can send instructions to the server without pausing the capturing and streaming process. If the user types the hotkey “Ctrl + \”, a menu pops up and the user can either type the selection of a new filter or exit the application. Thus, starting from the next video chunk that the web camera will capture, the video that will be shown in WorkStation2 will be filtered with the user’s selection.

4.2.2 Server

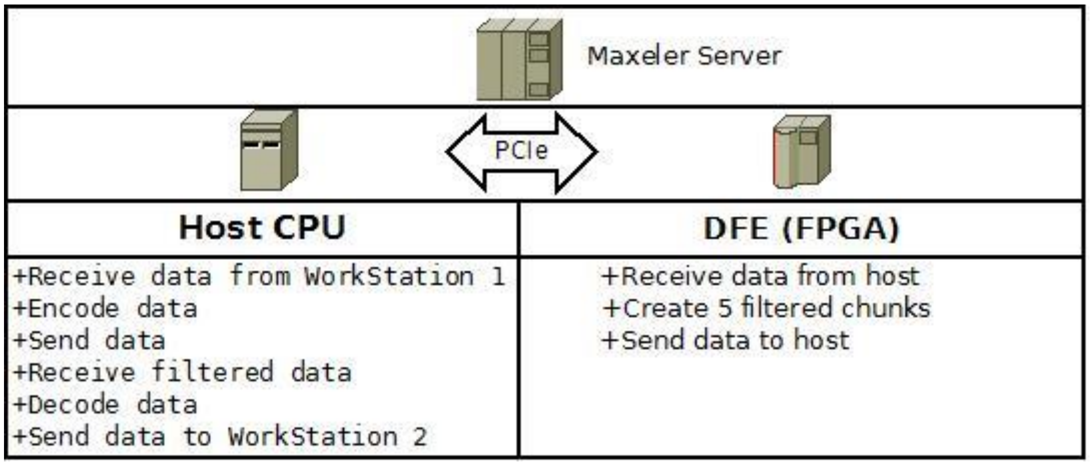

Figure 15: Server functionality

On The Host Computer on the Server, 3 threads are running simultaneously.

The first thread is using the Client-Server Library to create a TCP socket, in order to receive the video chunks. Then it saves them on the hard disk and fires a trigger. For the optimal usage of the available resources, the input video chunks cycle between 4 saved files. This means that, when the first video is received is saved as number 1, second as number 2, etc. When the fifth video chunk is received it is saved as number 1. The functionality of the previous first video is completed, as it has already been processed, so there is no need to maintain it. Thus, the amount of hard disk space required for my application is minimum. Finally, the first thread is responsible for receiving the filter selection from WorkStation1.

When the above mentioned trigger occures, the second thread calls the ffmpeg program in order to extract the JPEGs from the MJPEG video chunk, save them in a format appropriate for editing, the PPM and change the pixel format to rgb24.

The Portble Pixel Map (PPM) format is part of the Netpbm color image format which also consists of the Portable Graymap (PGM) and the Portable Bitmap (PBM) formats. Collectively, they are referred as the Portable Anymap (PNM) format. The PNM format was invented by Jeff Poskanzer in 1980 and it is free of use. A PPM file consists of a sequence of one or more PPM images. There are no data, delimiters, or padding before, after, or between images. Each PPM image consists of the following:

1. A “magic number” used for identification of the file type. For the PPM format it is “P6”
2. Whitespace (blancs, TABs, CRs, LFs)
3. A width, formatted as ASCII characters in decimal
4. Whitespace
5. A height, again in ASCII decimal
6. Whitespace
7. The maximum colour value, again in ASCII decimal. Must be less than 65536 and more than zero
8. A single whitespace character
9. A raster of Height rows, in order from top to bottom. Each row consists of Width pixels, in order from left to right. Each pixel is a triplet of red, green, and blue sample, in that order. Each sample is represented in binary code by either 1 or 2 bytes. If the maximum colour value is less than 256, it is 1 byte. Otherwise, it is 2 bytes. The most significant byte is first. For this thesis I used maximum colour value at 255.

Although, the PPM format is an exceptional format for editing images, I made one last transformation on the representation of each pixel’s value before streaming it to the DFE. Instead of 3 different values for one pixel (one for each colour component: Red, Green, Blue), I shifted the values of each component by 8 and zero filled. Thus, I got only one 32-bit integer per pixel. This minimizes the iterations between the input stream in the DFEs.

```
    int cnt2=0;

    for (cnt1=header_size;cnt1<chunk_size;cnt1=cnt1+3)
    {
         x[cnt2] = x1[cnt1]*65536 + x1[cnt1+1]*256 + x1[cnt1+2];

         if(cnt1%(ppm_size-3)==0) cnt1=cnt1+header_size;

         cnt2++;
    }
```

Code for the last trasformation on the data that was implemented. (zero fill not included.)

The reason for this last transformation is the image filters I implemented. Most of them make use of a window of the surrounding pixels, and this means that in order to edit one pixel, the DFE needs pixels in previous and following positions in the input stream.

The DFE implementation of the filters is described in the Chapter 5.3.

The DFE streams back to the host the edited vectors in the same format as they were sent to DFE. Then, the second thread decodes only the vector that corresponds to the filter choice made by the user when it was sent from WorkStation1.

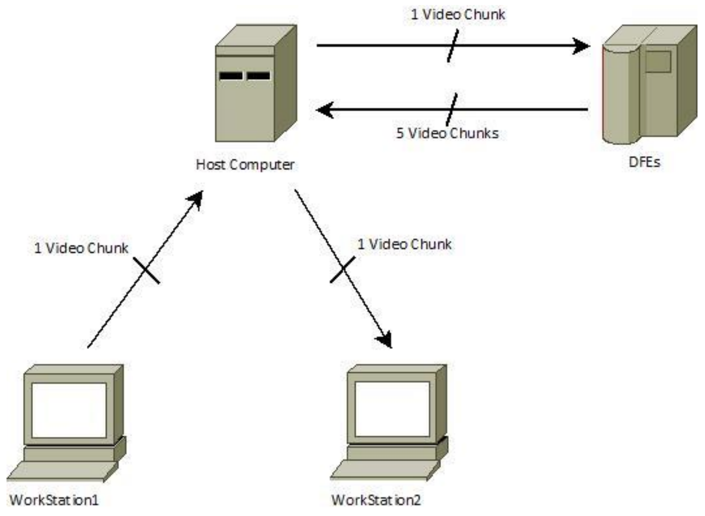

Figure 16: Each video chunk's route.

Afterwards, the thread creates the filtered video chunk using the MPEG-4 Part 10 codec.

The H.264 or MPEG-4 Part 10, Advanced Video Coding (MPEG-4 AVC) video codec has a very broad application range that covers all forms of digital compressed video from, low bit-rate Internet streaming applications to HDTV broadcast and Digital Cinema applications with nearly lossless coding. Compared to the current state of technology, the overall performance of H.264 is such that bit rate savings of 50% or more are reported. Digital Satellite TV quality, for example, was reported to be achievable at 1.5 Mbit/s, compared to the current operation point of MPEG 2 video at around 3.5 Mbit/s.

The codec specification itself distinguishes conceptually between a video coding layer (VCL) and a network abstraction layer (NAL).

The VCL contains the signal processing functionality of the codec; mechnanisms as transform, quantization, and motion compensated prediction; and a loop filter. It follows the general concept of most of today's video codecs, a macroblock-based coder that uses inter picture prediction with motion compensation and transform coding of the residual signal. The VCL encoder outputs slices: a bit string that contains the macroblock data of an integer number of macroblocks, and the information of the slice header (containing the spatial address of the first macroblock in the slice, the initial quantization parameter, and similar information). Macroblocks in slices are arranged in scan order unless a different macroblock allocation is specified, by using the so-called Flexible Macroblock Ordering syntax. In-picture prediction is used only within a slice.

The NAL encoder encapsulates the slice output of the VCL encoder into NAL units, which are suitable for transmission over packet networks or use in packet oriented multiplex environments.

The MPEG-4 Part 10 codec was selected because it offers good image quality at a minimum size.

Finally, the second thread fires the trigger for the third thread.

The third thread transmits, using a TCP socket created by the Client-Server Library, the next available filtered video chunk to WorkStation2.

4.2.3 WorkStation 2

Similar to WorkStation1, the role of the WorkStation2 is simple. Receive video chunks from the Server and show them.

The Client-Server Library was used, for the creation of a TCP socket in order to receive video chunks. After the chunk is received and saved on the hard disk, an internal trigger is fired and the chunk is enqueued in the current playlist. The media player that was used is the VLC media player.

VLC is a free and open source cross-platform multimedia player and framework that plays most multimedia files as well as DVDs, Audio CDs, VCDs, and various streaming protocols.

The reason I chose VLC media player is that it supports input commnads via UNIX domain sockets.

A Unix domain socket or IPC (Inter-Process Communication) socket [28] is a data communications endpoint for exchanging data between processes executing within the same host operating system. Unix domain sockets may be created either as connection-mode or as connectionless. Processes using Unix domain sockets do not need to share a common ancestry. The API for Unix domain sockets is similar to that of an Internet socket, but it does not use an underlying network protocol for communication. The Unix domain socket facility is a standard component of POSIX operating systems.

So, I created a Unix domain socket with the vlc, and when a chunk is saved at the hard drive, a command is sent to vlc to enqueue it at the current playlist.

Also, I intentionally delayed the start of playing the playlist for four chunks, which is 40 seconds. This was done in order to allow the playlist to make a temporary “buffer”, to achieve smoother transition between video chunks.

All the above mentioned triggers are implemented as named pipes.

A FIFO special file (a named pipe) [29] is similar to a pipe, except that it is accessed as part of the filesystem. It can be opened by multiple processes for reading and writing. When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem. Thus, the FIFO special file has no contents on the filesystem; the filesystem entry merely serves as a reference point so that processes can access the pipe using a name in the filesystem.

4.3 Client – Server Library

In general, a network protocol defines rules and conventions for communication between network devices. In depth, a network protocol includes mechanisms for devices to identify and make connections with each other, as well as formatting rules that specify how data is packaged into messages to be sent and received. Some protocols also support message acknowledgment and data compression designed for reliable and/or high-performance network communication. Hundreds of different computer network protocols have been developed each designed for specific purposes and environments. Common examples are the TCP/IP, UDP, HTTP and FTP.

The Internet protocol suite is the computer networking model and communications protocols used in the Internet and similar computer networks. It is commonly known as TCP/IP, because its most important protocols, the Transmission Control Protocol (TCP) and the Internet Protocol (IP), were the first networking protocols defined in this standard. Often also called the Internet model.

Internet Protocol version 4 (IPv4) **i**s the fourth version in the development of the Internet Protocol (IP) Internet, and routes most traffic on the Internet. It is a connectionless protocol for use on packet-switched networks. It operates on a best effort delivery model, in that it does not guarantee delivery, nor does it assure proper sequencing or avoidance of duplicate delivery. These aspects, including data integrity, are addressed by the Tranmission Control Protocol (TCP). IPv4 uses 32-bit (four-byte) addresses, which limits the address space to 2^{32} addresses.

Internet Protocol version 6 (IPv6) is the latest version of the Internet Protocol (IP). It uses a 128-bit address, allowing 2^{128} addresses. Additionally, the IPv6 provides some other technical benefits over its predecessor, the IPv4. In particular, it permits hierarchical address allocation methods that facilitate route aggregation across the Internet, and thus limit the expansion of routing tables. The use of multicast addressing is expanded and simplified, and provides additional optimization for the delivery of services. Device mobility, security, and configuration aspects have been considered in the design of this protocol..

The TCP/IP provides end-to-end connectivity specifying how data should be packetized, addressed, transmitted, routed and received at the destination. This functionality is organized into four abstraction layers which are used to sort all related protocols according to the scope of networking involved. From lowest to highest, the layers are the link layer, containing communication technologies for a single network segment (link), the internet layer, connecting hosts across independent networks, thus establishing internetworking, the transport layer handling host-to-host communication, and the application layer, which provides process-to-process application data exchange.

The User Datagram Protocol (UDP) is one of the core members of the Internet protocol suite.

UDP uses a simple connectionless transmission model with a minimum of protocol mechanism. It has no handshaking dialogues, and thus exposes any unreliability of the underlying network protocol to the user's program. There is no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network without prior communications to set up special transmission channels or data paths. UDP is suitable for purposes where error checking and correction is either not necessary or is performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system.

Applications use datagram sockets to establish host-to-host communications. An application binds a socket to its endpoint of data transmission, which is a combination of an IP address and a service port. A port is a software structure that is identified by the port number, a 16 bit integer value, allowing for port numbers between 0 and 65535. Port 0 is reserved, but is a permissible source port value if the sending process does not expect messages in response.

The Internet Assigned Numbers Authority (IANA) has divided port numbers into three ranges. Port numbers 0 through 1023 are used for common, well-known services. On Unix-like operating systems, using one of these ports requires superuser operating permission. Port numbers 1024 through 49151 are the registered ports used for IANA-registered services. Ports 49152 through 65535 are dynamic ports that are not officially designated for any specific service, and may be used for any purpose. They also are used as ephemeral ports, from which software running on the host may randomly choose a port in order to define itself. In effect, they are used as temporary ports primarily by clients when communicating with servers.

According to the above information, for the purposes of this thesis, the port number 8554 (RTSP alternate according to IANA) was selected for the WorkStation1 – Server communication. The port number 8556 (Unassigned according to IANA) is used by the Server – WorkStation2 communication, according to figure 11.

When one IP host has a large amount of data to send to another host, the data is transmitted as a series of IP datagrams. It is usually preferable that these datagrams be of the largest size that does not require fragmentation anywhere along the path from the source to the destination. This datagram size is referred to as the Path MTU (PMTU), and it is equal to the minimum of the MTUs of each hop in the path. The current practise is to use the lesser of 576 and the first-hop MTU as the PMTU for any destination that is not connected to the same network or subnet as the source. In many cases, this results in the use of smaller datagrams than necessary, because many paths have a PMTU greater than 576.

A PMTU discovery on the wireless network of the Technical University of Crete has been performed in the context of this thesis. The result is that the maximum transmission unit is 1472 Bytes. All the data transfers in this thesis use 1024 bytes per packet to avoid any hazard.

For the purposes of this thesis, two C language libraries were developed. They have been implemented as header files, in order to assist the future users of the Maxeler MPC-C Series, as well any other thesis or project that may require internet connection between two workstations.

Each of them creates either a TCP or a UDP socket. Users can include the header file that they want, TCP_socket.h or UDP_socket.h, and they can call the function they want by simply calling its name providing the required variables. The functions for the avoidance of any misunderstanding were called, create_tcp_socket() and create_udp_socket().

```
Int sockfd = create_tcp_socket("vergina.mhl.tuc.gr", "8554");
```

Example code for calling the method for creating a TCP/IP socket.

Both of the functions take as input two variables.

First, the address that they want to connect to, in the form of a character pointer, char*. For example, in order to connect to the vergina server, where Maxeler MPC-C Series is hosted, one shall give as first input “vergina.mhl.tuc.gr”.

Second, the port number that they want to use, also in the form of a character pointer, char*. For example, the string “8554” was used as port number in this thesis, as mentioned above.

The function that was developed, will recognize that the given address is either in IPv4 or Ipv6 form of the Internet Protocol and will return a socket specifically for that address and that port.

```
        if (nadr->ai_family == AF_INET)
        {
            struct sockaddr_in *ipv4 = (struct sockaddr_in *)nadr->ai_addr;
            ad = &(ipv4->sin_addr);
            ipver = "IPv4";
        }
        else
        {
            struct sockaddr_in6 *ipv6 = (struct sockaddr_in6 *)nadr->ai_addr;
            ad = &(ipv6->sin6_addr);
            ipver = "IPv6";
        }
```

This code snipet demonstrates how the distinction between IPv4 and IPv6 addresses is made.

During the development of this thesis and as seen in the Chapter 5 the libraries have been tested numerous times, and their functionality is flawless.

5. Performance and Validation

At this Chapter, there is a presentation of the capabilities of the Maxeler MPC – C Series, my systems' performance and validation of its correct functionality. First, I present the internet bandwidth of TUC. Following various experiments on the DFEs are shown. After that, there is an assesment on my system's performance. Closing, there are a few words about the type of the workload for the DFE that I chose, the video filters along with results of each of them.

5.1 Internet Bandwidth of TUC

A part of this thesis is to evaluate the internet bandwidth of the Technical University of Crete (TUC). This was done, so every member of TUC, either student or staff member, to be able to take in consideration these values for their projects.

The following table shows the average results I received, after performing experiments multiple times using the system described in the previous chapter.

Size of Dataset(bytes)	WorkStation1 Upload(MB/s)	Server Download(MB/s)	Server Upload(MB/s)	WorkStation2 Download(MB/s)
2,396,358	11.99	11.69	6.39	6.39
11,981,790	11.71	11.67	6.34	6.34
59,908,950	11.70	11.70	6.38	6.38
143,781,480	11.69	11.70	6.42	6.42
Average:	11.77	11.69	6.38	6.38

Table 1: Average values of TUC internet bandwidth

Although the bandwidth seems to be sufficient for many online applications, I would like to note that the TUC network, especially the wireless network connection, suffers from great fluctuation on its speed over the course of a day. This occurs mainly because of the large amount of users trying to connect to the network from certain places, like the classrooms and the school cafeteria. This ends up to the users experiencing many connection issues. Either random disconnections or not being able to connect to the network at all.

Also, I repeated the above experiment connected to TUC via Virtual Private Network (VPN). The table consists also of average values after a number of experiments.

Size of Dataset(bytes)	WorkStation1 Upload(MB/s)	Server Download(MB/s)	Server Upload(MB/s)	WorkStation2 Download(MB/s)
2,396,358	0.10	0.10	0.94	0.94
11,981,790	0.10	0.10	0.94	0.94
59,908,950	0.09	0.09	0.94	0.94
143,781,480	0.09	0.09	0.94	0.94
Average:	0.10	0.10	0.94	0.94

Table 2: Average values of TUC internet bandwidth (connection with vpn)

The internet bandwidth of the external connection provided by the ISP is:

- Download Speed: 24 Mbps
- Upload Speed : 1 Mbps

The real internet bandwidth of the external connection is as follows (provided by Ookla www.speedtest.net).

- Download Speed: 11.63 Mbps
- Upload Speed : 0.81 Mbps

These results show, that for an application that speed is critical, the connection to the TUC network via VPN is not a viable option.

5.2 System Performance

First, I will present general tests that were implemented in order to evaluate the capabilities of the DFE.

The first test is about the time that the DFE needed to get the input data, to process them, and to return them back to the Host Computer. I used various size of datasets in order to get a more complete idea on the scaling. All the data sets are in the form of BMP images, always as one input. Two different algorithms were implemented for this purpose and the average values are presented. The first algorithm is a mean video filter and the second one is the square root of each pixel. For a comparison, I tested the same filters on a computer implemented with the conventional control flow architecture.

Size of Dataset (bytes)	DFE: Mean Filter (MB/s)	CPU: Mean Filter (MB/s)	DFE: Square Rooting (MB/s)	CPU: Square Rooting (MB/s)
2,396,358	16.04	21.47	45.39	18.80
11,981,790	80.20	24.02	79.45	19.39
59,908,950	93.32	26.43	93.43	4.83
143,781,480	96.03	26.69	95.99	2.83
359,453,700	143.15	26.82	142.87	1.15
718,907,400	144.01	26.83	143.98	0.63
Average:	95.46	25.38	100.18	7.94

Table 3: Average values for processing times of DFE and CPU implementations of a mean filter and the square root of the input

As we can observe from the above table, the DFE are clearly faster compared to the conventional CPUs in processing time. Also, we shall note that the bandwidth of the DFE increases as the size of the data increases. Also, as discussed in Chapter 3, the overhead for one call at the DFE is relatively large (27 ms). These facts lead to the conclusion that the DFE work best for a small number of calls with the maximum available data size.

Furthermore, the DFE has the capability of getting as input more than one stream and returning as an output, also, more than one stream. The maximum number of input and output streams in one call to DFE is eight. Thus, a test on the speed flunctuation between the streams is in order. The logic on the datasets I used is the same as the above experiment. The tables presents the seconds that the implemention of a simple average filter on each dataset lasted on the DFE. (the overhead of sending and receiving the results is included in the values).

In the first table the number of input streams is always the same as the number of the ouput streams.

Size of each Stream (Bytes)	1 In 1Out (sec)	2 In 2 Out (sec)	3 In 3 Out (sec)	4 In 4 Out (sec)	5 In 5 Out (sec)	6 In 6 Out (sec)	7 In 7 Out (sec)	8 In 8 Out (sec)
2,396,358	0.044	0.045	0.047	0.047	0.052	0.051	0.052	0.048
11,981,790	0.107	0.116	0.115	0.119	0.128	0.128	0.132	0.135
59,908,950	0.442	0.456	0.466	0.474	0.504	0.519	0.527	0.538
119,817,900	0.851	0.876	0.889	0.916	0.962	0.986	1.006	1.048
359,453,700	2.511	2.538	2.634	2.698	2.828	2.9	3.006	3.054
718,907,400	4.992	5.073	5.177	5.357	5.599	5.664	5.801	5.848

Table 4: Time consumption on same input streams and output streams

In order to understand better the above values two figures are shown.

First, we present a figure that shows the best bandwidth that was achieved with the experiments that were held in this thesis per number of streams.

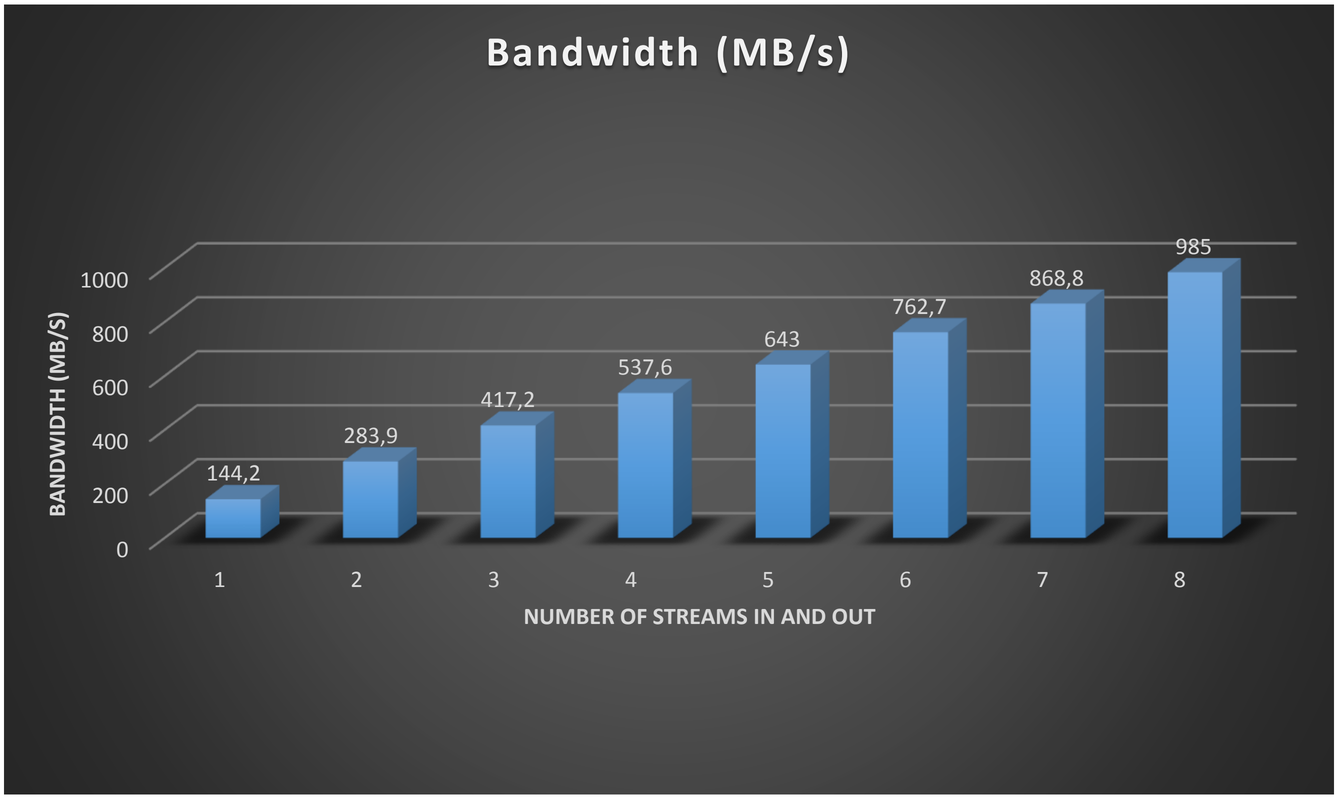

Figure 17: Bandwidth per streams

We can observe that the bandwidth increases almost linear with a rate of about 120 MB/s for every additional stream that was used.

The next figure presents the overhead that is required for every additional stream. We present three values all of them with the same number of input and output streams, for 1 stream, for 4 streams and for 8 streams.

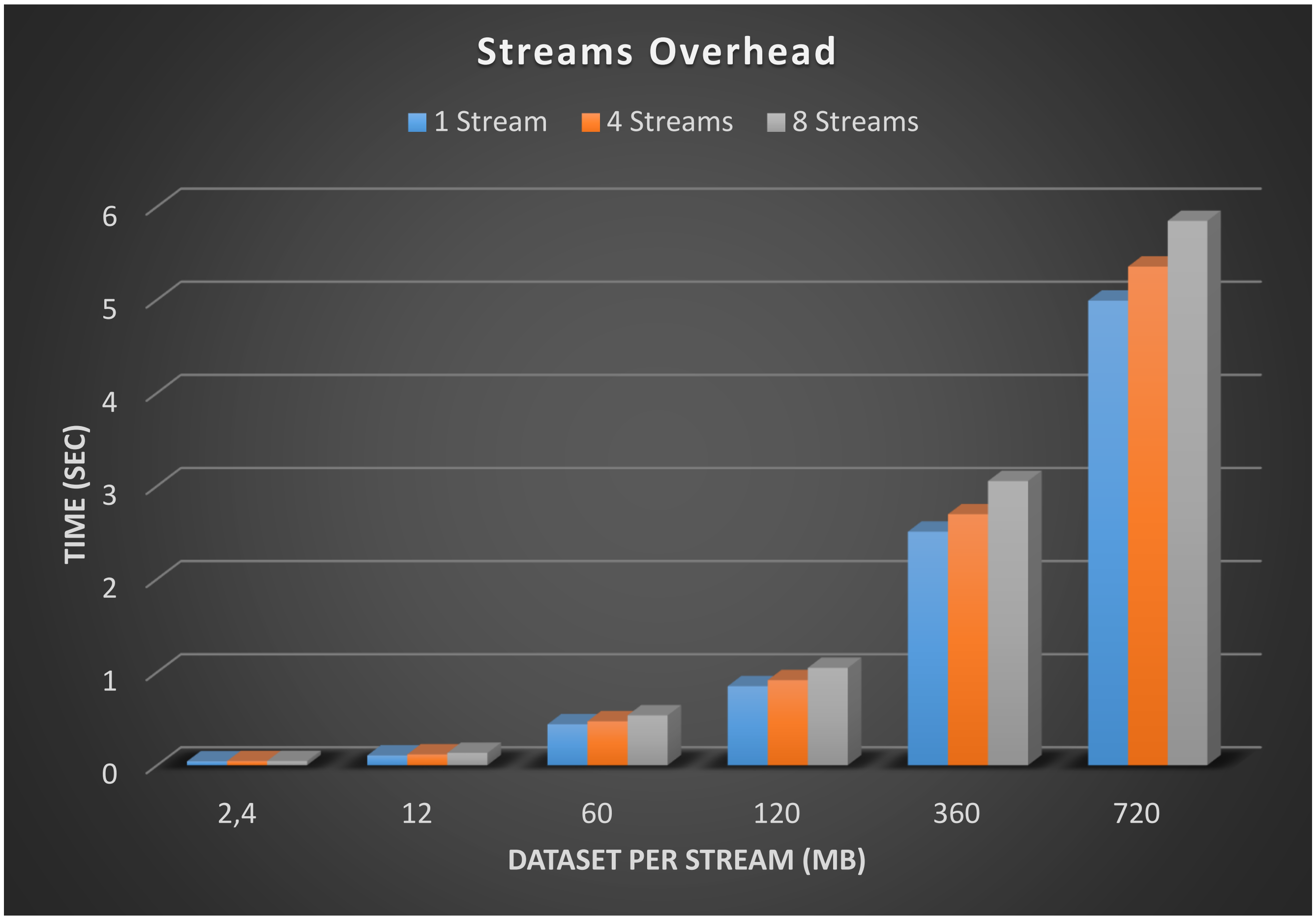

Figure 18: Streams Overhead

The overhead we have to pay increases accordingly the dataset we use. At the highest dataset that was used, 720 MB per stream, we observe that the overhead from 1 stream to 8 streams, has a difference of about 1 second.

In the next table the number of input streams is always the maximum (8) and the number of output streams changes.

Size of each Stream (Bytes)	8 In 1 Out (sec)	8 In 2 Out (sec)	8 In 3 Out (sec)	8 In 4 Out (sec)	8 In 5 Out (sec)	8 In 6 Out (sec)	8 In 7 Out (sec)
2,396,358	0.046	0.049	0.049	0.053	0.054	0.052	0.054
11,981,790	0.112	0.123	0.125	0.130	0.133	0.135	0.140
59,908,950	0.470	0.480	0.493	0.494	0.522	0.532	0.541
119,817,900	0.912	0.937	0.945	0.962	0.994	1.013	1.038
359,453,700	2.675	2.742	2.824	2.824	2.937	2.943	3.033
718,907,400	5.444	5.552	5.618	5.607	5.699	5.763	5.861

Table 5: Time consumption on maximum input streams and different output streams

For the last table regarding the streams, the number of output streams is kept at the maximum (8) and the number of input streams changes.

Size of each Stream (Bytes)	1 In 8 Out (sec)	2 In 8 Out (sec)	3 In 8 Out (sec)	4 In 8 Out (sec)	5 In 8 Out (sec)	6 In 8 Out (sec)	7 In 8 Out (sec)
2,396,358	0.052	0.053	0.053	0.056	0.056	0.055	0.055
11,981,790	0.134	0.136	0.135	0.138	0.139	0.141	0.141
59,908,950	0.518	0.516	0.526	0.538	0.545	0.546	0.553
119,817,900	0.989	1.008	1.018	1.014	1.034	1.035	1.034
359,453,700	2.827	2.839	2.921	2.930	2.931	3.016	2.984
718,907,400	5.493	5.668	5.813	5.785	5.819	5.900	5.948

Table 6: Time consumption on different input streams and maximum output streams

From all these experiments, we can conclude that the use of streams is highly encouraged. Let us consider the largest test case, approximately 719MB per stream. With 8 calls to DFE with one stream per call, the time required for finishing the process is about 8*4.992 = 39.936 seconds. While with a single call to the DFE with 8 streams for the input and the output data the time required is 5.848 seconds. We have a 6.8x speedup.

Furthermore, as a stress test for the DFE, I created an implementation of a 17x17 Sobel filter for 5 second duration video chunks.

The resources that the DFE used are as follows:

- Logic Utilization: 53034 / 297600 (17.82%)
    - LUTs: 34349 / 297600 (11.54%)
    - Primary FFs: 50269 / 297600 (16.89%)
- Multipliers (25x18): 1396 / 2016 (69.25%)
    - DSP blocks: 1396 / 2016 (69.25%)
- Block memory (BRAM18): 53 / 2128 (2.49%)

The streaming to DFE, the actual implementation of the filter and the streaming back to the Host Computer lasted on average 0.2816 seconds per video. In another implementation of the filter, now only on the Host Computer, apparently without the streaming from and to the DFE, the whole process lasted 4.8442 seconds. Thus, there is a speedup of 17.1x comparing the DFE and the conventional CPU implementation.

Finally, the system that was develloped for the Live Stream and discussed in Chapter 4 required:

- Logic Utilization: 13112 / 297600 (4.41%)
    - LUTs: 9227 / 297600 (3.10%)
    - Primary FFs: 12390 / 297600 (4.16%)
- Multipliers (25x18): 0 / 2016 (0.00%)
    - DSP blocks: 0 / 2016 (0.00%)
- Block memory (BRAM18): 47 / 2128 (2.21%)

These values occur after the optimization that the Maxeler MPC-C Series made automatically.

The DFE completes the streaming from and to the Host Computer and the processing of each video in 0.544 seconds on average. The first part of thread2, that is the encoding in an appropriate form for the DFE, lasts about 10.8 seconds. The last part of thread2, the decoding of the data and encoding to a new video file, lasts approximately 6.63 seconds. The threads 1 and 3, are proportional to the current bandwidth of the TUC network. Thus, we have a standard latency of about 18 seconds plus the time needed for streaming each video chunk to Server and to WorkStation2. That is the reason why my system show the videos to the final user with about 20 seconds delay.

5.3 Video Filters

Considering that the limiting factor on the system I developped is the Internet bandwidth, I chose as workload for the DFEs the implementation of video filters. The advantage of this choice, is that while the size of the data that are transmitted to the Server are relatively small, due to various video compression techniques, the computations can be very demanding. Now let us delve into the implementation of this workload.

Three of the filters that were implemented (Invert, Smooth and Colours Quantization) need to know all the colour components (R,G,B) that one pixel consists of. In order to achieve that, and considering the format of the data that were sent to the DFEs and is described in Section 4.2.2, I had to take slices of each input and to reformat them to 8-bit integers. Below is a code snippet that was actually used as part of the DFE code.

```
      static final DFEType type8 =  dfeInt(8);
      DFEVar in = io.input("x",type);

      DFEVar CentralR = in.slice(16,8).cast(type8);
      DFEVar CentralG = in.slice(8 ,8).cast(type8);
      DFEVar CentralB = in.slice(0 ,8).cast(type8);
```

Note, that all input data are zero filled in order to get 32-bit long integers and the first 8 zeros are irrelevant for the editing.

Furthemore, some filters (Smooth, Edge Enhancement and Sobel), in addition to the colour components, need a window of the surrounding pixels to operate. Thus, I had to temporarily access data (pixels) that were in a previous or in a subsequent position in the current stream. All the windows that were used are 3x3 and the size of each frame is 640x480. So, for example, the upper left pixel of the window is 641 positions before the current pixel for editing. The same logic applied for the other pixels in the window as well.

```
      static final DFEType type8 =  dfeInt(8);
      DFEVar in = io.input("x",type);

      DFEVar PAr = ((stream.offset(in,-641 )).slice(16,8)).cast(type8);
      DFEVar KDg = ((stream.offset(in,641 )).slice(8,8)).cast(type8);
```

This is an excerpt of the code that demonstrates how I accessed the Red component of the upper left pixel and the Green component of the lower right pixel and reformatted them to 8-bit integers.

Let us now review all the filters that were implemented as workload in the DFEs along with the result of each of them.

5.3.1 Edge detection

Edge detection is one of the most important studies of the Digital Image Processing field, because studying the edges can provide valuable information about the description, the formation and the identification of objects that are included in digital images.

The edge is defined as a curve, with both sides having significant differences in luminosity or one or more sourounding attributes. The attributes may be the density, the texture and the variation of the grey levels of the image. There is not a general mathematical model of the definition of the edge. Edges are characterized by their height, width and gradient.

The Sobel Operator, often called Sobel Filter, is an edge detection algorithm that creates an image with emphasis on the edges and the transitions. It was presented by Irwin Sobel in 1968 [30]. It is a discrete differentiation operator, computing an approximation of the gradient of the image intensity function. At each pixel in the image, the result of the Sobel operator is either the corresponding gradient vector or the norm of this vector. The Sobel operator is based on convolvin the image with a small, separable and integer valued filter in horizontal and vertical direction and it is therefore relatively inexpensive in terms of computations.

The operator uses two 3x3 kernels which are convolved with the original image in order to calculate approximations of the deratives; one for horizontal changes and one for vertical. Let

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * Im$$

Im be the original image, and G_x and G_y are two images which at each pixel contain the horizontal and vertical derivative approximations, the computations are as follows:

Figure 19: Sobel Operator: The coefficients on the X axis

Figure 20: Sobel Operator: The coefficients on the Y axis

where * is the convolution sign.

For the implementation of this filter in the DFEs, I multiplied each pixel on the window with its corresponding coefficient from the G_x and G_y arrays.Then I summed the results and took their

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * Im$$

absolute values.

```
        DFEVar SobelX,SobelY;
        int [][] Gx = { {-1, 0, 1}, {-2, 0, 2}, { -1,  0,  1} };
```

```
      DFEVar pvX= constant.var(0);

      pvX += Gx[0][0]*PAr;
      pvX += Gx[0][1]*Pr;
      pvX += Gx[0][2]*PDr;
      pvX += Gx[1][0]*Ar;
      pvX += Gx[1][1]*Kentror;
      pvX += Gx[1][2]*Dr;
      pvX += Gx[2][0]*KAr;
      pvX += Gx[2][1]*Pr;
      pvX += Gx[2][2]*KDr;
      SobelX = (pvX>0).ternaryIf(pvX, -pvX);
```

Each colour component of the final result is the concatenation of the summary of these values, SobelX for the X axis and SobelY for the Y axis.

```
Sobel = blanc8.cat(((SobelX+SobelY).cat(SobelX+SobelY)).cat(SobelX+SobelY));
```

Where blanc8 is an integer with eight zeros, necessary for the zero filling.

Finally, the result of my implementation is as follows.

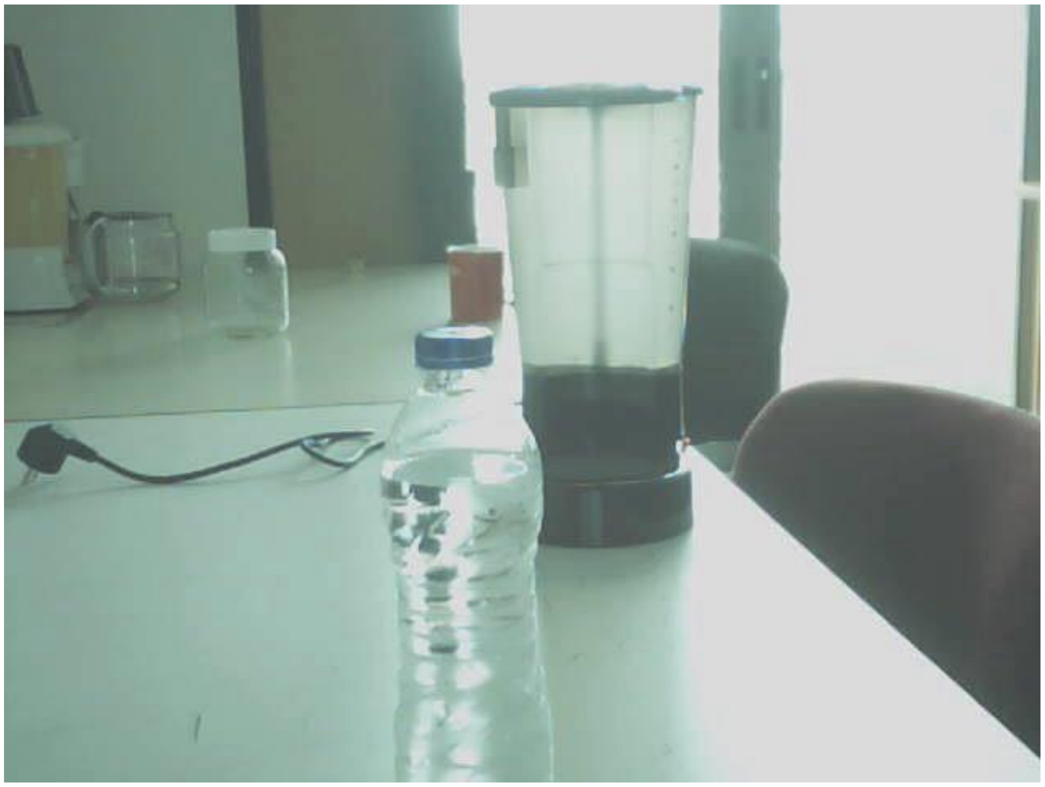

Figure 21: Sobel Operator: original Snapshot

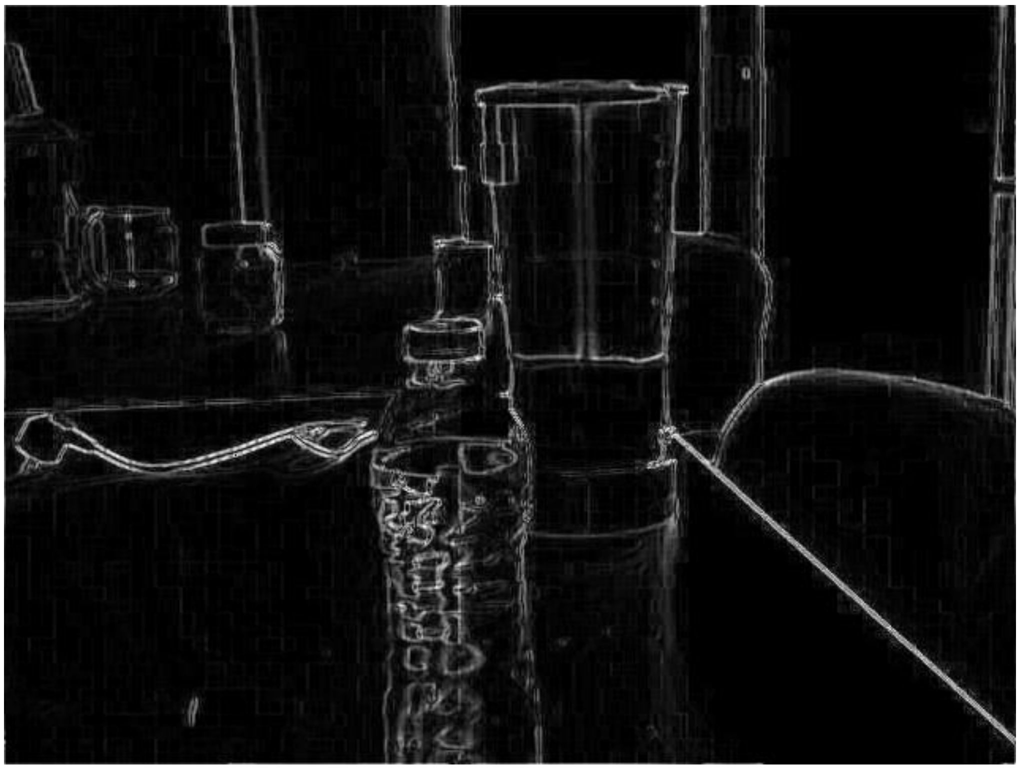

Figure 22: Sobel Operator: result

We can discern that even the smallest, like in the lower right corner, transitions are detected.

5.3.2 Edge Enhancement

Edge enhancement is an image processing filter that enhances the edge contrast of an image or video in an attempt to improve its acutance (apparent sharpness). The filter works by identifying sharp edge boundaries in the image, such as the edge between a subject and a background of a contrasting colour, and increasing the image contrast in the area immediately around the edge. This has the effect of creating subtle bright and dark highlits on either side of any edges in the image, called overshoot and undershoot, leading the edge to look more defined when viewed from a typical viewing distance. The most common uses of this filter are on televisions and computer printers.

Unlike some forms of image sharpening, edge enhancement does not enhance subtle detail which may appear in more uniform areas of the image, such as texture or grain, which appears in flat or smooth areas of the image. The benefit to this is that imperfections in the image reproduction, such as grain or noise, or imperfections in the subject, such as natural imperfections on a person's skin, are not made more obvious by the process. A drawback to this is that the image may begin to look less natural, because the apparent sharpness of the overall image has increased but the level of detail in flat, smooth areas has not.

The algorithm that was used in order to implement this filter dictates that for each pixel in the 3x3 mask, one of the colour attributes (Red was selected) is multiplied by the coresponding value in the matrix below.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 23: Edge Enhancement: The coefficient matrix

Finally, all products are added and the absolute value of the final summary was added to the initial value of each attribute (Red, Green, Blue).

The result of my implementation.

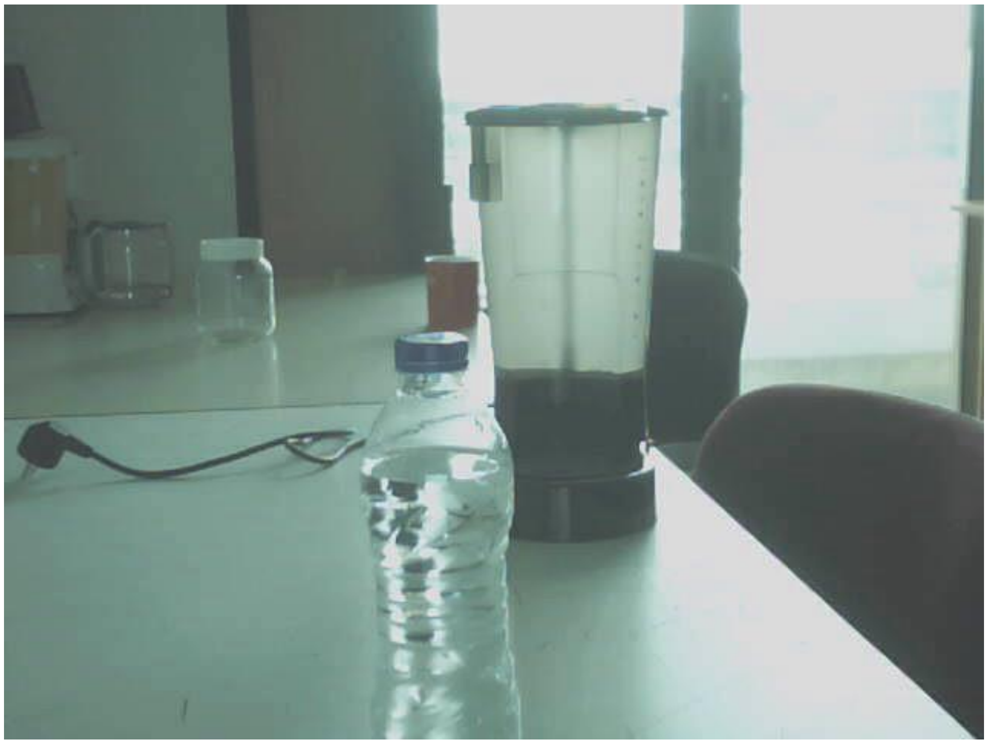

Figure 24: Edge Enhancment Filter: original Snapshot

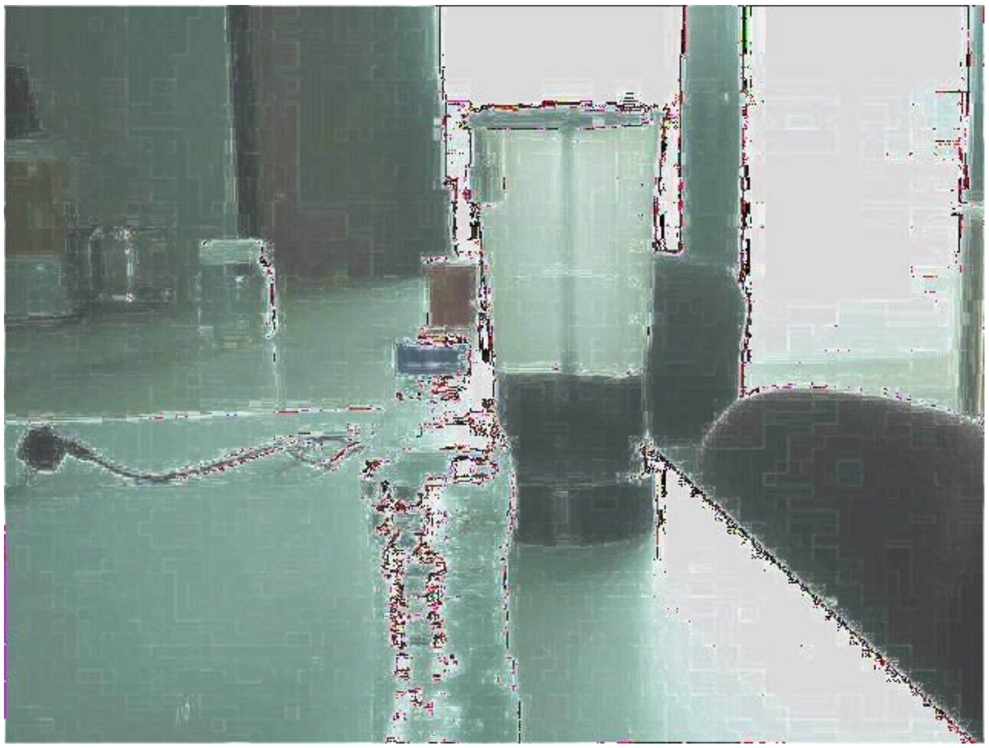

Figure 25: Edge Enhancment Filter: result

All the edges are highlighted either in white, for the small ones, or in dark for the larger ones.

5.3.3 Smooth Filter

In many experiments in physical science, statistics and digital image processing, the true signal (data set) amplitudes change rather smoothly as a function of the x-axis values, whereas many kinds of noise are seen as rapid, random changes in amplitude from point to point within the signal (data set). In the latter situation it may be useful in some cases to attempt to reduce the noise by a process called smoothing. In smoothing, the data points of a signal (data set) are modified so that individual points that are higher than the immediately adjacent points (presumably because of noise) are reduced, and points that are lower than the adjacent points are increased. This naturally leads to a smoother signal (data set). As long as the true underlying signal (data set) is actually smooth, then the true signal (data set) will not be much distorted by smoothing, but the noise will be reduced. Smoothing may be used in two important ways that can aid in data analysis

1. By being able to extract more information from the data as long as the assumption of smoothing is reasonable and

2. By being able to provide analyses that are both flexible and robust.

Linear smoothers

In the case that the smoothed values can be written as a linear transformation of the observed values, the smoothing operation is known as a linear smoother; the matrix representing the transformation is known as a smoother matrix. The operation of applying such a matrix transformation is called convolution. Thus the matrix is also called convolution matrix or a convolution kernel.

Mean Filter

Mean filter**,** or average filter is windowed filter of linear class, that smoothes signal (image). The filter works as a low-pass one. The basic idea behind the filter is for any element of the signal (image) take an average across its neighborhood.

```
SmoothR = (PAr.cast(dfeUInt(8))/c + Pr.cast(dfeUInt(8))/c + PDr.cast(dfeUInt(8))/c
+ Ar.cast(dfeUInt(8))/c + Dr.cast(dfeUInt(8))/c + KAr.cast(dfeUInt(8))/c +
Kr.cast(dfeUInt(8))/c + KDr.cast(dfeUInt(8))/c));
```

This code was used for calculating the average value of the current pixel for the Red component. Note that the MaxCompiler does not support, yet, the functionality of dividing a summary of variables inside a parenthesis with a divisor. The divisor must be divided individually with each addendum, otherwise it leads to unpredictable result. The result of this implementations are as follows.

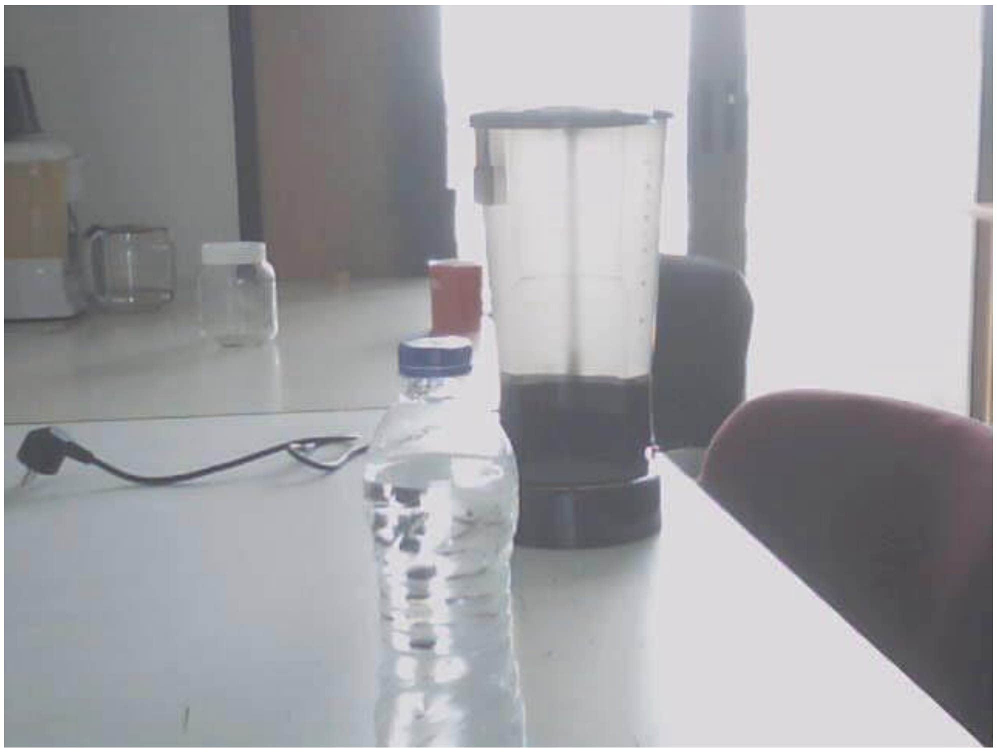

Figure 26: Smotth Filter: original Snapshot

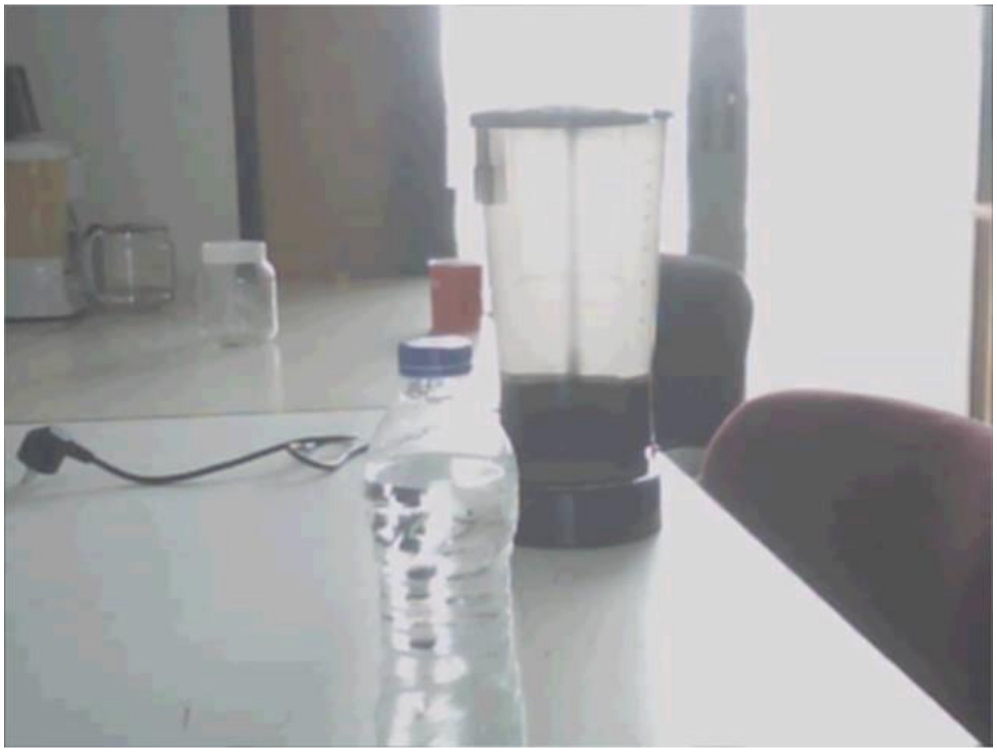

Figure 27: Smooth Filter: result

We can observe that the result has smoothier transitions from the brighter parts to the darker ones.

5.3.4 Invert Filter

The Invert filter inverts all the pixel colors and brightness values in the current image, as if the image were converted into a negative. Dark areas become bright and bright areas become dark. Hues are replaced by their complementary colors.

The algorithm that was used is quite simple; Subtract each colour attribute (Red, Green, Blue) from the maximum possible value of the attribute, which is 255. Following, there is the result.

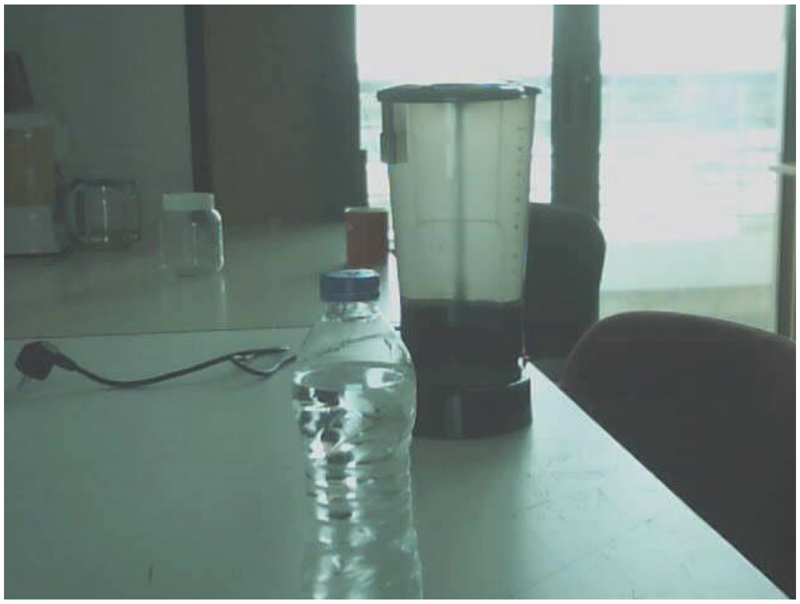

Figure 28: Invert Filter: original Snapshot

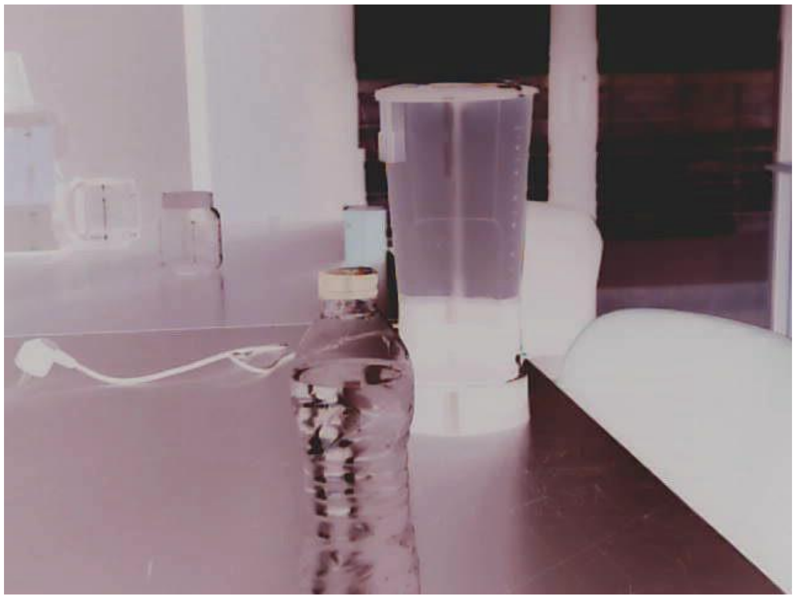

Figure 29: Invert Filter: result

We can observe that each colour has been substituted by their negative one. The overall red feeling on the result, is due to the blue one in the original.

5.3.5 Quantization Filter

Quantization in image processing, is a lossy compression technique achieved by compressing a range of values to a single quantum value. When the number of discrete symbols in a given stream is reduced, the stream becomes more compressible. For example, reducing the number of colours required to represent a digital image makes it possible to reduce its file size.

Color quantization reduces the number of colors used in an image; this is important for displaying images on devices that support a limited number of colors and for efficiently compressing certain kinds of images. Most bitmap editors and many operating systems have built-in support for color quantization.

It is a common compression technique used by many codecs and containers, such as JPEG[31] and MPEG[32].

For the implementation of this filter, the creation of a threshold was required. Each attribute of each pixel was quantised either at 64 or at 192, depending if its value was over or under the threshold of 128.

The results follow.

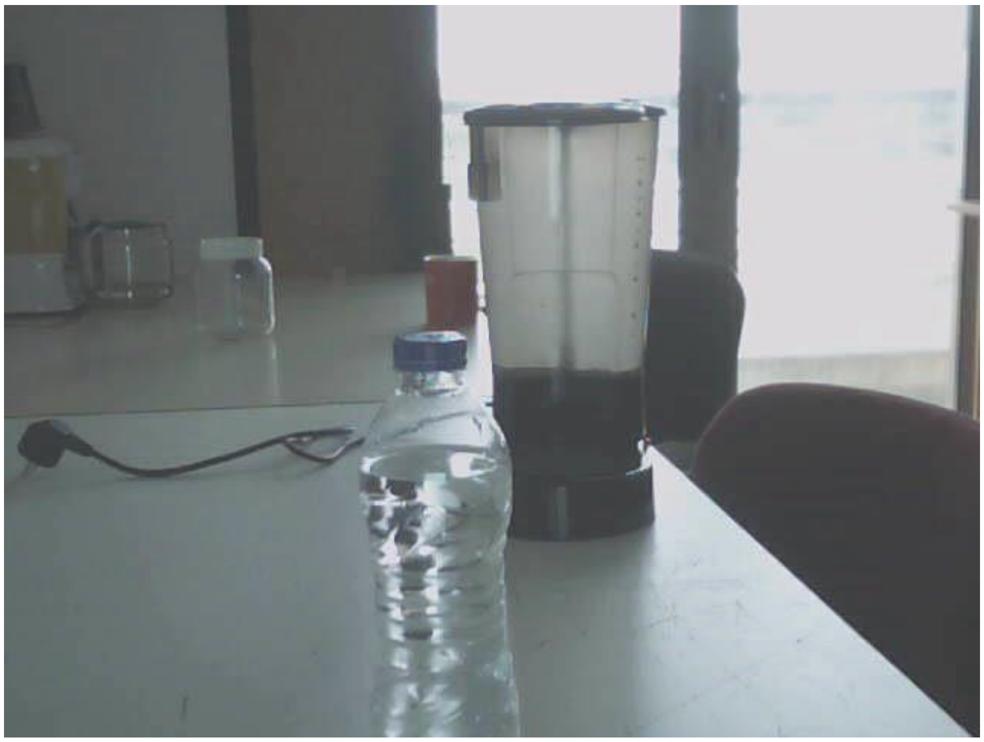
Figure 30: Colours Quantization Filter:original Snapshot

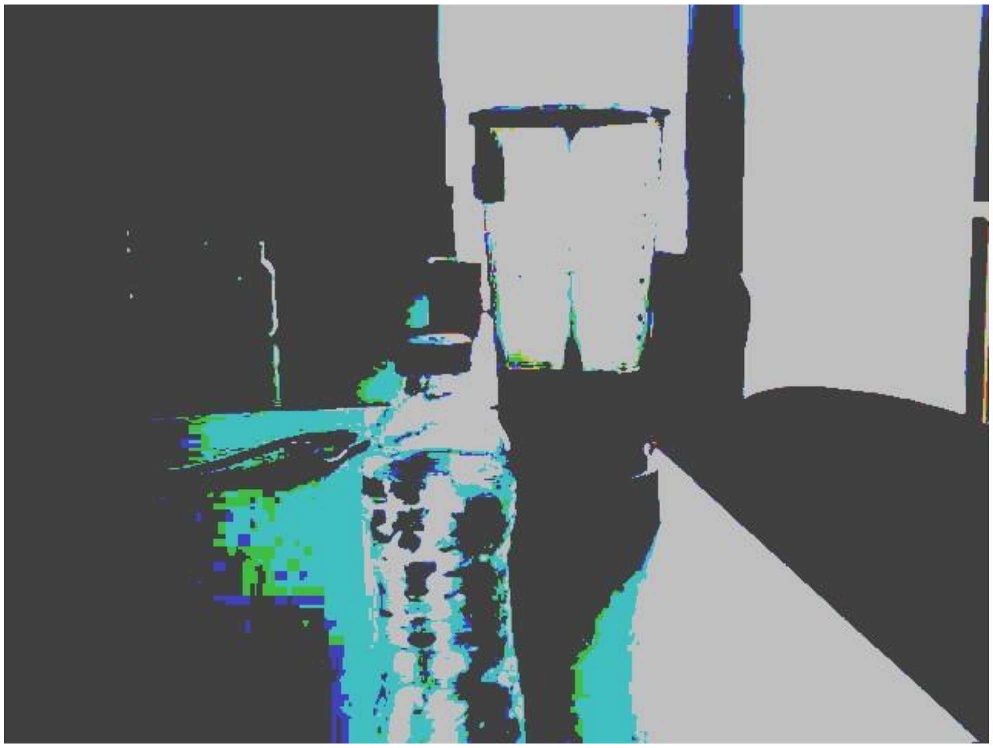

Figure 31: Colours Quantization Filter: result

As we observe, there are only four colours in the result image. The information in the image is visible even with such a small number of colours.

6. Conclusion and Future Work

6.1 Conclusion

This thesis demonstrates the capabilities of the Dataflow architecture, specifically the capabilities of the Maxeler MPC-C Series as a real-time coprocessor. The bottleneck of the Live Video Stream application, that was implemented as a test case, is the bandwidth of the network that was used, the network of T.U.C. As described in Chapter 5, the speedup that I got in the stress test is about 17% in relevance to a conventional control flow architecture, using approximately the 70% of the DSPs and only 12% of the LUTs available. Also, the result of the above mentioned experiments lead to the conclusion that the ideal use of the Maxeler System is to have a small number of DFE calls per application with the maximum possible size of dataset per system call. The use of all available streams per system call is highly encouraged. Furthermore, with this thesis, many functions that are commonly used were implemented. This will help the students and the staff members of T.U.C. to build on what I have developped and avoid reinventing the wheel. To sum up, it has been proven that it can cope really well with the workload I have feed it and is has no problem delivering really fast. It is clear that for certain kind of computations, the Dataflow architecture with its inherited parallelism offers great advantages.

6.2 Future work

In this thesis, only two protocols were used, the TCP/IP and the UDP, which have some benefits and some drawbacks, as described in Chapter 4. One could test the system with other protocols in order to avoid the bottleneck that is created in the Server – WorkStations communication.

Furthermore, a stress test that uses the resources of the Maxeler MPC-C Series, both the DSPs and the LUTs, as close as possible to 100% is in order.

Also, a speed test on the communication between the DFEs via MaxRing should be implemented.

One more issue is that although I have found that the throughput of the DFEs are 144 MB/s, one can implement specialised tests in order to achieve speed as close to the theoretical value of 250 MB/s as possible.

Finally, the whole system is Linux (CentOS) specific. For the convenience of the future users, it could be redevelloped to cross-platform.

7. References

[1] J. J. Wang, R. B.Katz, J. S. Sun, B. E. Cronquist, J. L. McCollum, T. M. Speers, W. C. Plants. “SRAM based re-programmable FPGA for space applications”. *IEEE Transactions on Nuclear Science*. Volume 46, Issue 6. ISSN: 0018-9499. 1999.

[2] D. Marculescu, R. Marculescu, N. H. Zamora, P. Stanley-Marbell, P. K. Khosla, S. Park, S. Jayaraman, S. Jung, C. Lauterbach, W. Weber, T. Kirstein, D. Cottet, J. Gryb, G. Troster, M. Jones, T. Martin, Z. Nakad. “Electronic textiles: A platform for pervasive computing”. *Proceedings of the IEEE.* Volume 91, Issue 12. ISSN: 0018-9219. 2003

[3] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, A. Vahdat. “PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric”. *In ACM SIG on Data Communications (SIGCOMM).* August 17-21 2009

[4] A. J. Plaza, C.-I Chang. *High Performance Computing in Remote Sensing.* Chapman & Hall/CRC computer & information science series. 2008. ISBN: 978-1-58488-662-4

[5] G. Seeber. *Satellite Geodesy: Foundations, Methods, and Applications 2nd revision.* Gottingen: Hubert & Co. GmbH & Co. Kg. 2003. ISBN: 3-11-017549-5

[6] K. Kosiba, J. Wurman, F. J. Masters, P. Robinson. “Mapping of Near-Surface Winds in Hurricane Rita Using Finescale Radar, Anemometer, and Land-Use Data”. *Monthly Weather Review from AMS.* Volume 141, Issue 12. ISSN: 4337-4349. December 2013

[7] M. Tanaka, T. J. Moriya, N. Yoshida, K. Nomoto. “Detectability of high-redshift superluminous supernovae with upcoming optical and near-infrared surveys”. *Monthly Notices of the Royal Astronomical Society.* Volume 422, Issue 3. ISSN: 2675-2684. 2012

[8] A. R. Thompson, J. M. Moran, G. W. Swenson Jr. *Interferometry and Synthesis in Radio Astronomy 2nd Edition.* Weinheim: WILEY-VCH Verlag GmbH & Co. KgaA. 2004. ISBN: 978-0-471-25492-8

[9] R. V. Tona, C. Benqlilou, A. Espuna, L. Puigjaner. “Dynamic Data Reconciliation Based on Wavelet Trend Analysis”. *Industrial & Engineering Chemistry Research Journal.* Volume 44, Issue 12. pp: 4323-4335. 12 May 2005

[10] J. B. Bremnes. “Probabilistic Forecasts of Precipitation in Terms of Quantiles Using NWP Model Output”. *Monthly Weather Review from AMS.* Volume 132, Issue 1. ISSN: 338-347. January 2004

[11] G. Rematska, K. Papadimitriou, A. Dollas. “A Low-Cost Embedded Real-Time 3D Stereo Matching System for Surveillance Applications”. *In IEEE International Symposium on Monitoring and Surveillance Research (ISMSR), in conjuction with the IEEE International Conference on Bioinformatics and Bioengineering (BIBE)*. Chania, Greece.November 2013

[12] S. Muthukrishnan, “Data Streams: Algorithms and Applications”, now Publishers Inc. 2015. Chapter 4: Data Streaming: Formal Aspects. Section 4.1: Data Stream Models. pages 8 – 10. ISBN: 9781933019147

[13] S. Bhattacharyya, S. Moon. “Network monitoring and measurements: Techniques and experience”. *In Tutorial at ACM Sigmetrics 2002.* Marina Del Rey, CA. June 2002

[14] P. C. Treleaven, R. P. Hopkins, P. W. Rautenbach. “Combining Data Flow and Control Flow Computing”. *The Computer Journal.* Volume 25, Issue 2. 1982

[15] B. Lee, A.R. Hurson, “Issues in Dataflow Computing”, *Advances in Computers vol.37*. Academic Press Inc. 1993. ISBN: 0-12-012137-9

[16] G. R. Trimble Jr. “A brief history of computing. Memoirs of living on the edge”. *IEEE Annals of the History of Computing.* Volume 23, Issue 3. pp 44-59. Summer 2001

[17] J. R. McGraw. “The VAL Language: Description and Analysis”. *ACM Transactions on Programming Languages and Systems (TOPLAS).* Volume 4, Issue 1. January 1982

[18] Arvind et al. “An Asynchronous Programming Language for a Large Multiprocessor Machine”. TR114a. Dept ISC. UC Irvine. December 1978

[19] J. T. Feo et al. “A Report on the SISAL Language Project”. *J Parallel and Distrib Computing.* Volume 10, Issue 4. pp 349-366. December 1990

[20] O. Mencer. “Maximum performance computing for exascale applications”. *In International Conference on Embedded Computer Systems (Samos) 2012.* E-ISBN: 978-1-4673-2296-6. 16-19 July 2012

[21] Maxeler Technologies Inc. “Dataflow Computing”. 2015. Available: http://www.maxeler.com/technology/dataflow-computing (Accessed 10/03/2015)

[22] New Wave DV. “Quad Channel 10 Gigabit Ethernet FPGA PCI Express Card Datasheet Revision 1”. V5031 datasheet. 2013. page 2

[23] Convey Computer Corporation. “Convey Coprocessor Reference Manual v1.1”. 21/05/2012. Chapter 2: HC System Organization, Section 2.1: System Architecture

[24] Convey Computer Corporation. “Convey PDK Reference Manual v5.2”. April 2012. Chapter 2: Coprocessor Architecture. page 2

[25] Maxeler Technologies Inc. “MaxGenFD: Tutorial v2011.3.1”. Chapter 1: Introduction to MaxGenFD, Section 1.7.2: Parallelism. page 9-11

[26] M. J. Flynn. “Dataflow SuperComputers”. In 22nd International Conference on Field Programmable Logic and Applications (FPL) 2012. Maxeler Technologies and Stanford University. page 25

[27] G. Chrysos, E. Sotiriadis, C. Rousopoulos, K. Pramataris, I. Papaefstathiou, A. Dollas, I. Kirmitzoglou, V.J. Promponas, T. Theocharides, G. Petihakis, J. Lagnel. “Reconfiguring the

Bioinformatics Computational Spectrum: Challenges and Opportunities of FPGA-Based Bioinformatics Acceleration Platforms”. *IEEE Design & Test*. Volume 31. Issue 1. ISSN: 2168-2356. October 2013

[28] D. A. Wheeler, “Secure Programming HOWTO v.3.65”. [Ebook]. 01/03/2015. Chapter 3: Summary of Linux and UNIX Security Features. Section 3.4: Sockets and Network Connections. Available: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html (Accessed 12/03/2015)

[29] S. Goldt, S. V. D. Meer, S. Burkett, M. Welsh. “The Linux Programmer’s Guide v0.4”. March 1995. Chapter 6: Linux Interprocess Communications. Section 6.3: Named Pipes (FIFOs – First In First Out). pages 27-30. Available: http://www.tldp.org/LDP/lpg-0.4.pdf. (Accessed 12/03/2015)

[30] I. Sobel, G. Feldman. “A 3x3 Isotropic Gradient Operator for Image Processing”. Presented at the Stanford Artificial Intelligence Project (SAIL). 1968

[31] Siu-Wai Wu. A. Gersho. “Rate-constrained picture-adaptive quantization for JPEG baseline coders”. In IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-93). 27-30 April 1993. ISSN: 1520-6149

[32] Wei Ding, B. Liu. “Rate control of MPEG video coding and recording by rate-quantization modeling”. *IEEE Transactions on Circuits and Systems for Video Technology.* Volume 6. Issue 1. ISSN: 1051-8215. February 1996.