TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

BECKERS LANDER, LAKIERE HENNING

EFFICIENT DISTRIBUTION OF A COMPUTATION INTENSIVE CALCULATION ON AN ANDROID DEVICE TO EXTERNAL COMPUTE UNITS WITH AN ANDROID API

Master of Science thesis

Examiner: Prof. Nurmi Jari
Examiner and topic approved by the
Dean of the Faculty of
Computing and Electrical Engineering
on 26th April 2017

ABSTRACT

BECKERS LANDER, LAKIERE HENNING: Efficient distribution of a computation intensive calculation on an Android device to external compute units with an Android API
Tampere University of Technology
Master of Science thesis, 82 pages
6th of June 2017
Major: Embedded Systems
Examiner: Prof. Nurmi Jari
Keywords: Parallel computing, SoC, CPU, FPGA, Android, Bluetooth WebSocket, OpenCL, C++, Java

English abstract

Is transferring computation intensive calculations to external compute-units the next trend? This master's thesis researches if it is worth the effort to transfer a matrix multiplication from an Android phone to a System-on-Chip (SoC), using Bluetooth or WebSocket as communication protocols. The SoC solution used in this work is an Intel Altera Cyclone V based board from TerASIC, equipped with a Field Programmable Gate Array (FPGA) including a Dual-core ARM A9 processor. Because the matrix size has a strong correlation to the number of calculations in a matrix multiplication, the calculation time on a CPU and FPGA will differ when the matrices grow in size. Comparing the multiplication times on Android and SoC, matrices with a matrix size above 1660x1660 are calculated faster on the SoC.

The matrix multiplication is accelerated using an OpenCL kernel on the FPGA, guided by a host program on the processor programmed in C++.

Experiments have shown that Bluetooth has a 500 times lower transfer rate than WebSocket, resulting in choosing only WebSocket for further investigations. Due to the transfer times, the minimum matrix size to win time by extending the multiplication to a SoC is 2338x2338. Although the implemented matrix multiplication does only support square matrices, future research could develop multiple kernels of different algorithms that support a variation in width and height.

Dutch abstract

Is processor intensieve berekeningen overbrengen naar externe rekeneenheden een nieuwe trend? Deze master thesis onderzoekt of het sneller is om een matrixvermenigvuldiging door te sturen naar en uit te rekenen op een externe rekeneenheid in plaats van direct op de smartphone. Matrices worden via Bluetooth of WebSocket tussen de smartphone en System-On-Chip (SoC) uitgewisseld. De gebruikte rekeneenheid is TerASIC's SoC, uitgerust met een Intel Altera Cyclone V Field Programmable Gate Array (FPGA) en Dual-core ARM A9 processor. Indien de matrices groter zijn dan 1660x1660, dan zal de SoC sneller de matrix vermenigvuldiging kunnen berekenen dan de Android smartphone.

De matrixvermenigvuldiging wordt geaccelereerd in een OpenCL kernel op de FPGA, gestuurd door een in C++ geschreven programma op de processor.

Experimenten toonden aan dat de overdrachtssnelheid van Bluetooth 500 keer trager is dan WebSocket, verdere experimenten zullen dus beter geen Bluetooth implementeren. Matrices groter dan 2338x2338 zijn sneller doorgestuurd, vermenigvuldigt en teruggestuurd op de SoC, dan op de Android smartphone zelf. Verder onderzoek zou andere OpenCL kernels kunnen ontwikkelen die andere berekeningen uitvoeren op niet-vierkante matrices.

PREFACE

This thesis is written in order to complete our master education in electronics and ICT engineering at UHasselt Belgium. The development and writing of this thesis has been done in the Tampere University of Technology, Finland. K. Aerts was the supervisor of our home university, Prof. J. Nurmi and PhD student K. Wang supervised us during the exchange period.

After several meetings with our supervisors we had a solid idea of what the thesis should be about and what the research question should be. We had some major problems with multiple software versions, but thanks to dedicated research and trial and error we were able to fix all of these issues.

We would like to thank our supervisors to contribute in our work and keep us motivated along the whole experience.

Thanks, to all our family members who visited us here in Finland to provide us with some comfort after surviving the cold winter and to all new friends we met here. Without the other exchange students, this experience would not have been this great.

Lander Beckers
Henning Lakiere

Tampere, (6th of June 2017)

Authors' Contributions		
Chapter	Section	Author
Poster		Lander
Abstract		Lander
Preface		Henning
1 Introduction		Lander
2 Theoretical background	2.1 Matrix multiplication	Lander
	2.2 Parallel computing	Henning
	2.3 Smartphone	Henning
	2.4 Communication	Henning
	2.5 SoC	Lander
3 Implementation	3.1 Communication protocol	Henning
	3.2 SoC	Lander
	3.3 Android Apps	Henning
4 Results	4.1 OpenCL performance	Lander
	4.2 Phone performance	Henning
	4.3 Bluetooth vs WebSocket	Henning
	4.4 Conclusion of the results	Lander
5 Conclusion		Lander

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ASIC	Application-Specific Integrated Circuit
CAD	Computer Aided Design
CPU	Central Processing Unit
DDR	Double Data Rate
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input Output
GPU	Graphical Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HPS	Hardware Processing System
IP	Intellectual Property
IC	Integrated Circuit
OS	Operating System
PCB	Printed Circuit Board
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on Chip
SOF	SRAM Object File
UART	Universal Asynchronous Receiver Transmitter
Verilog	Is a HDL like VHDL, but with a different syntax
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration

1. INTRODUCTION

Since Baron J.J.B discovered silicon in 1823, 125 years passed by before the first transistor was created. Transistors are the main components in all electronic circuits. It took another 25 years to publish the first microprocessor, named Intel 4004. The 4-bit Intel 4004 was able to perform 60.000 operations per second and was built with 2300 transistors. Five years earlier, in 1965, Gordon E. Moore made a statement in a paper [10], called "Moore's law". Moore's law is now a computing term, specifying that the number of transistors available on an integrated circuit doubles every two years. 50 years after he published his paper, his law is still standing. Every year since, processors kept increasing their clock frequency, internal memory and register width. The register width of the Intel 4004 was 4 bits. 4 bits increased to 8, 16, 32 and nowadays commercial available 64-bit architectures.

Ten years after the first commercial processor, Intel 4004 [14], was released, the first Field Programmable Gate Arrays (FPGAs) were created [35]. They where difficult to program, expensive and had a small amount of configurable logic blocks. Most designers avoided them and used processors and/or ASICs. ASICs are Application-Specific Integrated Circuits, designed for a specific purpose. When the application required parallel data processing, ASICs were used. Disadvantage of ASICs are the high initial production costs and difficult development. Although these days FPGAs have more logic blocks than ASICs and are partially reconfigurable after final configuration, ASICs are still used in high mass productions.

A next step in the evolution could be a combination of both Central Processing Unit (CPU) and FPGA. CPUs are amazingly stable, cheap and it is easy to develop programs for execution. FPGAs on the other hand are relative more expensive, more difficult to configure and consume more energy than comparable CPUs. Although right now you might think that CPUs are better than FPGAs, FPGAs have one advantage above all CPUs: they can massively process data in parallel. There exists something in-between, called a Graphical Processing Unit (GPU). GPUs differ from FPGAs with their fixed internal configuration and are able to compute the same instruction on multiple data. More about this can be found in 2.2: Parallel computing.

The following questions must be answered in this thesis, in order to answer our research question: "How to most efficiently distribute a computation intensive calculation on an Android device to external compute units with an Android API?". During this thesis a matrix mul-

tiplication from an Android phone is transferred to a System-on-Chip (SoC). The used SoC in this thesis is TerASICs DE1SoC, equipped with an Intel Altera Cyclone V FPGA and a Dual-core ARM A9 processor. The main purpose of extending this calculation to a SoC, is achieving a faster matrix multiplication result. The most interesting question will be: can a matrix multiplication, extended to a SoC, calculate a faster result than a multiplication on the Android device and/or at which matrix size is it profitable? Other interesting questions are, which communication protocol will be the most reliable and fastest to exchange matrices with the SoC. We will implement two communication protocols, Bluetooth and WebSocket.

This thesis comprises three chapters: Theoretical background, Implementation and Results. Firstly, we will describe which knowledge and resources are used to develop the programs for the SoC and the Android application. Secondly, the complete implementation is explained in depth. Beginning with the character error detection algorithm, this algorithm is mandatory to ensure zero package loss during Bluetooth communication. There are two other parts in this chapter: SoC and Android. Section SoC implements both Hard Processor System (HPS) and FPGA systems. The HPS is used as a host to start everything: receiving, calculating in OpenCL and receiving with both Bluetooth and WebSocket as communication protocols. During the calculations or when Bluetooth is used, the FPGA needs to be programmed and accessed from HPS. All calculations are performed by OpenCL on the FPGA side of our SoC. Android is our "customer", if a matrix calculation needs to be processed, then the Android application can connect to the SoC and transfer the matrix multiplication. Thirdly, we will discuss the results in chapter "results". We consider two questions useful to solve this thesis's problem. How does the OpenCL calculation perform compared to a single threaded CPU? The basic step is to verify if OpenCL is able to accelerate the matrix multiplication, otherwise our research question is already unfeasible. Afterwards, the communication to share the matrices has to be quick enough to ensure a faster result matrix on the Android. When the total matrix transmission times added with the OpenCL calculation time is larger than the single treaded matrix multiplication on Android, then already a final conclusion is achieved. The last chapter "Conclusion" will respond to all questions asked in this paragraph.

2. THEORETICAL BACKGROUND

All used knowledge and resources are explained in depth in this chapter. The basic matrix multiplication algorithm is explained in section 2.1 and a basic analysis is made to count the number of calculations depending on the matrix size. In the next section 2.2 describes and classifies parallel computing. In 2.3 is discussed how an Android application is compiled, developed and executed. Next, sections 2.4.1 and 2.4.2 explain respectively the basics of Bluetooth and WebSocket communication. Every component of the matrix multiplication accelerator "SoC" can be found in section 2.5, those components are the HPS, the FPGA and the bridges between the two combined with the development software used.

2.1 Matrix multiplication

Before explaining why we have chosen a matrix multiplication as a computation intensive algorithm, we will show how matrices A and B are multiplied in equation 2.1 [34].

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 1 \times 5 + 2 \times 7 & 1 \times 6 + 2 \times 8 \\ 13 \times 5 + 4 \times 7 & 3 \times 6 + 4 \times 8 \end{bmatrix} = \begin{bmatrix} 19 & 22 \\ 93 & 50 \end{bmatrix} \tag{2.1}$$

This is the most common method to multiply two matrices and it works in every situation. When a calculation is complex, people tend to divide the problem into a number of multiple smaller/easier problems. Using matrices to describe functions is one of those easier ways to deal with real life problems. That is the reason why most common tools in engineering use matrices. The numbers in a matrix represent the data from measurements or approximations given by mathematical equations. In many time-sensitive applications a faster method to solve matrix calculations could give faster approximations for real life problems [11]. We needed to choose a specific matrix calculation to implement and have set the shape of the matrices as square. The reason why we have chosen a matrix multiplication is concluded from equation 2.2. This equation clearly shows that the number of calculations grows faster than the matrixSize to the power of 3. By parallelizing, all calculations are placed in parallel, resulting in a bigger throughput.

$$numberOfCalculations = (2 \times size - 1) \times size^2 \tag{2.2}$$

$$M = \begin{bmatrix} A & B \\ .. & .. \end{bmatrix} \tag{2.3}$$

$$N = \begin{bmatrix} F & .. \\ G & .. \end{bmatrix} \tag{2.4}$$

$$R1 = \begin{bmatrix} A \times F + B \times G & .. \\ .. & .. \end{bmatrix} \tag{2.5}$$

This is how we derive equation 2.2. First, the number of calculations when calculating the first element of the result matrix R1 are counted. When processing the matrix multiplication M x N a count of three calculations, i.e. one addition and two multiplications, is achieved. After processing the same calculation for a matrix with size five, the result matrix R2 in equation 2.6 will be calculated. Result matrix R2 contains 4 additions and 5 multiplications for each element, resulting in 20 (4 x 5) calculations. The number of calculations for one element in the result matrix is always equal to "number of additions" added with "number of multiplications" and the number of elements in a result matrix are equal to matrixSize x matrixSize. When those are combined in a formula, this results in equation 2.2.

$$R2 = \begin{bmatrix} A \times F + B \times G + C \times H + D \times I + E \times J & .. & .. & .. & .. \\ .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. \\ .. & .. & .. & .. & .. \end{bmatrix} \tag{2.6}$$

2.2 Parallel computing

Parellel computing has made an uprising in the last couple of decades. Due to some technology constraints developers moved to multicore processors. This is not the only way of parallel computing, but more will be explained in the next sections.

2.2.1 What is parallel computing

The problem when facing large calculations is that they require a lot of computing power and thus time. Performing these calculations can be done with two types of computation, serial

and parallel computing. Serial computing means, you have one compute unit (e.g. a single-core CPU) available that performs all instructions on a certain set of data. This set of data will be broken into multiple smaller subparts that will be solved by a certain instruction. The single compute unit performs the instruction on every subpart in order to solve the whole part as shown in figure 2.1 [8].

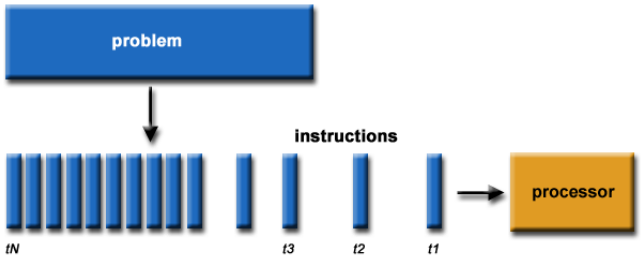


Figure 2.1 *Workflow serial computing*

Parallel computing, on the other hand, is the simultaneous use of multiple compute units, or a compute resource, to solve a computational problem. This compute resource can be a CPU with multiple cores, a combination of a CPU with different compute accelorators such as a GPU or even a whole network of computers and servers. We break the main problem into smaller subproblems as we did with serial computing. Now, since there are multiple compute units, we can distribute the subproblems among all these compute units. Every unit can now perform an instruction on their given subproblem simultaneously as shown in figure 2.2. With multiple

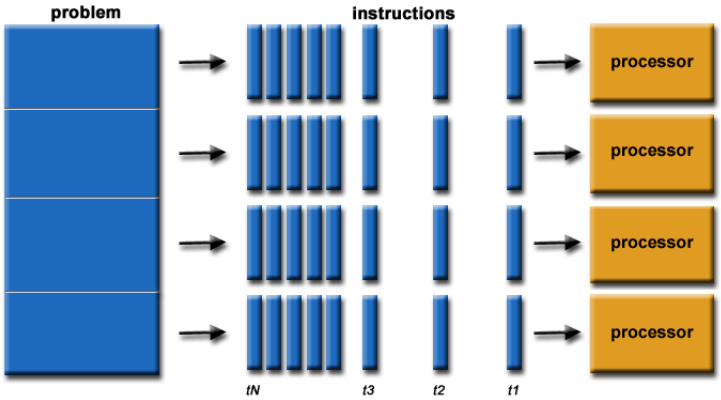


Figure 2.2 *Workflow parallel computing*

compute units executing one task, we will shorten the completion time and even have a potential cost saving. It also allows us to solve larger/complex problems since a single computer could suffer from limited memory. And last, we are able to access non-local resources in a network that would not be accessible from a local computer.

It is easy to conclude that the concept of parallel computing was to have a more efficient way to handle large sets of data such as huge databases, images or simulations that involve large

datasets. It is also easier to deal with complex data for example algorithms [8].

2.2.2 Classification of parallel computing

Parallel computing systems can be separated into different classes. According to Flynn's taxonomy, we can roughly place any of these systems in one of the four classes. This classification was first studied by Michael Flynn in 1972 [3]. The classifications are determined by two factors: instruction stream and data stream which both have two possible states being single or multiple. Figure 2.3 represents a the four possible classes from the Flynn's taxonomy [8].

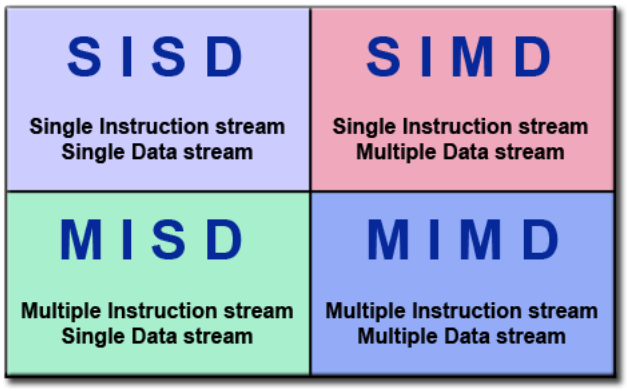


Figure 2.3 *Four possible classifications according to Flynn's Taxonomy*

- SISD

  The SISD class will have a single core processor executing a single data stream to operate on data stored in a single memory (figure 2.4a). This means that a parallel compute system cannot be classified as an SISD system, but Flynn's taxonomy was not made for just classifying parallel compute systems. Any traditional single-core processor falls into this category but it is usually old computers or older compute units that can be classified as SISD.

- SIMD

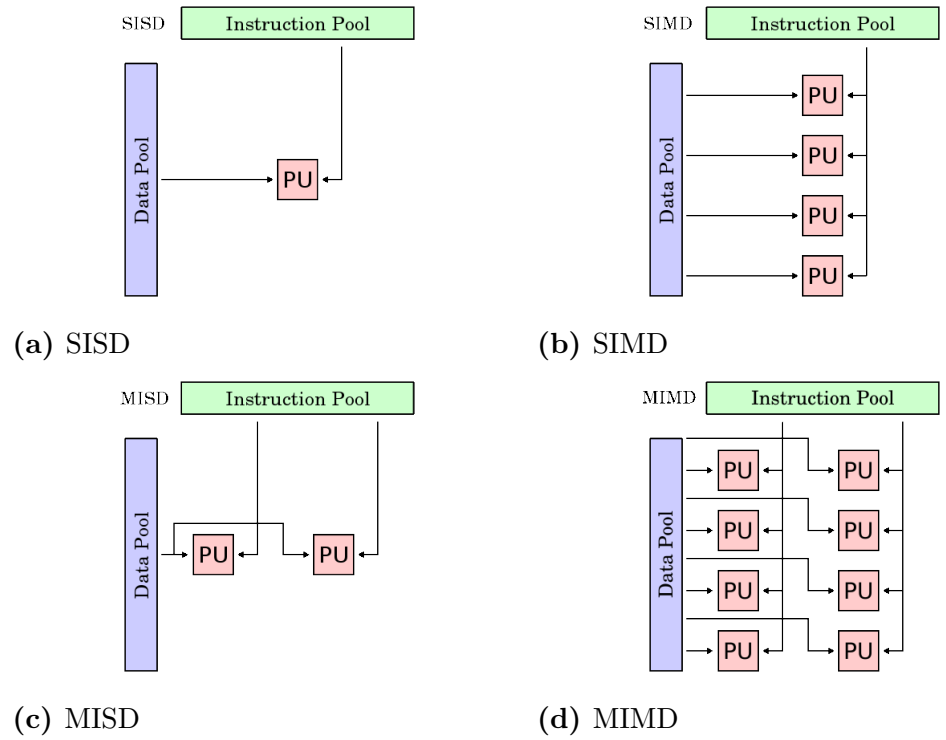
  Data is distributed amongst multiple processors who all execute the same instruction on this data (figure 2.4b). Since we have access to multiple compute units, parallel computing can be categorized in this class. Furthermore, this is the class where we can categorize our subject of the thesis in since we have a large set of data divided over multiple data streams (MD) and only one instruction stream (SI) since all processing units will perform the same instruction on the data. The SIMD class contains the most modern computers, particularly those with a graphics processor unit (GPU).

- MISD

  Each processing unit operates on the data independently via separate instruction streams while a single data stream is fed into multiple processing units (figure 2.4c). This class knows very few applications. An example of an application is the use of multiple cryptography algorithms attempting to crack a single coded message.

- MIMD

  This time every processing unit is able to execute a different instruction stream on a different data stream (figure 2.4d). This means that any instruction can be applied on any data package for every compute unit. Most supercomputers, networked parallel computer clusters and "grids" can be classified as a MIMD compute system. Also, many MIMD architectures include SIMD execution sub-components.



(a) SISD **(b)** SIMD

(c) MISD **(d)** MIMD

Figure 2.4 *Architecture classes from Flynn's taxonomy*

2.2.3 OpenCL

Companies worldwide constantly strive to improve computational performance. They start using GPUs, FPGAs and other compute accelerators that behave as a coprocessor to process

parallel workload. In order for these heterogeneous architectures to function properly, we need software that supports heterogeneous computing on hardware platforms from different vendors. To make this possible, developers use toolkits such as Threading Building Blocks (TBB), OpenMP, Compute Unified Device Architecture (CUDA), and others [29]. However, some of the existing toolkits were limited to either only being able to use a single microprocessor family or they did not support heterogeneous computing. OpenCL on the other hand provides a set of easy-to-use abstractions and a wide variety of APIs. OpenCL was developed by the Khronos group as a parallel computing API for Apple's OS X release of Snow Leopard back in 2009. This Khronos group is a mixture of people from different hardware vendors like ATI technologies, Intel, Nvidia to name a few [2].
One of the main reasons companies start using OpenCL is that in the past they would use GPUs, when they should be using FPGAs while others had the problem the other way around. The problem they had is that converting CUDA, which is the parallel computing platform from Nvidia (GPUs), to VHDL is difficult and annoying to do. More reasons on why OpenCL should be used to program FPGAs are listed below [22].

- Simplicity and ease of development
  Because most software developers are more familiar with C than low-level HDL language, OpenCL is easy to understand for the vast majority of developers worldwide.
- Code profiling
  OpenCL allows you to determine where exactly the performance-sensitive pieces in your code are. This way it is easy to assign these pieces of code to be executed by hardware accelerators as kernels.
- Performance & Efficiency
  Every developer wants to have his software build in the most efficient way to benefit from maximum performance. Due to the FPGA's parallelistic architecture, you only need to generate the logic the device needs to run the software to deliver high performance.
- Code reuse
  Since there are multiple devices that are supported by OpenCL, you can reuse your already written code on almost any of the other devices without having to change a thing.

2.3 Smartphone

Nowadays, almost everyone has a smartphone. These portable computers allow us to communicate with anyone across the world from almost any place. They can be used as entertainment devices to play music or video games and over the last couple of years even services like stock markets or banking systems have been integrating with smartphones. Furthermore, you can

make pictures with them, view all sorts of media, and the list goes on. Since the increasing popularity there have been many companies developing smartphones. The most common brands are Samsung and Apple accounting for over 37% of the market shares in 2017 of all smartphone brands [24].

2.3.1 Android OS

Like most communication devices, smartphones need an operating system (OS). While Apple's IPhones use their own OS called iOS, Samsung and many other smartphone manufacturers use the Android OS that was developed by Google. With a whopping 81,7% worth of market shares at the end of 2016, Android is definitely one of the market leaders when it comes to smartphone operating systems [15].
One of the main features of smartphones are applications or apps in short. Google's Play Store has over a million apps available for almost any Android device. Unlike Apple's App Store, basically anyone can upload their own apps on Google's Play Store. To upload apps as a developer for everyone to download and use you pay a one-time fee of 25$ to Google. Making apps on the other hand is free. Most apps are written in Java and there are multiple integrated development environments (IDE) available for the Android platform. The official IDE for Android is Android Studio which is described in the next section. Other IDEs available are AIDE (HTML, C, C++), Xamarin (C#) and many others.

2.3.2 Android Studio

Android Studio is the official IDE where you can develop apps for phones with an Android OS. Its main programming language is Java but since Android Studio version 2.2 it is possible to write and use C and C++ code by compiling it into a native library. With the Java Native Interface (JNI) you can call the C/C++ functions in your native library. Furthermore, Android Studio splits up front and back end of the application giving the developer a nice clean overview of the whole project. The front end can be edited through coding or with a visual interface where you can pick and place your required objects in a layout. The front end design is an xml-file that is attached to an activity. This activity is part of the back end where you write your code in order to interact with the front end of the application.

2.4 Communication

Communication in computer science is an act of exchanging information between two or multiple devices. It requires at least one sender, a receiver, a medium and a set of rules called a protocol. The next sections will briefly discuss some properties of the two mediums that were used for this thesis: Bluetooth and WebSocket.

2.4.1 Bluetooth

Bluetooth is a form of wireless communication that was developed in 1994 by Ericsson Mobile in Sweden. It is a radio frequency (RF) technology using the 2.4GHz industrial, scientific and medical (ISM) band, the same band where you can find ZigBee and WiFi aswell. It can be used to transmit data or voice communication over short distances. Bluetooth radios can be found in nearly every new smartphone and laptop device. It is easy to use, to setup and it has a lot of applications, for example hands-free devices, home heating systems, entertaining devices and so on. Bluetooth is designed to be low cost, for about 5-10$ per unit. The down side of this is the short connection range and the limited transmission speed of around 780 kb/s [26].

Bluetooth benefits

The introduction of Bluetooth allowed for many new applications in several areas. Even today it is still widely used, mostly for multimedia devices, keyboards, mices, printers. The following list explains some benefits for three general areas:

Data and voice acces points. Bluetooth allows a wireless connection between devices through which they can communicate. With Bluetooth, the devices are able to transmit voice and data packages in real-time.

Cable replacement. Some wired connections between devices require special cables or adapters. Bluetooth eliminates this hassle since any device can connect to another with the right communication protocol. The range of this connection is approximately 10m and doesn't require the devices to be in line of sight. With an optional amplifier the range can be extended to 100m.

Ad hoc networking. Devices with a Bluetooth radio can establish instant connections with each other as soon they come into range.

Master, slave and piconet

For a Bluetooth connection to exist, there has to be at least one master and one slave device. They use what is called the master/slave model. A master device can be connected to up to seven slave devices while a slave can only connect to one master device. A network of one master and one to seven slaves is called a piconet. The master device will coordinate all the

communication throughout the piconet. All slave devices are allowed to exchange data with the master device when granted premission, but cannot communicate with other slaves in the piconet. The connection between each device is encoded and protected to prevent other devices from eavesdropping and to prevent interference between other devices. Furthermore, in order for these devices to connect with each other, they require the same communication protocol. A device in one piconet can also exist as part of another piconet and can function as either a slave or a master in each piconet. This form of overlapping is called a scatternet [28]. An example of two piconets forming a scatternet is shown in figure 2.5.

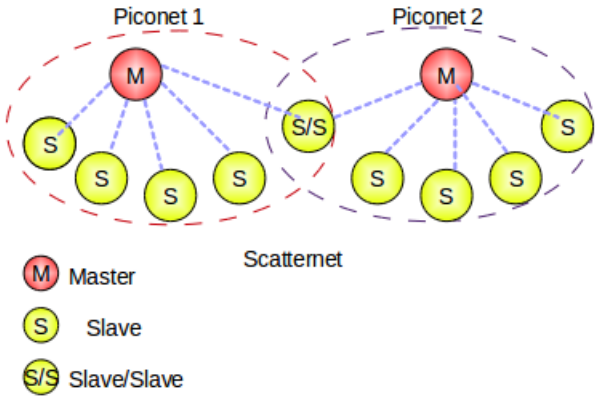


Figure 2.5 *Piconets and Scatternets*

The piconet/scatternet allows the devices to share the same physical area, allowing the network to make efficient use of the bandwidth. A Bluetooth system can use up to 79 different frequencies using a frequency hopping (from 2.402 to 2.480 GHz) [25] scheme with a carrier spacing of 1MHz. This allows a bandwidth of 80MHz. Without frequency hopping scheme, every single channel would have a bandwidth of 1MHz at their disposal. With frequency hopping, the sequence will define a logical channel. This allows to have an available bandwidth of 1MHz at any given time, with a maximum of eight devices sharing the bandwidth. This 80 MHz bandwidth can be shared by several different logical channels. Though, this can cause signal collisions when devices in different piconets, on different logical channels have the same hop frequency at a given time. Signal collisions degrade the performance, so we can state that the more piconets we have, the more collisions occur, the lower our total performance will be [28].

Protocol architecture

The Bluetooth protocol architecture consists of four basic layers: core protocols, cable replacement, telephony control protocols and adopted protocols. Figure 2.6 shows the architecture of the Bluetooth protocol stack.

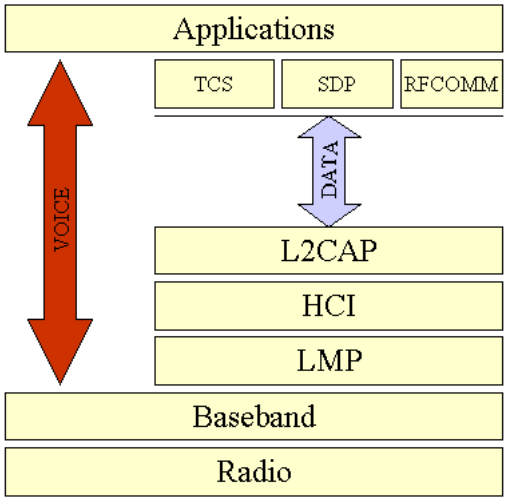


Figure 2.6 *Bluetooth protocol stack*

Core protocols. The core protocol is a five-layer stack. Every layer in the stack has its own responsibilities that are mentioned below.

- The *radio* layer is the wireless connection that specifies certain details about the air interface, including frequency, the use of frequency hopping, modulation scheme and transmit power.

- The *baseband* layer is responsible for the packet transmission to the radio layer. As mentioned before this data can contain data or voice packages. For the data packages, asynchronous connectionless (ACL) links are used while voice packages are transmitted with synchronous connection-oriented (SCO) links. The baseband layer maintains both ACL and SCO links. It is important for data packages to be transmitted correctly to maintain data integrity, while it is not a problem in case some voice packages get lost. That is why SCO packages are never retransmitted. If you would retransmit voice packages, every next package would suffer from a time delay restraining us from having real-time communication.

- The *Link manager protocol (LMP)* uses the links setup by the baseband and manages the connection between Bluetooth devices. Furthermore, it is responsible for monitoring service quality, security aspects such as device authentication, encryption plus the control and negotiation of baseband packet sizes.

- The *Host controller interface (HCI)* is the layer between the hardware and the software. The L2CAP layer and the layers above it are implemented in the software while all other layers under the HCI (LMP, baseband, radio) are part of the hardware. The HCI driver acts as a physical bus that connects the hardware with software. It is possible to access

the L2CAP layer directly by the application making it easier for application programmers. This makes the HCI, in some cases, an unnecessary component.

- The *Logical link control and adaptation protocol (L2CAP)* receives application data and transforms this to the Bluetooth format. Furthermore, Quality of Service (QoS) parameters are exchanged at this layer [25].

- According to [25], the *Service discovery protocol (SDP)* is not a part of the Core protocols. Though, it contains all the information, services and charasteristics in order to establish a connection between two or more Bluetooth devices. The LMP uses the SDP's first to find out what services are available from the acces point. Then information from the SDP is obtained by the LMP to create a L2CAP channel.

Cable replacement. The RFCOMM seen in figure 2.6 is the cable replacement protocol. It is a virtual serial port that is designed to replace cable technology. Serial ports are common types of communication interfaces used with computing and communication devices [28]. So with RFCOMM we eliminate the need for serial ports for communication between two devices, assuming both are equiped with a Bluetooth radio. EIA-232, once known as RS-232, is a widely used serial port interface standard. The RFCOMM will provide binary data transport and has to emulate EIA-232 control signal to the baseband layer.

Telephony control protocols. Telephony control specifications (binary) or TCS BIN, is a bit-oriented protocol that is necessary to define the call control services in order to establish speech or data calls between the Bluetooth devices.

Adopted protocols. Adopted protocols are protocols developed by other organizations. They are "adopted" into the overall Bluetooth architecture. They are usually standard protocols well known in applications other than Bluetooth. Bluetooth's strategy is to only invent necessary protocols and use existing standard protocols whenever possible. The following standards are the adopted protocols:

- The *PPP*, or point-to-point protocol is as a internet standard protocol for transporting IP datagrams over point-to-point links.

- *TCP/UDP/IP*, are the foundation protocols of the TCP/IP protocol suite.

- *The object exchange protocol*, or OBEX, is a session level protocol made by the Infrared Data Association (IrDA). It is used for exchaning objects. OBEX comes with quite similar

funcionalities as HTTP, but in a simpler way. There is also a model included in OBEX that is used for the representation of objects and operations.

- Bluetooth also adopts the wireless application environment *WAE* and the wireless application protocol *WAP* into its architecture.

2.4.2 WebSocket

WebSocket are used for fast, real-time communication between a server and a client. The HTTP model, which is also a communication protocol between a server and a client, allowed the client only to request data from the server, while the server was only able to fulfill these requests. WebSocket on the other hand, allow bidirectional communication between the server and the client. This means both client and server can request data and also respond to these requests. The main point of webSocket is to have true concurrency and to focus on the optimization of performance when it comes to communication and exchanging data. The WebSocket protocol that will be discussed is also known as the RFC6455 model [21].

WebSocket communication

Handshake. To communicate over WebSocket, the server and the client first have to connect with each other. The establishment of a WebSocket connection is done by a WebSocket handshake. Handshaking is the exchange of information between two devices and the agreement about which protocol will be used to exchange data after the connection is established. A well known example of this is the TCP three-way-handshake. The client sends a synchronization message (SYN) to the server as a request to synchronize with the server. If the server receives this message and allows the client to synchronize with it, it will send a similar SYN message and an acknowledgement message (ACK) to let the client know that it has received the request. When the client receives the SYN-ACK message, it will send an ACK message back to the server to acknowledge that it has received the server's message. The TCP connection is established whenever the server receives the ACK message from the client.
The WebSocket handshake is quite similar. The client sends a handshake request to the server, and the server will respond with a similar handshake request as seen in figure 2.7. The desktop, smartphone and tablet in figure 2.7 represent the clients that are connected to the server. The handshake request from the client-side is shown in program 2.1, while the response from the server is shown in program 2.2.
Program 2.1, the handshake request from the client, is a pretty standard HTTP request. It is built with multiple headers, some of which are mandatory for the request to be valid. If one of the headers is not understood, the server will reply with "400 Bad Request" and it will

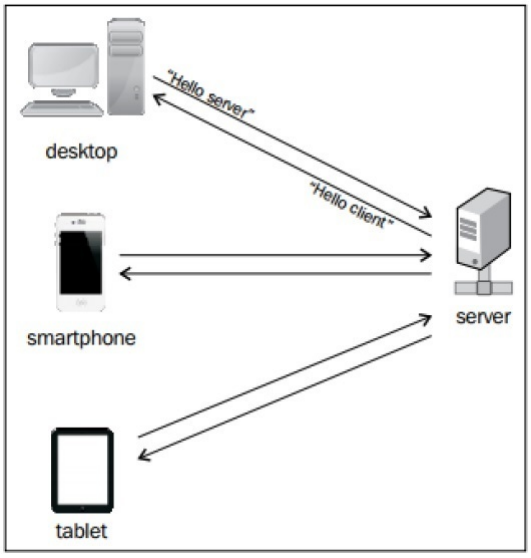


Figure 2.7 *WebSocket handshaking*

```
1| GET /chat HTTP/1.1
2| Host: example.com:8000
3| Upgrade: websocket
4| Connection: Upgrade
5| Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6| Sec-WebSocket-Version: 13
```

Program 2.1 *Client's request for WebSocket handshake*

close the socket afterwards. In some cases, it will also give a reason why the handshake failed, although browsers do not display these messages. If there is a problem with version numbers, the server adds a "Sec-WebSocket-Version" header in the HTTP response that contains the version it understands [23]. When the server receives a request handshake from a client with all

```
1| HTTP/1.1 101 Switching Protocols
2| Upgrade: websocket
3| Connection: Upgrade
4| Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
5|
```

Program 2.2 *Server's response for WebSocket handshake*

the necessary headers, it will reply with a HTTP response as shown in program 2.2. The "Sec-WebSocket-Accept" header is derived from the "Sec-WebSocket-Key" header from the client's handshake request. To get it, we combine the "Sec-WebSocket-Key" header and "258EAFA5-E914-47DA-95CA-C5AB0DC85B11" together. The second string is "a magic string". A magic string is predefined by the developer. It is made in a way where you would not expect it to be received from an input. When the "Sec-WebSocket-Key" header and the magic string are

combined, the SHA-1 hash is taken from the result, and the base64 encoding of the hash is returned [23]. The SHA-1 is a cryptographic hash function, while base64 is a binary-to-text encoding scheme.

WebSocket URIs. WebSocket defines two URI schemes. You can either use ws or the wss scheme. The ws (WebSocket) is a regular connection similar to http. While wss (web socket secure) is a secured connection similar to https. The schemes are built as follows:

ws-URI = "ws:" "//" host [":" port] path ["?" query]
wss-URI = "wss:" "//" host [":" port] path ["?" query]

The most important components of the ws or wss are the host and its port. The host is determined by the server's IP address, while the port defines which port the server uses for the communication. If there is no specified port, the standard port used for ws is 80, and the standard port used for wss is 443.

Data Frames. The main advantage of WebSockets is bidirectional communication. So at any point in time, either the client or the server can send a message. Every data frame that is sent from the client to the server, or vice versa, follows the same format as seen in figure 2.8.

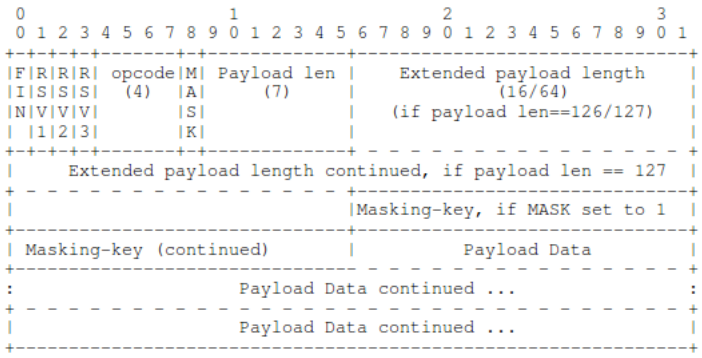


Figure 2.8 *Standard WebSocket dataframe format*

- FIN: 1 bit
  Depending on the value of this bit, it either tells the receiving end whether or not this is the final fragment of the message. If the bit equals "0", it is not the last fragment and the receiver will continue listening for more fragments. If the bit equals "1", it means it is the last fragment of the message and the server will consider the message being delivered.

- RSV1, RSV2, RSV3: 1 bit each
  All of these bits are reserved for WebSocket extensions. They should be "0" unless the client and server negotiated on whether or not a specific extension requires the use of any of the three bits. If any of these three bits is not zero while the client did not negotiate on any of these bits being non-zero, the receiving end will "fail" the WebSocket connection.

- Opcode: 4 bits
  These 4 bits will define how the receiving end should interprete the data. If the receiving end does not understand the opcode it will, as in the previous case, "fail" the WebSocket connection. The information about the different opcodes is found at [18].

  x0: continuation frame; this frame contains data that should be appended to the previous frame

  x2: binary frame; this frame (and any following) contains binary data

  x3 - x7: non-control reserved frames; these are reserved for possible WebSocket extensions

  x8: close frame; this frame should end the connection

  x9: ping frame

  xA: pong frame

  xB - xF: control reserved frames

- Mask bit: 1 bit
  This bit tells wether or not the frame uses a mask. If this bit is set to "1", a masking key is included in the message. This masking key is used to unmask the data in the payload. Every frame that is sent from the client to the server must have this bit set to "1".

- Payload length: 7 bits, 7+16 bits, 7+64bits
  The seven bits determine the length of the payload. If these seven bits equal 126, or "1111110", the actual length is determined by bits 16 to 31 (so 16 extra bits). These are the following 2 bytes. If the seven bits equal 127, or "1111111", the actual length is determined by bits 16 to 79 (so 64 extra bits). These are the following 8 bytes.

- Masking key: 4 bytes
  As mentioned previously, this field only exists if the mask bit is set to one. All the messages who have this field set to one, are masked by a 32-bit value. If the mask bit is set to zero, there will be no masking key in the first place.

- Payload data: x+y bytes
  The payload data is the combination of the extension data and the application data. These two are listed below.

- Extension data: x bytes
  The extension data is non-existent unless it was negotiated on the opening handshake between the server and the client. As mentioned earlier, the RSV1-3 bits are responsible for these extensions. Any extension that has been negotiated by the client and server must have a specified length of the "Extension data". It can also tell the receiving end on how to calculate this length. As said previously, the extension is part of the total payload data.

- Application data: y bytes
  The application data contains the actual data that has be to transmitted. It takes up the remaining space in the frame after any extension data. The application data is, like the extension data, part of the payload data.

2.5 SoC

SoC is the abbreviation of *System-on-Chip*. In our case that is a processor, FPGA and peripherals together on a single substrate inside of one chip. Industry calls this process *VLSI, Very Large Scale Integration*. The main advantages of using VLSI in a SoC are the low power consumption, its tiny size and the fast well shielded connections between the on-chip components [12]. On the other hand, using VLSI makes the chip design, production and service very complicated. Due to the high components density, a lot of heat is concentrated at the same location. The only way to cool down the chip is by increasing package size. To minimise the package size SoCs only drive low power GPIOs. If there is a need for high power controls, a series of buffers must extend the GPIO, General Purpose Input Output, signals.

The SoC used in this thesis, Cyclone V 5CSEMA5FF31C6N [6] in figure 2.9, is made by Altera and integrated in the DE1SoC development board by TerASIC [30]. The DE1SoC consist of a HPS- and a FPGA-part with both their own peripherals. *HPS* is the abbreviation of *Hard Processing System* and *FPGA* for *Field Programmable Gate Array*. Figure 2.9 shows most of those components and the system they are connected to. Orange peripherals are connected with the HPS, while green components are peripherals of the FPGA and everything in blue is commonly used. Inside the Cyclone V IC, three bridges are provided to distribute signals between FPGA and HPS. Because all the GPIO's are connected to the FPGA, all GPIO data are always transferred through the bridges, when they are needed by the HPS. The FPGA-part can be configured by a HDL, Hardware Description Language [13]. In contrary to regular programming languages, HDL describes digital hardware. Instead of programming a single thread and running each command at a time a FPGA implements parallel applications, resulting in a high throughput. HDL describes a full data path of registers, adders, multiplexers, etc. between multiple PLLs (Phase-Locked Loop), FSM (Finite State Machine) controllers and other modules.

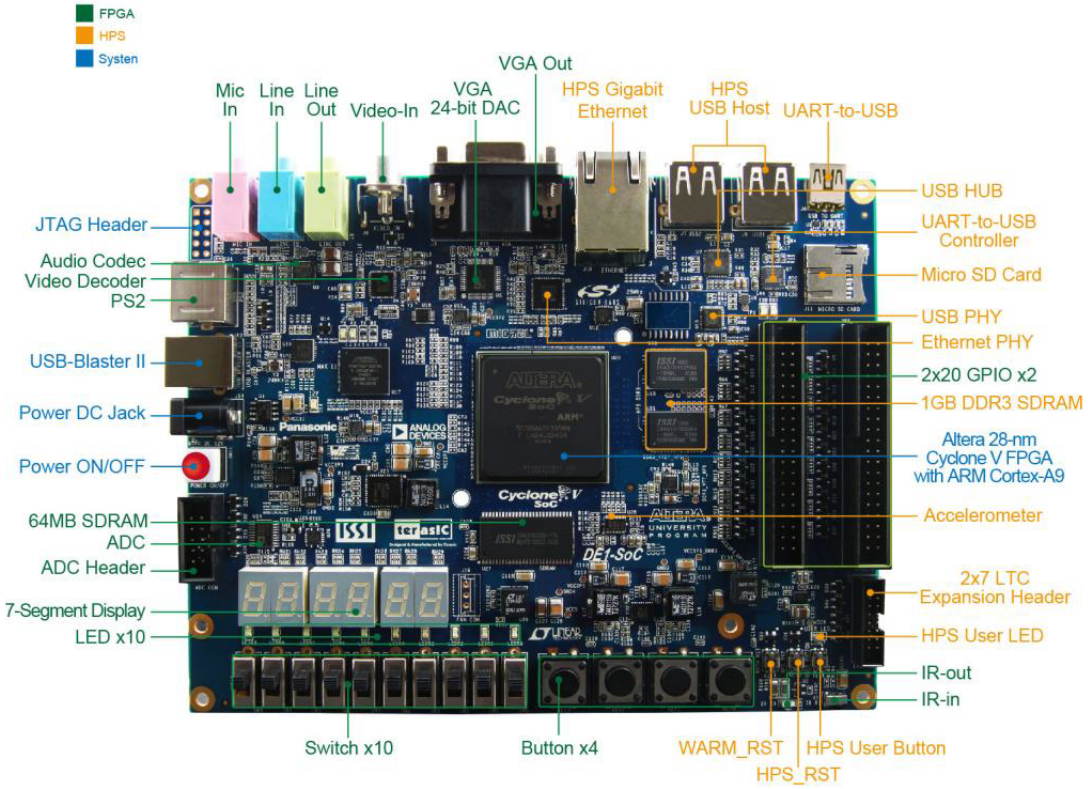


Figure 2.9 *Layout of the DE1SoC development board of TerASIC*

In the following sections we will discuss why a HPS, FPGA and bridges between the two are necessary.

2.5.1 Hard Processing System

Embedding one or more processors inside electronic systems gives the advantages of faster development time and in-the-field reprogrammability. The SoC we use includes a HPS with two ARM Cortex A9 cores, as can be seen in figure 2.10. Each processor uses its own L1 cache memory capable of storing 64 KB, of which 32KB is reserved for instructions and 32KB for data. L1 cache is relatively small, but provides a high speed read and write memory to the processor [22]. Dual-core applications need a shared cache memory when they exchange data between two processors. Shared cache memory is called L2 cache and is larger, but slower than L1 cache. DDR SDRAM is provided in the HPS, so the OS can boot. Developers can use the "Shared multiport DDR SDRAM controller" to read and write SDRAM data from the FPGA. There are three more connections, called bridges, in the SoC: "HPS to FPGA", "FPGA to HPS" and "FPGA Configuration". The third bridge gives developers the ability to upload a raw binary file to configure the FPGA from HPS. During runtime the "HPS to FPGA" and "FPGA to HPS" bridges can be called to exchange data. Other HPS peripherals are shown in

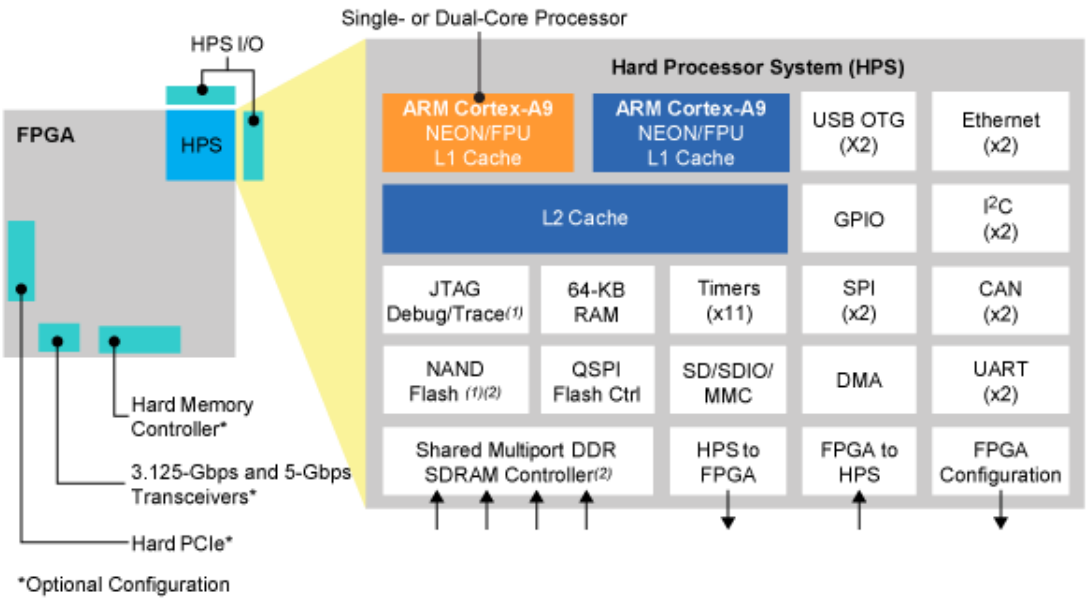


Figure 2.10 *Cyclone V Hard Processing System layout*

figure 2.10. We will not discuss them because the OS deals with them in the background.

Our SoC consists of a 32-bit, 800MHz dual-core ARM Cortex A9 MPCore architecture [5]. The data path size and width of the registers are both 32 bits. The term dual-core refers to two independent processing units with the ARM Cortex A9 MPCore architecture in the same package, running at a frequency of 800MHz. This architecture is highly recommended for low-power, cost-effective applications on a 32-bit platform.

In the next paragraphs we will explain why we need an Operating System and programming languages such as C++ and OpenCL and how we use them. The HPS has a processing unit and lots of peripherals, software is needed to use all of them in a structured way. This software is built on an interface that is called an Operating System. Operating systems are built with programming languages to execute programming languages. Embedded systems are mostly built of C or C++ source code. Besides C and C++ executables running on the HPS, we used OpenCL to program the FPGA with a kernel.

Operating system

As defined in "Research paper on operating system" [9]: "An OS is a collection of software that manages computer hardware resources and provides common services for computer programs. The operating system is an essential component of the system software in a computer. Application programs usually require an operating system to function".

The used OS on the DE1SoC board is provided by Altera and is a basic Linux kernel without

user interface. Because the kernel does only support command line, the impact of running an OS on the CPU is minimised. This kernel already has all the packages we need to run OpenCL, C++ and use the AMBA AXI bridges. More info about OpenCL can be found in paragraph 2.2.3. Altera has chosen the Angstrom Linux distribution for their SoC [7]. Angstrom is a very basic Linux distribution used in a variety of embedded devices. The popularity of embedded systems is due to the fact that it uses a binary package feed, allowing to simply install software distributed as OPKG packages. Those OPKG packages are precompiled on a host system.

Programming language

A programming language is defined as the communicator of instructions to the computer. Depending on the language used a certain knowledge of the architecture is needed. When programming in machine language for example, processor architecture must be well known. Each architecture has its own instruction set, with one instruction for each operation. All instructions correspond one on one to a physical command in the machines processor architecture. A command can be anything like loading data into registers or calculating a result from two data registers. Machine language is the most basic programming language. Programmers should have adequate experience with the specific architecture to program simple applications in an efficient way. Nowadays, with the wide variety of systems and architectures, programs are written in a higher level programming language. Those languages, such as C, C++, Java or C#, are easier to develop and distribute over different platforms. Higher level programming languages are still based on the same machine language for their specific architecture, but a compiler handles the conversion to machine language instead of the developer. The compiler generates an executable in different stages by using a preprocessor, compiler, assembler and linker.

The compilation process of a C++ program can be found in figure 2.11 [20]. A preprocessor copies the source code file and includes header files into a temporary expanded source code file. During the copying process all "#defines" are initialised with their values. Next, the temporary expanded source code file is translated to assembly language. This code is based on the desired architecture, resulting in a specific assembly code for each architecture. Assembly code passing through the assembler generates an object file. While the object file is generated, physical memory locations will be assigned to all variables and instructions are given by the assembly code. During the last step the previously generated object file is linked with other object files corresponding to the earlier included header files. By linking object files, one executable file is generated. Executables are started when the "./program" command is invoked. Executables can only be started when the user has given execution rights to the specific executable file. Users give the execution rights by the "chmod +x" command, used as "chmod +x executableFileName".

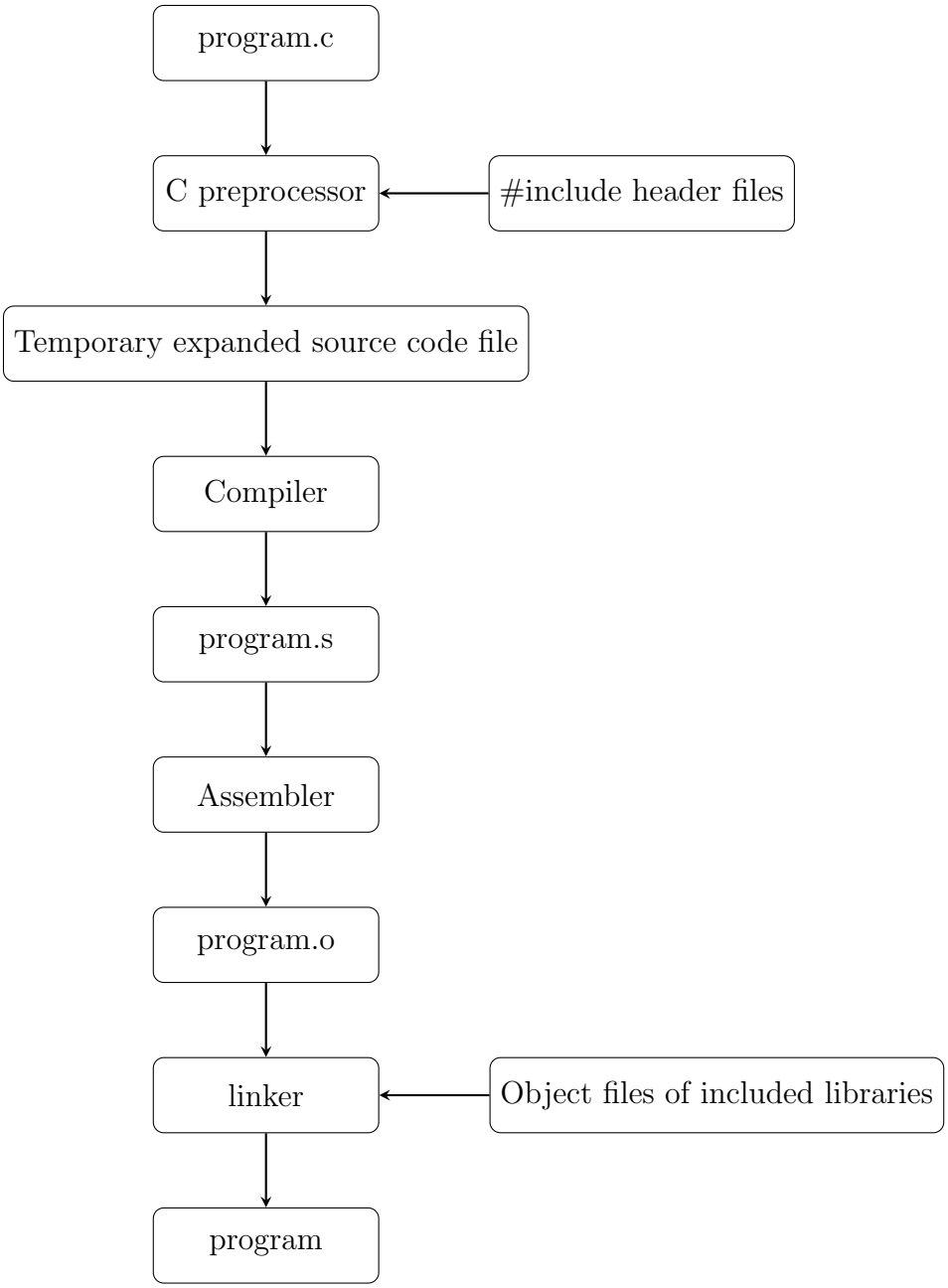


Figure 2.11 *Compilation process from High level programming language to an executable file [17]*

C++ programming: Bell Labs developed C in the early 1970's with the UNIX OS. For many years the book "The C Programming Language" [17], published in 1978, was the standard reference for the C language. In 1988 a second edition of "The C Programming Language" was published, after the use of C language spread beyond UNIX system. This second edition included a changed, platform-independent, standard. C became a general purpose programming language close to the machine hardware, using pointers to locate variables on a specific place in memory.

The reason the above book refers to the C programming language instead of the C++ pro-

gramming language, is the fact that pointers were better explained in this book [17] than in a C++ source. When a pointer points to a specific place in memory and a value is written into the pointer, the value is stored in that specific memory space. The OpenCL Kernels use pointers to arrays to transfer big data. Arrays in C are typically given a certain space by the "malloc()" function, with the address of the first element stored into the pointer.

OpenCL host: A general description of OpenCL can be found in subsection 2.2.3, in this paragraph we will discuss the OpenCL host program. The OpenCL host needs to invoke "clEnqueueTask()" for OpenCL kernel execution, but before the host can run "clEnqueueTask()" a kernel environment must be configured, so the organiser of the program performs all the following 13 tasks [32] to configure, initiate, catch results and finish the OpenCl kernel.

1. Get a list of available platforms: A platform is defined as the brand of devices such as Intel or AMD. OpenCL detects which platforms are available and stores them in variable "platform_id", see program 2.3. The first parameter of clGetPlatformIDs defines how many platforms are wanted, obviously this parameter needs to be greater than zero.

```
cl_platform_id platform_id = NULL;
cl_uint ret_num_platforms;
ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
```

Program 2.3 *Get a list of available platforms*

2. Select device: Devices are the products of a brand, like CPU's, GPU's or FPGA's. They belong to a specific platform. If multiple devices of different platforms are used, this step needs to be combined with step one. First, clGetDeviceIDs receives an id representing an available platform and next, a device type needs to specified. Device types can only be one of the following:

   - CL_DEVICE_TYPE_CPU
   - CL_DEVICE_TYPE_GPU
   - CL_DEVICE_TYPE_ACCELERATOR
   - CL_DEVICE_TYPE_DEFAULT
   - CL_DEVICE_TYPE_ALL

   The first three device types correspond to using the CPU, GPU and FPGA. The third parameter of the function, program 2.4, specifies the number of devices the host would like to use. "ret_num_devices" returns the amount of devices available of the specified device type.

```
cl_device_id device_id = NULL;
cl_uint ret_num_devices;
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT,
1, &device_id, &ret_num_devices);
```

Program 2.4 *Select device*

3. Create Context: Objects use a context in OpenCL. What the context does and why will be explained in memory objects. To create the context in program 2.5, clCreateContext needs to know how many and which devices need a context, respectively in parameters two and three. All the other parameters represent advanced properties that are not discussed in this thesis.

```
cl_context context = NULL;
context = clCreateContext(NULL, 1, &device_id,
NULL, NULL, &ret);
```

Program 2.5 *Create Context*

4. Create command queue: When a device is active, a medium to communicate with the device must be created. OpenCL calls this medium a *command queue*, see program 2.6. The command Queue needs arguments, in this order, to specify: the used context, which device will be execute in this command queue and some not discussed advanced parameters.

```
cl_command_queue command_queue = NULL;
command_queue = clCreateCommandQueue(context,
device_id, 0, &ret);
```

Program 2.6 *Create command queue*

5. Create memory objects: Kernels can not access memory outside their device. A solution is found by copying the data from host to device memory. OpenCL uses a buffer as medium to copy the data back and forth. In program 2.7 "clCreateBuffer" needs to know which context to use and the access rights the kernel has for the allocated device memory. The occupied memory size is passed as the third argument.

```
cl_mem memobj = NULL;
memobj = clCreateBuffer(context, CL_MEM_READ_WRITE,
MEM_SIZE x sizeof(char), NULL, &ret);
```

Program 2.7 *Create memory objects*

6. Read kernel file: In this step, things need to be split up. When a CPU or GPU is used, the OpenCL kernel can be read from source file. In contrast to the FPGA, Altera desires

to offline compile OpenCL kernels to a raw binary file. This and the next step mention the constructions for both Apple and Altera platforms.

- Apple: The host program 2.8 reads the OpenCL kernel file and puts the file in a huge array of characters with an allocated size of MAX_SOURCE_SIZE. An array of characters in C is comparable with a string datatype.

```
FILE *fp;
char fileName[] = "./hello.cl";
char *source_str;
size_t source_size;

/* Load kernel code */
fp = fopen(fileName, "r");
if (!fp) {
    fprintf(stderr, "Failed to load kernel.\n");
    exit(1);
}
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);
```

Program 2.8 *Read kernel file*

- Altera: Because the SoC is primitive, Altera prefers to compile the object file off-line with a specific license. To load the binary file, parameter KERNEL_NAME should contain the path to the binary file in the SoC directory. The last parameter in program 2.9 defines the device where the kernel is installed.

```
std::string binary_file = getBoardBinaryFile(KERNEL_NAME,
device);
```

Program 2.9 *Read kernel file*

7. Create program object: A kernel program can contain multiple kernel functions. If there are multiple kernel functions, each kernel should be converted to an object by the codes below.

- Apple: Once the source code is read from the kernel source file, it must be processed into an OpenCL kernel program. Function "clCreateProgramWithSource", program 2.10, creates a program object file in variable "program".

```
cl_program program = NULL;
program = clCreateProgramWithSource(context,
```

```
1, (const char **)&source_str,
(const size_t *)&source_size, &ret);
```

Program 2.10 *Create program object*

- Altera: After the offline compilation, a binary file was created. In the previous step the binary file was loaded into the host program. Now "createProgramFromBinary", program 2.11, converts the binary to an object file in variable "program".

```
program = createProgramFromBinary(context,
binary_file.c_str(), &device, 1);
```

Program 2.11 *Create program from binary file*

8. Compile kernel: At this point the kernel program needs to be built for a specific device with "clBuildProgram", program 2.12. In the step above, "clCreateProgramWithBinary" could be used instead of the "clCreateProgramWithSource". That way "clBuildProgram" would not be necessary for the Altera program kernel.

```
ret = clBuildProgram(program, 1, &device_id, NULL,NULL, NULL);
```

Program 2.12 *Compile kernel*

9. Create kernel object: For each kernel function an object should be created. The second argument of function "clCreateKernel", program 2.13, sets the name of the kernel object. In this case there is only one kernel, but multiple kernel objects could be generated if necessary.

```
cl_kernel kernel = NULL;
kernel = clCreateKernel(program, "hello", &ret);
```

Program 2.13 *Create kernel object*

10. Set kernel arguments: Setting kernel arguments is the main task of the OpenCL host program 2.14. The kernel expects a pointer to the memory objects, this pointer should be declared in the host side. When the pointer is not declared on the host side, the host can't manage the memory. The first argument is the kernel object itself, secondly the number of the argument to be set is specified. Next, the argument size and pointer to the argument are passed.

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem),
(void *) &memobj);
```

Program 2.14 *Set kernel arguments*

11. Execute kernel: The kernel will be executed when this function is started. Program 2.15, function "clEnqueueTask" takes the kernel and launches it into the queue. In this example we have only one queue, so the fifth argument can be "NULL". Otherwise the fifth argument has to be set as an event object. The event object will wait for the kernel to finish execution. Once executed it will give a notification to the host by using this event object.

```
ret = clEnqueueTask(command_queue, kernel, 0,
NULL, NULL);
```

Program 2.15 *Execute kernel*

12. Read memory object: Once the kernel is finished, return data must be read from the kernel. The return data will be available on the device side, where "clEnqueueReadBuffer", program 2.16, can copy the data back to the host side. This function uses a lot of arguments, but only some of them are important. The second argument points to the device side memory, while sixth argument points to the host side memory while the fifth argument determines the memory size.

```
char string[MEM_SIZE];
ret = clEnqueueReadBuffer(command_queue, memobj,
CL_TRUE, 0, MEM_SIZE x sizeof(char), string, 0,NULL, NULL);
```

Program 2.16 *Read memory object*

13. Free objects: Like in a regular C program, all objects needs to be freed. During this step is all of the allocated memory is deallocated, so it can be used by other programs. How to free all objects is shown in program 2.17.

```
ret = clReleaseKernel(kernel);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(memobj);
ret = clReleaseCommandQueue(command_queue);
ret = clReleaseContext(context);
```

Program 2.17 *Free objects*

This was a short introduction. When OpenCL is used in an application, the kernel is executed repetitively in a contiguous cycle as can be seen in figure 2.12. The cycle always starts with copying data from host to device, followed by executing the kernel and returning the calculated data back to the host.

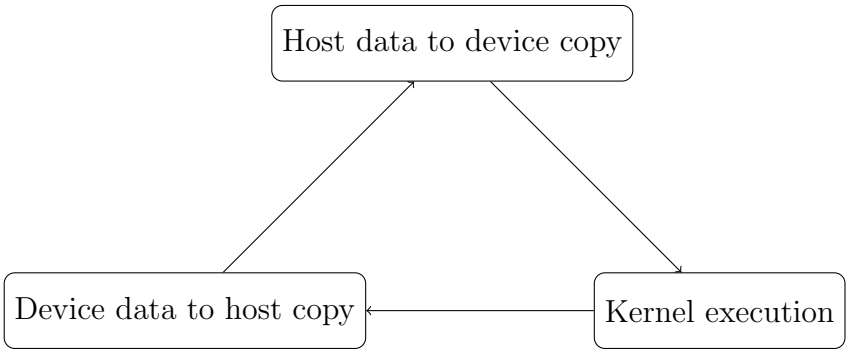


Figure 2.12 *OpenCL real life applications kernel cycle*

2.5.2 FPGA

Using a FPGA enables designers to program logic in the field. Altera, the manufacturer of our SoC, published a book called *FPGA for dummies* [22]. Everything in this subsection is referenced to this book.

FPGA design flow

Developing a FPGA implementation includes 5 main stages, as shown in figure 2.13. It all starts with a system design. Engineers decide which functions have to be implemented. They also keep the integration with the rest of the system in mind. Secondly all the needed inputs and outputs of the FPGA are matched to the other components in the system to inform the pin planner. The name Pin planner is self explanatory. It is planning which pins of the inputs and outputs are connected to which components on the PCB. The next stage is the most time consuming phase. Here, designers program in a HDL like Verilog, VHDL or a schematical editor to describe the logic circuit. Designers can implement IP-blocks, Intellectual Property blocks, to interact with the HDL. Some IP-blocks come with the design tool, others need to be bought from third party companies. Once the design is complete, two possibilities are left. When the design is considered small, developers start testing on the FPGA. If testing directly on FPGA is impossible or when dealing with a large design, test benches are desired. A test bench simulates the HDL as functional verification of the system. Although a HDL is tested by a test bench, there might still be errors after implementation. A test bench is as good as the test bench designer. Once the design has been defined, tested or not, synthesis is started. Synthesis starts checking the code on typos, not included packages, name mismatches, registers that need to be wires, etc. Once checking is complete, synthesis will optimise the code. The last step, shown in figure 2.13, is the design verification. Design verification could also be done by simulation, here the hardware of the specific implementation is tested.

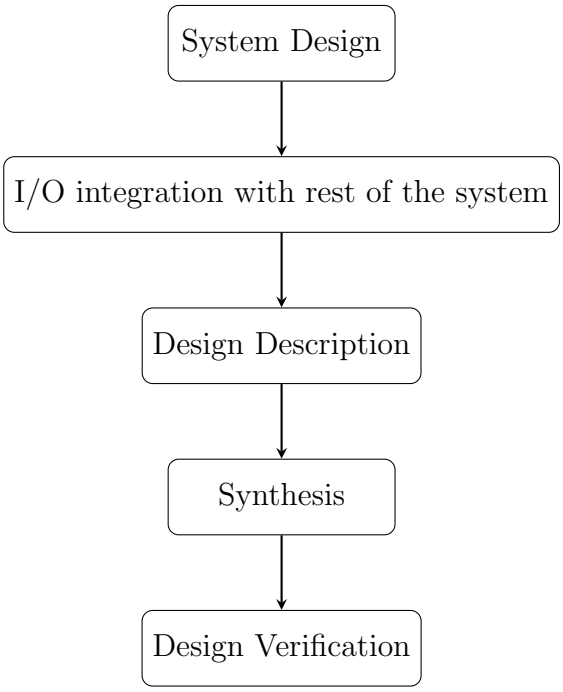


Figure 2.13 *FPGA design flow*

Hardware Description Languages

As mentioned earlier, HDLs describe digital hardware resulting in a physical hardware layout inside the IC. During the design of HDLs it is important to keep in mind that the source code is directly represented as hardware. There are two main HDL languages, Verilog and VHDL. Since we only used Verilog in our thesis, VHDL will not be discussed here. Verilog is difficult to implement, due to the clocks that keeps everything in synchronization. OpenCL is a C-syntaxt based alternative to program GPUs and FPGAs. Besides these, other devices are supported, like CPUs.

Verilog: Both HDL and programming language use variables. Verilog uses two main kinds of variables, i.e. wires, *wire* and registers, *reg*. A "wire" is a wired connection in the IC and is used in assignments or modules. Assignments and modules have strong connections between each other, while registers are considered weak connections. Registers or *regs*, are only used when a signal changes inside an "always" statement. However, unlike in programming languages, a signal called clock is used. The clock signal is the most important signal in all HDLs. Giving an example: when horses A, B and C race against each other on different racetracks, they will probably never finish at the same time. Imagine the horses being electric pulses A, B and C. There will be a race between the three pulses, because they all travel as fast as they can. Some pulses travel faster than others, the solution for this is using a clock in order to read all pulses at the same time. On the rising edge of the clock, referring to figure 2.14, all signals

leave register1. Obviously signal A will be the first to reach register2 and there will be a race between pulses A, B and C to get the second and third place of B and C in register2. Which one will arrive first is unpredictable, because of the different delays in and between the not predefined place of the logic elements. This phenomenon is called a race in HDL. Using a clock makes sure there will be enough time for every signal to arrive, as long as all signals arrive before the rising edge.

The system also needs a default state. When the reset signal is engaged, the whole system is set to default. A reset can happen synchronously (Program 2.18) or asynchronously (Program 2.19), meaning respectively resetting only at the edge of the clock or resetting whenever the reset signal is engaged. The only difference between the two is the added "posedge reset" in Program 2.18. When a designer is not consistent and changes between synchronous and asynchronous reset or puts another signal in the always construct, the behaviour will probably be unpredictable.

Inside the "always" function in Verilog syntax *if, case, while, for* and *repeat* statements can be used. All used signals inside an "always" need to be declared as registers "reg" and need to have a default value defined in "if(reset)".

Apart from the "always" statement two possibilities can be considered. The first one is an assignment between wires and the second one is wiring another module into the current module. Assignments are used when a signal needs to be delivered immediately when it is present, two examples can be seen in Program 2.20. Assignment to "result1" is a "logical or" between two wires "a" and "b". While "result2" looks like an if-else statement without clock. Whenever "conditionBit" is true, the "valueBit" will be returned in wire "return2", otherwise "valueFalse" will be returned. Designers need to remember that in an assignment no clock is used. HDL-races can result in unpredictable behaviour, If they are not handled carefully. Another possibility when using assignments is to connect a register and a wire. Assignments provide also the usage of a module inside another module as can be seen in Program 2.21, where object "objectAbc" of module "abc" is wired to wires "A", "B" and "C". The inputs and outputs of module abc are "a", "b" and "c".

Verilog-syntax is very simple, we have already discussed almost all the important syntaxes except for what modules look like. A module is based on regs, wires, assignments, alwayses, other modules, inputs, outputs and inouts. The last three have some special rules. An input always has to be of type net, when used externally they are connected to regs or wires. Outputs have the opposite rules. Inside the module they can be a wire or reg, but when used externally the outputs have to be connected to a wire. Lastly inouts are always wired internally and externally. An elementary example of a module can be found in Program 2.22.

In the following paragraph OpenCL kernels are explained, they are an alternative way to

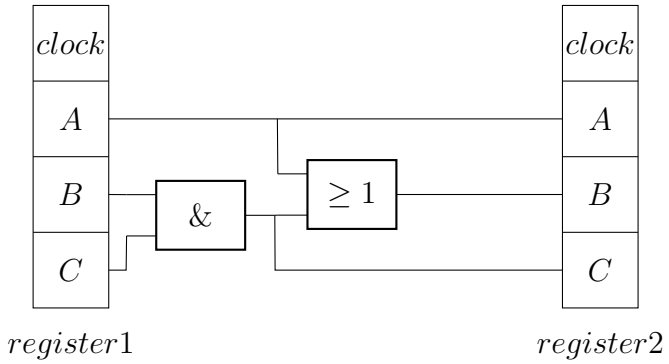


Figure 2.14 *HDL races figure*

```
always @(posedge clk) begin
    if(reset)
        ... <= 1'b0;
    else
        ... <= ...;
 end
```

Program 2.18 *Verilog example, synchronous reset*

configure FPGAs.

OpenCL kernel: An OpenCL kernel is a C/C++ based programming language used to rapidly extend a compute exhausting task to a hardware accelerator, such as a GPU or in our case a FPGA. In order to use a kernel, a host in C or C++ should be coded. The host side, explained in 2.5.1 paragraph OpenCL Host, checks and configures the environment before initiating the kernel. Kernels are initiated on a device of a specific platform. In our case, the device is an FPGA. FPGAs are ideal devices for algorithms that parallelize their problems. We will explain how the kernel works based on figure 2.15 as a representative example. The idea of this example is found on a website referenced by [27]. This example uses the structure of a school to calculate a sum of multiplications. Once the example is explained, it will be linked with the real kernel using figure 2.16. Imagine a very large sum of multiplications, like equation 2.7, calculated by an algorithm like the structure of a school.

$$result = (A \times B + C \times D + E \times F + G \times H) + (I \times J + K \times L + M \times N + O \times P) \quad (2.7)$$

There could be one person doing all the multiplications and adding them to the previous ones, but he would take a long time doing the same thing over and over again. It would be easier if the school extends parts of the calculation to different departments, as can be seen in figure 2.15. Department A takes the first 8 numbers and department B the last 8 numbers. Each department distributes the multiplications to classes of two students. A student goes to the

```
always @(posedge clk or posedge reset) begin
    if(reset)
        ... <= 1'b0;
    else
        ... <= ...;
end
```

Program 2.19 *Verilog example, assynchronous reset*

```
assignment result1 = a or b;
assignment result2 = (conditionBit) ? valueTrue : valueFalse;
```

Program 2.20 *Verilog example assignments*

blackboard in front of the class, calculates the multiplication and returns the result to the blackboard. Once all the students are finished, the teacher of the class returns the results to the department. The department waits for all the other classes to finish and calculates the sum of all results. When each department has returned its sums, the school director can calculate the sum of all values returned by the departments.

Figure 2.16 represents the real kernel situation of the previous example. The outer circle represents the platform of the vendor, like Intel/AMD/Altera. The vendor's platform could be seen as the campus, with different schools doing a specific parallelized algorithm, called a kernel. Devices belong to a vendor platform, but that is not important. It is just a practical way of structuring devices and managing device driver codes for all vendors. Each device has its own global memory which is the only memory the host side has access to, so all input and output data are passed here. Besides the memory needed by the device, it is mandatory for at least one department to be able to perform the calculation. If there is no department, there will be no place provided to calculate. Our school example uses two departments A and B. Compute units can work directly with global memory data, but they are more efficient when data are transferred to local memory. The differences between local and global memory are their speed and their size. Global memory is bigger than local memory, but local memory is faster than global memory. So the department should always copy their data from global to local memory. Local memory can be represented as the shared blackboard in front of the class. Every student can take notes from the board, calculate and return his result to the board. Notes are stored in private memory and the result is returned to local memory as soon as the work item, in this case a student, finishes calculating. When all workgroups are finished, a "master workgroup" is assigned to do the job of the school director, who will calculate the final sum with the returned results from every department.

Understanding a kernel's code is more difficult than the theory described above. That is why we

```
abc objectAbc (
    .a(A);
    .b(B);
    .c(C)
);
```

Program 2.21 *Verilog example, using a module*

```
module abc ( a, b, c );
    input     a;
    input     b;
    output    c;

    assign c = a and b;
endmodule
```

Program 2.22 *Verilog example, defining a module*

will only explain the kernel code, program 2.23, for a simple matrix addition. Three function arguments are provided, the first two are the input matrices and the third one corresponds to the output matrix. To calculate the addition, each element of matrix A should be added to the corresponding element of matrix B. The sum is saved in the corresponding element of matrix R. As can be seen by declarator "__global", only global memory is used to store the matrices in one dimensional float arrays. In this implementation every sum is made in one workgroup with one work item. Increasing the work items per group could make the kernel execution faster in terms of parallel computation. Sometimes it takes more time to copy data from global to local memory than the execution itself. Efficient kernel design has the need for research and testing. TUT PhD student Kui Wang has written a paper about using OpenCL to rapidly prototype FPGA designs [33]. He adjusted the number of work items in a workgroup and reduced hardware resource usage to replicate more OpenCL compute units. He concluded that in his Mandelbrot use case: "adjusting workgroup size has little impact on the speed of computation". He also concluded that: "by reducing the hardware resource usage per OpenCL compute unit, the number of compute units that can be replicated on the DE1-SoC board is increased".

When we return to our matrix addition example, it is important to understand how all workgroups iterates the data. Because every work item needs to be able to restore its result into global memory, pointers are commonly used. As explained in 2.5.1, a pointer points to an address of the register containing a specific value. In order to access each value of an array in C or C++, a for loop would be used. In OpenCL we use something equal, the function "get_global_id(0)". This function returns the number of the current workgroup, specifying which elements of the matrices should be accessed. A kernel code must be seen as the code

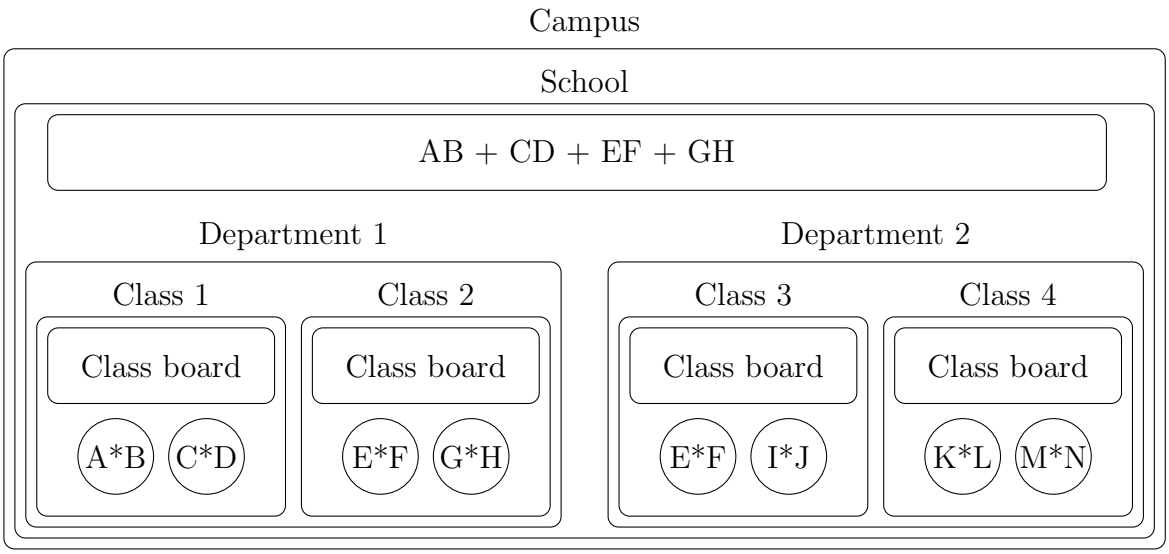


Figure 2.15 *OpenCL kernel representative school layout*

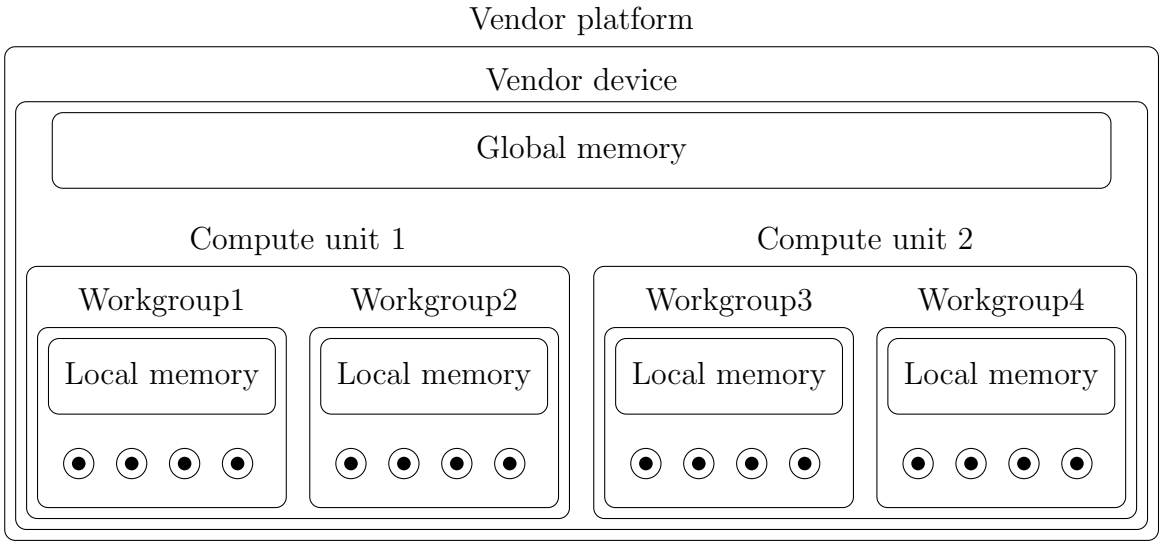


Figure 2.16 *OpenCL kernel layout: The circles represent work items with a dot in the middle as private memory*

for one element of a parallelization, using the global identifiers to iterate all the elements of the result "matrixR". Because OpenCL has influences of C-syntax, pointer notation is used to point to global memory.

2.5.3 Bridges

Between the HPS and the FPGA three bridges, mentioned in 2.5.1, are used to exchange data during runtime. By default only the FPGA configuration bridge is enabled, the HPS2FPGA and FPGA2HPS bridges should be configured in Qsys. Exchanging data from HPS to FPGA and back requires almost the same source code, but it was hard to find info about the FPGA2HPS

```
__kernel void matadd(    __global float* matrixA,
                         __global float* matrixB,
                         __global float* matrixR) {
    int i = get_global_id(0);
    matrixR[i] = matrixA[i] + matrixB[i];
}
```

Program 2.23 *OpenCL kernel example: matrix addition*

bridge. Both bridges can be configured with three different bus widths: 32, 64 and 128 bits.

Altera uses the ARM AMBA AXI bus to implement the bridges. Both first use the "mmap" function, program 2.24, in HPS to call a page of memory into the process's memory space [19]. Where "memoryBaseAddress" is the most important argument, it specifies where the bridge starts on the AXI bus. The base address is specified in Qsys, we use 0xC0000000. The base address needs to be added with an offset, for both HPS2FPGA and FPGA2HPS bridges. Explanation about the base memory offset is given in 2.5.4, paragraph *Qsys*. The second important argument specifies the size that needs to be reserved starting from the memory base address, called "PAGE_SIZE". The other parameters change with different setups. Parameter "bridge_map" is the virtual mapped memory address of the bridge. Writing the bridges assumes

```
bridge_map = mmap(NULL, PAGE_SIZE, PROT_WRITE,
    MAP_SHARED, fd, memoryBaseAddress);
```

Program 2.24 *mmap function*

basic knowledge of pointers, this can be found in 2.5.1, paragraph *C programming*. The pointer valueAdress, in 2.25, represents the location to find transmitted values. It needs an offset, specified during configuration in Qsys, to separate the AMBA AXI bus in different bridges. To write a value to the bridge, it needs to be written to the address of pointer valueAdress. Programs 2.25 and 2.26 show the code to write the HPS2FPGA and read the FPGA2HPS bridges respectively.

```
valueAdress = (unsigned char *) (bridgeMap + offsetHPS2FPGA);
*valueAdress = writeValue;
```

Program 2.25 *HPS2FPGA, transmit over bridge*

Reading is done in the same way, by transferring the pointed value of memory address "valueAdress" into the program.

```
valueAdress = (unsigned char *) (bridgeMap + offsetFPGA2HPS);
readValue = *valueAdress;
```

Program 2.26 *FPGA2HPS, receive over bridge*

FPGA Configuration: The FPGA can be configured in two ways. The most common way to configure the FPGA during development is to use "Quartus programmer". This tool uploads the SOF, SRAM Object File, to the FPGA. It is impossible to configure the FPGA with "Quartus programmer" each time the SoC is booted. The Cyclone V HPS can configure the FPGA by using the FPGA configuration bridge. Before the FPGA can be configured, all the AMBA AXI bridges have to be disabled. When they are not disabled a system crash of Linux will occur. Program 2.27 disables the bridges, it is written in the Bash programming language. By using "echo" text can be written into a file. In this case the text is a "0" and the file has a specified file path. Note the not yet discussed "lwhps2fpga", abbreviation for *light-weight HPS to FPGA* bridge. It is slower than the HPS2FPGA bridge and has a fixed width of 32bits. Programming the FPGA is done by copying the Raw Binary File with "dd"

```
echo 0 > /sys/class/fpga-bridge/hps2fpga/enable
echo 0 > /sys/class/fpga-bridge/fpga2hps/enable
echo 0 > /sys/class/fpga-bridge/lwhps2fpga/enable
```

Program 2.27 *Disable the AMBA AXI bridges from HPS*

into the device called "fpga0". In program 2.28 the "dd" command is shown. "dd" uses two arguments: "if" and "of", input file and output file. Once the FPGA is configured, the bridges

```
dd if=/home/root/FPGAConfigurationFile.rbf of=/dev/fpga0
```

Program 2.28 *Configuration of the FPGA with Linux dd command*

need to be enabled again. Otherwise they can't be used.

2.5.4 Quartus

Design structure

Pin Planner: Every design has to be fit into the device architecture and so do the connection to the peripherals of the embedded system. The pin planner assigns a signal name and directions to all used pins in the package. Directions can be Input, Output or Bidirectional. Pin planner adds other information automatically during "Assignment & Synthesis".

```
echo 1 > /sys/class/fpga-bridge/hps2fpga/enable
echo 1 > /sys/class/fpga-bridge/fpga2hps/enable
echo 1 > /sys/class/fpga-bridge/lwhps2fpga/enable
```

Program 2.29 *Enable the AMBA AXI bridges from HPS*

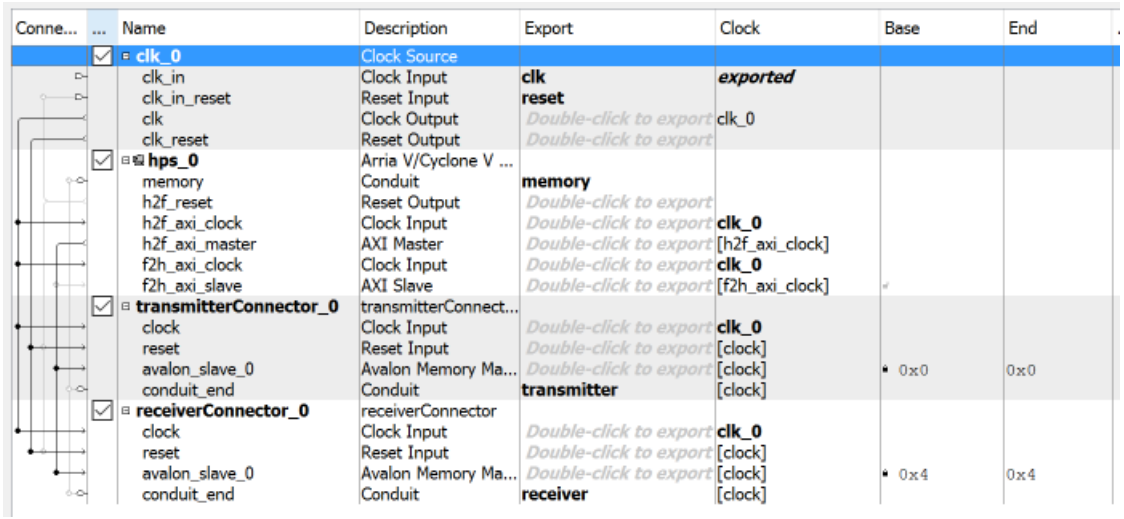


Figure 2.17 *Qsys internal connections*

Qsys: As Verilog is a complicated HDL, implementing a lot of modules in the system is a confusing task. Qsys simplifies this in figure 2.17, by providing a graphical tool to included IP. Figure 2.17 pictures all modules shown by name and all possible connections in the far left column. The highlighted connections are connected. Columns "Base" and "End" define the base and end address of the AMBA AXI bus used by the bridges between HPS and FPGA. Qsys generates one top module to include the whole Qsys system at once in the developer's his Verilog module. All the inputs and outputs of that Qsys top module are defined in column "export". Bridges are easy to configure in Qsys. The only choice is whether to use 32, 64 or 128 bits, as shown in figure 2.18. Once selected, Qsys will generate all the connections. When a Qsys design includes the AMBA AXI bridges, some connections need to be added to the pin planner. Luckily Quartus generates a .tcl file to add those connections to the pin assignments. However the .tcl file is only generated during the "Assignment & Synthesis", i.e. every time an AMBA AXI bridge is added, removed or changed in the Qsys-tool. The developer should restart compilation process "Assignment & Synthesis" followed by running the .tcl file.

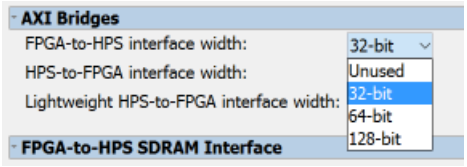


Figure 2.18 *Qsys configuration of bridges*

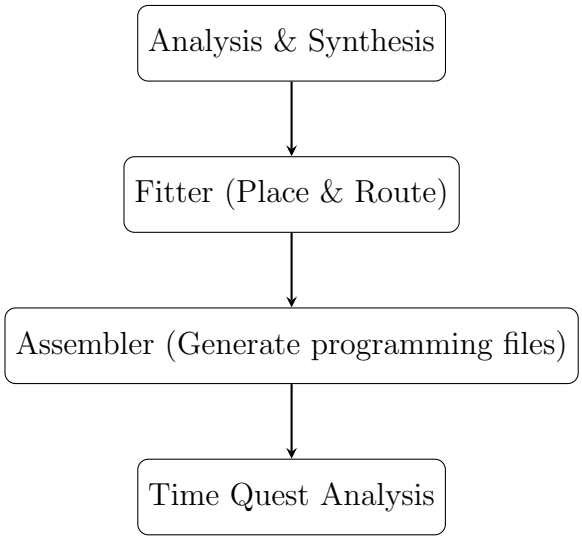


Figure 2.19 *FPGA design flow*

Compilation flow: Quartus compilation flow consists of 4 steps shown in figure 2.19 [4].

Analysis and synthesis checks the design files and overall design for errors. A design hierarchy is created and a single design database is built. During this step the design is changed to a minimum resource usage and uses the fixed logic modules as much as possible. This part is also used to perform a compilation check, because once this step has been completed, errors are rare. When errors occur in the next steps, it will be a problem on Quartus's end.

The fitter places and routes the developed logic design into a device architecture. Depending on the architecture, components need to be repositioned and connections to the components are routed in different ways.

The assembler creates an image, called *SOF*, to program the device. It can be compared with an executable in the C compilation flow.

In the last phase the design assistant checks the reliability of the design. Predefined design rules are used.

SignalTap II Logic Analyzer: If the configured FPGA does not work, SignalTap can help you. SignalTap is a logic analyser as shown in figure 2.20, where developers can review included signals. SignalTap needs two signals to be configured, the clock and a signal to trigger a "hold". Triggering conditions can be set on a signal by rising, falling or either edge. When the trigger condition occurs, all signals are plotted on a time span of a preset number of clock cycles. Logic analyzers give a good view into systems with a high clock speed.

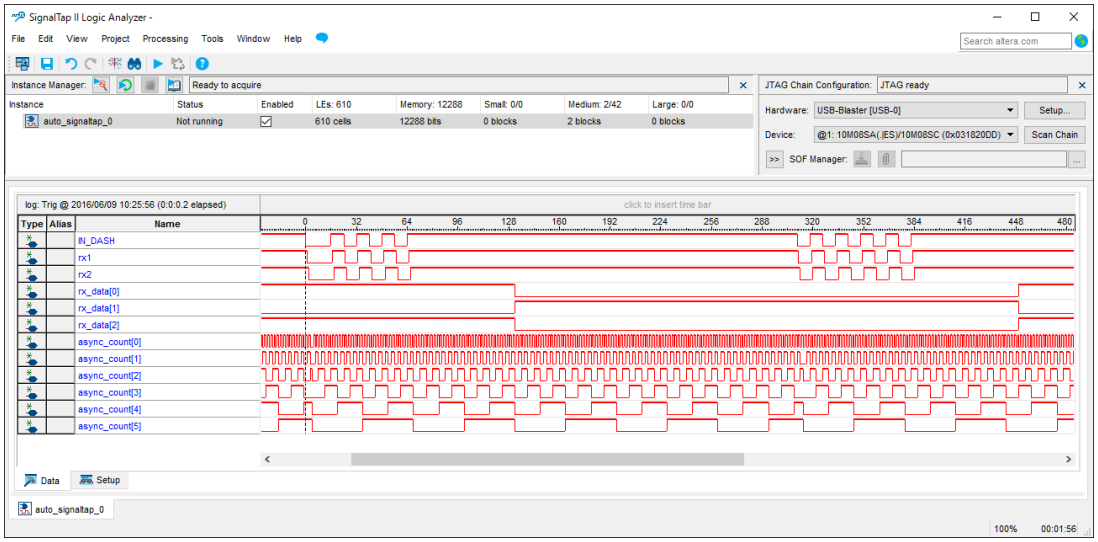


Figure 2.20 *SignalTap II Logic Analyzer layout*

Operating system issues

Quartus has some annoying issues. When using Quartus 16.1 installed on Red Hat 6.5, basic Verilog code and an OpenCL kernel can be compiled. However, Qsys, a tool to rapidly integrate IP of Altera in Quartus, gives synthesise errors during compilation of a program without faults. Those errors seemed to be unknown in the community, resulting in a trial and error problem search. Eventually the solution was installing multiple versions of Quartus on both Linux and Windows, Quartus 16.1 installed on Windows 10 seemed to work perfectly with Qsys-tool, but the OpenCL kernel did not compile anymore. Finally we used Quartus 16.1 installed on Windows 10 and Linux Red Hat 6.5 in order to successfully compile the system. Expensive commercially available programs like Quartus should work at all times. So when there is an error, developers expect they created the error, instead of Quartus itself creating error messages.

3. IMPLEMENTATION

This chapter discusses all the implementations accomplished during our thesis divided in three main parts: Communication protocol, SoC and Android. The SoC and Android can communicate using Bluetooth or WebSocket, resulting in a Bluetooth and WebSocket implementation in both the SoC and Android. The communication protocols are the same in SoC and Android, but the implementation is accomplished in different languages. The SoC is programmed in C++ and the OpenCL framework, whereas the Android API is programmed in Java. In our SoC the OpenCL framework is used to accelerate the matrix multiplication in a kernel on the FPGA. This chapter starts with the implemented character error detection. Character error detection is needed because we discovered data losses in the Bluetooth communication. In the next two sections, SoC and Android, both the receiver and transmitter are implemented with the use of this character error detection algorithm. Besides Bluetooth, we implemented a WebSocket. Since we did not discover any data losses using the WebSocket, character error detection was not implemented. Section SoC will also explain how the matrix multiplication is calculated in the OpenCL kernel.

3.1 Communication protocol

If we want to communicate between the devices, we have to define a set of rules that each of the devices have to follow. These rules should be made in a way that 100% of the transmitted data will arrive at the receiving end. For example, if data gets lost through transmission, the device on the receiving end will not receive all the necessary data. This can be prevented by adding a checksum that is different then from all other receivable data. If the received message looks incorrect, the receiving device has to respond by asking to resend this piece of data. We are trying out two different communication technologies so we have to set up a communication protocol for each one of them since they both have different performance aspects. To clarify future images, matrix A and B are the matrices sent to the SoC in order to calculate their multiplication. Matrix C is the calculated matrix sent back to the phone.

3.1.1 Bluetooth communication protocol

To define a set of rules we first analyzed how good the Bluetooth transmission is. We did some testing and came to the conclusion that there was quite a lot of data loss. We first tried sending eight characters at once but nearly 80% of the time there was at least one of the characters missing. Missing one number would mean that the matrix we need is incorrect and this would lead to an incorrect outcome. We did some further testing and realized it was the best option to send four numbers at a time. With this we can set up our first rule. The first rule being, we let both devices know that they should always send four numbers at once. Both devices also know they should receive four numbers. If this is not the case we have to resend the message. In order to make sure all characters would arrive at the receiving end, we added an extra character ':' at the end of each message. This character lets the receiving end know that it is the end of a message. If we would not define the end of a message and a number would be missing, the receiving end would wait for the next number to come through. The problem is that the next number would be part of the next message causing all numbers to be messed up with each other. An example of a message would look like this: 4372: .
Now for the communication, we start off by connecting both devices with Bluetooth. As soon as the devices are connected with each other, the SoC will send message "C" to the Android phone notifying it is ready to receive the two matrices. As soon as the phone receives message "C", it will start sending the first number or in some cases a part of the number. In case the message is received correctly, the SoC sends the message "O" to let the phone know the message was received. Figure 3.1 shows an example of how a correct message should look like.

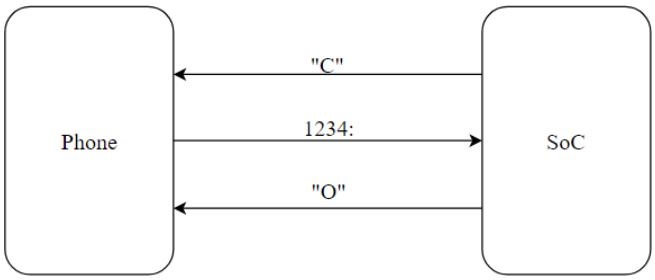


Figure 3.1 *Correct message transfer*

In case we are missing a number due to data loss, the receiving end will respond with the message "R" as shown in figure 3.2

There is also the possibility of a device not responding to a request. The receiving end is waiting for a response from its request. If the wait time is too long compared to a predefined waiting time, the receiving end will send a message "R" as seen in figure 3.3.

As for now, we work with square matrices with a predefined size. Both devices know at the start of the session how large both matrix A and matrix B are. This way the SoC can calculate how

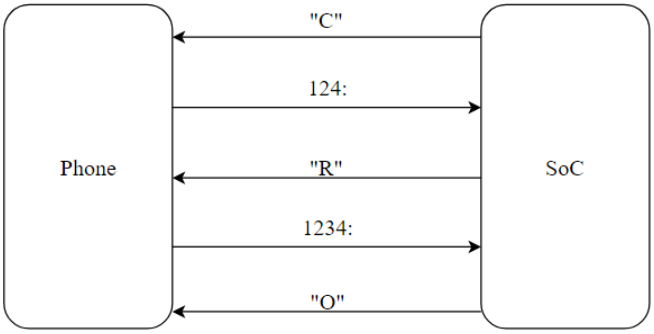


Figure 3.2 *Incorrect message transfer*

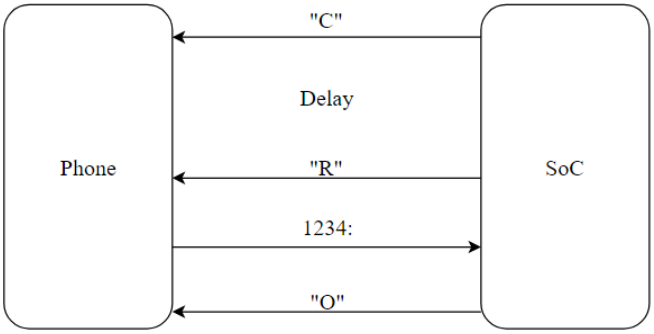


Figure 3.3 *Delayed message*

many numbers it should receive from the phone. Whenever matrix A is transferred succesfully, the Soc will send message ”D” to notify the phone it should start sending matrix B (Figure 3.4). The same procedures as mentioned above are applied on matrix B as well.

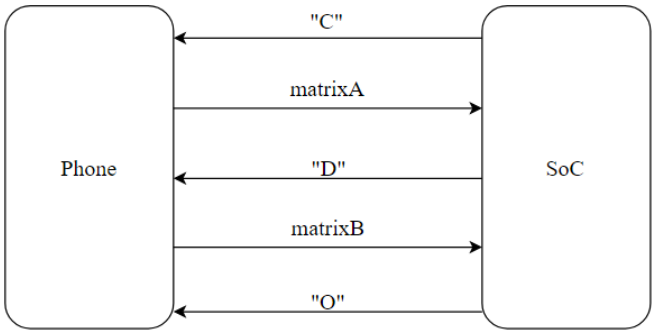


Figure 3.4 *Switch matrix message*

When both matrices are transferred, the role of transmitter and receiver are switched. The phone will now act as a receiver while the SoC becomes the transmitter. The SoC will transmit matrix C after doing the multiplication of matrices A and B. For this proces, the previous rules are also applicable. Although this time, the phone will start off by sending a message ”C” to notify the SoC that it is ready to receive matrix C. The SoC proceeds by sending the data in the same way as the phone did. Whenever the transmission experiences data loss, the phone will send message ”R” to the SoC exactly like the SoC does the other way around. Message ”D” is not used by the phone since there is only one matrix it should receive from the SoC.

When the phone has received matrix C, the Bluetooth connection between the devices will be suspended.

3.1.2 WebSocket communication protocol

The WebSocket communication has to achieve the same goal as the Bluetooth communication being transmitting matrix A and B. Depending on the efficiency and performance of this WebSocket communication we set up a similar set of rules. Compared to Bluetooth, the WebSocket transmission is much more efficient. We can easily transmit over eight characters without having any data loss. This means that we do not have to send acknowledgement messages like the "O" or "R" message that were necessary with the Bluetooth communication in order to have a solid data transmission.

If we look at subsection 2.4.2 under data frames, we see that the payload length of a message can be either 16 or 64 bits. However, this does not mean we are limited to sending messages smaller than 64 bits at a time since the socket will just split up the data in separate messages. This allows us to basically have any message size. Because of limited memory we chose to use integer numbers between 0-99. Next, we decided to combine 4 numbers and make one message with them. The message now contains 4 integers (16 bytes). We convert the integers to strings and add a "0"-string in front of numbers that are smaller than 10 to make sure the full message has a constant length of eight characters.

In order to have any communication at all, we start off by connecting both devices through WebSocket technology. As soon as both devices are connected with each other, the SoC will again start off by sending a message. The message will ask the phone to send matrix A. The phone then will start transferring all data from matrix A to the SoC. Whenever matrix A is transmitted completely, the SoC will ask to transfer matrix B and the phone will send matrix B. After the last number of matrix B being transferred, the SoC will confirm that all data has been received. In figure 3.5 we can see the exact messages needed from the SoC in order to have a correct communication process. The messages from the phone to the SoC are examples of how the messages could look like. The first message "12345678" would be numbers 12, 34, 56 and 78 being transmitted to the SoC.

After the SoC sends its last message it will start doing the matrix multiplication of matrix A and B with the outcome being matrixC. When the SoC is ready it will send the message "matrixR =" to notify the phone it will start sending over the values within matrix C. Unlike the phone, the SoC will send each number one by one since the number length varies with the size of the matrix. Figure 3.6 shows both types of messages being sent by the SoC. The phone should receive all numbers without having to give any feedback to the SoC. Whenever the full result matrix is transmitted by the SoC, the WebSocket connection is terminated and

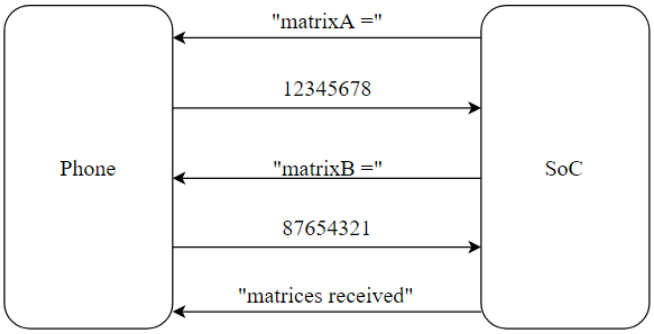


Figure 3.5 *Sending matrix A and B*

the complete process is finished.

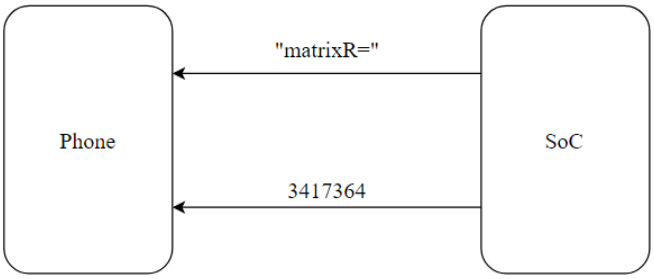


Figure 3.6 *Sending result matrix*

3.2 SoC

As described in section 2.5, a HPS and FPGA are provided in the DE1SoC development board. We use the HPS, see paragraph 2.5.1, to run a master program, where all the used peripherals are accessed and controlled. The master program is cross-compiled into an executable and the executable is started immediately after logging into the OS. To start a program automatically after login, the next code line must be added at the end of the file "/etc/profile".

```
exec ./matrixMultiplication
```

Command "exec" starts the execution of executable "matrixMultiplication". In this executable the Bluetooth module, WebSocket module and OpenCL kernel are initialised, executed and released from the OS. Figure 3.7 represents the complete implemented SoC. Chapter 2, "Theoretical background", gives an introduction to the layout of a SoC. The two main parts are the HPS and the FPGA connected with bridges to share data. In the next two points we will explain the data flow in case we choose to transmit our data through the Bluetooth or WebSocket connection.

Bluetooth Our DE1SoC board does not have on board Bluetooth, but TerASIC provided the RFS daughter card shipped with the HC-01 Bluetooth module. This card is connected to

the GPIO1 socket, which is directly and only connected to the FPGA, see figure 2.9. We used HDL Verilog to interact with this Bluetooth module through UART in file "TopLevelModule.v". Once the data are received, they need to be transmitted to the HPS. Altera provides bridges for this, but they need to be configured with Qsys. Qsys provides just a connection with the bridges, so two extra modules, "receiverConnection.v" and "transmitterConnection.v", are needed to connect Qsys with the topLevelModule. When we take another look at the implementation block diagram in figure 3.7, we can see that these two modules are not in between Qsys and the topLevelModule. We did not want to place them in between, because they are added as a custom component inside Qsys. The main benefit of importing these files in Qsys, are the uncluttered connections in topLevelModule. Once the data are transferred to the HPS, the FPGA is reconfigured with "OpenCL.rbf" and class "openclHost" will configure the environment for the OpenCL kernel. Next, the kernel receives two matrices from the bridges, calculates the matrix multiplication and returns the result back to the HPS. Lastly, our developed "matrixMultiplicatoin.rbf", is used to reconfigure the FPGA and send the result matrix back to the Android device using Bluetooth.

WebSocket The implementation of the WebSocket is much simpler. Classes wsReceiver and wsTransmitter will respectively receive the two matrices and send the result matrix. The Class openclHost calculates the result matrix by configuring the FPGA with "OpenCL.rbf" in the same way as explained above in "Bluetooth".

All the used classes and HDL, for matrix multiplication and both Bluetooth and WebSocket communication are explained in depth in the next paragraphs.

The main function can idle in two states, determined by the first command line option. When the option is a "b", then Bluetooth is enabled as communication protocol. If it is "w", the WebSocket will be used. In either case, the FPGA needs to be configured to use Bluetooth or OpenCL. Function "setBridges()" with a string as parameter, indicating Bluetooth or OpenCL, configures the FPGA with respectively "matrix_multiplication.rbf" and "opencl.rbf". Raw Binary File "matrix_multiplication.rbf" is designed for the Bluetooth communication in this thesis, more info about the design can be found in 3.2.3 "Bluetooth on the FPGA". The second RBF "opencl.rbf" is provided by Altera to configure the AXI AMBA bridges and FPGA, in order to make them both ready for an OpenCL kernel execution.

3.2.1 Websocket

The websocket server package used in our SoC is called WebsocketD. WebsocketD is the simplest WebSocket available and can be used on almost all platforms by all languages that can write

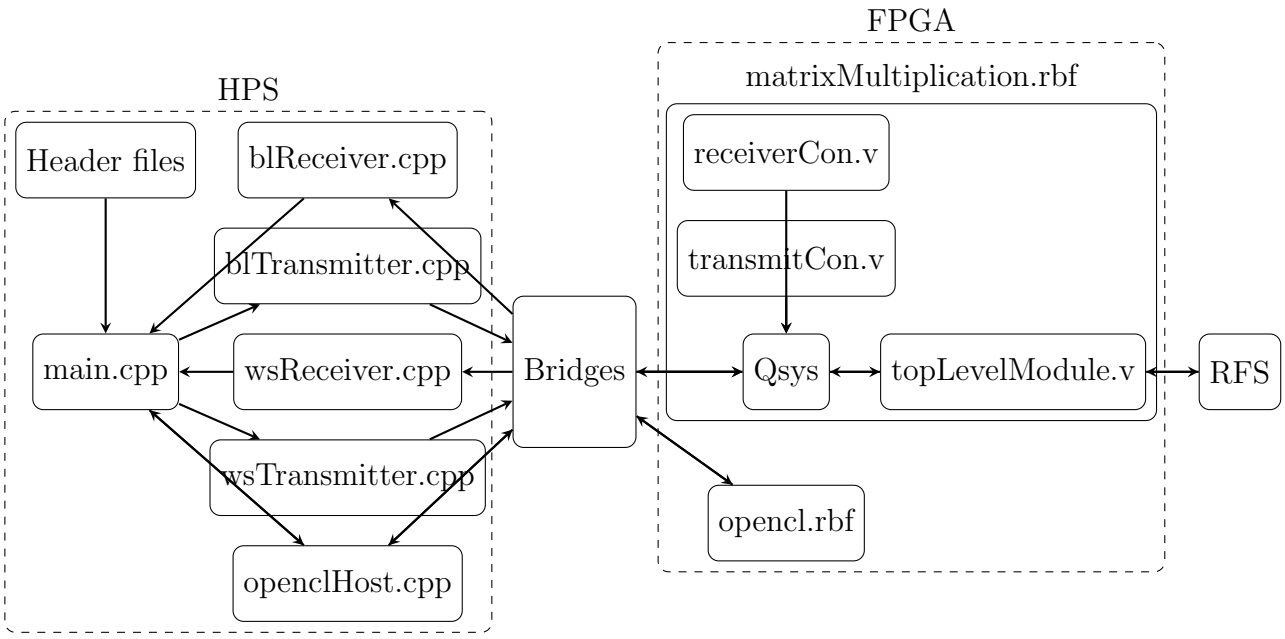


Figure 3.7 *Block diagram of the implemented system on the SoC with both the HPS and FPGA modules*

a "printf" to the shell. The only requirements are a valid network connection between server-client and an executable called "websocketD". The executable replaces messy code with libraries by the two required command line arguments listed below.

- Port: All WebSockets uses a specific port to communicate. This port can be chosen randomly as long as that port is unused and enabled in the OS. We use port: 8080.
- Executable: The executable of the developed program, in our case matrixMultiplication. This is also the place to give command line arguments to the main program executable, as can be seen in the example below.

```
./websocketd --port=8080 ./matrixMultiplication w
```

There is only disadvantage using this WebSocket setup. It is not practical to debug your program 'old school', by printing debug information to the shell, because every printf will be transferred to the WebSocket client.

receiver

Receiving data in a c++ WebSocket program is simple with WebsocketD. Including the "std" library makes the developer able to use "cin", a variable that refers to the standard input stream, to read data from the shell. The example code line below pauses the program execution until

there is a new text entry in the shell followed by a 'new line' to end the word. Variable "cin" will be transferred to string "s", once the 'new line' is detected.

```
cin >> s;
```

transmitter

The transmitter uses the same library as the receiver to write to the shell. Variable "cout" prints the data given in string "s" to the shell followed by a "\n", as can be seen in the example code line below. The "\n" creates a new line. Each new line is considered as a "send now" signal to the WebSocket.

```
cout << s << "\n";
```

3.2.2 Bluetooth on the HPS

Bluetooth is implemented as one of the communication protocols between the Android API and the SoC. We implemented a receiver and transmitter C++ class, where the data are read from and written to the bridges. The main function in figure 3.7 invokes the receiver and transmitter classes. The character error detection algorithm embedded in those two classes is explained in the next two paragraphs.

Receiver in C++

The receiver class has two public functions, one called "pullMatrix" that returns a pointer to an array, pointing in the pulled matrix and "receiveChar" used on the character error detection to receive only one character at a time. Function "receiveChar" is an almost identical version of function "receiveData" discussed in "pullMatrix". Function "pullMatrix" receives one character at a time from private function "receiveData()" and organises all the characters into an array of integers, representing an input matrix.

When function "receiveData()" is called, then data are read from the bridges by the following three code lines.

```
fd = open("/dev/mem", O_RDWR | O_SYNC);
bridgeMap = mmap(NULL, PAGE_SIZE, PROT_READ, MAP_SHARED,
        fd, bridge_base);
character = (unsigned char *) ((char*)bridgeMap +
        RECEIVEROFFSET);
```

First, system call "open" handles the receiving of data by opening memory file "/dev/mem", with option "O_RDWR" to give the program read and write permission. The second option "O_SYNC" makes sure that all data have been transferred, when a read or write is requested, before the program will continue. The value returned into integer "fd" after the function is ended, is the identifier of the newly opened file. This identifier is always the smallest available integer greater than zero.

Secondly parameter "bridgeMap" contains the base address of the used memory block. Function "mmap" links the base address of the bridge register "bridge_base" with the base address of the used memory. Parameter "PAGE_SIZE" defines the space needed for the bridges to be able to write all the data. The next two parameters are necessary options, they give read-only access rights to the memory and make sure that the memory updates are immediately available to other mmap functions.

Thirdly, pointer "character" points to the begin address of the virtual memory linked with the bridges register. In other words, the value of pointer "character" contains the character located in the bridges. The address of "character" is formed by adding an offset to the base address of the memory, "bridge_map".

Once character points to the base address of the receiver bridge, our error detection algorithm can start. Program 3.1 is a copy of the algorithm, before we start the algorithm a timer is invoked. This timer will make sure that, when the algorithm is idling too long without receiving a character, the algorithm is reset and the data will be repeated. The while loop will be repeated WORDSIZE times plus two, because there should be received WORDSIZE characters closed with a ":" which represents one extra character. A second extra character does not exist, but adding it to WORDSIZE allows us to detect an error. This error occurs if there are "DATASIZE" characters received and the next characters is not an ":". Next, two if-statements are used: one to check if the received character is a number and another one to detect errors in the communication. The next step is important for a correct execution, the value pointed to by "character" is fixed into integer "c". This step is mandatory, due to the fact that "character" is declared as volatile. When a variable is declared as volatile it can be changed at any time, in our case it allows us to change the variable during runtime from an external source. The external source is the bridge. Because the variable can be changed at any moment, we need to make sure that it will not change during one cycle of the while loop in program 3.1, resulting in using integer "c" for the rest of the cycle. The two if-statements will be discussed independently:

- First if-statement: Here a check will occur if the received value is a number. When the received value is a number, it can be converted from ascii to integer and added to array "preArray". Once the while loop has finished without any error from the second

if-statement, preArray will be returned to function "pullMatrix()". Only when a number is detected successfully, the variable "numberOfChar" is increased.

- Second if-statement: During every iteration of the while-loop it is checked if the "numberOfChar" is equal, lower or higher than WORDSIZE. When it is equal to WORDSIZE and the currently received character is a ":", then the communication was successful. After a successful communication command "OK" is sent using function "transmitData" of object "trans". In all other cases of the second if-statement an error occurred.

  All errors are handled in the same way: send a repeat request using function "transmitData" of object "trans" and repeat the same function "receiveData" concurrently.

```
startTimer();
while(numberOfChar < (WORDSIZE+2)) {
    int c = *character;
    if((c != 'a') && (c != ',') && (c != ':')) {
        int charValue = asciToInt(c);
        if(charValue == -1) {
            tran.transmitData(REPEAT);
            receiveData();
            break;
        } else {
            preArray[numberOfChar] = charValue;
        }
        numberOfChar++;
    }
    if((numberOfChar == WORDSIZE) && (c == ':')) {
        tran.transmitData(OK);
        numberOfChar=0;
        break;
    } else if((numberOfChar < WORDSIZE) && (c == ':')) {
        tran.transmitData(REPEAT);
        receiveData();
        break;
    } else if(numberOfChar > WORDSIZE) {
        tran.transmitData(REPEAT);
        receiveData();
        break;
    } else if(duration() > PASSEDTIMEREPEATRECEIVEDATA) {
                tran.transmitData(REPEAT);
        receiveData();
        break;
    }
}
```

Program 3.1 *Class receiver: receiveData algorithm*

Transmitter in C++

The moment a matrix is pushed in the main function, the function "pushMatrix" of class transmitter is called. This function will first establish the connection in order to enable transmitter-mode. StartTransmitter sets the connection in transmitter-mode by sending a "DONE". If the next received value is a "COME" transmitter-mode is active. Next, the transmitter will send each element of the matrix separately. A problem occurs if the number of an element has less digits compared to the "DATASIZE", because the character error detection algorithm does not allow changes in "DATASIZE". In order to make sure that all array elements have the same "DATASIZE" we add a couple of zeros. Imagine for example an element with value '123', while a defined "DATASIZE" equals to four characters. Because of the algorithm an error will occur while the fourth character is sent, but the fourth character does not exist. Our solution for this problem is adding zeroes to the beginning of the number, resulting in: '0123'.

We continue with transferring the word into function "sendWord". Here each character of the string is transmitted by function "transmitData". When all the characters of a word are sent and ended with a semicolon, then the algorithm of program 3.2 will verify the transmission. It begins with a request to receive a character from object "rec" of the class receiver. If the receiving device answered with an "OK", it returns a "1" to the function. In all other cases, where one or more characters of the transmitted package are lost, a "REPEAT" command is returned to the transmitter. During the last scenario the function "sendWord(x, c)" is called again, concurrently, with the whole word as parameter. This function is called every time something goes wrong. When the communication succeeds, returning a "1", then all concurrently called functions return a "1" into each other. By using this return value, function "pushMatrix" knows whether the transmission was successful or not. Lastly, the characters are pushed into the bridges. This is simply done by writing the character value into the pointer pointing to the bridge mapped memory.

```
while(!received) {
    int receivedValue = rec.receiveChar();
    if(receivedValue == OK) {
        return 1;
    } else if(receivedValue == REPEAT) {
        return sendWord(x, c);
    }
}
```

Program 3.2 *Class transmitter, function sendWord: algorithm to send a word*

3.2.3 Bluetooth on the FPGA

We used the RFS daughter card plugged into the 2x20 GPIO socket to provide our SoC with Bluetooth. That socket is a FPGA-only peripheral, resulting in a Bluetooth data path through the FPGA. To configure the FPGA, HDL Verilog code is developed. To share the data between FPGA and HPS, the AMBA AXI bridges are configured using Qsys. A detailed explanation can be found in 2.5. First the configuration in Qsys of the HPS-FPGA bridges will be clarified. Secondly and thirdly the Verilog and C++ code of the receiver and transmitter are investigated, respectively.

Qsys module

Qsys is a graphical programming tool provided in Quartus, paragraph 2.5.3 introduces Qsys with HPS-FPGA bridge configuration. Figure 3.8 contains 4 modules: clock_0, hps_0, transmitterConnector_0 and receiverConnector_0.

- Clock: The module clock is set as default, because Qsys is a smart graphical interface to develop a HDL. All HDL designs include a clock signal, so Qsys provides the clock by default, see paragraph 2.5.2.
- HPS: In Qsys all the internal connections are made in the connections column, left most column in figure 3.8. This way the clock signal is distributed to all clock inputs of all the modules. Module called "hps_0" belongs to the IP of Quartus, here the bridges are implemented and HDL communication with the bridges is provided. Since in our design the HPS controls everything, we have chosen to use a master connection for both the receiver and transmitter, see "h2f_axi_master" instead of "f2h_axi_slave" in figure 3.8.
- ReceiverConnector: When there is no IP provided by Quartus suitable for the developers's design, then developers can include their own HDL into a Qsys system. We made our own receiver and transmitter HDL modules and included them in the Qsys design. Program 3.3 shows the transmitter module in Verilog. It shows two inputs and two outputs used to read data from FPGA to HPS. Signal "inRead" gives the command that the Bluetooth module wants to send data provided in 8-bit signal "inReadData". Assign is used to assign 8-bits of zeros to the exiting signal when "inRead" is low, while the incoming data is directly connected to the output data at the moment "inRead" becomes high. The read signal is set at each positive edge of the clock in the always statement. If "inRead" is high, "outRead" will become high too. We want to prevent HDL races, see paragraph 2.5.2, at any cost, but still the input data signal will be assigned to the output data whenever signal "inRead" is high. This, however is irrelevant, because the data will be stuck at

the next buffer, waiting for the "read" signal arriving at the buffer and transmitting the buffered signal through the bridges.

```
module receiverConnector (
    input                clk, reset, inRead,
    input [7:0]          inReadData,
    output wire [7:0]    outReadData,
    output reg           outRead );

    assign outReadData = inRead ? inReadData:8'b0;

    always @(posedge clk) begin
        if(reset)
            outRead <= 1'b0;
        else if(inRead)
            outRead <= 1'b1;
        else
            outRead <= 1'b0;
    end

endmodule
```

Program 3.3 *Module receiverConnector Verilog code*

- TransmitterConnector: The transmitter uses a similar program to write data from the bridges to FPGA, program 3.4 shows the Verilog code. There is an input and output signal for both write and writeData. Basically the only thing that changes, except for the direction of data flow, is that in this case the module will need to block the data signal until the moment the signal "write" is high on a positive edge of the clock. If the transmitterConnector module would not do this, races would occur which results in package loss.

Qsys is integrated as one module into the top-level-module, as can be seen in figure 3.7. In the top-level-module all wires are assigned to each input or output of the object named "SoC_System". The third column of figure 3.7 sets a name to export external connections, those connections correspond to the connections of the previously mentioned assignments.

The only thing in between the data from Qsys and the RFS daughter card of TerASIC is the UART connection. We implemented an UART module and some extra code, programs 3.5, 3.6 and 3.7, to make a data flow between Qsys and UART. To write and read data in the UART module, two times three signals are provided. Respectively "write" and "read" must be high when writing or reading data. Signals "writedata" and "readdata" are 8-bits signals to transport the data and signals "wrfull" and "rdempty" sign that the UART communication

```
module transmitterConnector (
    input clk, reset, inWrite,
    input [7:0] inWriteData,
    output [7:0] outWriteData,
    output outWrite );

    reg write;
    reg [7:0] writeData;

    assign outWrite = write;
    assign outWriteData = writeData;

    always @(negedge clk) begin
        if(reset)
            write <= 1'b0;
        else if(inWrite)
            write <= 1'b1;
        else
            write <= 1'b0;
    end

    always @(posedge clk) begin
        if(reset)
            writeData <= 8'b0;
        else if(inWrite & write)
            writeData <= inWriteData;
        else
            writeData <= 8'b0;
        end
endmodule
```

Program 3.4 *Module transmitterConnector Verilog code*

buffer is full or empty. The three last module arguments are a 25MHz clock input and the two "tx" and "rx" communication lines. Our main clock of the FPGA has a frequency of 50MHz, by using program 3.5 the 50MHz clock is converted to a 25MHz clock. In this program "cnt" is a single bit register. When "cnt" is zero and an one is added, then the result will be one. However, if "cnt" is one and another one is added, then "cnt" becomes two. Since the number two is "10" in binary and only the least significant bit is assigned to one-bit register "cnt", "cnt" will be zero again.

```
always@(posedge clk)
    cnt <= cnt + 1;
```

Program 3.5 *FPGA 50MHz clock divider to 25MHz clock for the UART module*

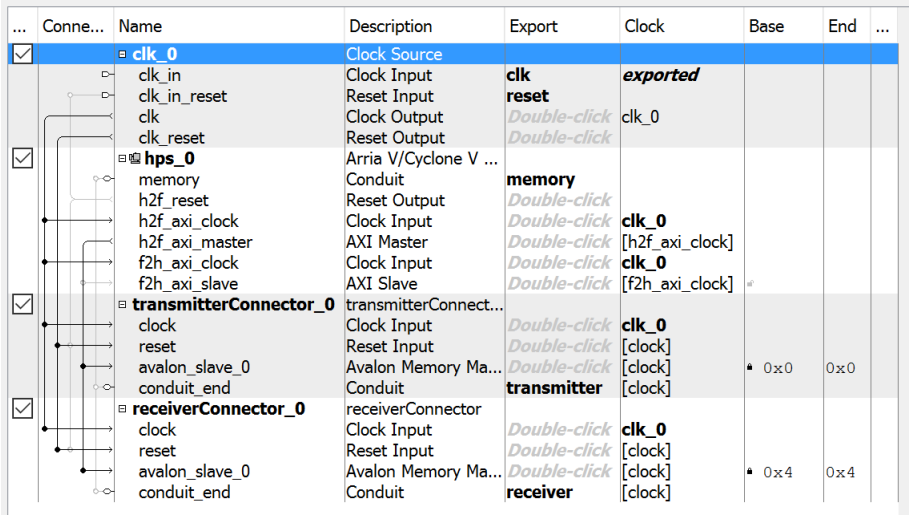


Figure 3.8 *Graphical programming layout of the system in Qsys*

Receiver implementation in top-level-module

To connect the UART module with the Qsys system program 3.6 is needed. Because this is the receiver, the bridge data are written from FPGA when the "writeBridge" signal is high. When UART gives the "read" as high and the "rdempty" as low, then the incoming data from UART are forwarded to the register "readDatahold". At all other moments, the hexadecimal value h61, ascii for "a", is transmitted. Detecting "a" is mandatory in case the UART module is not sending useful data. These situations can happen, because the UART and Qsys modules are working independently and constantly near each other. At any time, Qsys can ask for data by setting "receiveRead" high and checking wire "receiveData", these data always come from register "readDatahold" and will be either the UART data or an "a" that is filtered out in the HPS. The wire "receiveData" is reset to eight zeros when Qsys does not want to read data. It is confusing that in the second always statement of program 3.6 the register "read" is set by the use of wire "rdempty", while in the second assignment both of them are mandatory to determine the value of wire "writeBridge". This is because the second always statement gives a one clock cycle delay to the Qsys system before reading the data.

Transmitter implementation in top-level-module

The transmitter has a data path in the opposite direction of the receiver data path. The second always statement of program 3.7 implements a delay-check. It checks if the wire "nextWriteData", connected to Qsys, was previously high and currently low. If this situation occurs, then the data coming from Qsys in "transmitterWritedata" will be connected to register "writedata" in the UART module. At the same time signal "write" connected to the UART module is set high. When the writing condition is not achieved, ascii value "NULL" is sent by the

```
assign receiveData = (receiveRead) ? readDatahold:8'h0;
assign writeBridge = ( read & (~rdempty));

always@(posedge clk) begin
    if(!reset)
        readDatahold <= 8'h30;
    else if(writeBridge)
        readDatahold <= readdata;
    else if(receiveRead)
        readDatahold <= 8'h61;
end

always@(posedge clk) begin
    if (~rdempty)
        read <= 1;
    else
        read <= 0;
    end
```

Program 3.6 *Integration of the receiver in the top-level-module*

hexadecimal number h0.

3.2.4 OpenCL host

Our OpenCL uses a host in C++ to configure the environment before implementing an OpenCL kernel. Chapter "Theoretical background" handles a general description about the host class, this paragraph will describe our implementation. It all starts by calling function "startHost()" with two input matrices A and B as parameters. First, OpenCL must detect which devices of which platforms are connected. Next, a context needs to be created for every device used, this is a medium to connect host program and device to talk with each other. The command queue on the other hand is a specific data-stream on top of the context, it enables the host program to talk directly to a kernel running on a device. Up to this point only an environment has been created, but how the kernel should behave has not yet been defined The code after the white space in program 3.8 configures the kernel with an offline pre-compiled kernel. Firstly, the binary file is loaded from the user directory in the OS, running on the SoC. Secondly a program will be created and built from this binary file, the desired device and corresponding context. Developers can include specific options during the kernel build. We don't use these, but to give an example: "-cl-opt-disable". This option will disable the standard enabled compiler kernel optimimalization. Thirdly and lastly a kernel object will be created. Kernel objects are used in the next part to easily code the data-stream to the kernel.

```
always@(posedge clk) begin
    if(!reset)
        oldNextWriteData <= 1'b0;
    else if(nextWriteData)
        oldNextWriteData <= 1'b1;
    else
        oldNextWriteData <= 1'b0;
end

always@(posedge clk) begin
    if(!reset) begin
        write             <= 1'b0;
        writedata <= 8'h0;
    end else if(nextWriteData == 1'b0 &&
          oldNextWriteData == 1'b1) begin
        writedata <= transmitterWritedata;
        write             <= 1'b1;
    end else begin
        write             <= 1'b0;
        writedata <= 8'h0;
    end
end
```

Program 3.7 *Integration of the transmitter in the top-level-module*

```
platform = findPlatform("Altera");
dev.reset(getDevices(platform, DEVICE_TYPE, &num_devices));
context = clCreateContext(NULL, 1, &dev[0], NULL, NULL, &err);
queue = clCreateCommandQueue(context, device, OPTIONS, &err);

binFile = getBoardBinaryFile(KERNEL_NAME, device);
program = createProgramFromBinary(context, binFile.c_str(),
                &dev[0], 1);
clBuildProgram(program, 0, NULL, options, NULL, NULL);
kernel = clCreateKernel(program, KERNEL_NAME, &err);
```

Program 3.8 *Declaration of OpenCL kernel environment in OpenCL host*

3.2.5 OpenCL kernel

The OpenCL kernel is the main subject of our thesis. If there was no OpenCL, then we would be forced to use a HDL to accelerate the matrix multiplication. HDL descriptions are way more difficult to develop and to implement compared to a programming language. Chapter "Theoretical background" gives the most simple OpenCL kernel example, a matrix addition, in 2.5.2. All kernel codes explained in this chapter are based on the vector addition example. Each

kernel will do the same matrix multiplication with their own advantages and disadvantages. In the following item list, all used kernel codes are explained in depth. They all have the same parameters A, B, result and "ARRAYSIZE", corresponding respectively to the two input matrices, the output matrix and the dimension of the square matrices. OpenCL can not handle pointers to pointers, resulting in one-dimensional kernel function arguments, instead of two-dimensional function arguments.

- The first OpenCL kernel we developed is the most simple one, represented in program 3.9. When programming a matrix multiplication in a single thread, multiple for-loops are used to provide the iterators used in the calculation. Because a kernel represents the behaviour of one element, called a workgroup, of all the parallelized elements, a constant iterator parameter is needed in each workgroup. More info about how workgroups fit in the OpenCL architecture can be found in paragraph 2.5.2. Constant integers "r" and "c" correspond to these iterators. To store a result in between calculations, global integer "sum" is used. Table 3.1 represents the basic principle of our kernel algorithm for a 2 x 2 matrix multiplication. As can be seen "r" and "c" are iterating all possible situations for each matrix. For each iteration, derived by equation 2.2, "ARRAYSIZE" multiplications, represented by "i", need to be calculated and added into global parameter "sum". When the "ARRAYSIZE" multiplications are added together, then the result "sum" is returned into the corresponding place of array "result".

- Where the first kernel uses global memory to store the result in between calculations, this second kernel will use local memory. Paragraph 2.5.2 describes the advantages and limitations of using global vs local memory. Global memory is bigger but slower than local memory. We expect to have a shorter calculation time when doing a matrix multiplication on bigger matrices. Program 3.10 represents the second kernel. Local memory parameter sum cannot be accessed by a pointer like a global parameter, because it is stored in a register on the FPGA side. Although data on the FPGA side are equally stored as data in software, by using register addresses, OpenCL does not allow pointing to local memory.

- Our third kernel implementation, program 3.11 is more advanced. The second kernel writes all the sums to local memory, but it reads global memory twice every time a multiplication is done. We tried to reduce the amount of time we read from global memory to the minimum, by copying the input matrices into two-dimensional arrays in local memory. This step will add some time to the process, but we think that it pays off when calculating big matrices, because reading local memory is much faster than global memory. The algorithm stays the same, but one additional step is required. All elements are processed in this kernel in parallel, but the algorithm can only be invoked when all the data are transferred from the global into the local memory. Function "barrier()" with

parameter "CLK_GLOBAL_MEM_FENCE" makes sure that each workgroup has written to local memory, before they can start the matrix multiplication algorithm.

```
kernel void multiplicationSum(global int* restrict A,
        global int* restrict B, global int* restrict result,
        int ARRAYSIZE, global int* restrict sum) {
   const int r = get_global_id(0);
   const int c = get_global_id(1);
   *sum = 0;

   for (int i=0; i<ARRAYSIZE; i++) {
      *sum += A[r x ARRAYSIZE + i] x B[i x ARRAYSIZE + c];
   }
   result[r x ARRAYSIZE + c] = *sum;
}
```

Program 3.9 *OpenCl kernel: global memory implementation*

r	c	i = 0	i = 1	sum	r x Size + c
0	0	A x E	B x G	A x E + B x G	0
0	1	A x F	B x H	A x F + B x H	1
1	0	C x E	D x G	C x E + D x G	2
1	1	C x D	D x H	C x D + D x H	3

Table 3.1 *OpenCL kernel: matrix multiplication implementation flow table with matrixSize equal to two*

```
kernel void multiplicationSum(global int* restrict A,
        global int* restrict B, global int* restrict result,
        int ARRAYSIZE) {
   const int r = get_global_id(0);
   const int c = get_global_id(1);
   local int sum;
   sum = 0;

   for (int i=0; i<ARRAYSIZE; i++) {
      sum += A[r x ARRAYSIZE + i] x B[i x ARRAYSIZE + c];
   }
   result[r x ARRAYSIZE + c] = sum;
}
```

Program 3.10 *OpenCl kernel: local memory implementation*

```
kernel void multiplicationSum(global int* restrict A,
        global int* restrict B, global int* restrict result,
        int ARRAYSIZE) {
    const int row = get_global_id(0);
    const int col = get_global_id(1);

    local int subA[64][64];
    local int subB[64][64];
    local int sum;
    sum = 0;

    subA[row][col] = A[row x ARRAYSIZE + col];
    subB[row][col] = B[row x ARRAYSIZE + col];
    barrier(CLK_LOCAL_MEM_FENCE);

    for (int k=0; k<ARRAYSIZE; k++) {
        sum += subA[row][k] x subB[k][col];
    }
    result[row x ARRAYSIZE + col] = sum;
}
```

Program 3.11 *OpenCl kernel: global to local memory copy implementation*

3.3 Android apps

The android phone needs proper software in order to function well in our expirement. To build this software we use Android Studio. It is an easy-to-use java environment with a lot of information that can be found on the internet. We will make three different applications using Android Studio. One application to test the matrix mulitplication with the phone itself to compare the speed with the SoC. A second application is responsible for the Bluetooth communication and a third application is used for the WebSocket communication.

3.3.1 Matrix multiplication

This app will multiply two square matrices that are filled with random integers converted to strings (see subsection 3.1.2 on why we do this). We start off with creating two matrices. Program 3.12 shows how this is done. First, we make an array of integers that is "arraySize" large. We start a for loop that will cycle "arraySize" times. In line 4 we create a random integer between value 0 and 99 and put this in the array "matrixA" in the next line. MatrixB is made in exactly the same way.

We then go through program 3.13 which is a basic way of multiplying two matrices.

```
1| matrixA = new Integer[arraySize];
2| int random;
3| for (int i=0; i<arraySize;i++){
4|   random = (int) (Math.random()*100);
5|   matrixA[i] = random;
6| }
```

Program 3.12 *Creating matrix*

```
1| t1 = (int) System.currentTimeMillis();
2| for(int i=0; i<arraySize; i++){
3|   for(int j=0; j<arraySize; j++){
4|     sum = 0;
5|     for(int k=0; k<arraySize; k++){
6|       sum = sum + matrixA[i*arraySize+k]*matrixB[k*arraySize+j];
7|     }
8|     matrixC[i*arraySize+j] = sum;
9|   }
10| }
11| t2 = (int) System.currentTimeMillis();
```

Program 3.13 *Code for matrix multiplication on Android phone*

First, we determine the system time and assign this value to t1. In section 2.1, we explain how matrix multiplication works so this should clear up lines 2-10 in the program above. After the multiplication is finished we determine another system time and assign this value to t2. If we substract t1 from t2, we know how many milliseconds it took to complete the full multiplication. The layout of this app can be seen in figure 3.9.

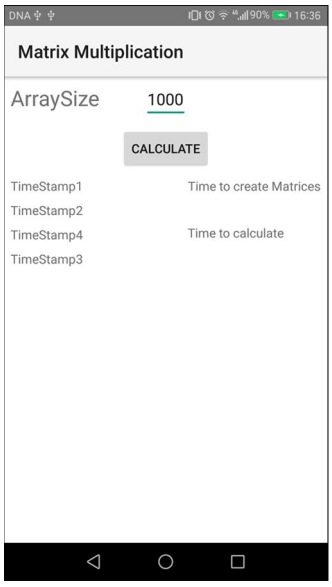


Figure 3.9 *Matrix multiplication app*

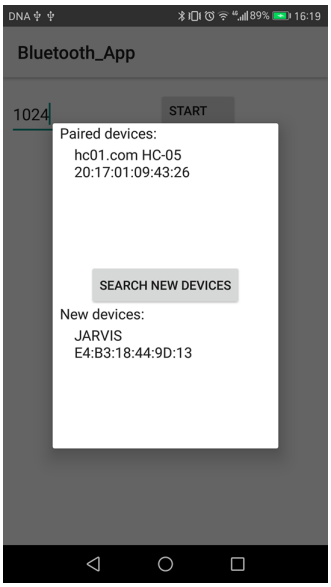


Figure 3.10 *Bluetooth menu*

3.3.2 Bluetooth application

In the next two apps, we will transfer the matrics over to the SoC that will perform the matrix multiplicaiton. First, we will explain how the Bluetooth app works.
Since this app works with Bluetooth, we have to set up the permissions for the app to use the phone's Bluetooth adapter in the AndroidManifest.xml file like so:

<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.BLUETOOTH"/>

The first thing the app will do is check wether or not your phone's Bluetooth is enabled. If not, it will ask you if you want to enable the Bluetooth. If you press allow, you will be directed to the main screen. If the Bluetooth was already enabled, nothing will pop up and you will see the main screen in front of you. The front end of the app is simple, consisting of only one textview and two buttons. The textview has the purpose to allow you to input a custom size for your matrix's row and column. The first button "Start" will make 2 matrices with the given size. The other button will open a menu with 2 lists and a third button. The first list will show you all the already paired Bluetooth devices on your phone. The other list is empty, but after pressing the button "Search new devices", the list will fill up with discoverable devices around you as you can see in figure 3.10.Tapping on one of the new devices will result in an attempt to pair with this device. Tapping on a paired device in the list will result in an attempt to connect to that device. The device you selected will be the device you will communicate with. Android Studio provides libraries that easily gives you acces to the Bluetooth adapter in the phone. More information on how the Bluetooth adapter works can be found on the Android developer website [1]. When the connection is established, the protocol described in 3.1.1 will be executed.
The Bluetooth adapter has an input and an output stream to respectively read and write data. Both of which run in a thread so whenever there is new data coming from the SoC, the app will be able to read that. The received messages go to the message handler. In this message handler, various things happen depending on which message was received. Some of the effects of the messages are described in 3.1.1 and here we will explain what happens in the background of the app.
Program 3.14 is a piece of the handler code. First, we set up some byte array variables. One of these will read and copy whatever is in the buffer, while the other is used to send our message to the SoC. The following if-statement asks if the phone is the device that is currently "sending" matrices. If so, a second if-statement awaits with cases for all possible messages the phone can receive from the SoC. As we already know, when the phone receives the message "C", we will be starting to send over the matrices. We define an integer "i" that functions as an index to go through the matrix that contains the messages we have to send. We define a new message

with the first element of the array and we define a first timestamp that indicates the beginning of the whole process. Next up is the "O" message. This message is received whenever the SoC acknowledges that it has received our previous data. We increment our index and check if the index is not exceeding our matrix length. If the index is ok, we send a message with the next array component. When the SoC did not receive our message correctly, we should receive the message "R". If this is the case we will not increment the index so it will just send the previous message again. As for now, we only send one matrix over to the SoC for the sole purpose of testing the transmission time since sending the second matrix would take the same amount of time because they have the same size. Whenever the matrix has been transferred, the SoC will send an "X". At that point we make another timestamp so we know the elapsed time for sending one matrix from the phone to the SoC. The time will be printed in the log with the code shown in program 3.15. Meanwhile, the phone sends a message "C" to the SoC in order to switch the roles. The SoC will become the transmitter and the phone will act as a receiver.

```
 1| byte[] readBuf = (byte[]) msg.obj;
 2| byte[] newMsg;
 3| String readMessage = new String(readBuf, 0, msg.arg1)
 4| if(sending){
 5|   if (readMessage.equals("C")) {
 6|     i = 0;
 7|     newMsg = (String.valueOf(matrixD[i]) + ":").getBytes();
 8|     t1 = System.currentTimeMillis();
 9|   }else if (readMessage.equals("O")) {
10|     i++;
11|     if(i<matrixD.length){
12|       newMsg = (String.valueOf(matrixD[i]) + ":").getBytes();
13|     }
14|   }else if (readMessage.equals("R")) {
15|     newMsg = (String.valueOf(matrixD[i]) + ":").getBytes();
16|   }else if (readMessage.equals("X")) {
17|     t2 = System.currentTimeMillis();
18|     newMsg = "C".getBytes();
19|     printTime();
20|     i = 0;
21|   }
22|   else{
23|     newMsg = ":".getBytes();
24|   }
25| mChatService.write(newMsg);
```

Program 3.14 *Bluetooth message handler as transmitter*

Program 3.16 starts of with the else-statement that is related to the if-statement from program 3.14. Now that the phone is the receiver the first message we would receive from the SoC should

```
public void printTime(){
   Log.i(TAG, "----------------------------------------------------");
   Log.i(TAG, "Time = " + String.valueOf(t2 - t1));
   Log.i(TAG, "----------------------------------------------------");
}
```

Program 3.15 *Prints elapsed time for sending matrices*

be a "D", indicating the SoC is done with the matrix multiplication. We send over message "C" to tell the SoC that the phone is ready to receive data. We also set up a timestamp to indicate the beginning of the receive time. Further messages that will be received will be a set of numbers with a ":"-symbol at the end to indicate that it is the end of the message. In subsection 3.1.1 we explain why we do this. If the received number follows the rules of the protocol we confirm that we received the number correctly by sending an "O" as a message to the SoC. In case we suffered from data loss or any other error, we send an "R". Line 10 in program 3.16 will check wether or not our matrix is full. If it is, we make a second timestamp and print the time it took to receive te matrix.

```
 1| else{
 2|    if(readMessage.equals("D")){
 3|      newMsg = "C".getBytes();
 4|      t1 = System.currentTimeMillis();
 5|    }else if(readMessage.length()==wordSize+1 && readMessage.endsWith(":"))
 6|       try{
 7|        matrixC[counter] = Integer.parseInt(readMessage.substring(0, wordS
 8|        counter++;
 9|       }catch(Exception e){}
10|        if(matrixC[elements-1] != null){
11|           t2 = System.currentTimeMillis();
12|           printTime();
13|        }
14|        newMsg = "O".getBytes();
15|    }
16|    else{
17|      newMsg = "R".getBytes();
18|    }
19|    mChatService.write(newMsg);
20| }
```

Program 3.16 *Bluetooth message handler as receiver*

3.3.3 WebSocket application

Just like in the Bluetooth application, the SoC has to solve the matrix multiplication. Again, the front end of the app is really simple, containing only a textview and a button (figure 3.9). The textview is for defining a matrix size for the rows and columns. With the first tap on the button, you confirm the matrix size and create 2 square matrices with the chosen size containing random values from 0 to 99. The second tap will connect you to the WebSocket whose IP address is predefined as you can see in line 4 of program 3.17. In order for us to connect the phone to the WebSocket server, we need to create a WebSocket client first. The WebSocket client can be created with the code shown in program 3.17. For the WebSocket client, we used the "org.java-websocket:Java-WebSocket:1.3.0" library since it is good and easy to use. Also, to allow the app to use the phone's internet connection, we have to put the following line in the AndroidManifest.xml file:

<uses-permission android:name="android.permission.INTERNET"/>

Now, whenever we call the method to initiate the WebSocket client, the URI from the server is required (line 4). Line 9-50 are shown in program 3.18 that gives the detailed code on which methods the socket has to contain in order to function properly. In line 51 we try to establish a connection between the client and the server.

```
 1| public void connectWebSocket() {
 2|   URI uri;
 3| try {
 4|   uri = new URI("ws://130.230.144.40:8080");
 5| } catch (URISyntaxException e) {
 6|   e.printStackTrace();
 7|   return;
 8| }
 9|   socket = new WebSocketClient(uri, new Draft_17()) {...};
51|   socket.connect();
52| }
```

Program 3.17 *Making a WebSocket client*

The socket requires certain methods that are invoked on a call from the server. For example, if the client connects succesfully with the server, method "OnOpen" is invoked. In this method we print a message to our Log that the phone succesfully connected to the server. The second method required for the WebSocket client is "onMessage". This method is invoked whenever the client receives a message from the server. What happens next depends on the message this method received. All different types of messages are put into an if-statement so we execute different code depending on the received message. You can read more about the messages

on line 16, 19 and 23 in subsection 3.1.2 where the transmission protocol is explained. If we receive "matrixA=" or "MatrixB=", we respectively start transferring matrix A and matrix B. However, before we start sending them over, we create a timestamp t1 that will measure the system time in milliseconds at that point. We do the same thing for t2 as soon as we start sending matrixB. Having a timestamp for matrixB seems redunant but why this is done will be explained in the results. A third timestamp t3 is defined after receiving the message "matrixR=" which will represent the start of receiving data. The message in the next case is to determing how long the SoC's execution time was for calculating the matrix multiplication. When we receive message "done :D", we measure our last timestamp and close the WebSocket connection.

```
 9| socket = new WebSocketClient(uri, new Draft_17()) {
10|   @Override
11|   public void onOpen(ServerHandshake serverHandshake) {
12|     Log.i(TAG, "Connected to: " + uri.getHost() + ":" + uri.getPort());
13|   }
14|   @Override
15|   public void onMessage(String s) {
16|     if(s.equals("matrixA=")){
17|       t1 = System.currentTimeMillis();
18|       sendMatrixA();
19|     }
20|     else if(s.equals("matrixB=")){
21|       t2 = System.currentTimeMillis();
22|       sendMatrixB();
23|     }
24|     else if(s.equals("matrixR=")){
25|       t3 = System.currentTimeMillis();
26|     }
27|     else if(s.contains("Execution time in")){
28|       exeTime=s;
29|     }
30|     else if(s.equals("done :D")){
31|       t4 = System.currentTimeMillis();
32|       displayTimes();
33|     }
34|     else{
35|       try{
36|         int i = Integer.parseInt(s);
37|         matrixC[counter] = i;
38|         counter++;
39|       } catch(Exception e){}
40|     }
41|   }
42|   @Override
43|   public void onClose(int i, String s, boolean b) {
44|     counter = 0;
45|   }
46|   @Override
47|   public void onError(Exception e) {
48|     //Log.i(TAG, "Error " + e.getMessage());
49|   }
50| };
```

Program 3.18 *Detailed look at creating the WebSocket client*

4. RESULTS

Corresponding to the research question of this thesis, two questions needs to be answered. Firstly, is it possible to accelerate the matrix multiplication on the SoC, in order to make it faster than the same multiplication on a single threaded Android application? Section 4.1 describes the outcome during SoC performance testings on three different OpenCL kernels. This section is ended by an interesting fact, discovered during the OpenCL kernel development on the MacBook Pro. Secondly, in section 4.3 a speed comparison between data transfer protocols Bluetooth and WebSocket is described. Besides the comparison of the communication speed only, a total performance comparison is made between: the total data transmission plus calculation time and the calculation time of the Android matrix multiplication. Here the second and main question is answered: is it possible to receive a faster matrix calculation result by extending the matrices to a SoC, instead of performing the calculation inside the Android application?

4.1 OpenCL performance

The implemented OpenCL kernels have been developed and tested on a MacBook Pro, before they were implemented on the DE1SoC. We discovered a huge difference in matrix multiplication calculation time on both architectures, as can be seen in graph 4.1. We will discuss the performance and resource usage of each kernel implementation in the next paragraphs. The implementation of each OpenCL kernel is explained in paragraph 3.2.5. All kernels, except the one copying global to local memory, support an experimentally determined maximum matrix size of 3516x3516. The SoC is theoretically able to allocate maximum 402 653 184 bytes in memory, equation 4.1 shows the calculated amount of memory that must be allocated to store all the data. There are 7 060 992 bytes allocated, but not used to store matrices. These bytes are used by other OpenCL parameters. All the data are consist of two input matrices, one output matrix and one buffer matrix. The last paragraph discusses the general evolution of the calculation times on the three architectures: Android, DE1SoC and MacBook Pro.

$$4 \times 3516^2 \times 8 = 395592192 bytes \tag{4.1}$$

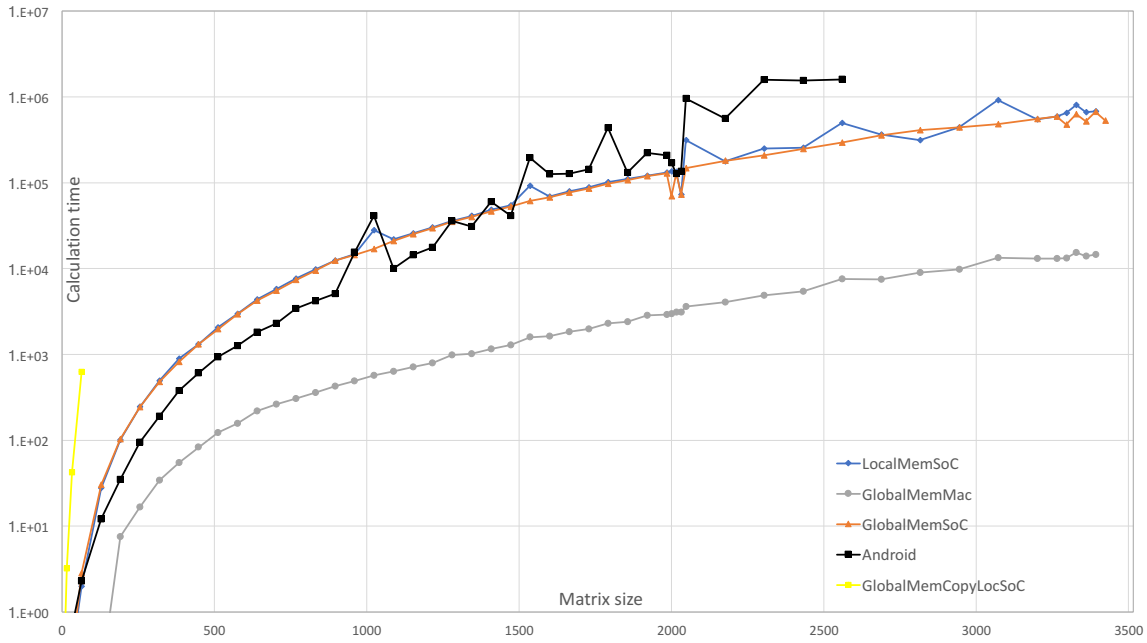


Figure 4.1 *Comparison between OpenCL kernel and Android matrix multiplication calculation times. The vertical axis is in logarithmic scaled.*

4.1.1 Global memory sum storage

The first developed OpenCL kernel uses global memory to store the temporary result of sums. We consider this the most simple and expected it to be the slowest implementation. As can be seen in figure 4.1, the orange line, representing the global memory implementation, looks equally fast as the local memory implementation discussed next. This, however, is not true. Our global memory implementation is slightly faster than the local one, because they are plotted in the same graph it seems that they are equal. Although the global memory seems to have a smooth curve in figure 4.1, strange things happen around matrix size 2048 and above matrix size 3246. We cannot explain what exactly happens in these points, but similar issues seems to happen in both the local memory implementation on the SoC and the Android phone. Matrix sizes 1024, 1536, 2048, 2560 and 3072 are problematical points, with a specific correlation between all points: addition of matrices of size 512x512. Although the amount of elements in 512x512 (=262 144) do not correspond to a commonly used amount of memory, we hypothesize that it is caused by memory allocation.

Table 4.1 reports, at offline compilation time, all the estimated resources used by our global memory implementation in the DE1SoC. We conclude that there are a lot of possibilities to enhance the matrix multiplication, because there still are a lot of resources that remain unused. The next implementations are an attempt to use more resources.

Resource	Usage
Logic utilization	32%
ALUTs	19%
Dedicated logic registers	14%
Memory blocks	28%
DSP blocks	7%

Table 4.1 *Resource usage in global memory implementation*

4.1.2 Local memory sum storage

In theory, the local memory storage is smaller and faster than global memory. When the matrix size increases, then memory is accessed more often. Because the local memory is faster much faster in reading and a little faster in writing than global, it seemed a good idea to use local memory to store a result in between calculations. If we take a look at figure 4.1, we can conclude that there is no significant difference between using global and local memory. We derive this from the fact that reading memory is always faster than writing memory. Because we write "ARRAYSIZE" times and read only one time the local memory per result matrix element, the difference might be negligible. Matrix sizes 1024, 1536, 2048, 2560 and 3072 show the same weird behaviour as mentioned in 4.1.1.

The resource usage in table 4.2 results in less resource usage for the local memory implementation. This is due to the fact that OpenCL requires global memory variables to be declared in the argument list of the kernel function. When looking at the architecture of the SoC, this OpenCL requirement is easily explainable. All global data is stored into the shared DDR3 SDRAM, so every wire of global memory is routed all the way to shared memory. Those wires take a bit extra resourec usage.

Resource	Usage
Logic utilization	28%
ALUTs	16%
Dedicated logic registers	13%
Memory blocks	25%
DSP blocks	7%

Table 4.2 *Resource usage in local memory implementation*

4.1.3 Global to local memory copy

Because we access the two input matrices "ARRAYSIZE" times in order to calculate the result for every element of the result matrix, it seemed a good idea to copy the two input matrices

into local memory, before executing the matrix multiplication. The disadvantage is that it will take a while to write all the data from global into local memory, but the algorithm will receive the data faster during calculation. When we look again at the yellow curve in figure 4.1, we see that this last implementation results in the worst developed kernel. This can be explained by the delay as a result of copying the whole matrices into local memory. Because the local memory is smaller than global memory, the maximum matrix size is limited by 64 x 64. That size is determined by looking at the resource usage. When the matrix size equals 128, table 4.3 shows that 115% of the memory blocks are used. By limiting the matrix size to 64, only 40% of the memory blocks are used in table 4.4.

During the testing of this kernel on the DE1SoC, we discovered that copying data from global to local memory has a big influence on our total calculation time. The yellow line in figure 4.1 represents this kernel. Although this kernel was definitely not a faster solution due to the limited size in local memory, we can conclude that access speed of memory has a big influence in kernel execution time. Future kernels for matrix multiplication should implement global, local and private memory in such a way that global memory access is held to a minimum.

Resource	Usage
Logic utilization	23%
ALUTs	13%
Dedicated logic registers	11%
Memory blocks	115%
DSP blocks	5%

Table 4.3 *Resource usage global to local memory copy implementation for 128 x 128 matrices*

Resource	Usage
Logic utilization	23%
ALUTs	13%
Dedicated logic registers	11%
Memory blocks	40%
DSP blocks	5%

Table 4.4 *Resource usage global to local memory copy implementation for 64 x 64 matrices*

4.1.4 Global memory sum storage on the MacBook Pro

Although all kernels are developed and tested on the MacBook Pro, comparing their performance with all kernels on the DE1SoC is not in the scope of this thesis. We did find an interesting quest. The execution time of the global memory kernel on the MacBook Pro, grey curve in figure 4.1, has an amazingly shorter calculation time compared to the DE1SoC. OpenCL was initially developed to be used on GPUs combined with a host on CPU, not FPGAs. This

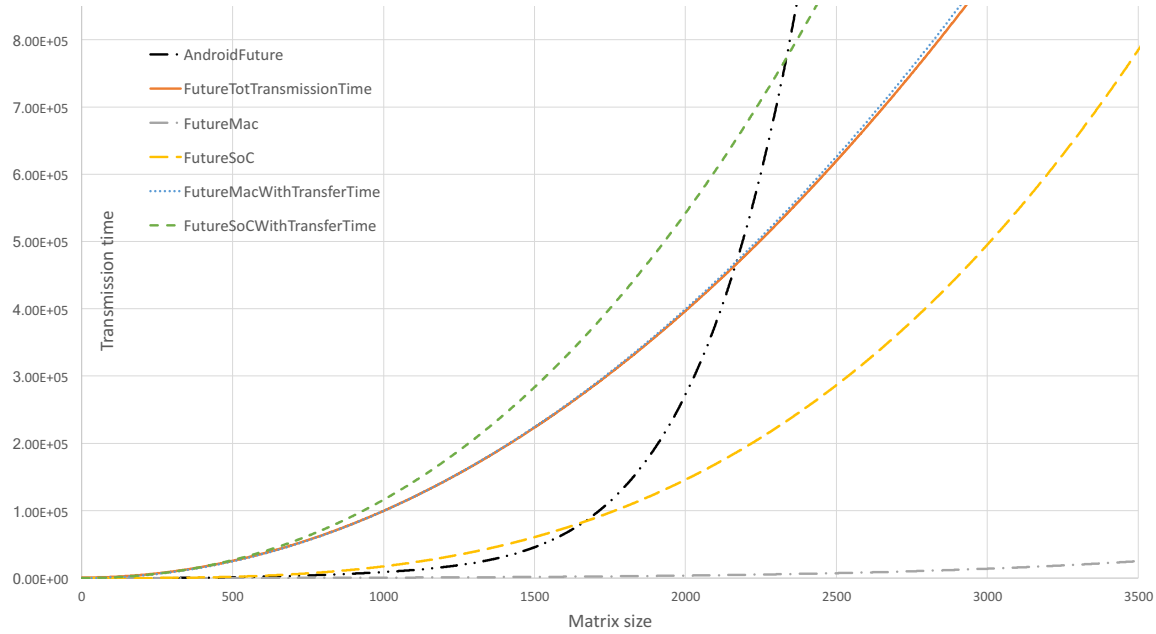


Figure 4.2 *Comparison between matrix multiplications on different architectures, with and without the matrix transportation time*

explains why the MacBook Pro was able to compute a much faster result. We suggest that in further research OpenCL kernels could be extended to other external computing sources like computers with their GPU.

4.1.5 Comparison between matrix multiplications on different architectures in Java and OpenCL

The last curve to discuss in figure 4.1 is the black one, representing the Android device. We talked already about strange behaviour on certain matrix sizes, but in the Android application the calculation time becomes really weird from matrix size 960. Between the already discussed spikes at 1024, 1536, 2048 and 2560, there are a lot of unexplainable spikes. In order to compare results for bigger matrices, a 6th grade polynomial trend line, equation 4.2, was generated by Excel from the black curve in figure 4.2 in between 4 and 896. Equations 4.3 and 4.4 represent both the global memory OpenCL kernel execution on the MacBook Pro and DE1SoC, respectively.

$$y = 1\times10^{-14}\times x^6 - 1\times10^{-11}\times x^5 - 2\times10^{-8}\times x^4 + 4\times10^{-5}\times x^3 - 1,2\times10^{-2}\times x^2 + 1,4\times x - 18,1 \tag{4.2}$$

$$y = 3\times10^{-10}\times x^4 - 1\times10^{-6}\times x^3 + 2,1\times10^{-3}\times x^2 - 7,7\times10^{-1}\times x + 45,6 \tag{4.3}$$

$$y = -3 \times 10^{-10} \times x^4 + 2 \times 10^{-5} \times x^3 - 2,1 \times 10^{-3} \times x^2 - 0,2801 \times x + 142,83 \quad (4.4)$$

All the not yet discussed curves on figure 4.2 are discussed in the next part of this chapter and used in the end conclusion. Until now, we can conclude that calculations on the SoC with a matrix size bigger than 1660 are faster than on the Android phone.

4.2 Phone performance

In this section we will describe the performance and the results we got from our applications from section 3.3. The phone we used is a Huawei P9 (EVA-L09) that has following specifications:

- CPU: HiSilicon Kirin 955 2.52 GHz
- RAM: 2780MB
- Android version: 7.0

4.2.1 Matrix multiplication

The Huawei P9 is quite a powerful device and was able to outperform the SoC when calculating with medium sized matrices. When the app is executed correctly the screen should look like figure 4.3. While testing, we came accros some interesting findings when calculating matrices with the size of 1024. The time skyrocketed when the matrix had an exact size of 1024. One column/row more or less and the app would behave as expected. From there on, the time for certain matrix sizes is inconsistent. Because we work with large matrices, we can hardly determine whether or not the phone is executing everything correctly making the results in figure 4.5 for large matrices unreliable.
Figure 4.4 shows us the CPU and RAM usage over time. In the RAM graph we can clearly see where the matrices are generated. Over time the RAM usage increases and it drops after all matrices in the app are cleared. The CPU usage is around 12-13% throughout the whole process.

4.2.2 Bluetooth app

After connecting the devices, the SoC will request for the transmission of the data. There is not a lot happening in the front end but the transmission time and receive time are displayed in the Logcat like in figure 4.7 when the full process is done. Again we will check out the CPU and RAM usage of the app as we did with the previous app. If we look at figure 4.6 we notice

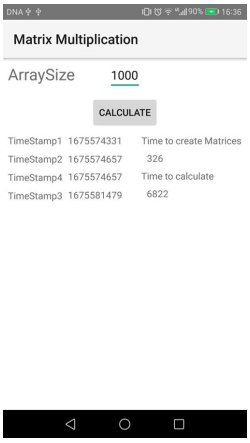


Figure 4.3 *Result screen matrix multiplication*

Figure 4.4 *CPU and RAM usage*

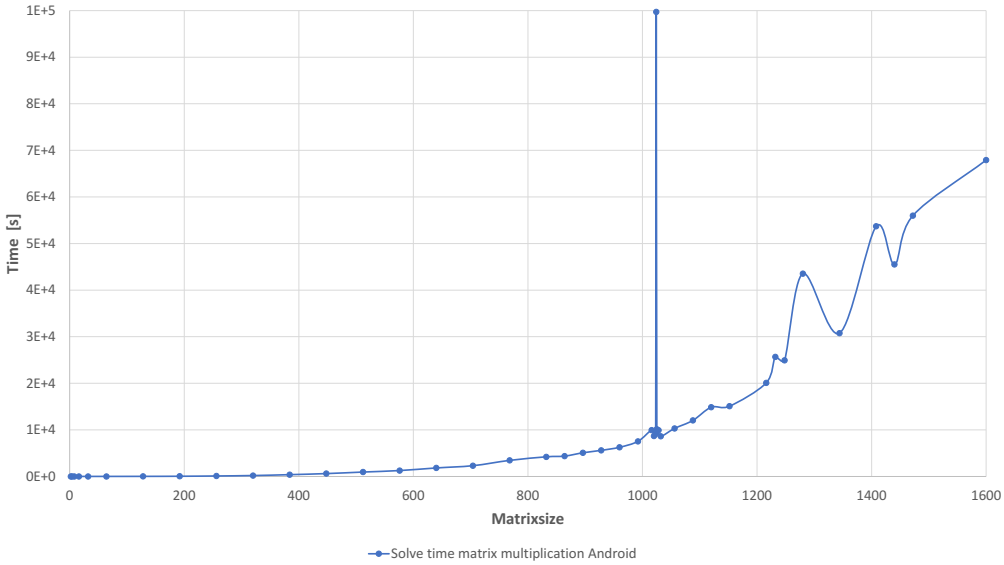


Figure 4.5 *Results for Android matrix multiplication*

a lot of changes. The app barely uses and CPU power at all and only requires a small amount of RAM to create the matrices. As for the Bluetooth app we were only able to test matrices that were smaller than 20x20 as the SoC would give the error: "Too many files open".

4.2.3 WebSocket app

As for the WebSocket, there is a little more interesting data to show. If we look at the RAM usage in figure 4.9, the first increase in RAM is the app startup. The next one is where we create the matrices. Around 8 seconds the matrices are created and the network graph indicates we are transmitting the matrices over to the SoC. The decrease around 18 seconds indicates that we clear the matrices that have been sent. After that we are in an idle state where we send the last few messages that are in the WebSocket queue. When the SoC has executed the

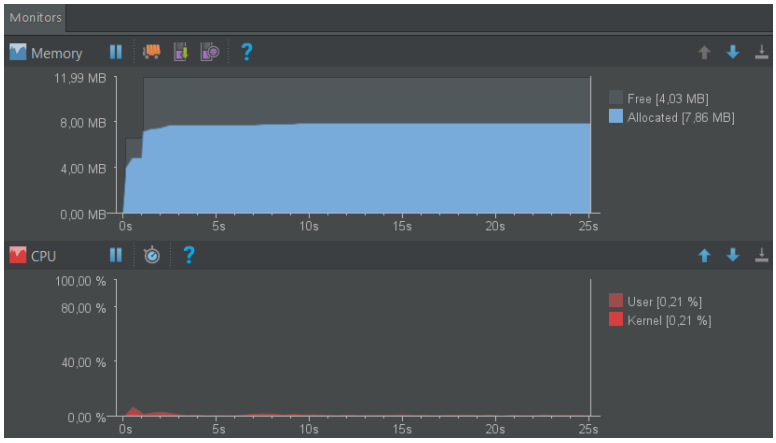


Figure 4.6 *CPU and RAM usage Bluetooth*

```
06-02 04:14:14.641 I/BluetoothChatFragment: ------------------------------------------------------
06-02 04:14:14.645 I/BluetoothChatFragment: Transfer time = 2583
06-02 04:14:14.647 I/BluetoothChatFragment: ------------------------------------------------------
06-02 04:16:24.855 I/BluetoothChatFragment: ------------------------------------------------------
06-02 04:16:24.860 I/BluetoothChatFragment: Receive time = 2693
06-02 04:16:24.863 I/BluetoothChatFragment: ------------------------------------------------------
```

Figure 4.7 *Bluetooth app log*

multiplication it will send the solution back to the phone, hence the receive (Rx) spikes in the network graph. When we look at the CPU, it is mostly used when we receive data from the SoC. This is probably CPU intensive since we slowly fill up the solution matrix with the data that is received from the SoC.
The log file in figure 4.8 shows all time intervals mentioned in 3.3.3.

```
06-01 14:50:58.446 I/ContentValues: Connected to: 130.230.144.40:8080
06-01 14:51:31.756 I/ContentValues: ------------------------------------------------
06-01 14:51:31.756 I/ContentValues: Transfer time =         19894 ms
06-01 14:51:31.756 I/ContentValues: Execution time in milliseconds = 1981.023 ms
06-01 14:51:31.756 I/ContentValues: Receive time =      12370 ms
06-01 14:51:31.756 I/ContentValues: Total time =       32974 ms
06-01 14:51:31.756 I/ContentValues: ------------------------------------------------
```

Figure 4.8 *WebSocket app log*

4.3 Bluetooth vs WebSocket communication speed

In order to choose the best communication protocol, we have to test both Bluetooth and WebSocket communication preferable with large sized arrays. First, we perform the actual communication in order to find a relation between the arraysize and the elapsed time. When we find this relation we can set up a formula that allows us to calculate an estimate value for the elapsed time for any matrix size.

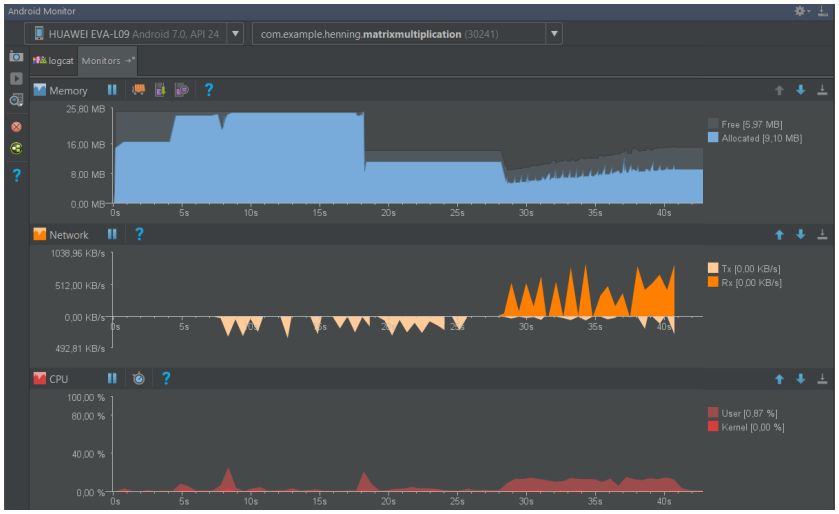


Figure 4.9 *RAM, network and CPU usage WebSocket*

4.3.1 WebSocket transmission

We first performed multiple calculations with WebSocket communication. We separated the elapsed time it took the phone to transmit the data and the elapsed time it took the SoC to transmit the data. If we divide the total number of separate symbols in one matrix with the elapsed time, we know the number of symbols that are transmitted per second (sym/s). For small matrices, the sym/s is not reliable since there are background processes that influence these results. However, these background processes barely affect the sym/s with larger matrices. So in order to determine the relation, we will take an average of all the sym/s with all matrix sizes, apart from those that are highly affected by background processes. This resulted in a sym/s value of 86843,61 for the phone transmission and 127228,7 for the SoC transmission. Knowing these values we can make graph 4.10. In the graph we compare the transmission time for the phone with the time for the SoC. The small dent in the SoC curve is the change in average number length in the matrix from 6 to 7.

4.3.2 Bluetooth transmission

Next we determine the transmission time using the Bluetooth communication. We only mention the practical elapsed time to send over one matrix from the phone to the SoC, as the elapsed time the other way around was very similar. We were only able to test small matrices of up to a size of 20x20. The sym/s for this communication protocol was around 258 sym/s including a lot of repeated messages due to package loss. Although we are not able to transfer large matrices, we are able to set up a formula that allows us to make an estimate of the elapsed time for higher matricex sizes. The results for this are shown in graph 4.11.

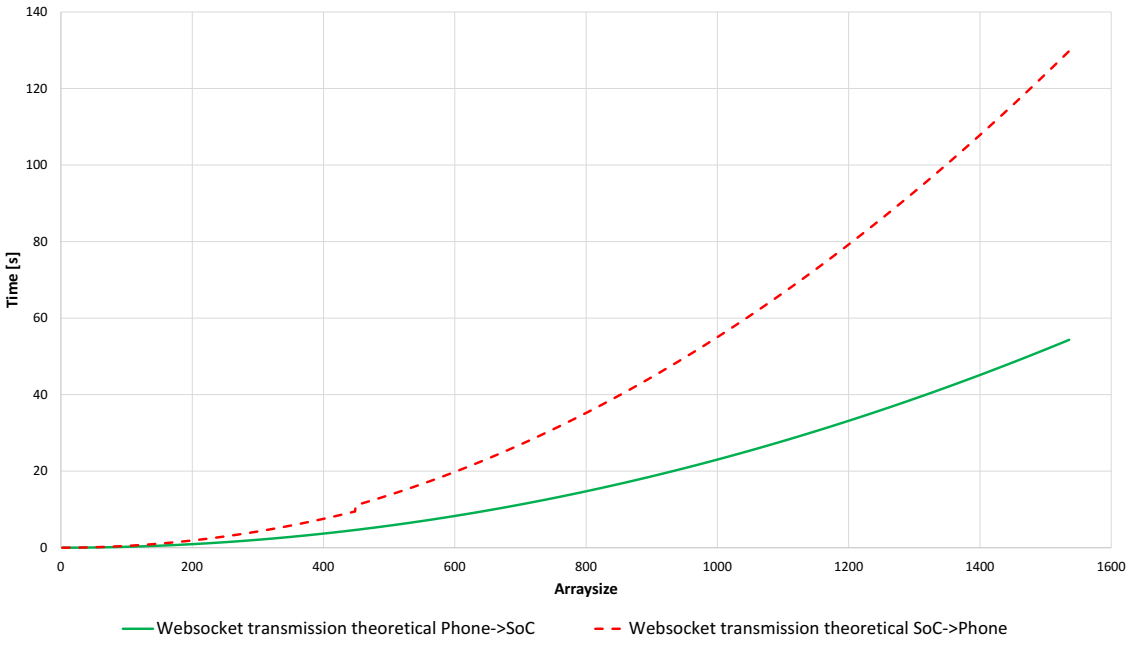


Figure 4.10 *WebSocket transmission time for phone to SoC and vice versa*

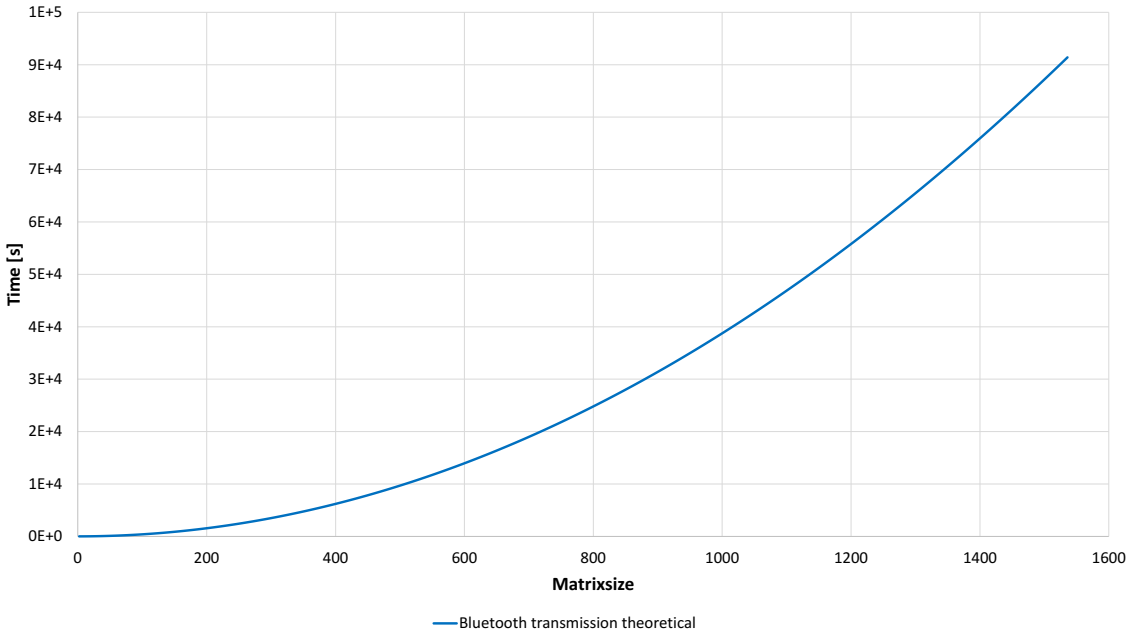


Figure 4.11 *Bluetooth transmission time*

4.3.3 Bluetooth vs. WebSocket

Comparing the sym/s and looking at both graph scales, it is very clear that the WebSocket is way faster than Bluetooth communication. The WebSocket transfer time from the phone to the SoC is around 340 times faster while vice versa the WebSocket is almost 500 times faster. The WebSocket also allowed us to practically transfer much larger matrices, making it superior compared to our Bluetooth communication. Graph 4.12 visualizes the difference in scale for

both communication protocols. We can clearly see that the WebSocket requires less time to transfer one matrix compared to Bluetooth.

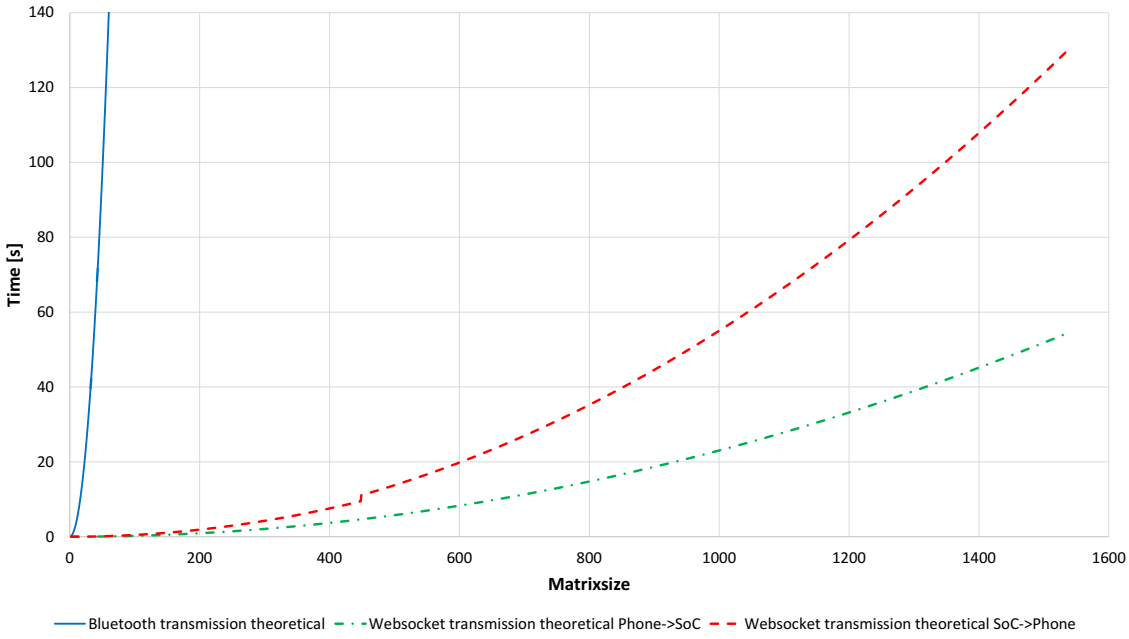


Figure 4.12 *Bluetooth transfer time vs. WebSocket transfer time*

4.3.4 WebSocket and SoC vs. Android smartphone

As discussed in previous subsection, we can conclude that there is no way of Bluetooth being worthwhile for this project. Thus, we are only testing the WebSocket protocol combined with the SoC's ability to multiply matrices. Subsection 4.1.5 gives us a comparison between the phone and the SoC on how fast they can process the matrix multiplication. Now we want to know at which point the Android phone will be slower than transferring all three matrices and performing the matrix multiplication on the SoC. If we look back at graph figure 4.1, the orange curve represents the estimate of the total transfer time of all three matrices. While the yellow and gray curves respectively represent the calculation time it takes to perform the multiplication with the SoC and the Mac. The yellow and gray curves are trendlines derived from data we achieved from performing multiple matrix multiplications with different matrix sizes. The equation for the yellow curve trendline is equation 4.4 and the grey curve's trendline is equation 4.3. Combining these gives us the blue curve which represents the combination of the transfer time and the Mac calculation time, while the green curve shows the combination of the transfer time and the SoC calculation time. If we look at the green curve we see it crosses the black curve, which is the trendline made with equation 4.2 around a matrix size value of 2350. If we want a more correct value for this we take a look at the equation that defines both curves. Equation 4.2 represents the used function to generate the trend line of the Android phone execution time. Equation 4.6 is a combination of the data transfer time from both the

phone and the SoC. The transfer speed from the phone is 43,42 numbers/second while the SoC can transfer the larger numbers back to the phone at a rate of 18,85 numbers/second. In order to find the the value for the matrix size where the phone execution time will be the same as the full process time from the SoC, we have to find x in y=v+w, with y equal to equation 4.2.

$$v = -3 * 10^{-10} * x^4 + 2 * 10^{-5} * x^3 - 0,0021 * x^2 - 0,2801 * x + 142,83 \tag{4.5}$$

$$w = \frac{2 * x^2}{43,42} + \frac{x^2}{18,85} \tag{4.6}$$

If we extract x from y=v+w we get a value of 2338. This means we have an equal time for both processes in case we try to do a matrix multiplication with two 2338x2338 matrices.

4.4 Conclusion of the results

In the first section, we concluded that calculations on the SoC with a matrix size bigger than 1660 are faster than on the Android phone, due to the increasing amount of calculations when the matrices grow.

The previous section gave us a determined answer about the speed difference between Bluetooth and Websocket. Results of experiments showed that Bluetooth is around 500 times slower than WebSocket, because of this we will not do any other test with Bluetooth. The implemented WebSocket resulted in an easy to develop and rugged system. Using the WebSocket, we were able to achieve a faster matrix multiplication above matrix size 2338x2338 on the SoC than the statistical values of the Android phone. Statistics were used on the Android phone in order to create a trustworthy result at bigger matrix sizes.

Further research could investigate the development of more efficient OpenCL kernels or implementing multiple kernels to accelerate other algorithms. Algorithms like the "Mandlebrot fractal", accelerated in the paper of K. Wang and J. Nurmi [33].

5. CONCLUSION

By the use of the theoretical background a Bluetooth and WebSocket communication between Android and SoC could be developed. The used Bluetooth module is the by Altera provided RFS TerASIC daughter card directly connected to the FPGA side of the DE1SoC. AMBA AXI bridges are configured to share Bluetooth data between HPS and FPGA on the DE1SoC. The WebSocket communication is directly programmed on the HPS by the use of the 'WebsocketD' executable. Once all the data is received, it is saved into two one-dimensional arrays representing the matrices. Next, the OpenCL host initiates the OpenCL kernel, in order to accelerate the matrix multiplication on the FPGA-side. Once the kernel result is returned, result data are returned back to the Android phone.

Chapter result responds to the research question, "How to most efficiently distribute a computation intensive calculation on an Android device to external compute units with an Android API?", in two stages. Firstly, is verified if the accelerated multiplication on a FPGA can be faster than on the Android phone. Results proved that matrix multiplications are faster on a FPGA than Android phone, as long as the matrices exceed size 1660x1660. Due to this conclusion, the next question can be verified. Secondly, the total times needed to transfer all matrices for different matrix sizes are observed. Here two interesting facts were discovered. Starting with: matrix sizes bigger than 2338x2338 are faster transferred and calculated on the DE1SoC than on the Android phone. The second interesting discovery is that the MacBook Pro is able to calculate an even faster matrix multiplication than the DE1SoC, with the same OpenCL kernels. This discovery was found due to the long offline OpenCL kernel compilation times, using Xcode speeds up the OpenCL kernel development. The calculation OpenCL kernel matrix multiplication time is way faster on the MacBook Pro than on the DE1SoC. Since the calculation times differ in function of the matrix sizes, it is difficult to determine exactly how much faster the MacBook Pro is compared to the DE1SoC. Lastly, the communication protocols are discussed. Both Bluetooth and Websocket have a constant data transfer speed, but Bluetooth seems to be 500 times slower than WebSocket. The future research discussed in the next paragraph should definitely be implemented using the WebSocket.

Our final conclusion is that it is possible to extend a matrix multiplication to the DE1SoC to speed up calculation times by the use of OpenCL, but it would even be much better to use the MacBook Pro with a GPU. There are two ways future research can be done based on this

thesis. Firstly, a lot of resources remain unused, which could lead to an OpenCL kernel with higher performance. Secondly, this systems performance can be tested with other calculations, algorithms like a "Mandlebrot fractal".

BIBLIOGRAPHY

[1] Bluetoothadapter. [Online]. Available: https://developer.android.com/reference/android/bluetooth/BluetoothAdapter.html

[2] Kronos members. [Online]. Available: https://www.khronos.org/members/list

[3] *MCSE-011: Parallel Computing*. Gullybaba Publishing House, 2008.

[4] Altera, "About compilation." [Online]. Available: http://quartushelp.altera.com/15.0/mergedProjects/comp/comp/comp_view_comp.htm

[5] ——, *Cyclone V SoC Hard Processor System*. [Online]. Available: https://www.altera.com/products/fpga/features/cyv-soc-hps.html

[6] ——, "Cyclone v device datasheet," *altera.com*, 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_51002.pdf

[7] Angstrom, "Angstrom main page." [Online]. Available: http://www.angstrom-distribution.org

[8] B. Barney, "What is parallel computing," *Introduction to Parallel Computing*, 2012.

[9] S. Bhardwaj, S. Arora, and S. Malik, "Research paper on operating system," *IJIRT*, vol. 1, no. 5, pp. 774–783, 2014. [Online]. Available: http://ijirt.org/vol1/paperpublished/IJIRT100384_PAPER.pdf

[10] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, Apr. 1965. [Online]. Available: http://www.monolithic3d.com/uploads/6/0/5/5/6055488/gordon_moore_1965_article.pdf

[11] L. Hardesty, "Concepts familiar from grade-school algebra have broad ramifications in computer science," *MIT news*, 2013. [Online]. Available: http://news.mit.edu/2013/explained-matrices-1206

[12] H. W. Heil and D. M. Harris, *CMOS VLSI Design: A circuit and system perspective.*, H. W. Heil and D. M. Harris, Eds. Pearson, 2013, no. 181-182. [Online]. Available: http://ic.sjtu.edu.cn/ic/dic/wp-content/uploads/sites/10/2013/04/CMOS-VLSI-design.pdf

[13] ——, *CMOS VLSI Design: A circuit and system perspective.*, H. W. Heil and D. M. Harris, Eds. Pearson, 2013, no. 38-59. [Online]. Available: http://ic.sjtu.edu.cn/ic/dic/wp-content/uploads/sites/10/2013/04/CMOS-VLSI-design.pdf

[14] Intel, "The story of the intel 4004." [Online]. Available: http://www.intel.com/content/www/us/en/history/museum-story-of-intel-4004.html

[15] V. James, "99.6 percent of new smartphones run android or ios," *The Verge*, 2017.

[16] J. Jenkov. (2015) Amdahl's law. [Online]. Available: http://tutorials.jenkov.com/java-concurrency/amdahls-law.html

[17] B. Kernighan and D. Ritchie, *The C programming language*, B. Kernighan and D. Ritchie, Eds. Prentice hall, 1978.

[18] J. Lawson. (2012) Websockets, a guide. [Online]. Available: http://buildnewgames.com/websockets/

[19] H. Mao, *Exploring the Arrow SoCKit*, 2013. [Online]. Available: https://zhehaomao.com/blog/fpga/2013/12/27/sockit-3.html

[20] K. McMahon, "The c++ compilation process." [Online]. Available: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

[21] A. Melnikov. (2011) The websocket protocol. [Online]. Available: https://tools.ietf.org/html/rfc6455

[22] A. Moore and R. WIlson, *FPGA for dummies*, A. Moore and R. WIlson, Eds. intel, 2017. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/misc/fpgas_for_dummies_ebook.pdf

[23] Mozilla. (2017) Writing websocket servers. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/Writing_WebSocket_servers

[24] S. M. Patterson, "Q1 2017 smartphone shipments: Samsung rebounds, apple goes sideways, chinese makers roar," 2017.

[25] M. H. Qusay. (2003) Wireless application programming with j2me and bluetooth. [Online]. Available: http://www.oracle.com/technetwork/systems/index-156651.html

[26] T. Saggu, "Bluetooth technology," *S.U.S.CET*.

[27] M. Scarpino, "A gentle introduction to opencl," website, Aug. 2011. [Online]. Available: http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854?pgno=2

[28] W. Stallings. (2001) Introduction to bluetooth. [Online]. Available: http://www.informit.com/articles/article.aspx?p=23760

[29] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[30] TerASIC, *DE1-SoC Getting Started Guide*, 2016. [Online]. Available: http://www.ee.ic.ac.uk/pcheung/teaching/E2_experiment/DE1-SoC_Getting_Started_Guide.pdf

[31] ——, *Quartus Prime Standard Handbook v16.1*, 16th ed., Altera, 2016. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/qts-qps-handbook.pdf

[32] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, J. Son, and S. Miki, *The OpenCL Programming Book*, S. Tagawa, Ed. Fixstars Corporation, 2010. [Online]. Available: https://www.fixstars.com/en/opencl/book/OpenCLProgrammingBook/basic-program-flow/

[33] K. Wang and J. Nurmi, "Using opencl to rapidly prototype fpga designs," *ieee*, 2016.

[34] E. Weisstein, *Matrix multiplication*, MathWorld–A Wolfram. [Online]. Available: http://mathworld.wolfram.com/MatrixMultiplication.html

[35] R. Wilson, "In the beginning," Altera site. [Online]. Available: https://www.altera.com/solutions/technology/system-design/articles/_2013/in-the-beginning.html

[36] M. wolman, "Compilers, assemblers, linkers, loaders: A short course," 1997. [Online]. Available: https://courses.cs.washington.edu/courses/cse378/97au/help/compilation.html