Laboratory SoC Design - Implementation of Nyuzi in Vivado

Johannes Blatnik, Ioannis Daktylidis, Jonas Ferdigg, Daniel Haslauer, Markus Kessler, Edwin Willegger
Institute of Computer Technology
TU Wien, Vienna, Austria
e1325707@student.tuwien.ac.at, e1128193@student.tuwien.ac.at, e1226597@student.tuwien.ac.at,
e1026814@student.tuwien.ac.at, e1225380@student.tuwien.ac.at, e1326324@student.tuwien.ac.at

***Index Terms*—Open Core, Graphics Processing Unit (GPU), General Purpose Graphics Processing Unit (GPGPU), Vivado, ZCU102, Ultrascale, Nyuzi.**

I. Project Goal

The purpose of this project was to identify an open source GPU and implement it on an Field Programmable Gate Array (FPGA). The resulting system should contain one or more GPU cores in conjunction with a general purpose Central Processing Unit (CPU) acting as a host for the graphics processor.
As a proof of the functionality, the final presentation should contain the following points:

1) show three different 3D-models
   a) Pyramid with single color
   b) Jar containing a liquid with two colors
   c) Pinocchio with multiple colors
2) rotate these models directly on the development platform with at least 24 frames per seconds (FPS) and a resolution of 640x480 pixels.
3) switch between the different scenes without a host computer.

II. Development Environment

Vivado Design Suite 2016.4 was chosen as a development environment to support the integration of the design to the FPGA of the universities development boards. Netlist simulation was done using the integrated simulator of Vivado, simulations on RTL level were processed with Intel Modelsim 10.5c. Software was developed and tested in Vivado SDK and Eclipse C/C++ Development Tooling (CDT). These tools were run on Windows 7 and 10, as well as on Ubuntu 16.4, 17.10 and Debian Stretch.
The project for the hardware platform started on the ZedBoard development board, featuring a ZYNQ-7000 device. In the next step, the platform was changed to a Xilinx ZCU102 board featuring an Ultrascale+ FPGA. This board made it possible to achieve a higher clock frequency and use more hardware.
To perform initial tests, an Intel (former Altera) Cyclone IV E FPGA platform was used.
At the end of the project, the development environment was updated to Vivado 2017.4 and the entire design was successfully tested again.

III. Nyuzi

After initial research, Nyuzi was identified as the most suitable solution for the project. The other two candidates were Wireframe 3d and MIAOW. Nyuzi is a multicore GPGPU. It has been started started and is still being extended by Jeff Bush. https://github.com/jbush001/NyuziProcessor contains the source code, documentation, a wiki and an issue tracker.
The Git repository was cloned in October 2017 and not updated with potentially updated sources.

IV. Project team and project organization

The team consisted of six students. The communication view of the project is shown in Figure 1.
The implementation view of the project is shown in Figure 2.

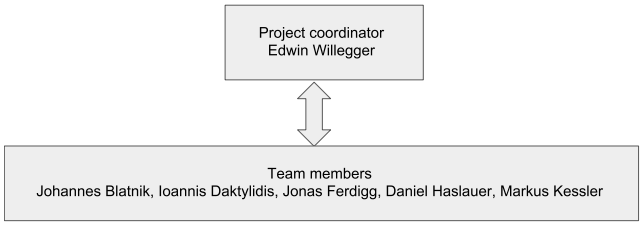

Fig. 1: Communication view

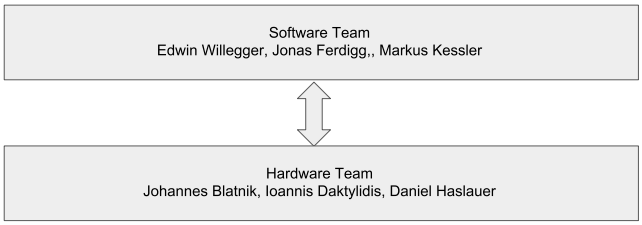

Fig. 2: Implementation view

V. Hardware

This section states an overview of the hardware and its functional components. For this implementation no new Intellectual Propertys (IPs) were developed, but improvements were made on different IP cores, because they were faulty or

did not fit the needs of the project.

A. System Overview

Figure 3 shows an overview of the final implementation. Xilinx differs between the Programming System (PS) and the Programming Logic (PL) on their boards. The ZCU102 development board offers a dedicated Random Access Memory (RAM) slot for the PL area, which will be referred to as PL-RAM in this section. The entire design needed 182.557 Configuarable Logic Blocks (CLBs) and each CLB consists of two slices. The used GPU Nyuzi needed 25.500 CLBs and consumed an area of 2550 A.
The Advanced RISC Machines (ARM) CPU uses the PS-RAM part as main memory. The components that require a higher bandwidth are connected to a 32-bit wide Advanced eXtensible Interface (AXI)-bus. An AXI-lite bus was used for configuration purpose. The main system components are:

- Video Graphics Array (VGA) controller: Reads data frames from the PL-RAM and outputs standard VGA signals to the chips pins. Because the ZCU102 does not offer a VGA port, a Digilent Peripheral Module (Pmod) adapter was used, which has a reduced color depth of 4 bits per channel. https://store.digilentinc.com/pmod-vga-video-graphics-array/
- Nyuzi: 4 cores of the GPU were implemented in this project. Each of them has a dedicated Level 1 (L1)-cache and a shared Level 2 (L2)-cache.
- ARM processor: The on-chip processor manages and controls the GPU. In addition, it is possible to run the conversion from a 3D scene to a binary file on this CPU instead of a host machine.
- PL-RAM: The dedicated RAM for the programmable logic contains the triple-frame-buffer and is used as the main memory for the Nyuzi cores.
- Bootrom: This Read Only Memory (ROM) stores a simple boot sequence for the Nyuzi cores.
- Universal Asynchronous Receiver/Transmitter (UART): Nyuzi UART interface, multiplexed with the ARM processors UART on ZCU102.
- Secure Digital Memory Card (SD)-Card: The SD-Card is only accessible from the ARM core and therefore not shown in this diagram, it stores the 3D-models and the program for Nyuzi.

B. Top Level Components

This section states the top level components, explains their functionality and lists the interfaces.
1) VGA Controller: The VGA controller fetches data from the triple-frame-buffer stored in the PL-RAM and drives the board external Pmod VGA adapter. The controller differs from Jeff Bushs implementation in the following way: it uses an AXI lite interface for configuration purpose instead of the proprietary Input Output (IO)-Bus and it writes white pixels to the VGA Porch areas, to support a higher range of displays. The clock resides in a separate clock domain to make it possible to boost

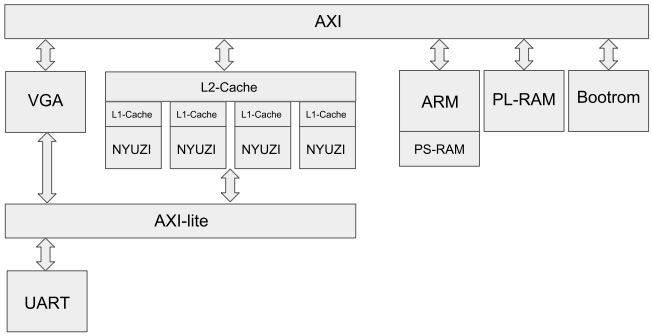

Fig. 3: System Overview

up Nyuzis clock frequency. Clock domain crossing is done using a First In First Out (FIFO)-Buffer.
Table I lists the VGA Controllers top level signals.

TABLE I: VGA top level signals

Signal	Direction	Description
VGA_clk	in	Clock for VGA module, resides in a different clock domain than the main system.
VGA_reset	in	Asynchronous module reset.
VGA_[r,g,b]	out	VGA color channel outputs.
VGA_hs	out	VGA horizontal synchronization signal.
VGA_vs	out	VGA vertical synchronization signal.
interrupt	out	VGA interrupt signal, the module works interrupt driven.
M_AXI*	in/out	AXI master interface to fetch data from frame memory.
S_AXI*	int/out	AXI lite Slave configuration interface.

2) Nyuzi: Please refer to the official website for a very good documentation. The authors did some minor changes and adjustments to the core but did not change the architecture. https://github.com/jbush001
The core was implemented in a 4-core fashion, each core featuring 2 threads, with an extended shared level 2 cache.
The cores IO-bus interface was replaced by an AXI-lite interface to be able to decouple the core clock from the lower speed clocks of the peripheral components.
3) PL RAM: The PL RAM contains the triple frame buffer, 3D model data as well as stack/heap of the Nyuzi cores. The RAM is decoupled from the ARM processor to prevent unintentional memory access of the CPU. Even though the ARM processor can access the PL RAM to store the 3D models via the AXI bus.
Table II lists the memory partitioning.
4) Bootrom: The Bootrom contains a simple boot sequence for the Nyuzi cores which points to the start address of the executable in the PL RAM.
5) ARM Processor: The on chip ARM processor (PS system) reads all the necessary data from a SD-card and loads the 3D-model as well as Nyuzi executables into the PL-RAM. During operation the processor controls the Nyuzi cores. It uses the PS RAM as a main memory and is capable of compiling the

TABLE II: PL RAM memory partitioning

Base Address	Description
0x41fffffe4	Shared Memory (Nyuzi, Zynq) for communication
0x405000000	Ressources (3D-Models)
0x400500000	Nyuzi Heap (unverified information)
0x400458000	Framebuffer 3
0x40032c000	Framebuffer 2
0x400200000	Framebuffer 1
0x400200000	Nyuzi Stack
0x400000000	Nyuzi Program

GPU executables on chip without a host computer. The CPU can access the full AXI address space.

6) UART: The UART interface of the GPU is connected to the AXI-lite bus. To transmit data to the outside world an AXI UARTlite IP core by Xilinx is used (https://www.xilinx.com/support/documentation/ip_documentation/axi_uartlite/v2_0/pg142-axi-uartlite.pdf).
The ARM processors and Nyuzi cores UART interfaces are multiplexed on the ZCU102 board.

C. Vivado Block Design

During project development it was decided to switch to Vivados Block Design approach because it simplifies the configuration of the PL RAM Double Data Rate (DDR)-controller, clock domain crossing and AXI bus configuration/interconnection extensively. Functional components listed in section V-B were implemented as Vivado IP cores to support further system development and easy reuse.

VI. Software

The software development was done for two different architectures. The Software Development Kit (SDK) was used to access the ARM processor of the ZCU102 development board and to upload the software containing the Nyuzi code.
The software development for both systems was done using the cross compilation approach and C/C++ was used as main programming language. The bootloader in the bootrom was written in assembly language. Xilinx glsarm v8 GNU-Toolchain was used within SDK and a specialized LLVM toolchain for the software of Nyuzi. The LLVM toolchain has an unchanged front-end side which allows one to write a C/C++ programs without any other changes. The LLVM back-end side was adapted to fit the Instruction Set Architecture (ISA) of Nyuzi.
The goal of the software development was to use the ARM processor as a CPU and Nyuzi as a GPU.

A. Developed Software for the ARM processor

Four main tasks had to be accomplished with this software:

1) Communication with the SD card. The Xilinx Generic Fat File System Library (XILFFS) was used to implement the SD card interface. The aim of this interface is to get the ability to read the different files from the SD-card with the CPU and write them to the DDR RAM. This was done with different functions within the sdcard.cc and sdcard.hpp files.
2) Controlling of the General Purpose Input/Outputs (GPIOs). Three GPIOs of the ZCU102 board were used. The push buttons were used as a user interface. With the different push buttons the user could rotate the different 3D models or turn left and right and move forward or backward in the Mario Kart environment. The GPIO also uses the interrupt interface. If the user is pressing a push button, an interrupt service routine is called which triggers an appropriate communication to Nyuzi to react to the interaction. This way of communication between the user and a GPU is very common, but there is one big difference: in other designs the CPU sends the right instructions to the GPU. This is a possible issue for further improvements of the design.
   The 8 switches were used to select the different 3D scenes, which could be exchanged, from the SD-card. Currently it is possible to select three different 3D scenes and the fourth scene is the Mario Kart environment. The three scenes are selected by setting switch 8, 7 or 6 to *on* and if you set switch 4 to *on* you select the Mario Kart environment (every other switch needs to be set to *off*). At the moment the software recognizes which scene is selected and different codes are sent to Nyuzi to adapt the controlling mechanism of the 3D scene accordingly.
3) Communication between CPU and GPU. The communication is currently done with a so called shared memory data interface. It is a dedicated address space within the DDR RAM, and both Nyuzi and the CPU has write and read access to this address space. The CPU writes the start address of the Nyuzi program, the size of the program, the start address of the resource file, the size of the resources, the button code and the scene number to the shared memory section. Nyuzi on the other hand sends the current frame rate back to the CPU in the same way. This kind of communication is very near to the communication of modern GPU architectures. In another currently used communication technique between CPU and GPU the CPU sends all instructions for the GPU to a dedicated memory area and the GPU fetches the instructions from there and starts with the computation.
   In that design a FIFO queue is used and the GPU counts the number of executed instructions and the CPU has read access to the number of executed GPU instructions.
   This implementation was deemed too time consuming for the short period of the lab. If we were to implement communication on the instruction level, it means that already during the compilation process of the software of the ARM CPU, the instructions for Nyuzi have to be implemented into the machine code for the CPU. This means that we would have to adapt the Xilinx GNU Toolchain and insert the specific commands for Nyuzi into the ISA. At the moment, the level of abstraction

in the communication between Nyuzi and the CPU includes rotation and movement.

Currently Nyuzi does not communicate to the CPU when it stopped the execution or it reached the end of the program. This could be done with an interrupt triggered at the CPU. The interrupt approach is already used in other GPU architectures.
Another drawback of the current design is, that the GPU is not able to communicate with more than one CPU. This has never been a goal of this laboratory but it is a possible field for further improvement.

B. Developed software for Nyuzi

An existing rendering library was used to process the different 3D models. This library was written in C/C++ and offers important 3D computations running on the Nyuzi hardware. Based on this library, a new C++ class was written to provide a high level Application Programming Interface (API).
The main function of the program is to manage the communication between the software running on the ARM processor and Nyuzi. It also offers the possibility to switch between the different 3D scenes and modify the camera position.

An important step is to flush and/or invalidate the Nyuzi cache after writing to the shared memory interface or before reading from it. Without this, memory changes are only stored locally in the data cache of Nyuzi but are never written back to the DDR RAM or the DDR RAM has changed in the meantime but the cache is never updated as the cache control does not register changes of the DDR RAM. To perform a rotation, it was necessary to start with a rotation angle unequal to zero, otherwise the 3D library would compute a wrong camera viewing angle for $\frac{n\pi}{2}$ and $\frac{3n\pi}{2}$, where n is an integer number. The software receives the starting address and length of the resource file from the shared memory data base. After the 3D scene is loaded, it performs the instructions on the data according to the user input.
At the moment three frame buffers are used to prevent flickering on the screen. This approach guarantees Nyuzi dedicated access to a single frame buffer while the VGA-controller can read from the other two. The *libos* library from the Nyuzi environment has been altered for this approach. An additional function within the library was added so that the VGA controller switches between the three different frame buffers.
The frame rates for the different 3D models were between five and twenty FPS.

VII. How to Run the Example Files

The following steps were tested using Vivado 2017.4.

1) Install Instructions
   a) Download and Install Vivado 2017.4
   b) To install the device drivers open Vivado and go to *Help/Add Design Tools or Devices*. Follow the instruction and install *Engineering Sample Devices for Custom Platforms/Zynq UltraScale+MPSoC ES*.
   c) Install Git
2) Open the Vivado project file in `syn\ZCU102_BD\Vivado\Nyuzi_BD.xpr`
3) Under IP Integrator use Open Block Design to open the Block Design and validate the design (Button F6).
4) Generate the Bitstream.
5) Export the Hardware and include the Bitstream locally to the project.
6) Launch SDK from Vivado.
7) Generate a standalone Board Support Package (BSP) with a 64-bit Compiler which includes XILFFS in SDK.
8) Generate a new Software project in SDK. Select standalone Operating System (OS), C++, 64-bit Compiler and the already created BSP and select empty project.
9) Delete main.cc
10) Import the C++ files from the folder `sw\zcu102\scene_change\src` into your src folder.
11) Build the project and set the Debug options, the SDK project should reset the entire system, the core you are using and it should program the FPGA-board.
12) Now you are ready to copy the different resource files on the SD-Card.
13) The different 3d-examples, used for this demo, are stored in the folder `sw\nyuzi_sw\apps`.
14) Copy the file *scenev.bin* from the folder `scene_viewer\obj` to the root folder of your SD-card. This file is the program which runs on the Nyuzi and renders the 3D scenes.
15) Copy the file *cup.bin* from the folder `LowResCup\obj` to the root folder of you SD-card. Also copy the files *pyram.bin*, *pinoc.bin* and *luigi.bin* from `Pyramide\obj`, `Pinoccio\obj` and `Luigi_Circuit\obj`.
16) Insert your SD-card with the files to the board and start the debugging of the software again in the SDK. Debug the program and it should find the files on the SD-card.
17) Run the program with your debug options, if it does not work at the first try, try a second time, sometimes this works already.
18) You can select the scene with the switches 8 to 5.
19) Push the middle button to start Nyuzi, and then you will see the selected scene on the screen.
20) Rotate, move or turn around with the other four push buttons.

VIII. Hints and Pitfalls

1) Updating IPs in Vivado
   - When selfmade IP cores should get updated in Vivado, it is not sufficient to just change the underlaying HDL code. One must open it in the IP packager view, change the code and re-pack the IP there. Afterwards it is necessary to upgrade the

IP version, since the revision number has changed. Vivado guides you through the second step. (Show IP status, Upgrade Selected)

2) Creating Board Support Package (BSP) for SDK
   - If you create your software project in SDK, you need to include the Generic Fat File System Library (xilffs), to access the SD-card. In Vivado 2016.4 you got an error, if you create the BSP together with the software application project, if you change some parts of the source code. The error tells you that the xilffs library is not included into the BSP, but it is. The best workaround is to create a BSP with the xilffs separately and then reference to it from the application project.
3) Wrong addresses in SDK for GPIO devices for more than a 32bit address
   - If you use Windows 10 and Vivado 2016.4 and you export your hardware description file and bit file and launch the SDK project, your addresses in the file *xparameters.h* are only four byte long (instead of eight). As a result of this truncation error, you can not initialize your GPIO devices correctly. A workaround of this is to change the addresses in the *xparameters.h* file manually to the addresses which you see in the address editor in Vivado 2016.4. This error was solved with Vivado 2017.4 and Windows 10. The *xparameters.h* file is a file in the board support package, and you find it in the include folder of the used CPU.
4) Vivado 2016.4 GUI crashes on Wayland (maybe also later versions)
   - Vivado GUI crashes on Wayland (Ubuntu >= 17.10). Solution: Switch from Wayland to Xorg: https://itsfoss.com/switch-xorg-wayland/
5) ZCU102 USB hub drivers
   - It is necessary to install the drivers of the included USB hub in Windows, otherwise nothing will work and Windows does not complain about anything by itself.
6) Vivado block design automation
   - Do not use the automated Vivado wiring or block automation, unless you know what you want and what you are doing.
7) ZCU102 HDMI IC
   - The Xilinx HDMI IP is needed, which is not free. You cannot drive the Transmitter IC directly without using the FPGA transceivers. A from scratch implementation must configure and use the chips transceivers.
8) Byte-Endianness
   - Nyuzi uses little-endian and the PS system uses big-endian memory access. This has to be considered when reading from or writing to memory sections that are used by both Nyuzi and PS system (like the shared memory interface).
9) Cache coherency
   - Nyuzi uses a multi level cache architecture. This has to be considered when reading from or writing to external memory sections. The assembly instructions to be used for flushing or invalidating cache sections are *dflush* and *dinvalidate* (for data memory) or *iinvalidate* (for instruction memory). https://github.com/jbush001/NyuziProcessor/wiki/Instruction-Set
10) Xilinx simulation libraries for Modelsim on Ubuntu
    - To be able to compile and simulate Xilinx IPs in Modelsim it is necessary to compile the responsible Xilinx libraries. Modelsim Intel starter edition is officially not supported but there is a workaround. This workaround + an installation guide for Modelsim and Vivado in Ubuntu can be found here:
      https://www.lvoudouris.com/installing-xilinx-vivado-2016-4-and-intel-modelsim\-starter-edition-16-1-on-64-bit-ubuntu-16-10-2/
      (also works in Ubuntu 17.10)
      It is not possible to choose the Zynq Ultrascale+ architecture in the Vivado GUI but the following TCL command does the job. Please make sure to use the correct paths in the command:

```
compile_simlib -force -language all \
  -dir {EDIT/modelsim_ase/xilinx\} \
  -simulator modelsim \
  -simulator_exec_path \
  {EDIT/modelsim_ase/bin\} \
  -32bit -verbose -library all \
  -family zynquplus
```

11) OpenCL Tutorials
    - Here are some videos that break down OpenCL programming and GPU programming in general. Not exactly what we need but a good start to get an idea about the concept.

      OpenCL 1.2: High-Level Overview
      https://www.youtube.com/watch?v=8D6yhpiQVVI
      OpenCL GPU Architecture
      https://www.youtube.com/watch?v=1Yr-E9w4tGI
12) Graphics Pipeline Tutorials
    - Design and implementation of a graphics pipeline including choosing and implementing the necessary shader cores, host to device communication and host API functions.

      Some reference videos to get started:
      https://www.youtube.com/watch?v=bgckX62f4EA
      https://www.youtube.com/watch?v=csKrVBWCItc
      https://www.youtube.com/watch?v=0jML5fMBDGg

13) NYUZI Debugging hints from Jeff Bush

- With a hang, I've found the most useful place to look first is at the program counters. They're in `core\ifetch_tag_stage`, called *next_program_counter*. If the whole system is hung, it may be sufficient to look at just one of them. A few ideas:

  a) I've tried synthesizing a small FIFO (maybe 64 or 128 entries) and logging the program counter into it every cycle, then dumping that out via serial (or JTAG) when externally triggered.
  b) Since you have an ARM core that is connected, you might be able to expose the FIFO data to it via memory mapped registers.
  c) If you have a logic analyzer, you could just route those signals to external pins.
  d) I've also tried connecting the program counter to external LEDs, then adding an external switch to freeze the processor clock to inspect them.

  These can suggest a few places to look:

  a) If a program counter is not changing, the thread may be blocked. This can happen for a few reasons:
     i) In *ifetch_tag_stage*, there is a register *icache_wait_threads*. When a bit is set, the thread is waiting for an icache fill.
     ii) If the instruction FIFO in the thread select stage is full, it will deassert *ts_fetch_en*. This can happen if the thread is not issuing because *thread_select_stage.thread_blocked* is set. This is set by a data cache miss, while waiting for memory mapped I/O, or if the store buffer is full.
     iii) This can also happen if the thread is not enabled, indicated by the *thread_en* signal going into *thread_select_stage.sv*. If *exit()* is called (in `software/libs/libos/bare-metal`), this will halt all threads by writing to the *REG_THREAD_HALT* register. This can happen because of an *assert()* failure, for example. I'd expect to see something out of the serial port.
  b) If the program counter is advancing, it can be useful to look at which code it is running. If the build system didn't create an assembly listing file, you can generate one manually with the llvm-objdump command included in the toolchain:

```
$ llvm-nyuzi/bin/llvm-obdump \
    -d elf_file
```