[Date]

– Group 05

FINAL REPORT

Student name	Student ID	Student email
Trinh Gia Huy	H290290	giahuy.trinh@tuni.fi
Phan Vu Thien Quang	H281249	quang.phan@tuni.fi

Overview of the System

In the exercise project, we implemented a system consisting of an ARM-based processor and a FPGA accelerator. The system is capable of video streaming which collects real-time data and output it in the computer program using Kvazaar - a HEVC coding technique. Hence, the FPGA is used to accelerate video encoding and decoding.

Hardware

The project leverages the Cyclone® V SoC Development Board (VEEK-MT-C5SoC). The board employs a dual-core ARM Cortex-A9 MPCore processor up to 925 MHz. In addition, it has up to 13.59 Mb of embedded memory and several buses. It has also on-chip RAM, FPGA-to-HPS SDRAM controller and HPS-FPGA bridges. Those are where FPGA communicates with ARM cores. In addition, there are camera and touchscreen on the backside of the board for video capturing.

Software

The board is controlled via a Putty connection. Here, the driver for Kvazaar (***ip_acc_driver.ko***) is loaded to control the encoder, then the camera driver (***camera_driver.ko***) is uploaded for controlling the camera.

1. Summary of the Exercises

Exercise 1

The main objective of exercise 1 is introducing us to new tools and basic knowledge of system design such as profiling Kvazaar performance with gprof profiling and encoding HEVC video. Kactus2 is used to generate Makefile for Kvazaar compiling, configure different Kvazaar systems (for e.g. PC-system, Veek-system, …), integrate and reuse the Intellectual-Properties (IP) in order to increase the hardware development productivity. At first, we configure the library and prepare some video input for encode. Next step, we compile and run the Kvazaar using pre-made script ***encode_run.sh***. The output video encoding result will be logged into external files for data parse. By analyzing these log files with different quantization values QP and running profiling build to create the ***gmon.out*** output file, we made some conclusions on encoded frames per second (FPS), the effect of decreasing/increasing quantization value, and average PSNR in both profiling and non-profiling build run. In addition, the generated file from profile_image.sh script would provide us with the illustrating profiling image which we could summarize the executing time used within as well as the time used for recursion of each functions in percentage.

Exercise 2

The main purpose of this exercise is to get us familiar with SystemC modeling, simulating and visualizing using GTKWave. At the very beginning, we start checking the functionality of SystemC library. Next, we run and examine the PC-application by dividing it into the inter-communication SystemC process and adding timing to the model. To be precise, we verify the application's functionality using pre-made synchronized testbench by checking the difference between the input and output. The purpose of the application is to receive, permutate, encrypt the input with a key and reverse all these processes to produce the output. Therefore, they must be similar. Compiling the system requires a Makefile from Kactus2 as well. We then implement the decrypting processing (process 3 and 4) based on stub code used in the encrypting procedure (process 1 and 2) as image below.

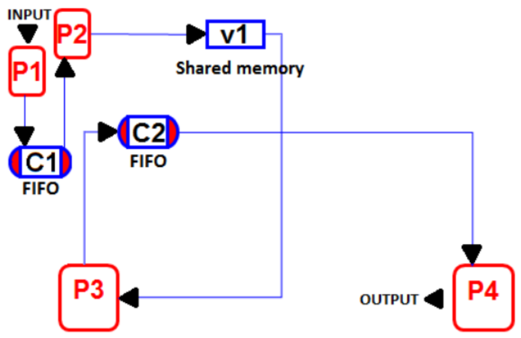


***Figure 1**: The application structure*

Channel 1 and 2 will be modeled as FIFO to implement the communication between process 1 & 2 and process 3 & 4, while data polled for metadata can be accessed via shared memory v1. The GTKWave software simulates the application by generating waveforms. Next task is to optimize the overall application performance by interchanging the CPU between process 1 and 4 as well as the bus interface of all 4 process from I2C to AMBA.

Exercise 3

This exercise comprises 3 parts, which are implementing the inter-process communication of the previous application (CrypterSimApp) using TLM Socket, some system calls (read/write) used as interface between the encoder and hardware accelerator that increases the system performance. The socket transfer between modules at transaction level modeling is more complicated compared to previous exercise which is FIFO. To be precise, in socket transfer, the process 1 would be the master (initiator) while the process 2 acts as the slave (target). Besides, the data passed in this payload must be of native C++ data type and memory reservation should be taken into account.

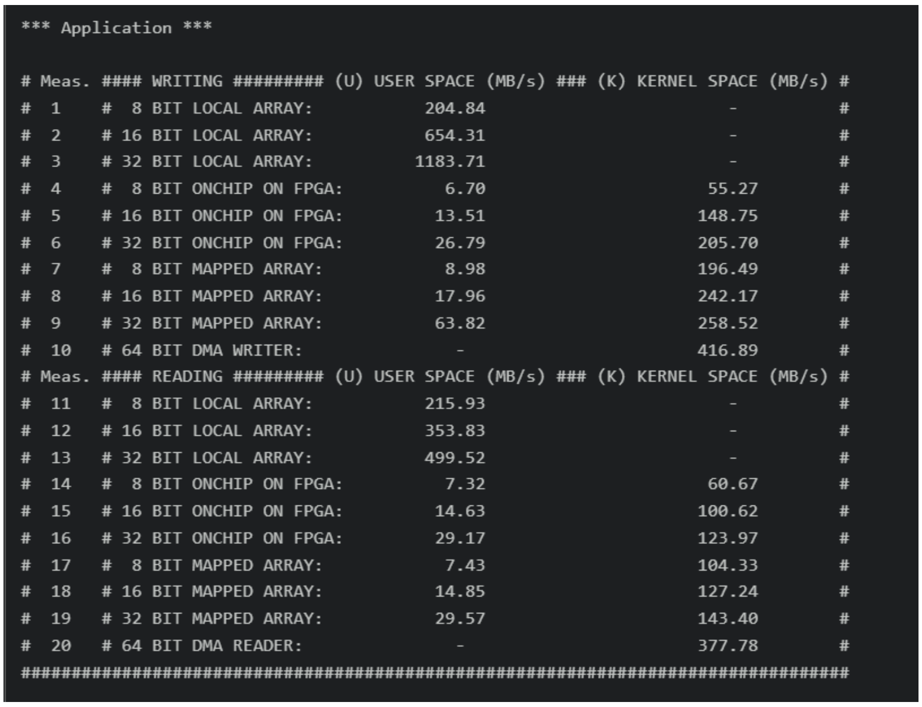
The encoder would run on User space while accelerator calculates the prediction on FPGA. The function read() and write() access the correct selected base addresses which will be used by b_transport() function when transferring data. To handle event, the read() function has to wait for interrupt request from accelerator when data is ready. The average PSNR result that is yielded from running the system using script with group QP value should be the same compared to previous exercise. In addition, the bonus task is to optimize the performance of Kvazaar intra-prediction by dividing the function intra_get_angular() and its SystemC thread for parallelism.

Exercise 4

The main objective of this exercise is to introduce the communication between FPGA and Hardware Processor System (HPS) by controlling LEDs with buttons and getting familiar with Quartus environment (Prime and Qsys). Qsys is the bus design tool that is integrated inside Quartus Prime software which allows the connection to Altera Avalon bus and provides bridges to HPS via AXI bus. To begin with, we add FPGA components into pre-made SoC system design using Qsys tool. In detail, we add 2 parallel input/output components (PIO) from the library for LED and button with their corresponding clock, reset and ports connected to HPS ports. The ***s1*** port (acted as slave) will be connected to ***h2f_lw_axi_master*** of HPS via light-weight AXI bus and base addresses can be set. Next, we generate Qsys and compile the Quartus project into raw binary file (.rbf) which will be uploaded into board. Additionally, we need to implement and test the blinking software on virtual machine and transfer the resulting executable to the board. While executing, the function read_btn() from ***BlinkerApp.c*** will read the button interactivity and set_led() function will write the value into the LED for displaying. Since physical memory cannot be directly accessed by the software, it should be mapped to virtual memory with correct regions. In addition, we use Kactus2 to configure the ARM Cortex, generate and compile Makefile for the system. Since BlinkerApp_0 file will not be executed on VM due to the lack of binary file, we would have to transfer and run the executing file on the board. The LED should be bright when we push either the button or both simultaneously.

Exercise 5

This exercise aims to examine the memory architecture and mapping which includes the DMA and Kernel space measurement using Signal Tap II tool. We start by generating the Makefile using Kactus2 for the pre-made benchmark with configuration for ARM Cortex and compiler gcc-tool-chain. The provided image of FPGA (which is a raw binary file) and application should also be uploaded to the board. Running the executable on the board prints out the benchmarking result which comprises different memory reading & writing speeds. However, some zeros indicate the need for a new kernel driver made for the benchmarking hardware to register. The implemented kernel driver ***benchmark_driver.ko*** can be loaded, thus running the application yields a complete benchmarking result as below.



```
*** Application ***

# Meas. #### WRITING ######### (U) USER SPACE (MB/s) ### (K) KERNEL SPACE (MB/s) #
#  1    #  8 BIT LOCAL ARRAY:          204.84                      -             #
#  2    # 16 BIT LOCAL ARRAY:          654.31                      -             #
#  3    # 32 BIT LOCAL ARRAY:         1183.71                      -             #
#  4    #  8 BIT ONCHIP ON FPGA:         6.70                    55.27           #
#  5    # 16 BIT ONCHIP ON FPGA:        13.51                   148.75           #
#  6    # 32 BIT ONCHIP ON FPGA:        26.79                   205.70           #
#  7    #  8 BIT MAPPED ARRAY:           8.98                   196.49           #
#  8    # 16 BIT MAPPED ARRAY:          17.96                   242.17           #
#  9    # 32 BIT MAPPED ARRAY:          63.82                   258.52           #
#  10   # 64 BIT DMA WRITER:              -                     416.89           #
# Meas. #### READING ######### (U) USER SPACE (MB/s) ### (K) KERNEL SPACE (MB/s) #
#  11   #  8 BIT LOCAL ARRAY:          215.93                      -             #
#  12   # 16 BIT LOCAL ARRAY:          353.83                      -             #
#  13   # 32 BIT LOCAL ARRAY:          499.52                      -             #
#  14   #  8 BIT ONCHIP ON FPGA:         7.32                    60.67           #
#  15   # 16 BIT ONCHIP ON FPGA:        14.63                   100.62           #
#  16   # 32 BIT ONCHIP ON FPGA:        29.17                   123.97           #
#  17   #  8 BIT MAPPED ARRAY:           7.43                   104.33           #
#  18   # 16 BIT MAPPED ARRAY:          14.85                   127.24           #
#  19   # 32 BIT MAPPED ARRAY:          29.57                   143.40           #
#  20   # 64 BIT DMA READER:              -                     377.78           #
##################################################################################
```

Figure 2*: Benchmarking result*

From the figure above, there are two Direct Memory Accesses (DMAs) which are implemented on FPGA and can read (write) from (to) the HPS memory (DDR) directly. We need to measure this DMA reading/writing speed as accurately as possible using SignalTap II for later use. In detail, we measure the number of clock cycles it takes from the moment address read/write starts to be valid until the end of the simulation. At this point, all these measurements could be marked down using Sticky Note Tool in Kactus2. Additionally, the speculative processing power should be taken into consideration by comparing 16-bit multiplication (in MIPS) of CPU and FPGA. We also need to measure accelerated FPS with the assumption that 40% of executing time is on ARM CPU. In the sequel, the pure software FPS can be measured using the relation between ARM MIPS and VM MIPS. Finally, Kvazaar is executed on the board with both profiling and non-profiling Makefile with the transferred group video sequence. This will encode 10 frames with different average PSNR value comparing to previous exercise, which results from the above assumption.

Exercise 6

The objective of this exercise is to explore the parameters to match the performance of Kvazaar on Terasic Cyclone V Veek board, figure out the challenges when designing Kvazaar space with SystemC simulation, and experiment what it takes to encode a 25 fps full HD video. At first, we need to test and find the right value of ***delay_c*** (17000) so that the simulated FPS of untimed model matches the one on ARM. Next, we will run Kvazaar on ARM using timed HW-accelerator with declared variables from previous exercise. In addition, we may add the calculation to function ***b_transport*** in ***sc_kvazaar_ip_sub.c*** source file. Since the gprof will not reveal the execution time (in %) of different parts in the encoder, it is recommended to implement a simple profiler in function ***sc_kvazaar_main*** with EXPLORATION_SW/EXPLORATION_HW defined in ***sc_kvazaar.c*** file. Thus, we can verify its functionality by simulating Kvazaar on ARM and compared with the one added to ***exploration.h***. The Big Simulation task guild us into explore the issue during a simulation by comparing different generated waveform of HW accelerator's signal with and without the delay. By not using wait(), the simulation time is shorter. (60 us vs. 3 ms), and the handshake signals have less time to communicate. At last, we evaluate the code defined the the ***Kvazaar_sim_support*** SW component file sets to find out what it takes to encode the full encode a 25.22 fps full HD video.

Exercise 7

This exercise contains multiple tasks, specifically creating the QSYS component with intra-prediction HW accelerator and camera control. The HW accelerator driver will be used with Kvazaar while camera driver will be loaded with User space application. At the beginning, we start creating and compiling the ***Kvazaar_qsys*** component following the instruction using Qsys tool in Quartus Prime. When the HW is ready, we can export the top-level design, generate Quartus project for synthesis and HTML documentation. When synthesizing the Quartus Project, we have to set our devices' assignments as global. Besides, the timing constraint file needs to be added as well. After that, we run Analysis & Synthesis to verify its functionality using provided Tcl scripts. The resulting raw binary file (.rbf) can be loaded to the board for synthesis. The accelerated version of Kvazaar is generated and compiled as before. The camera driver (***camera_driver.ko***), kernel module (***ip_acc_driver.ko***) along with other files and scripts have to be loaded into the board. To test the HW, we load the kernel driver and run scripts to encode the video. The same thing is done with camera driver. The recorded result is displayed below.

Figure 3: The accelerated output encoded video

Figure 4: The camera encoding output

Exercise 8

In this final exercise, we stream the video from the board and profile the hardware accelerated encoder. First, we start creating profiling build for accelerated Kvazaar and run it similarly to exercise 7, which also yields ***gmon.out***. In VM, we run encode script to create the profiling image. In next task, we start by creating a non-profiling build of accelerated hardware. The derived Kvazaar executable file and the previous encoding video are then loaded to the board for streaming. Also, we create our own shell scripts for streaming and viewing stream on VM with different resolutions and number of threads. Finally, from the screen captures, we could measure the FPS and latency of the system.

2. Suggestions and Feedback

What Did We Learn

We have learned the basic concept of designing system (for i.e: gemerating Makefile, writing driver, measuring the memory accessing speed,…), HEVC technique and the IP reuse. Furthermore, we are familiar with new software and tools such as Quartus Prime, Qsys, SignalTap II and Kactus2. We have learned most of the topics from the course and the course cover almost necessary system design topics.

What was Easy/Hard

Most of the designing tasks with clear instructions are quite easy but time consuming while the writing driver, TLM modeling is hard since it requires us to learn more from book/references pages.

What Needs Improvement

We think the guiding TA sessions should be increased since 2-hour guidance is not enough to ask questions and cover almost groups that need help. If possible, employing more TAs would be perfect.