CE 435
Embedded Systems

Spring 2018

Lab 5

SoC design and optimization

Introduction

FPGA fabric can be used to implement hardware accelerators to offload computationally demanding tasks from the CPU. In this lab you will first run and profile a software application (Sobel filter) on the ARM Cortex-A9 processor and you will profile the application to assess its performance footprint. You will perform software optimizations to improve its execution time.
However, software running on a processor, no matter how well optimized it is, is slower than a hardware accelerator implementing the same functionality. Based on the outcome of SW profiling you will implement the Sobel filter in an accelerator to reduce its execution time using Vivado HLS (High Level Synthesis). Addional optimizations in hardware can provide additional performance gains.
It is important that you are familiar with the Vivado HLS flow. Therefore, you need to do the exercises in the HLS tutorial focusing on the following chapters: ch.1 (tool flow), ch.2 (HLS Introduction), ch. 4 (Interface Synthesis), ch. 6 (Design Analysis), ch. 7 (Design Optimization), ch. 9 (Using HLS IP in IP Integrator) and ch. 10, (Using HLS IP in a Zynq Processor Design, lab1 only).

Sobel filter

Sobel filter is an image processing operator which receives an input image and creates an image emphasizing edges (see Fig. at the bottom of the page). It performs a 2-D spatial gradient measurement on an image and emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image. Sobel filter operator consists of a pair of 3×3 convolution kernels as shown below. One kernel is simply the other rotated by 90°.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

For each pixel in the input image, Sobel filter applies horizontal and vertical convolution using

the two kernels Gx and Gy, respectively.

Step 1 – Sobel in Software

In this phase you will port the Sobel filter and execute it on the ARM processor using the C code and the input image given to you. You may start from the hardware you built in one of the previous labs, export the hardware HDF file and invoke SDK. Then, you will profile the code to measure its performance using *gprof* or ARM counters. You can optimize your software code using *gcc* compilation flags or modifications to the source code.

Note that you need to include the *xilffs* (basically, a Xilinx basic file system) when the BSP (board support package) is being built. You also need to link the math libraries with the *–lm* flag to be able to use *sqrt* function or any math function.

Step 2 – Sobel in Hardware

In this phase, you will try to improve the execution time of the Sobel software implementation by using a hardware accelerator to offload the computation. Use the Vivado HLS toolflow to implement the Sobel filter (function *sobel*) and to integrate it with the ARM core. Note the granularity of acceleration execution: each invocation of the accelerator will apply the Sobel filter to the whole image. The accelerator should use the AXI4 bus to read the input image and write the output image from/to the DDR3 memory.

Even if the hardware will do most of the heavy duty computation in step 2, the ARM core will still execute all tasks related with initalization and validation. It will read the image from a file to an array, will perform configuration and triggering of the hardware accelerator, and will block until the hardware acceletor finishes execution. Then the ARM core will read the output image back from the accelerator and will write it in an output file.

Finally, the software running on ARM core should validate the correctness of the accelerator output. You should compare the output of the hardware accelerator to the correct result using one of two methods: a) use the *golden.grey* file which is the correct output, or b) run the Sobel filter in software (reuse software of step 1) and compare the two arrays. If the two outputs match, the *main* function will return 0 to signal correct hardware execution. If the two outputs do not match, the *main* function will return 1 to signal an error.

Read chapter 10 of the HLS tutorial to understand how the accelerator is mapped to an AXI4 bus. Vivado HLS will also automatically generate template driver files to facilitate communication between ARM CPU and the hardware accelerator. The new IP (hardware+software drivers) should be imported to the Vivado IP repository and instantiated in a Vivado design (chapter 9 provides details).

Note also that the Sobel accelerator is not coherent to the ARM Data cache. When ARM reads the image from the disk and stores it in *array*, that *array* will be in the L1 Data cache of the ARM (since it is resides in a cacheable memory space), but not in the main memory. Since the Sobel accelerator does not have access to the L1 ARM Data cache, the *array* should be written back to the main memory before the invocation of the Sobel accelerator. In other words, you will need to

flush the Data cache to the main memory before invoking the accelerator. Function *Xil_DCacheFlush()* flushes the Data Cache to the main memory. Function *dsb()* is a barrier that freezes execution until the Data cache is written back to the main memory.

Step 3 –Hardware Optimizations

As we have mentioned in class, hardware design gives you a lot of potential to trade-off performance versus area. Step 3 requires that you perform hardware optimizations to reduce execution time. You should detect the performance bottleneck of the first hardware solution (of step 2) and apply a series of optimizations to improve performance. Some ideas for optimizations:

a) Apply HLS directives to increase performance (even at the expense of extra area). You will realize that HLS optimizations are more profitable when the HLS compiler knows exactly the number of loop iterations (i.e. the *height* and *width* of the image). This is because the HLS compiler can safely do loop unrolling and pipeling, two optimizations that tend to be exploit parallelism and improve performance.

b) Avoid conditionals in the source code (if-then-else). Try to replace short conditionals such as the cropping (0,255)

c) Determine which operations are in the critical path and try to circumvent their effect. For example, by running them in parallel or even replacing them with cheaper (even approximate!) operations.

d) Most importantly: compute how many times your first hardware solution (step 2) accesses each pixel of the input image to produce the output image. Redundant memory accesses are a major bottleneck in performance degradation, especially when there is no cache hierarchy to reduce latency (as we noted, the hardware accelerator connects to the main memory without the benefit of a cache). Bandwidth improvement is critical especially if you have already improved parallelism. There is no point in parallelizing computation if you cannot read and write pixels fast enough to always keep the accelerator busy.

   You should re-write your source code to improve bandwidth (directives are not enough). You may consider how to restructure the code to only read the input pixels only once (instead of nine times).

Compare the software and hardware solutions. You should handin a detailed and graphical account of your solutions together with performance and area numbers.