Convolution Co-processor for ZYNQ7000 processing system

Joey De Smet Sam Decorte

Faculty of Engineering Technology, KU Leuven - Bruges Campus
Spoorwegstraat 12, 8200 Bruges, Belgium
{joey.desmet, sam.decorte}@student.kuleuven.be

Abstract

Keywords— Co-Processor, SIMD

I. Introduction

Image processing is everywhere: take programs like Photoshop or Photopea for example. But for real-time applications like video processing or shaders, this becomes a lot more difficult.

If a CPU has to do all this sequentially, it would struggle to get real-time performance. Take this simple example: we have HD video (1920 pixels by 1080 pixels, 60 fps) we want to process. We would have to process $1920 \cdot 1080 \cdot 60 = 124416000$ pixels per second, giving us only 8,04 ns per pixel. Doing this sequentially on a CPU, means that the CPU has a very high workload, or might even be impossible.

Instead, you can offload this work to a co-processor, that will process an image in parallel. This makes the whole system faster, and the CPU has time to do other work.

In image processing, a kernel is a small matrix that can be used to add an effect to an image, by convoluting the kernel over the image. Different kernels achieve different tasks, like blurring, sharpening, embossing... The output of a pixel A does not have any effect on a different pixel B, meaning we can fully parallelize the process.

II. Implementation

A. High level overview

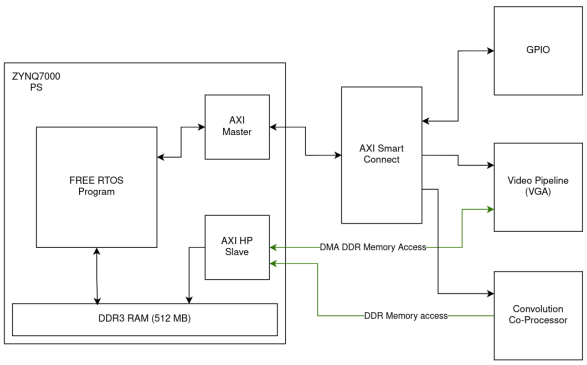

Fig. 1. Overview interconnect architecture

A high level overview is given in Figure 1. The system contains four main parts: the MCU, the co-processor, video output and the AXI bus to connect everything together.

The MCU contains data for the image to be processed, inside DDR3 RAM. It splits the image data into four quadrants, and sends the corresponding pixels for each quadrant to the co-processor via AXI.

The processed image is sent from the co-processor to the MCU via AXI. The MCU then sends it to the video pipeline to display the image.

B. Convolution unit

The convolution unit is the heart of our system. It loads pixels into its buffer, and calculates the result of each pixel when the buffers are full enough. The system is shown in Figure 2.

Some kernels require fractional coefficients. This is implemented as a bit shift at the end of the calculation. This is not a catch-all solution, as only divisions by 2^n can be constructed, but it's much easier to implement and faster than using fixed-point or floating-point arithmetic.

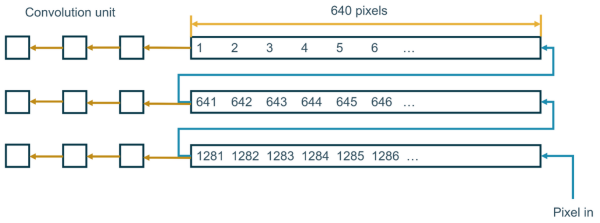

Fig. 2. Convolution unit schematic

The convolution unit contains three buffers, each 640 pixels in length. A pixel is shifted into the lower buffer each clock cycle. When the buffer is full, ejected bits are inserted into the middle buffer. When the middle buffer is full, the pixels are shifted into the upper buffer. This nicely arranges the pixels, allowing the module to ingest pixels from three image rows simultaneously.

To further increase speed, the convolution process is pipelined.

1) Stage 1: Calculate P_1c_1, P_2c_2, P_3c_3 in parallel
2) Stage 2: Calculate $P_1c_1 + P_2c_2$
3) Stage 3: Calculate $P_1c_1 + P_2c_2 + P_3c_3$
4) Stage 4: Calculate $(P_1c_1 + P_2c_2 + P_3c_3) >> s$

The convolution unit RTL module exposes inputs for the kernel coefficients, and the bit shift that is applied after the convolution step. This makes the kernel configurable at runtime.

C. Effect of block size

To calculate the convolution of a block of size n by n pixels, we need the surrounding pixels as well, as shown in Figure 3. To calculate the convolution for the n^2 pixels, we need to fetch $(n+2)^2$ pixels, making the efficiency $\eta = \frac{n^2}{(n+1)^2}$. For low block sizes, there are a lot of pixels that overlap between adjacent blocks, resulting in wasted re-fetching. However, larger block sizes mean less parallelism, resulting in a slower system.

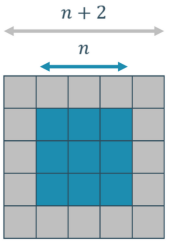

Fig. 3. Pixels to calculate (blue) and apron (grey)

III. Performance Analysis

In this section, we evaluate the performance of the proposed convolution co-processor. Metrics include processing throughput, latency, resource utilization, and energy efficiency. Comparisons are made with a reference CPU-only implementation on the ZYNQ7000 processing system.

A. Experimental Setup

The experiments were performed on a Digilent ZedBoard development board with the following specifications:

- Processing System: Dual-core ARM Cortex-A9, 667 MHz
- FPGA: XC7Z020 (Artix-7), 53k LUTs, 106k FFs, 4.9 Mb BRAM
- Clock frequency of co-processor: 100 MHz
- Test images: resolution 640×480, 32-bit RGBA
- Convolution kernel: 3×3

B. Latency and Throughput

For this performance analysis, we'll compare the co-processor speed to a naïve sequential algorithm on a CPU. This algorithm is shown in Section III-B.

```
for y from 1 to H-2:
  for x from 1 to W-2:
    acc = 0
    # Convolute current pixel with kernel
    for ky from -1 to +1:
      for kx from -1 to +1:
        acc += img[y + ky][x + kx] \
          * K[ky + 1][kx + 1]
    out[y][x] = acc
```

In the naïve implementation, we need 9 multiplications and 9 additions per pixel. Taking one cycle per addition, one cycle per multiplication, and 10 to 15 cycles for the loop logic and memory, we need about 28 to 33 cycles/pixel in the sequential example.

In our parallel implementation, we have an initial three line buffers that need to be filled first (taking 640 cycles each), and an additional three cycles to load the needed data into the convolution unit. After this first delay, we can process pixels at a rate of 1 cycle/pixel.

Processing the image by splitting it into four quadrants, we get an additional speedup of 4x.

When we look at our co-processor architectural implementation, we only need 1 cycle/pixel after the initial latency for filling the buffers and the data going trough the pipeline. So our implementation already has an architectural speedup of 28-33X, when also taking into account that we split the image in four we can multiply this speedup buy four and have a total architectural speedup of 112-132X.

However, the speedup is limited by the speed of AXI.

$$T_{AXI} = 2 \cdot \frac{N_{pixels}}{\text{pixels\_per\_AXI\_cycle} \cdot f_{AXI}} \tag{1}$$

The latency T_{latency} of the co-processor is measured as the time between issuing a convolution request and receiving the processed data:

$$T_{\text{latency}} = T_{\text{transfer}} + T_{\text{compute}} + T_{\text{response}} \tag{2}$$

Throughput $R_{\text{throughput}}$ is calculated as:

$$R_{\text{throughput}} = \frac{\text{Number of pixels processed}}{T_{\text{latency}}} \tag{3}$$

TABLE I. Latency and throughput for processing new versus in-memory images

In memory	Latency [ms]	Throughput [MPix/s]
No	–	–
Yes	–	–

C. Resource Utilization

For a single convolution unit (which consists of three line buffers of 640 pixels depth, and a processor unit), the estimated resource utilization of the synthesized design is summarized in Table II.

TABLE II. FPGA Resource Utilization

Resource	Used (%)
LUTs	4
Flip-Flops	1
BUFG	3
LUTRAM	8

D. Comparison with CPU Implementation

For reference, a CPU-only implementation, as a FreeRTOS task with highest priority, was run on the ARM Cortex-A9 core. Table III summarizes the speed-up achieved:

TABLE III. Speed-Up of FPGA Co-Processor vs CPU

CPU Latency [ms]	FPGA Speed-Up
–	–

E. Energy Efficiency

Energy consumption was measured for the convolution co-processor using onboard power monitoring or external measurement tools. The energy efficiency η is defined as the number of pixels processed per joule of energy consumed:

$$\eta = \frac{\text{Number of pixels processed}}{E_{\text{total}}} \quad \text{[MPixels/J]} \tag{4}$$

where E_{total} is the total energy consumed during the convolution operation.

TABLE IV. Energy efficiency of the co-processor for CPU and FPGA

Platform	Energy [mJ]	Efficiency [MPix/J]
$FPGA$	–	–
CPU	–	–

IV. Conclusion

The coprocessor has the potential to significantly speed up image processing, but is currently limited by the AXI bus. AXI would be less limiting if there was more processing needed per pixel.

V. Future work

- Splitting the data into the different buffers to allow for more parallelism, is now managed by the processor. A hardware implementation could make it possible for data to be streamed in bigger burst which would decrease te delay for data transfer.
- Currently only 3×3 kernels are supported some minor changes could be done to expand this to a $n \times n$ kernel.
- The image is sent to the co-processor via AXI. However, a direct connection to the processors DDR3 RAM would be much faster.

ACKNOWLEDGMENT

The autors used generative AI tools to assist with language refinement, LaTeX table template generation and grammar correction during the preparation of this paper.

REFERENCES

[1] Digilent, *ZedBoard User's Guide*, 2014. Available: https://files.digilent.com/resources/programmable-logic/zedboard/ZedBoard_HW_UG_v2_2.pdf

[2] ARM, *AXI specification*, 2025. Available: https://developer.arm.com/documentation/ihi0022/latest/

[3] AMD, *Zynq 7000 SoC Technical Reference Manual*, 2023. Available: https://docs.amd.com/r/en-US/ug585-zynq-7000-SoC-TRM/Register-ICCIDR-Details