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

Image processing is used everywhere, from image editing to shaders requiring real-time performance. In this paper, an FPGA-based co-processor is presented, allowing $2\times$ image processing speedup compared to a CPU-based algorithm. The results are only theoretical, as the hardware implementation was never fully realized due to time constraints.

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… [5]. The output of a pixel A does not have any effect on a different pixel B, meaning we can fully parallelize the process.

II. Literature study

To implement the design, we first looked into how generic GPU programming is done. In [4], a kernel image processor is implemented in CUDA. In this example, the image is split into blocks and given to a number of GPU cores. Each multiprocessor has access to a 16 kB ultra-fast shared memory, allowing it to store 64×64 pixels, with 4 bytes per pixel. Those 64×64, after its loaded in from global memory, can then be processed in parallel.

In this paper, a hardware version of the algorithm described in [4] is implemented.

III. Implementation

A. High level overview

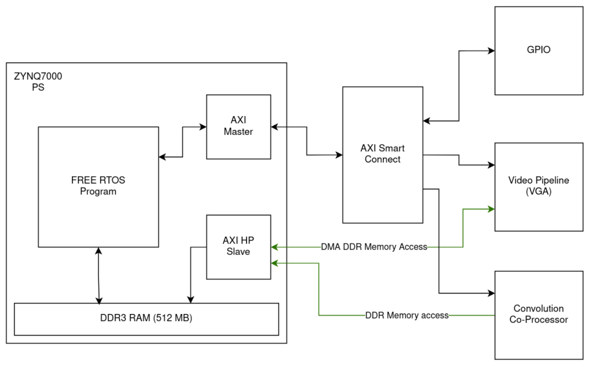

Fig. 1. Overview interconnect architecture

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.

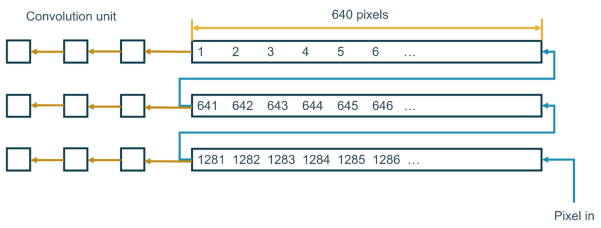

Fig. 2. Convolution unit schematic

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 $\eta = \frac{n^2}{(n+1)^2}$. A plot of this function is shown in Figure 4. 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.

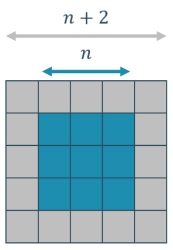

Fig. 3. Pixels to calculate (blue) and apron (grey)

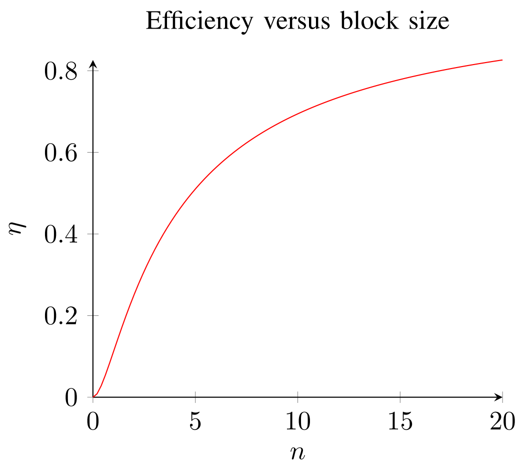

Fig. 4. Efficiency versus block size

D. Video Pipeline

The idea to be able to showcase the co-processor was to display the unprocessed and processed images on a screen with timing information using VGA. This was implemented using existing Vivado AXI periherial IPs shown in figure Figure 5.

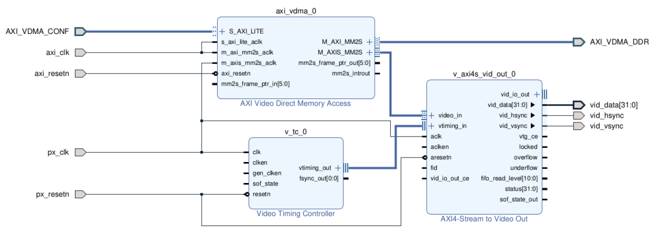

Fig. 5. Video Pipeline Overview

We can see three blocks.

1) *Video Timing Controller*

To generate a 640x480p video signal at 60 Hz, a pixel clock of 25.175 MHz is required according to the VGA timing specification. This clock determines the rate at which individual pixels are transmitted. Each clock cycle corresponds to one pixel period, including both the active video region and the blanking intervals.

Following the specifications the following parameters where set in the controller seen in Table I.

TABLE I. VGA 640×480 @ 60 Hz Timing Parameters

Parameter	Horizontal (pixels)	Vertical (lines)
Active video area	640	480
Front porch	16	10
Sync pulse width	96	2
Back porch	48	33
Total	800	525

In practice the pixel clock of 25.175 MHz is not achivable exact but a clock of 25 MHz was used and gives a small deviation in refresh rate. But this is all within the tolerance of standard VGA monitors.

2) *AXI Video Direct Memory Access*

To reduce the processing load on the CPU, an AXI Video Direct Memory Access (VDMA) controller is used to transfer video data directly between the DDR3 memory and the video pipeline. By offloading continuous frame transfers to dedicated hardware, the CPU remains available for higher-level control tasks.

The VDMA was configured to operate in memory-to-stream mode using three frame buffers. This multi-buffering scheme allows one frame to be displayed while another is being prepared or updated, reducing the risk of visual artifacts such as tearing. The controller is configured via software by providing the base addresses of the frame buffers loacated in DDR3 memory.

Once initialized, the VDMA autonomously handles the read transactions on the AXI bus and streams pixels data into the AXI-stream to Video out controller.

3) *AXI-Stream To Video Out*

The AXI-Stream To Video Out controller takes in the Video Timing signals and the stream from the VDMA Controller and synchronises these to output the correct VGA timing signals and pixels.

IV. 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

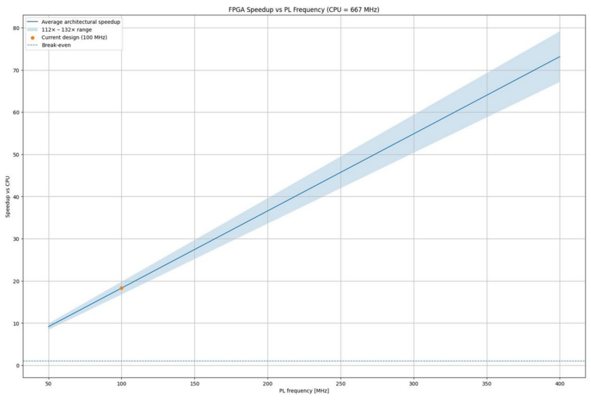

Fig. 7. Theoretical architectural speedup

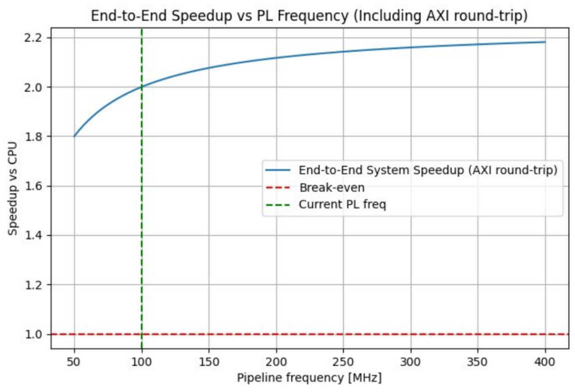

Fig. 8. Plot of Equation (1); speedup is limited by AXI

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 Figure 6.

```
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
```

Fig. 6. Implementation of the sequential convolution algorithm

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.

This speedup would theoretically scale linearly with the clock speed, as shown in Figure 7.

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

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

In Figure 8, it's possible to see that AXI limits the system to around $2.2\times$ speedup, even with very high clock speeds.

If this were a real GPU-like system, we would expect a needed data rate of $1920 \cdot 1080$ pixels/frame $\cdot$ 32 bits/pixel $\cdot$ 60 frames/second $= 497.7$ MB/s (!).

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

$$\begin{aligned} T_{\text{latency}} &= 640 \cdot 3 + 3 \\ &= 1923 \text{ clock cycles} \end{aligned}$$

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

Additionally, the convolution unit has a worst negative slack of 4,363 ns, well within the constraints for a 100 MHz FPGA clock speed.

D. Comparison with CPU Implementation

For reference, we would've liked to run a CPU-only implementation, but due to time constraints this hasn't happened. Instead, this was done theoretically (see Figure 6). There, we assumed 10 to 15 cycles needed for memory access and the loop logic. A summary of these findings is shown in Table III.

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

	Latency (cycles)	Throughput (max, cycles/pixel)
CPU	15	33
FPGA	1923	1

For the sequential algorithm, processing a 640×480 pixel image, we would expect it to take $15 + 33 \cdot 640 \cdot 480 = 10137615$ cycles. Compared to the FPGA, where the image is split into four blocks that's processed in parallel, meaning we would expect $1923 + 1 \cdot 640 \cdot 480 \cdot \frac{1}{4} = 78723$ cycles, which is still significantly faster ($128.8\times$). Off course, the clock speed of these implementations will be different, resulting in a slightly different outcome.

The clock speed of the CPU is set at 667 MHz, while the FPGA is at 100 MHz. Still, the theoretical speedup of the parallel algorithm significantly outweighs this difference.

V. 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 to do on the image data (for example, in the context of shaders). This could all be improved by increasing data width and speed of the AXI bus, but beacause of lack of time and no working implementation, no further investigation was done for this.

VI. 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

[4] NVIDIA, *Image Processing & Video Algorithms with CUDA*, 2008, Available: https://www.nvidia.com/content/nvision2008/tech_presentations/game_developer_track/nvision08-image_processing_and_video_with_cuda.pdf

[5] Wikipedia, *Kernel (image processing)*, 2025, Available: https://en.wikipedia.org/wiki/Kernel_(image_processing)