Linear Image Filtering using HLS

Reconfigurable Computing Project

Cristian Cristea

Faculty of Electronics, Telecommunications and Information Technology
National University of Science and Technology POLITEHNICA Bucharest

June 2024

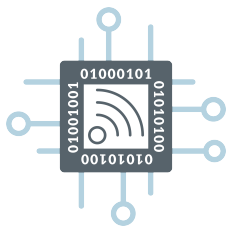

Table of Contents

General Information

Linear Image Filtering (Convolution)

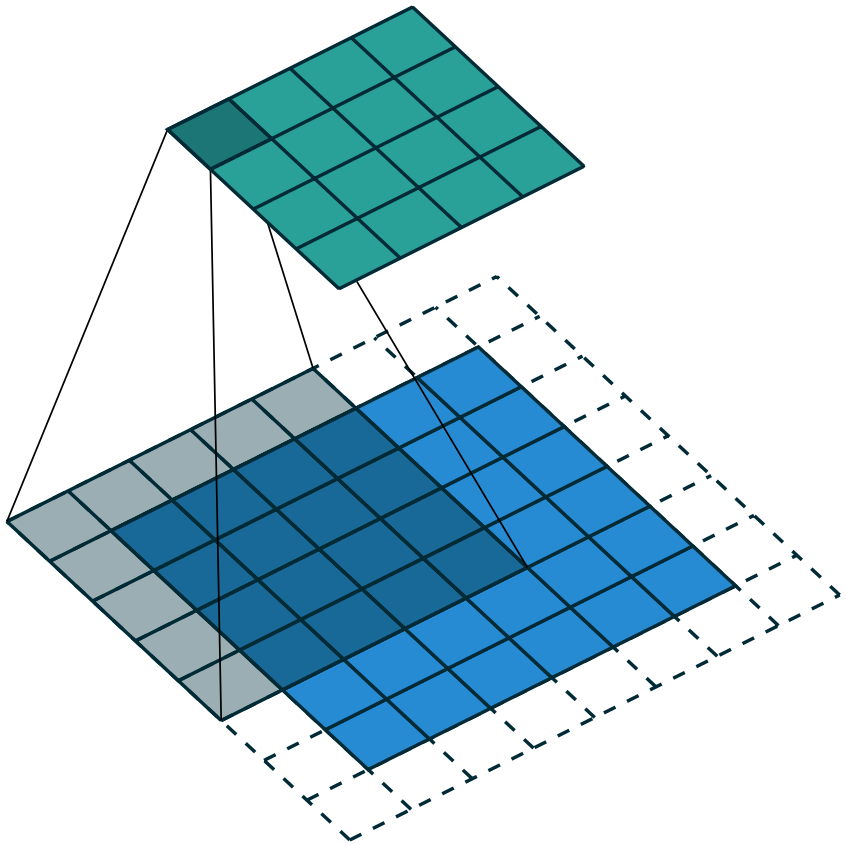

Figure 1: Convolution with a 5×5 kernel (filter)

Padding Methods

- **Zeros** – Adds rows and columns of zeros around the input image, preserving the spatial dimensions during convolution.
- **Edge** – Pads the input with the edge values of the image, replicating the border elements.
- **Reflect** – Extends the input by reflecting the values at the boundaries, creating a mirror effect.

Mean Filtering

$$\boldsymbol{K} := \frac{1}{n^2} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix}_{n \times n} , n \bmod 2 \equiv 1.$$

Mean Filtering

(a) Original black and white Lena picture

(b) Filtered with a 11×11 mean kernel

Figure 2: Input and output of a convolution

Progress

Optimisation Changelog

Optimisation Changelog

- **Version 1** – Initial implementation of the project by transforming the reference C++ code into a Vitis project.

Optimisation Changelog

- **Version 1** – Initial implementation of the project by transforming the reference C++ code into a Vitis project.
- **Version 2** – Implement caching for the input image and the kernel matrix in BRAM memory to reduce the memory access time.

Optimisation Changelog

- **Version 1** – Initial implementation of the project by transforming the reference C++ code into a Vitis project.
- **Version 2** – Implement caching for the input image and the kernel matrix in BRAM memory to reduce the memory access time.
- **Version 3** – Unroll the columns loop in the image processing loops to increase the parallelism of the algorithm.

Optimisation Changelog

- **Version 1** – Initial implementation of the project by transforming the reference C++ code into a Vitis project.
- **Version 2** – Implement caching for the input image and the kernel matrix in BRAM memory to reduce the memory access time.
- **Version 3** – Unroll the columns loop in the image processing loops to increase the parallelism of the algorithm.
- **Version 4** – Change the unroll to the rows loop in the image processing loops.

Optimisation Changelog

- **Version 1** – Initial implementation of the project by transforming the reference C++ code into a Vitis project.
- **Version 2** – Implement caching for the input image and the kernel matrix in BRAM memory to reduce the memory access time.
- **Version 3** – Unroll the columns loop in the image processing loops to increase the parallelism of the algorithm.
- **Version 4** – Change the unroll to the rows loop in the image processing loops.
- **Version 5** – Change the input image cache to have shorter lines and more of them, reduce the unroll factor, and pipeline the padding function instead of inlining it.

Optimisation Changelog

- **Version 1** – Initial implementation of the project by transforming the reference C++ code into a Vitis project.
- **Version 2** – Implement caching for the input image and the kernel matrix in BRAM memory to reduce the memory access time.
- **Version 3** – Unroll the columns loop in the image processing loops to increase the parallelism of the algorithm.
- **Version 4** – Change the unroll to the rows loop in the image processing loops.
- **Version 5** – Change the input image cache to have shorter lines and more of them, reduce the unroll factor, and pipeline the padding function instead of inlining it.
- **Version 6** – Change the data type from `float` to `uint8_t`, inline the padding function, remove the loop unrolling, and reduce the number of cache lines.

Results

Results

Implementation	**3 × 3**	**5 × 5**	**7 × 7**	**9 × 9**	**11 × 11**
Reference on PC[1]	491 ms	1 083 ms	1 974 ms	3 154 ms	4 594 ms
Reference on Laptop[2]	376 ms	696 ms	1 793 ms	2 919 ms	4 050 ms
Reference on RPI 5[3]	380 ms	1 013 ms	1 978 ms	3 146 ms	4 993 ms
Reference on PYNQ-Z2[4]	7 744 ms	19 930 ms	39 428 ms	63 288 ms	93 335 ms
Version 1 on PYNQ-Z2	29 158 ms	61 238 ms	109 352 ms	173 334 ms	252 413 ms
Version 2 on PYNQ-Z2	14 131 ms	22 194 ms	34 288 ms	50 413 ms	70 568 ms
Version 3 on PYNQ-Z2	13 830 ms	21 892 ms	33 987 ms	50 112 ms	70 266 ms
Version 4 on PYNQ-Z2	14 132 ms	22 195 ms	34 288 ms	50 412 ms	70 568 ms
Version 5 on PYNQ-Z2	14 204 ms	22 266 ms	34 359 ms	50 859 ms	71 398 ms
Version 6 on PYNQ-Z2	10 693 ms	12 450 ms	15 019 ms	18 378 ms	22 550 ms

Table 1: Results comparison for different kernel sizes

[1] Intel® Core™ i5-6600K (4 Cores, 4 Threads, 3.5 GHz - 3.9 GHz, 6 MB Cache)
[2] Intel® Core™ i7-9750H (6 Cores, 12 Threads, 2.6 GHz - 4.5 GHz, 12 MB Cache)
[3] Broadcom BCM2712 - ARM® Cortex-A76 (4 Cores, 4 Threads, 2.4 GHz)
[4] Xilinx Zynq-7000 - ARM® Cortex-A9 (2 Cores, 2 Threads, 0.65 GHz)

Conclusions

Conclusions

- The FPGA High-Level Synthesis (HLS) designs exhibited subpar performance compared to implementations executed on a PC, Laptop, and Raspberry Pi 5.

Conclusions

- The FPGA High-Level Synthesis (HLS) designs exhibited subpar performance compared to implementations executed on a PC, Laptop, and Raspberry Pi 5.
- Implementing caching within the FPGA HLS designs significantly improved performance by reducing memory access times.

Conclusions

- The FPGA High-Level Synthesis (HLS) designs exhibited subpar performance compared to implementations executed on a PC, Laptop, and Raspberry Pi 5.
- Implementing caching within the FPGA HLS designs significantly improved performance by reducing memory access times.
- Optimisation techniques such as loop unrolling and pipelining did not decrease execution times.

Conclusions

- The FPGA High-Level Synthesis (HLS) designs exhibited subpar performance compared to implementations executed on a PC, Laptop, and Raspberry Pi 5.
- Implementing caching within the FPGA HLS designs significantly improved performance by reducing memory access times.
- Optimisation techniques such as loop unrolling and pipelining did not decrease execution times.
- A substantial increase in performance was achieved by changing the data type from floating point to integer-based.

Conclusions

- The FPGA High-Level Synthesis (HLS) designs exhibited subpar performance compared to implementations executed on a PC, Laptop, and Raspberry Pi 5.
- Implementing caching within the FPGA HLS designs significantly improved performance by reducing memory access times.
- Optimisation techniques such as loop unrolling and pipelining did not decrease execution times.
- A substantial increase in performance was achieved by changing the data type from floating point to integer-based.
- Among the FPGA HLS designs, all except the one without caching outperformed the reference implementation running on the ARM core of the PYNQ-Z2 when large kernel sizes were employed.

Conclusions

- The FPGA High-Level Synthesis (HLS) designs exhibited subpar performance compared to implementations executed on a PC, Laptop, and Raspberry Pi 5.
- Implementing caching within the FPGA HLS designs significantly improved performance by reducing memory access times.
- Optimisation techniques such as loop unrolling and pipelining did not decrease execution times.
- A substantial increase in performance was achieved by changing the data type from floating point to integer-based.
- Among the FPGA HLS designs, all except the one without caching outperformed the reference implementation running on the ARM core of the PYNQ-Z2 when large kernel sizes were employed.
- Further analysis is needed to evaluate performance in the context of power consumption.

Thank you for your attention! Questions?