# Design and implementation of an image processing heterogeneous digital system

**Laboratory Project Report**

**ASH 2024/2025** **Class: 1MEEC_T01**

**António Lopes 202006725, Francisco Vilarinho 202005500**

## 1 Introduction

This project focuses on designing a hardware-software heterogeneous system for image processing on the ZedBoard development platform. The primary goal is to optimize a C-based image processing application by leveraging the Zynq platform's capabilities to integrate custom hardware accelerators. The project emphasizes achieving performance improvements while adhering to the logic resource constraints of the Zynq device.

The software application in this project is a C-based image processing program designed to handle 8-bit grayscale images and convert them into binary outputs. The program performs a sequence of image processing operations, including blurring, brightness adjustment, contrast enhancement, edge detection, binarization, and erosion. The input image is provided as a text file, processed by the software, and then outputted as a binary text file.

Programs like this often encounter performance bottlenecks in compute-intensive operations, particularly in convolution-based tasks such as blurring. These stages involve repetitive calculations over large datasets, which makes them resource-intensive and time-consuming when executed on general-purpose processors. Memory access overheads and lack of parallelism further decreases inefficiencies.

To address these issues, profiling tools were used to analyze the performance of the application. The workflow involved the use of Xilinx's Vivado, Vitis and Vitis HLS, providing a comprehensive suite for hardware-software co-design. Vivado was used for hardware implementation and optimization, Vitis enabled initial profiling and integration and Vitis HLS facilitated the high-level synthesis of critical functions.

## 2 Development Phase

The development phase of this project involved a detailed analysis of the software application, identifying performance bottlenecks, and implementing targeted optimizations using hardware acceleration. Each step of this process contributed to achieving an optimized design that met performance goals while ensuring compliance with the system constraints.

### 2.1 Profiling on the ZedBoard and Target Function Selection

Profiling was conducted to evaluate the execution times of various functions within the software application and identify the potential of hardware acceleration. By assessing the execution time of the different functions, insights on which functions are limiting the application's performance are provided.

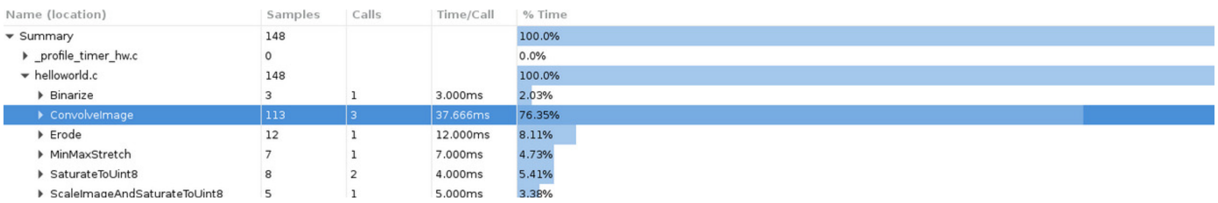
The profiling was configured with gprof in Vitis to collect detailed runtime statistics. The process began by creating a basic hardware platform, consisting only of the Zynq processing system, in Vivado and exporting it to Vitis, where profiling resources were enabled by setting the `enable_sw_intrusive_profiling` flag and adding the `-pg` compiler option. After configuring the profiling frequency for accurate sampling, the application was executed on the FPGA, and profiling data was visualized.

The profiling results for the application under different optimization levels `No Optimization`, `-O1`, `-O2`, and `-O3` are summarized in Table 1. Each optimization level provided insights into the performance of critical functions.

Table 1: Profiling Results for Different Optimization Levels

| **Function** | **Optimization Level** | **Time/Call (ms)** | **% Time** |
|---|---|---|---|
| `ConvolveImage` | `No Optimization` | 290.000 | 83.25% |
| | `-O1` | 37.666 | 76.35% |
| | `-O2` | 39.666 | 81.51% |
| | `-O3` | 40.333 | 81.76% |

The `ConvolveImage` function was selected as the target for hardware acceleration based on two primary factors. Firstly, it consistently consumed the majority of the runtime across all optimization levels, Figure 1 shows the profiling of the best optimization, indicating it as the most computationally intensive operation. Secondly, a detailed analysis of the function's code revealed opportunities for improvements through parallelism and memory access optimization. By implementing this function in hardware, significant performance gains were expected, reducing the overall application runtime.



```
Name (location)                     Samples  Calls  Time/Call  % Time
▾ Summary                           148                        100.0%
  ▸ _profile_timer_hw.c             0                          0.0%
  ▾ helloworld.c                    148                        100.0%
    ▸ Binarize                      3        1      3.000ms    2.03%
    ▸ ConvolveImage                 113      3      37.666ms   76.35%
    ▸ Erode                         12       1      12.000ms   8.11%
    ▸ MinMaxStretch                 7        1      7.000ms    4.73%
    ▸ SaturateToUint8               8        2      4.000ms    5.41%
    ▸ ScaleImageAndSaturateToUint8  5        1      5.000ms    3.38%
```

Figure 1: Best Profiling Optimization

## 2.2 High-Level Synthesis and Results Analysis

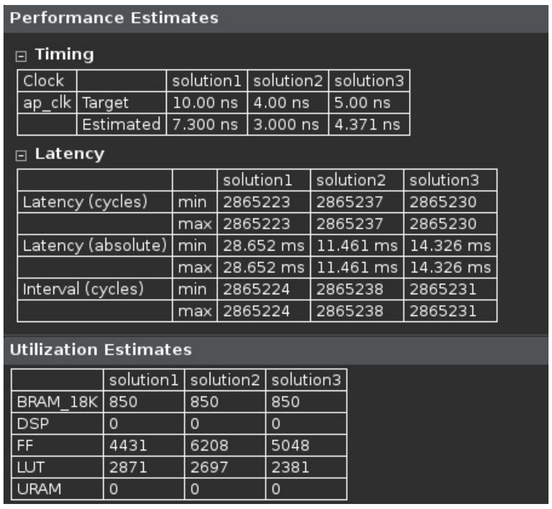
High-Level Synthesis (HLS) was performed using Xilinx Vitis HLS to accelerate the function. During the course of the project, several source code versions were developed. In this subsection, the main ideas behind the thought process of each implementation will be provided as well as the reasons behind the need of developing other versions.

Our first implementation aimed to achieve two primary objectives: reducing the function's execution time and minimizing the overhead caused by memory accesses. To achieve this, we started out by configuring the function to use AXI interfaces, with AXI4-Master (MAXI) for handling the matrix data (Min, Mout, and Kernel) and AXI4-Lite (AXILite) for scalar variables such as KernelSize and scalef. This interface design facilitated efficient data communication between the hardware accelerator and external memory.

To address memory access overhead, the matrices were explicitly transferred between memory and the hardware accelerator using memcpy operations, ensuring that the matrices were stored in local memory for faster access during computation. Additionally, loop unrolling and pipelining pragmas provided by Vitis HLS were applied to the nested loops within the function, based on the documentation provided by Xilinx in [1]. This optimization enabled loop threading, overlapping of operations and reduced overall latency.

The initial implementation presented significant improvements in computation time. However, storing entire matrices in local memory resulted in excessive usage of BRAM18K resources. As shown in Figure 2, even though function latency times are promising, Vitis HLS estimated the implementation would require 850 BRAM18K blocks, however, the board's available BRAM18K resource limit was 280. This resource overutilization meant that further optimizations to the function were needed in order to comply with hardware constraints.

To tackle the issue regarding this limitation, we implemented a new cycle that would divide the Min and Mout matrixes in 4 equal parts and compute the convolution one at a time, which


Performance Estimates

Timing

| Clock | | solution1 | solution2 | solution3 |
|---|---|---|---|---|
| ap_clk | Target | 10.00 ns | 4.00 ns | 5.00 ns |
| | Estimated | 7.300 ns | 3.000 ns | 4.371 ns |

Latency

| | | solution1 | solution2 | solution3 |
|---|---|---|---|---|
| Latency (cycles) | min | 2865223 | 2865237 | 2865230 |
| | max | 2865223 | 2865237 | 2865230 |
| Latency (absolute) | min | 28.652 ms | 11.461 ms | 14.326 ms |
| | max | 28.652 ms | 11.461 ms | 14.326 ms |
| Interval (cycles) | min | 2865224 | 2865238 | 2865231 |
| | max | 2865224 | 2865238 | 2865231 |

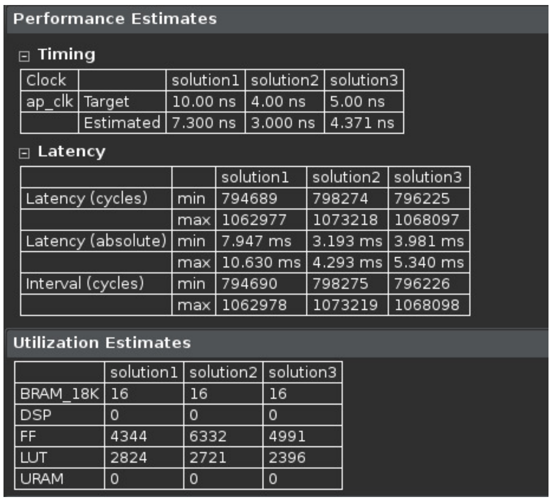
Utilization Estimates

| | solution1 | solution2 | solution3 |
|---|---|---|---|
| BRAM_18K | 850 | 850 | 850 |
| DSP | 0 | 0 | 0 |
| FF | 4431 | 6208 | 5048 |
| LUT | 2871 | 2697 | 2381 |
| URAM | 0 | 0 | 0 |

Figure 2: Implementation 1 simulation metrics for a 3x3 kernel

provides a 4x decrease in BRAM18k utilization. However, this implementation proved to be non-equivalent to the original function as the kernel matrix would not be convoluting the pixels in the edges of the partitioned matrixes.This discrepancy highlighted the need for further modifications to the function's logic.

To tackle both issues efficiently, the function was redesigned to process the Min matrix row by row instead of storing the entire matrix in local memory. A local buffer was used to hold only the rows needed for the current computation, updating its contents as the convolution progressed. The kernel matrix was also fully copied into a local buffer for fast access. This approach significantly reduced BRAM18k usage and latency while maintaining equivalent functionality, as shown in Figure 3. However, additional logic was introduced to handle edge cases and ensure the kernel properly convolves pixels at the matrix boundaries.


Performance Estimates

Timing

| Clock | | solution1 | solution2 | solution3 |
|---|---|---|---|---|
| ap_clk | Target | 10.00 ns | 4.00 ns | 5.00 ns |
| | Estimated | 7.300 ns | 3.000 ns | 4.371 ns |

Latency

| | | solution1 | solution2 | solution3 |
|---|---|---|---|---|
| Latency (cycles) | min | 794689 | 798274 | 796225 |
| | max | 1062977 | 1073218 | 1068097 |
| Latency (absolute) | min | 7.947 ms | 3.193 ms | 3.981 ms |
| | max | 10.630 ms | 4.293 ms | 5.340 ms |
| Interval (cycles) | min | 794690 | 798275 | 796226 |
| | max | 1062978 | 1073219 | 1068098 |

Utilization Estimates

| | solution1 | solution2 | solution3 |
|---|---|---|---|
| BRAM_18K | 16 | 16 | 16 |
| DSP | 0 | 0 | 0 |
| FF | 4344 | 6332 | 4991 |
| LUT | 2824 | 2721 | 2396 |
| URAM | 0 | 0 | 0 |

Figure 3: Implementation 2 simulation metrics for a 3x3 kernel

We then performed the RTL export of the design to generate the hardware description from our HLS code. Having completed this step in Vitis HLS, we are now ready to proceed with the hardware implementation in Vivado/Vitis.

## 2.3 Hardware/Software Interface and Integration

After completing the HLS and optimizations in the previous steps, the next critical phase of the project was the integration of the hardware accelerator with the software system.

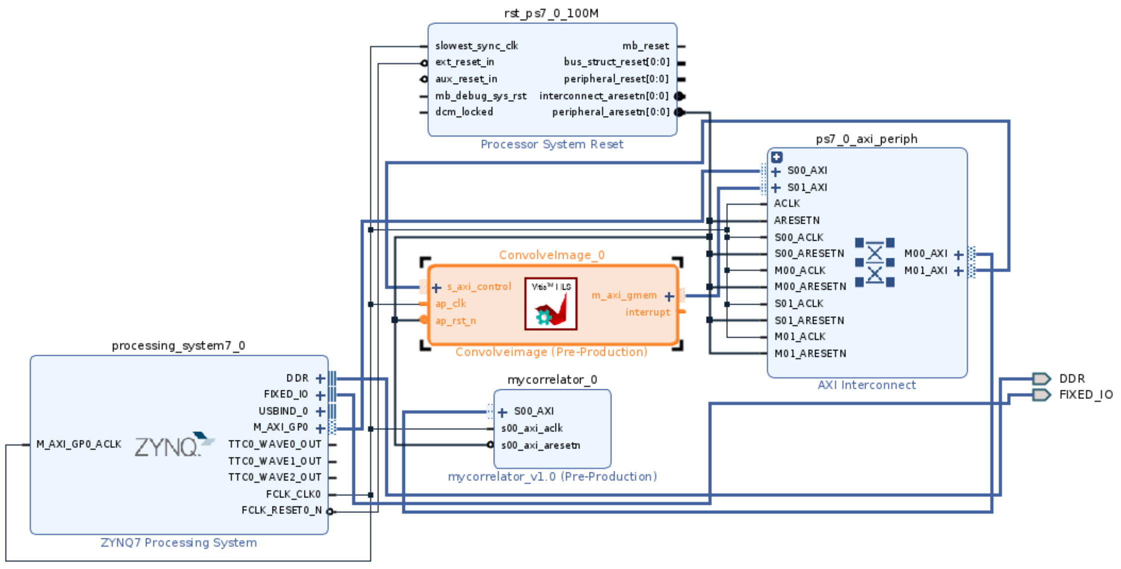


Figure 4: System's block diagram

### 2.3.1 Integration Process within the Zynq System

The integration of the hardware accelerator with the Zynq system was a multi-step process that required careful consideration of both the software and hardware components. Figure 4 illustrates the system architecture, where the integration between the processor and the hardware accelerator is shown.

The primary objective of the hardware/software integration was to ensure efficient communication between the software running on the processor in the Zynq Processing system and the hardware accelerator. To achieve this, we utilized AXI interfaces, which allow high-performance, low-latency communication between them.

- **AXI4-Master (MAXI):** these interfaces were used for transferring large data blocks, specifically the input image matrix Min and the kernel matrix. These matrices were transferred to the hardware for convolution in order to minimize the overhead due to memory access.

- **AXI4-Lite (AXILite):** these interfaces were employed for the scalar variables used to configure the kernel size and scaling factor for the convolution operation.

The choice of AXI interfaces ensures that both the software and hardware components could efficiently exchange data and control signals. The software side of the system is responsible for preparing the input data (image matrix and kernel), loading it into the appropriate memory regions, and triggering the hardware computation. Once the convolution operation is completed by the hardware, the results are sent back for further processing.

This setup allows for high-throughput data transfers. By using AXI interfaces, the system can ensure minimal delays in communication between the processor and the accelerator.

The FPGA fabric clock frequency was carefully optimized to maximize performance while ensuring stable operation. This was achieved by analyzing the worst negative slack (WNS) metric during the bitstream generation process. The clock frequency was adjusted to a value that provided a balance between achieving high throughput and meeting timing constraints.

Additionally, a clock counter was added to the system. This peripheral worked in a way such that its base address would get incremented at each clock cycle. This allowed us to get the execution time of each of the `ConvolveImage` functions implemented in hardware.

For functionality testing, we developed a function that would compare the output matrix of the original convolve function and the hardware implemented one, allowing us to assess if these matched or not, providing the necessary insights on code functionality.

To debug the system, we used printf statements strategically placed in the C code running on the processor. These statements allowed us to ensure that the hardware accelerator was being triggered as expected.

## 3 Final Analysis

The final analysis section presents the outcomes of both the initial implementation and a refined one based on feedback from the professors during the project presentation, showcasing the limitations, and system performance improvements achieved through hardware/software co-design.

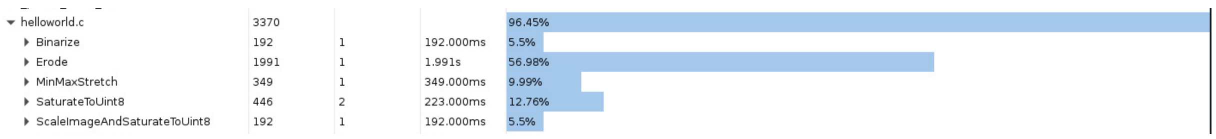
### 3.1 Pre-Presentation Results

The initial implementation of the system provided did not result in good performance metrics for the `ConvolveImage` function. When running in the integrated hardware / software system, the function exhibited execution times of 177 ms for a 3x3 kernel and 440 ms for a 5x5 kernel, resulting in 264,7 ms per Call (Time / Calls), which is much slower than the best optimization that took 37.7 ms per Call. These results highlight the inefficiencies in the initial hardware implementation, compared to the expected outcomes.

$$\text{Initial } \texttt{ConvolveImage} \text{ Speedup} = \frac{37.7}{\frac{177+177+440}{3}} \approx 0.14$$

As the `ConvolveImage` Speedup is lower than 1, the global speed-up for the initial implementation will also be lower than 1 resulting in a global slow-down. Based on the profiling results presented in figure 5, it was calculated as:

$$\text{Initial Global Speedup} = \frac{3 + 37.7 + 12 + 7 + 4 + 5}{192 + 1991 + 349 + 223 + 192 + \frac{177+177+440}{3}} \approx 0.021$$



```
▾ helloworld.c                 3370                         96.45%
  ▸ Binarize                   192     1    192.000ms       5.5%
  ▸ Erode                      1991    1    1.991s          56.98%
  ▸ MinMaxStretch              349     1    349.000ms       9.99%
  ▸ SaturateToUint8            446     2    223.000ms       12.76%
  ▸ ScaleImageAndSaturateToUint8 192   1    192.000ms       5.5%
```

Figure 5: Initial Profiling

These results indicate that the hardware acceleration of `ConvolveImage` did not effectively improve global system performance. Contributing factors include resource contention, unoptimized memory access patterns, and integration inefficiencies.

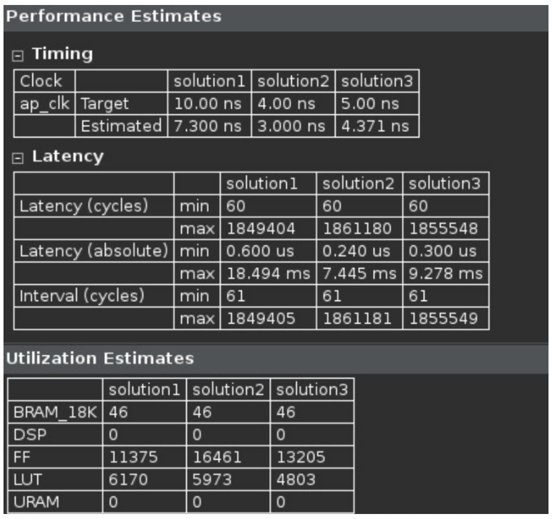
### 3.2 Post-Presentation Code Adjustments

During the presentation of our initial results, feedback from the professors helped us identify that the increase in execution time for the `ConvolveImage` function was due to the fact that the hardware was being generalized as it was implemented to handle any possible kernel size up to 4 bytes. This design choice meant that the circuit required significantly more cycles to execute, as it introduced unnecessary complexity to support all potential kernel sizes, even if they were not used.

#### 3.2.1 High-Level Synthesis

To address this issue, we modified the source code in Vitis HLS by introducing an if statement to set a fixed kernel size based on the value passed to the function.

Since only two kernel sizes were being used, the if statement ensures the kernel size is set to either 3 or 5, depending on the input. By restricting the design to these two specific sizes, we significantly reduced the circuit's complexity and improved execution time. Figure 6 shows the Vitis HLS estimations for the new implementation.



**Performance Estimates**

**Timing**

| Clock | | solution1 | solution2 | solution3 |
|---|---|---|---|---|
| ap_clk | Target | 10.00 ns | 4.00 ns | 5.00 ns |
| | Estimated | 7.300 ns | 3.000 ns | 4.371 ns |

**Latency**

| | | solution1 | solution2 | solution3 |
|---|---|---|---|---|
| Latency (cycles) | min | 60 | 60 | 60 |
| | max | 1849404 | 1861180 | 1855548 |
| Latency (absolute) | min | 0.600 us | 0.240 us | 0.300 us |
| | max | 18.494 ms | 7.445 ms | 9.278 ms |
| Interval (cycles) | min | 61 | 61 | 61 |
| | max | 1849405 | 1861181 | 1855549 |

**Utilization Estimates**

| | solution1 | solution2 | solution3 |
|---|---|---|---|
| BRAM_18K | 46 | 46 | 46 |
| DSP | 0 | 0 | 0 |
| FF | 11375 | 16461 | 13205 |
| LUT | 6170 | 5973 | 4803 |
| URAM | 0 | 0 | 0 |

Figure 6: Last implementation simulation metrics for a 3x3 kernel

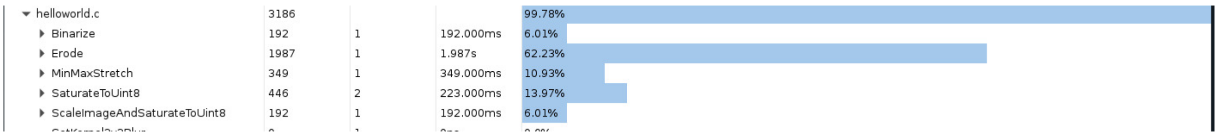
### 3.2.2 Refined Results After Code Adjustments

We then implemented the design on the FPGA and measured the execution time for the `ConvolveImage` function using the counter peripheral. The execution time was reduced to 17 ms for a 3x3 kernel and 23 ms for a 5x5 kernel. These results represented a significant reduction in runtime compared to the initial implementation and reflected the effectiveness of the adjustments.

$$\text{Final } \texttt{ConvolveImage} \text{ Speedup} = \frac{37.7}{\frac{17+17+23}{3}} \approx 1.98$$

The global speedup for the refined implementation was also calculated based on the profiling results presented in figure 7, as:

$$\text{Final Global Speedup} = \frac{3 + 37.7 + 12 + 7 + 4 + 5}{192 + 1987 + 349 + 223 + 192 + \frac{17+17+23}{3}} \approx 0.023$$



```
▾ helloworld.c                      3186                        99.78%
  ▸ Binarize                        192      1    192.000ms    6.01%
  ▸ Erode                           1987     1    1.987s       62.23%
  ▸ MinMaxStretch                   349      1    349.000ms    10.93%
  ▸ SaturateToUint8                 446      2    223.000ms    13.97%
  ▸ ScaleImageAndSaturateToUint8    192      1    192.000ms    6.01%
```

Figure 7: Final Profiling

While the speedup of the `ConvolveImage` function in the refined implementation demonstrated the potential of hardware acceleration, the global speedup remained low. This was attributed to several trade-offs and limitations in the current implementation. Resource contention on the FPGA resulted in slower execution times for other functions, which led to an imbalance in the overall system performance.

In summary, the project demonstrated the effectiveness of hardware/software co-design but also underscored the complexity of achieving holistic system performance improvements. Future efforts should focus on addressing the trade-offs and limitations to maximize the benefits of hardware acceleration.

## 4 Conclusion

This project successfully demonstrated the potential of hardware/software co-design to enhance the performance of compute-intensive image processing tasks. By integrating a hardware accelerator for the `ConvolveImage` function within a heterogeneous system on the Zynq platform, significant speedup was achieved in the targeted function, with execution times reduced to a fraction of their original values. The project's key outcomes include a deeper understanding of profiling, high-level synthesis, and efficient integration of hardware and software components using AXI interfaces. These outcomes validated the effectiveness of targeted hardware acceleration for computationally demanding operations.

Throughout the project, several valuable lessons were learned. Firstly, the importance of profiling and selecting appropriate functions for acceleration was highlighted, as not all operations benefit equally from hardware implementation. Secondly, the challenges of balancing resource utilization and performance were evident, particularly with respect to BRAM18K limitations and the need for careful memory access optimization. Finally, the significance of holistic system performance, rather than isolated function optimization, became clear, as the overall speedup remained constrained by resource contention and pipeline inefficiencies.

The project also faced certain limitations. Resource constraints on the FPGA limited the scalability of the hardware accelerator for larger kernels or higher resolution images. Additionally, the performance of other functions suffered due to bandwidth contention and an unoptimized software control flow, reducing the global speedup of the system.

Future directions for this work include further optimization of memory access patterns, exploration of parallel execution strategies for other functions, and better workload balancing across hardware and software components. Implementing techniques such as double buffering or hardware-based prefetching could mitigate bandwidth bottlenecks, while offloading additional tasks to hardware could distribute the computational load more evenly.

Overall, this project underscores the potential and challenges of hardware/software co-design and serves as a foundation for future endeavors aimed at achieving high-performance embedded systems.

## References

[1] Xilinx, *Vitis high-level-synthesis user guide*, Accessed: 2025-01-10, 2022. [Online]. Available: `https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf`.