Deliverable D3.1

1st Version of Robotics Simulation

2022-06-30

Work package:	WP3 Development of 1st Version of Applications	
Author(s):	Yannick Goumaz, Olivier Michel	CYB
Reviewer #1	FZJ	
Reviewer #2	EXA	
Dissemination Level	Public	
Nature	Report	

Confidentiality

This document may contain proprietary material of certain OPTIMA contractors. The commercial use of any information contained in this document may require a license from the proprietor of that information.

This project has received funding from the European High-Performance Computing Joint Undertaking Joint Undertaking (JU) under grant agreement No 955739. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Greece, Germany, Italy, Netherlands,Spain,Switzerland.

Executive Summary

In the ever-evolving field of robotics, simulations are becoming increasingly popular and useful for developers. Accelerating the execution of these simulations is particularly important as it would allow for faster and more accurate testing and thus more efficient prototyping. Previous research proposes solutions for deploying various simulators on cloud providers and high-performance computers to improve performance. However, most of them remain very superficial or focus only on very specific implementations. The initial work of Cyberbotics reported in this deliverable aims at addressing robotics simulations with complex control algorithms involving machine learning, which slow down computation considerably, by accelerating them in HW. In D5.1, we will further optimize the HW acceleration in order to provide the final version.

The robot simulations are created using Webots, an open-source robotics simulator developed by Cyberbotics. The hardware acceleration was performed progressively in two steps. The first one shows the achievable performance on the JUMAX HPC system for simple multilayer perceptrons. The second step is more concrete and complex, as it accelerates the controller of a simulated autonomous car implementing a convolutional neural network. The difference in the execution time with an equivalent implementation on a CPU is significant and shows the importance of FPGA-based HPC systems in the field of robot simulations and machine learning.

Table of Contents

List of Abbreviations

HPC	High Performance Computing
FPGA	Field Programmable Gate Array
DFE	DataFlow Engine
GPU	Graphics Processing Unit
CPU	Central Processing Unit
MLP	Multi-layer Perceptron
CNN	Convolutional Neural Network

1. Introduction

This work was mainly carried out by Cyberbotics Ltd. [1], a spin-off of the EPFL, specialized in the development of robot simulations. Cyberbotics has created an open source software called Webots [2] that simulates robots of all types, such as drones, industrial robots or autonomous cars. Webots allows developers to create a wide variety of environments and to add robots on which all kinds of sensors can be installed, like cameras, distance sensors, etc. Robot controllers can be programmed in many languages like C, C++, Python, Java or even using the Robot Operating System (ROS) [3].

Simulations play an important role in the robotics industry. Indeed, they allow developers to quickly and cheaply prototype applications involving different types of robots. This approach makes it easy to test an idea or a robotics project of varying complexity and to prove its feasibility and efficiency, before investing in a real robot for example. Also, in some cases, they can accelerate the tests by executing them at a speed higher than real time. They bring the advantage of test reproducibility: they can be indefinitely and instantly re-executed despite potential hardware damage or environmental changes during execution. As a result, an important advantage of robot simulations is the time saving. Indeed, the speed with which the environment can be modified, the ease with which the simulation can be restarted or the speed of its execution superior to real time implies a considerable time saving in the prototyping of robotics applications.

For some applications, it is necessary to further optimize the execution time of the simulation because, for example, the number of desired results is very large or the complexity of the algorithms implemented on the robots is very high.

Machine learning, and more precisely deep learning, are increasingly popular techniques to improve the intelligence of robots and the complexity of their applications. But one of their big disadvantages is the large amount of computations to be performed each time a result is desired. If these results have to be computed with a high frequency, they can become an important bottleneck during the simulation. This deliverable shows how robot simulations on Webots can be accelerated on the Jumax DFEs compared to the Jumax host CPU. Jumax is equipped with a MPC-X card, which has 8 DFEs of MAX5 generation, each containing a Virtex UltraScale VU9P FPGA from Xilinx. This chip contains 2,586,000 logic cells, 6,840 Digital Signal Processors for multiplications and 340Mb of memory blocks. Jumax also contains a double *AMD EPYC 7601 @ 2.7 GHz×64T* as the host CPU [4][5].

2. Definition of the task for hardware acceleration

The main goal of this deliverable is to accelerate a Webots robot simulation which uses deep learning in its controllers on a FPGA-based system and provide an initial performance comparison with a purely CPU execution, while the final optimized version will be provided in D5.1. The remaining chapters summarize the work performed on the Jumax machine, provided by Maxeler and the Juelich Research Center, to adapt and evaluate a deep-learning robot simulation in FPGAs.

The structure of the work includes two parts.
In the first part, we evaluate the multi-layer perceptrons (MLP) inference running on the Jumax CPU and on the Jumax DFEs. The simple neural network classifies the images of the MNIST dataset [6]. The goal of this step is to have a first global overview of the Jumax programming process and the estimated performance gains that can be achieved using FPGAs. No simulation is involved in this step.
The second part of the contribution aims at extending the performance evaluation to a more concrete application involving a simulation on Webots. The neural network structure is extended to Convolutional Neural Networks in order to obtain a higher complexity and a more interesting and complete implementation on Jumax. The robot application consists of a self-driving car using a front camera to follow a given track. Part of the simulation content, as well as the neural network structure, is taken from a reference paper [7]. The car controller code is optimized on CPU and FPGA to measure the impact on the simulation speed.

3. Implementation details

3.1 First Part: Multi-layer perceptron

The goal is to design a very basic neural network, composed of fully connected layers, capable of classifying the digits in the MNIST dataset. The dataset consists of images showing handwritten numbers. The chosen framework contains one hidden layer of 64 neurons, and one input and one output layer of size 784 and 10 respectively. The creation of a training framework in C++ allows the training of the network parameters without having to use external libraries like PyTorch or Tensorflow.

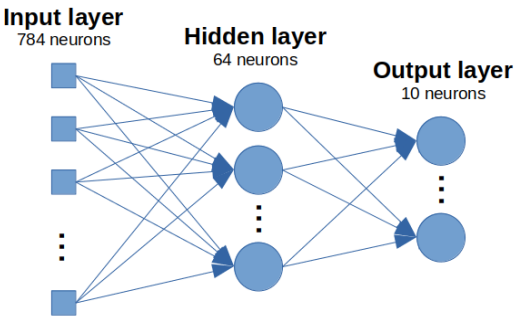

Figure 1: MLP neural network architecture used to classify MNIST digits.

The same network is then implemented in inference mode in C++ to evaluate the performance of the classification of thousands of images when running on the Jumax CPU. The inference is next ported on the JuMax DFEs in order to be accelerated in hardware. Both codes are optimized as much as possible to get a fair comparison between both implementations.

Concerning the CPU version, two main optimizations allow to significantly accelerate the code:

- Parameters trained with the framework are originally saved in floating point representation, but operations on fixed-point numbers are faster on CPU.

Therefore, for inference, the weights and inputs are converted to fixed-point representation.

- Multi-threaded programming aims at parallelizing the execution of an application to a given number of threads of the CPU. So, we have parallelized the execution of the inference by using one thread per image. For example, on a 8-thread CPU, the application could compute the classification of 8 images (depending on the CPU usage) in parallel.

As mentioned above, the inference code for classification is then translated into MaxJ (Maxeler programming language), for execution on Jumax DataFlow Engines (DFEs). Multiple optimization steps are performed in order to use the maximum resources of the FPGA chips and to obtain the fastest classification possible. Improvements include input and output parallelization using vectors and loop unrolling, usage of multiple DFEs and fixed-point representation. The chip frequency is also increased as much as possible. The overall structure of the accelerated code is as follows: the CPU host code is responsible for loading the input images and network parameters, as well as initializing the DFE through the .max file. The inputs and weights are transferred to the FPGA. The DFE application normalizes the inputs and performs the inference in hardware. Each layer computes multiple dot product operations in parallel to take advantage of the hardware parallelism offered in the FPGA. The results are transferred back to the CPU host code and the execution time is recorded.

3.2 Second Part: Autonomous car simulation

Literature on robotics applications involving convolutional neural networks (CNNs) includes several interesting applications implemented on Webots. For example, [7] exposes a self-driving car that computes its speed and steering from a front camera. This application is a good basis for a performance comparison between CPU and FPGA. The neural network is less complex than in most applications but it is still demanding in terms of number of operations. In addition, car simulation is a more and more popular domain on Webots. Moreover, this is a very illustrative example of the capabilities of neural networks in an everyday application.
The authors of the paper do not give access to their dataset. Furthermore, their research was focused on the Udacity car simulator, which is different from our Webots simulation environment. Therefore, the first step is to collect a ground truth dataset and use it to train the network described in the paper. A simulated car is driven on a training track using trajectory planning. Images taken by front cameras and the corresponding steering and speed target values are recorded. The network parameters are then trained using the PyTorch [8] framework with the images as input and the driving values as target output. The training track (**Figure 2**) and the neural network architecture (**Figure 3**) are shown below.

Once trained, the neural network is able to accurately drive the car on an unknown track without getting out of the road.

Now that the simulation environment is set up, the inference part can be translated into C++ and into dataflow computing in order to be accelerated on JuMax and evaluate the performance improvement of FPGAs over CPUs.

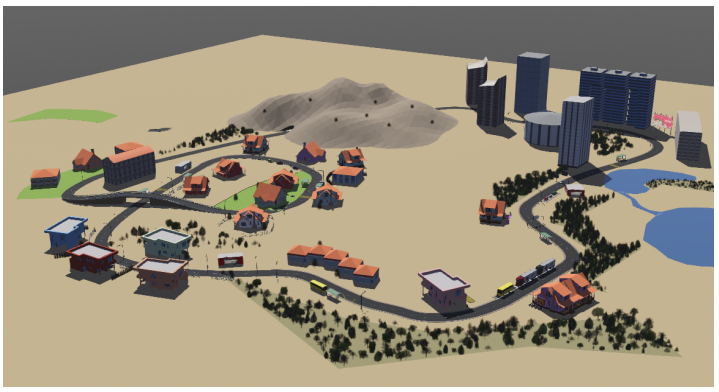

Figure 2: Webots world used to collect the dataset.

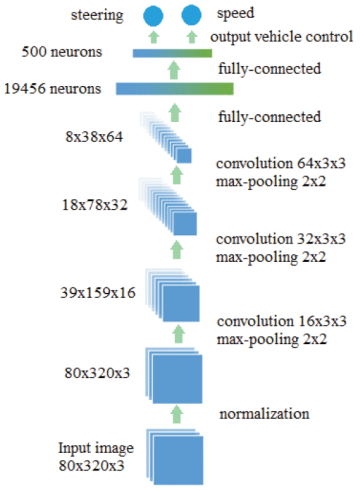

Figure 3: CNN architecture

The C++ part gets the same optimization steps as in the first part, namely fixed-point representation and multi-threading. In this application, only one image is going through the network at each timestep of the simulation, while the threads compute multiple channels of a convolutional layer in parallel. At each time step of the simulation, the front camera image is captured, normalized using the mean and standard deviation values computed from the training dataset and passed through the forward propagation to compute the velocity and direction of the car. These output values are further applied to the driving car.

On JuMax, parts of the controller code run on the host CPU, in C++, namely the normalization, the parameters loading, and the driving. In addition, the CPU is responsible for loading the .max file to the JuMax DFEs. The inference is executed on the FPGA, where the hardware parallelism allows us to increase the performance. Multiple dot products with different convolutional filters are performed in parallel at each clock cycle. Pooling and linear layers are also parallelized at the cost of more logic usage. The clock frequency is also pushed as high as possible to further decrease the execution time. Moreover, three strategies on the CPU side are explored to execute parts of the code asynchronously with the DFE. This is possible using the non-blocking DFE functions. These strategies are shown on **Figure 4**. Each of them shows two sets of tasks running in parallel, one executing two processes sequentially and the other executing a third process in parallel. The execution time of the strategies depends on each individual process runtime. Running the slower process alone in parallel with the two others allows to obtain the fastest strategy.

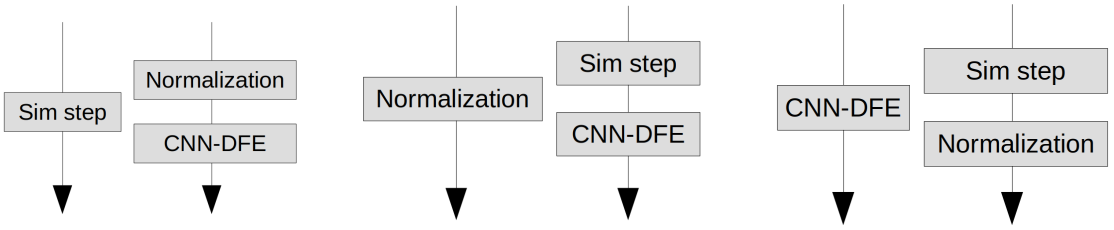

Figure 4: Three different strategies to parallelize controller operations.

The first strategy uses two CPU threads to execute the Webots simulation step in parallel of the normalization and FPGA inference. The second and third one take advantage of the non-blocking DFE calculation to execute FPGA and CPU codes in

parallel. It is not possible to combine both solutions (Webots multi-threading and non-blocking functions) to run all three processes in parallel because of the latency it would introduce. Indeed, applying the controls to the car and retrieving the camera images in multi-threading implies compromises. The results of the CNN would be available only two timesteps after the corresponding image is captured. The car would therefore drive less accurately on the road.

4. Numerical results on Jumax prototype

4.1 First Part: Multi-layer perceptron

Table 1 shows the execution time of the MLP network described in **Section 3.1** on the Jumax CPU. Different numbers of images are evaluated on the most optimized C++ code.

Number of images	Execution time [ms]
10'000	42
30'000	80
60'000	110

Table 1: Inference execution time for different number of images

Figure 5 shows the intermediate results obtained for each optimization step of the code for both the CPU execution (dashed lines) and the FPGA acceleration (solid lines) using different batch sizes of images.

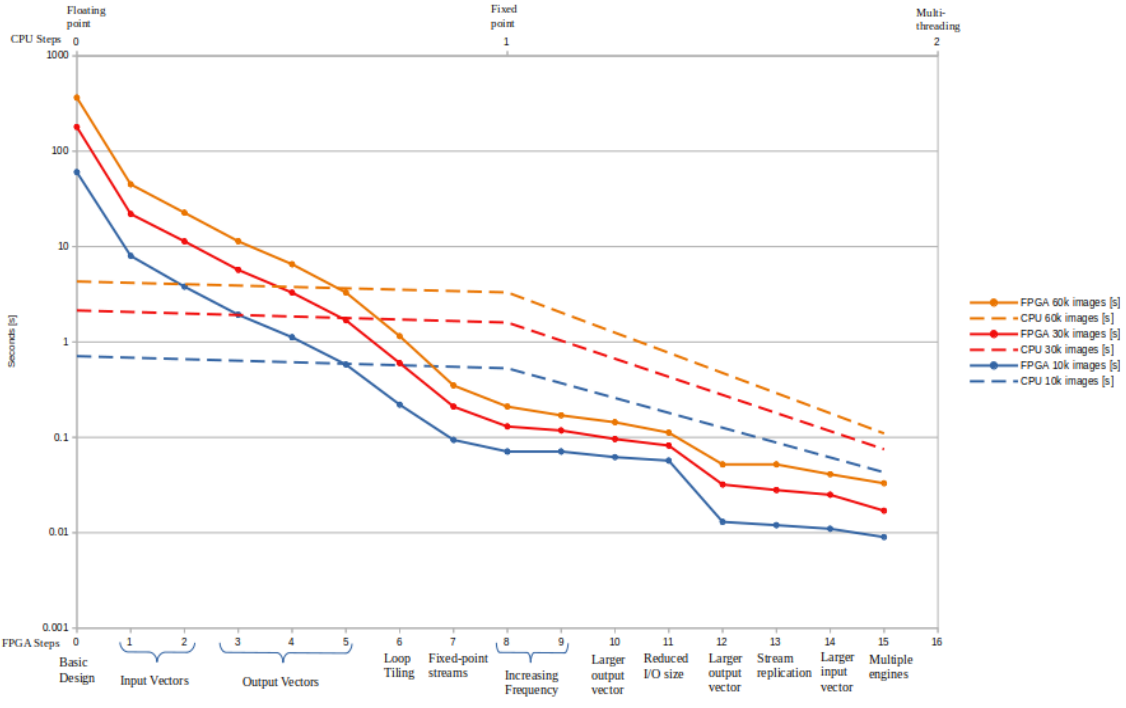

Figure 5: Execution times for each presented optimization on different MNIST images batch sizes with the corresponding CPU best result (dashed lines).

Results from **Table 1** are actually the last values of the dashed lines (best CPU performance). The solid lines show the results for the FPGA execution of the inference. The bottom horizontal axis represents the FPGA improvement steps (described in **Section 3.2**, they mainly consist of using more and more FPGA logic cells by taking advantage of different parallelization techniques of the neural network layers), while the top one shows the two accelerations of the CPU code. The final results (last value of each line) show 3.5 to 5 times faster FPGA execution than CPU execution, depending on the number of images.

4.2 Second Part: Autonomous car simulation

The trajectory planning algorithm is robust enough to drive the car perfectly on the training track, which allows the collection of an accurate dataset for the training of the CNN. The dataset is processed to keep a good distribution of the data and is augmented with image flipping and other techniques. The training phase is performed using CUDA on the PyTorch framework and takes less than 2 hours.
Authors of [9] give more detailed implementations of dataset augmentation to gain robustness. In the scope of this project, having a perfectly driving autonomous car is neither mandatory nor the primary goal. The car is capable of driving several laps on an unknown track without deviating from the right lane, which is more than enough for the purposes of the project, which is focused on the execution time and not on the robustness of the network.

The Webots rendering and the CNN inference are first executed on the Jumax CPU. The results are given in **Table 2**. The execution time represents the time it takes to evaluate 1 image through the network. The simulation speed gives the ratio between simulated time and real time for a simulation with a timestep of 30ms. The CNN part, which is executed in parallel with the simulation rendering, is the only bottleneck.

	Execution time	Simulation speed ratio
Optimized CNN	29.3ms	1.02

Table 2: Simulation performance on Jumax CPU.

At each timestep, it takes 29.3ms to evaluate the image given by the car camera. All other operations are executed in parallel or are negligible. Therefore, when the timestep of the simulation is set to 30ms, the simulation runs a little faster than real time (1.02x).

The FPGA implementation on the Jumax DFEs gives the following results (**Table 3**). The third strategy of **Figure 3** is used**.**

	Execution time	Simulation speed ratio
CNN on DFE	5.5ms	1.31

Table 3: Simulation performance on Jumax CPU with DFE for CNN inference.

This result seems strange. While the CNN inference time of each time step is now reduced by a factor of 5.3, the simulation speed ratio is almost the same as running the inference on the Jumax CPU. The reason is that the CNN is no longer the bottleneck. This is because there is no GPU on Jumax, and thus the simulation rendering is performed on the Jumax CPU. The camera rendering takes 22ms and becomes the new bottleneck. Thus the low ratio value of 1.31.
A GPU has been installed on Jumax since the collection of these results. However, it has not been installed on the same machine as the one that has access to the FPGAs. Because of this, Webots must be adapted to be able to run the car controller on a different machine than the one where Webots rendering takes place. This configuration will be tested in D5.1.
However, despite the lack of GPU, the expected result can still be accurately predicted. With a decent GPU, the rendering of the camera would take around 1ms instead of 22ms (measured on a local machine, it could be slightly different with the Jumax GPU). The CNN inference is becoming once again the main bottleneck and the simulation speed ratio is accelerated by a factor higher than 5 compared to the CPU-only version.
Table 4 and **Figure 6** compare the results obtained and the predicted ones. The simulation speed is increased up to 5.45.

	CNN time	Rendering time	Simulation speed ratio
No GPU	5.5ms	22ms	1.31
GPU	5.5ms	1ms	5.45

Table 4: Prediction of the performance if a GPU was implemented on Jumax.

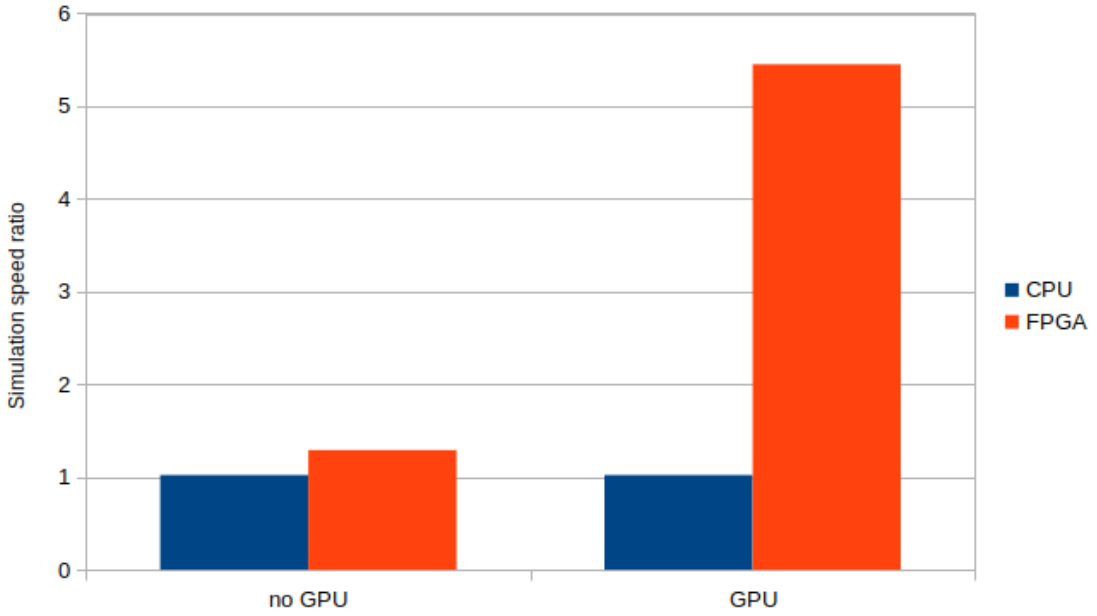

Figure 6: Comparison of simulation speed ratios between CPU and FPGA with and without GPU.

5. Concluding remarks

Using Jumax, our FPGA implementation of the convolutional neural network we tested in a real-world robotics simulation is more than five times faster than the equivalent optimized CPU version. As deep learning becomes omnipresent in robotics applications, it is important that the execution time bottleneck introduced by these algorithms can be overcome to allow for very fast execution of simulations. This acceleration can also be used to improve the accuracy of the simulation. The time step of the simulation can be reduced by a factor of five and still allow a real time simulation speed. Thus, the neural network could send new commands to the car five times more often. The FPGA's high performance can also benefit other robotics applications, like localization, image processing or more complex tasks. The contribution also highlights the importance of hybrid systems. The lack of a GPU on the Jumax HPC system at the time of these experiments slows down the simulation considerably. The addition of the GPU should allow us to observe the predicted results in D5.1.

6. References

[1] Cyberbotics Ltd., “Webots; OPEN SOURCE ROBOT SIMULATOR.” https://cyberbotics.com/. [Online; accessed 01.02.2022].

[2] Cyberbotics Ltd., “Webots Robot Simulator.” https://github.com/cyberbotics/ webots. [Online; accessed 01.02.2022].

[3] Open Robotics, “ROS - Robot Operating System.” https://www.ros.org/. [Online; accessed 01.02.2022].

[4] Xilinx, “UltraScale+ FPGAs; Product Tables and Product Selection Guide.” https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf. [Online; accessed 01.02.2022]

[5] Maxeler, “JuMax DFEs (MAX5 gen).” https://indico-jsc.fz-juelich.de/event/78/session/0/contribution/1/material/slides/0.pdf#page=43. [Online; accessed 01.02.2022]

[6] Y. LeCun, “THE MNIST DATABASE.” http://yann.lecun.com/exdb/mnist/. [Online; accessed 02.02.2022].

[7] M.-T. Duong, T.-D. Do, and M.-H. Le, “Navigating self-driving vehicles using convolutional neural network,” in 2018 4th International Conference on Green Technology and Sustainable Development (GTSD), pp. 607–610, 2018.

[8] Meta, “FROM RESEARCH TO PRODUCTION” https://pytorch.org/. [Online; accessed 22.04.2022].

[9] E. Forson, “Teaching Cars To Drive Using Deep Learning ˆa Steering Angle Prediction.” https://towardsdatascience.com/teaching-cars-to-drive-using-deep-learning-steering-angle-prediction-5773154608f2. [Online; accessed 03.02.2022].