RISCkant: A high performance, low power heterogeneous AI computing platform

Sebastian Fritsch Simon Klier Christian Künzle
Jan-Niklas Weghorn

Artificial Intelligence Research Lab at Hardenberg Gymnasium Fürth[1]

October 2019

[1] www.hardenberg-gymnasium.de

License

RISCkant repository: `https://gitlab.com/risckant`

Contents

Acknowledgements

We would like to acknowledge our indebtedness and render our warmest thanks to our physics teacher, Dr. Rüdiger Janner, who made this work possible by introducing FPGAs to us. His friendly guidance and expert advice have been invaluable throughout all stages of the work. Without our long and extensive conversations this project would not have been possible.

We also wish to express our gratitude and utmost respect to the great administration of our school. Without them always providing excellent working conditions this project would not have been achievable. Their support opened many doors for us. Thank you all so much.

The development of this paper has also benefited from comments and suggestions made by Toni Erdenkäufer. He taught us basic concepts of programming and fostered our dedication in computer sciences. We take this opportunity to thank him.

The persons with the greatest indirect contribution to this work are our parents, who have taught every single of us a spirit of discovery and research. We want to thank them, as well as the rest of our families, for their constant encouragement.

This document has been written during our senior grades at Hardenberg-Gymnasium Fürth. We would like to thank Andreas Landgraf and Xilinx, Inc. for providing financial and material support. Special thanks be due to Christian Ernst, the chairman of our school's friends association, for taking care of our research group.

Fürth, Oktober 2019

Sebastian Fritsch
Simon Klier
Christian Künzle
Jan-Niklas Weghorn

1 Overview

It's hardware that makes a machine fast. It's software that makes a fast machine slow.

(Craig Bruce)

To many people, the inner workings of a computer are a deep mystery. Even a simple question like the difference between software and hardware might be a tremendous challenge for those unfamiliar with the art of computer architectures. A common metaphor for the distinction which might be given by a IT expert to one could be:

> The human body with it's complex structure of organs, bones, muscles, nervous system, blood vessels, cells and many more parts of the most advanced and almost magical nature is entirely useless without the brain to give it guidance, intelligent thought and wisdom. The structure of a computer is similar. The hardware is the body, it is the vessel in which software lives. Software is the instructions, literally, which guide the hardware to unfold there true potential.

This quite beautiful metaphor is very well suited to explain the design of RISCkant. The two parts of the architecture, software and hardware are usually designed by two completely separated entities. Therefore they rarely are built with the same design

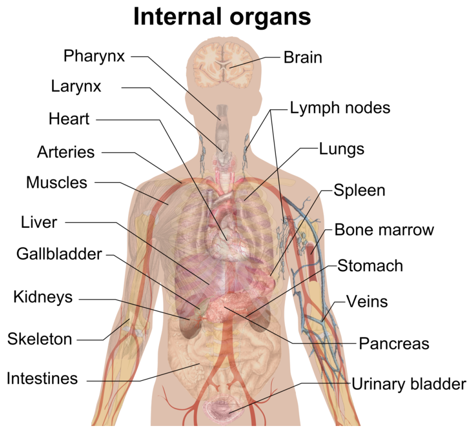


Figure 1: The human body

goals. RISCkant however was designed by one team with one goal in mind: accelerate AIs.

Henceforth the software and hardware could be designed so it would work in unison. Another method we were able to use is specialisation, which is also employed in the human body. It means to separate a generalized system in to multiple system which each are able to perform a subset of the original systems features. This allows each system to be more agile than before as it can now focus one doing one task well instead of many tasks badly. A example of this employed in the human body is the muscular apparatus. The common skeletal muscle consists of two types of muscle fibers, the fast and slow twitching types.

This allows the body to perform tasks like locomotion both at a fast and a slow speed. In the hardware of RISCkant specialisation is used accelerate AI by developing a special chip which is only used to compute matrix multiplications. Before RISCkant the CPU, a chip that is able to do many different types of computation, had to do all processing, including the matrix multiplication. In this way we applied specialisation to accelerate neural networks.

1.1 Hardware

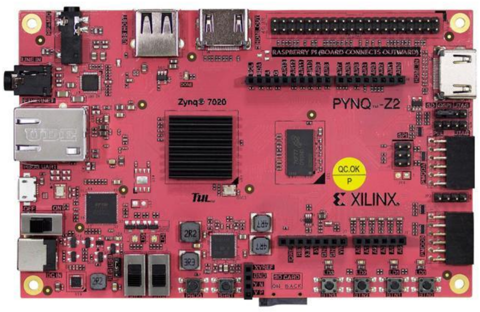


Figure 2: PYNQ-Z2 – a FPGA board suitable for RISCkant

The engine of the RISCkant AI platform is a Zynq-7000 SoC by Xilinx. The Zynq-7000 SoC consists out of a dual-core Cortex-A9 processing system and an Artix-7 or Kintex-7 FPGA.

The FPGA enables RISCkant to offload computationally intensive tasks which are commonly used in neural networks like matrix multiplication to the FPGA. This enables us to develop high performance fixed function hardware which runs on the FPGA and can be used to increase the performance of the RISCkant platform dramatically.

Figure 3: Darknet – an advanced AI framework

1.2 Software

The software of the RISCkant platform is centered around the AI framework "Darknet"[2] by Joseph Redmon. It is a high performance AI framework which is commonly used for state-of-the-art image recognition and classification.

The framework is written in C and therefore well suited for the application in low power embedded systems like RISCkant. It has been used for many innovations in the computer vision space like the YOLO image classification network.

This network is also utilized in our demonstrators and enables them to achieve excellent performance in a wide range of image recognition tasks. Thanks to the modular and scalable architecture of Darknet it is easy to integrate RISCkant into many tasks that require the use of cutting-edge neural networks.

[2] https://pjreddie.com/darknet/

1.3 Demonstrator

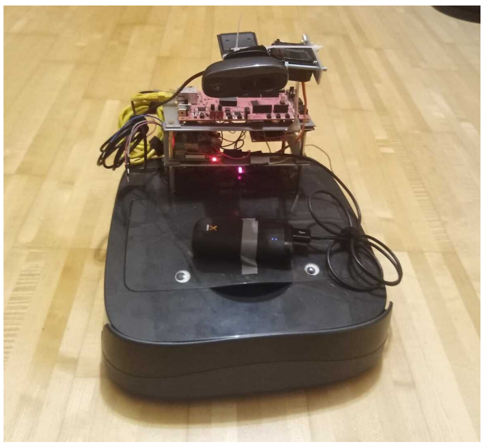


Figure 4: nicolAI – "*Could these eyes lie?*"

In order to demonstrate RISCkant hands-on, we decided to build a cigarette-butt collecting robot. This robot, which is called "nicolAI", is able to navigate autonomously through the room and to detect pieces of litter. By doing so, the energy efficiency of RISCkant is shown in a decent way.

2 Achievements

The aimless suffers his fate
— the aimful shapes it.
(Immanuel Kant)

With RISCkant we set out to develop an AI platform that would enable new innovative ideas in the field of low-power devices to be evolved. This section of the project documentation highlights the achievements and innovations of the RISCkant project and showcases its strengths and weaknesses.

2.1 Efficiency

The efficiency of RISCkant was one of the most important characteristics we considered during the development. This decision was made because there are many interesting applications of AI in low power systems. It was our goal for RISCkant to be able to detect and classify an image. We achieved this with using under 2 W. In comparison the NVIDIA Jetson development board has a power draw of 15 W this means that RISCkant only uses

$$\frac{2\,\mathrm{W}}{15\,\mathrm{W}} \approx 13.3\,\% \tag{1}$$

of the power of the NVIDIA system.

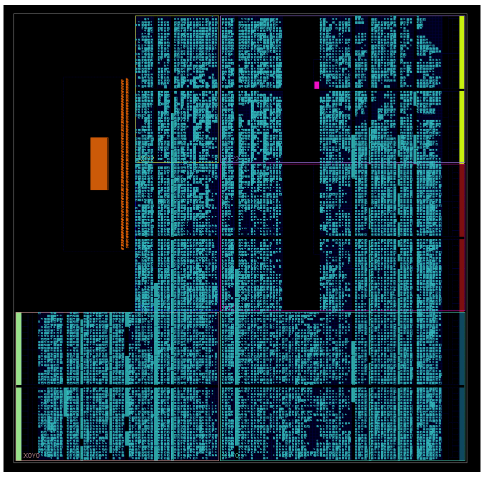
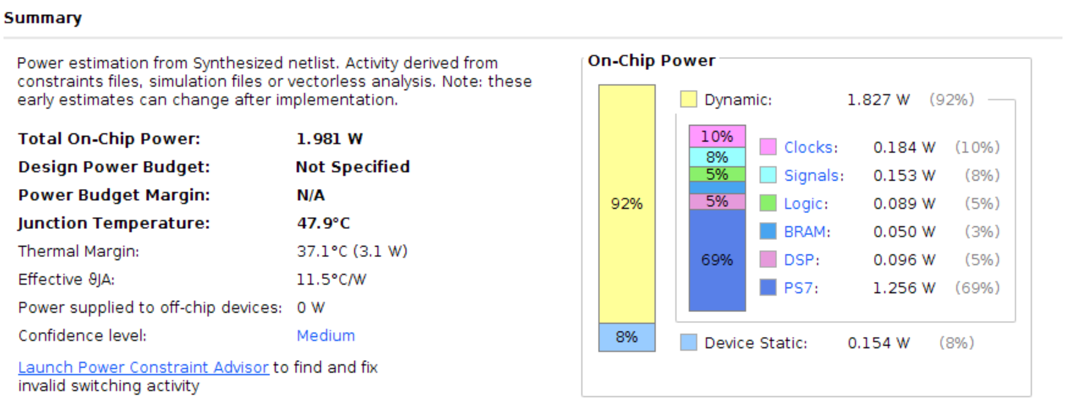


Figure 6: The RISCkant chip – a very complex design!



Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**1.981 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**47.9°C**
Thermal Margin:	37.1°C (3.1 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

	Dynamic:	1.827 W	(92%)
	Clocks:	0.184 W	(10%)
	Signals:	0.153 W	(8%)
	Logic:	0.089 W	(5%)
	BRAM:	0.050 W	(3%)
	DSP:	0.096 W	(5%)
	PS7:	1.256 W	(69%)
	Device Static:	0.154 W	(8%)

Figure 5: The power envelope of the RISCkant chip

2.2 Facts and figures

The hardware of RISCkant is built on a FPGA and a generic CPU. To get most out of the FPGA, RISCkant is designed to be scalable. This enables us to utilize almost all of the FPGA which can be seen in figures 5 and 7. This also leads to a very complex design. Solely for the FPGA design over 20000 lines of synthesized VHDL code were generated. The software of the RISCkant project which runs on the processor of the SoC envelops over 50000 lines of code.

While being able to be used on nearly every FPGA chip, RISCkant offers a wide range of applications: Due to its tremendously low power consumption, this framework facilitates the utilization of machine learning technologies in mobile environments.

$$\frac{2\,\mathrm{W}}{200\,\mathrm{W}} = 1\,\% \tag{2}$$

Chart 8 illustrates RISCkant's advantage in electricity usage: By consuming only approximately one percent of a GPU's power and yet processing the same rate of frames, RISCkant overrules common GPUs in image recognition.

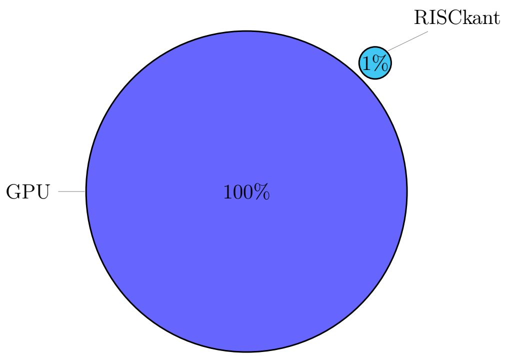


Figure 8: RISCkant power consumption compared to a generic GPU

2.3 Opening object classification to new applications

Reducing the power needed significantly, RISCkant introduces a new era for high performance image recognition: From now on it is possible to use object classification software in non-stationary, narrow-space surroundings.

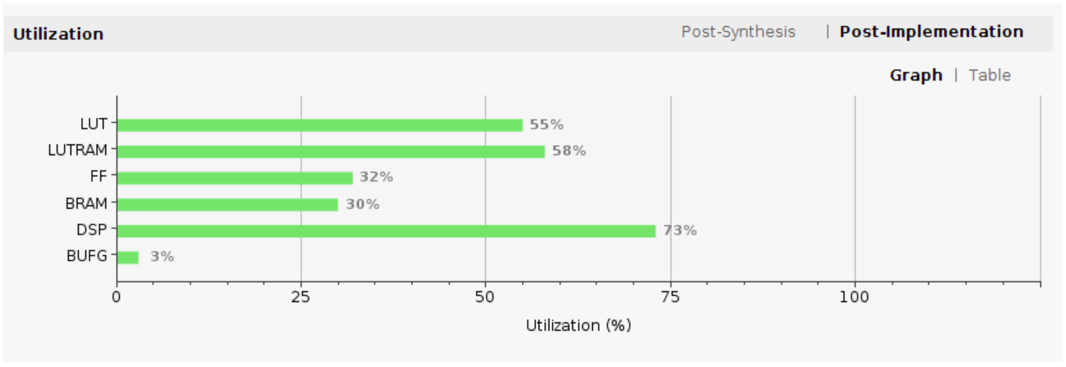


Figure 7: RISCkant's scalability helps it getting the most performance out of the FPGA

RISCkant is the key technology to develop a whole new field of apparatuses, which includes diminutive autonomous robots as well as energy-efficient, driverless and battery-driven vehicles. A specific example is given in section 6.

2.4 Performance

In general, one can state, that a image recognition unit equipped with RISCkant is able to process the video stream at least fifty, but in average a hundred times and more faster than concurring systems while using the same amount of energy.

This means, an application's performance can be increased significantly without changing other parameters like the power supply or the thermal budget. As a result, RISCkant can easily be (retro-)fitted into existing machinery.

2.5 Universality in applications

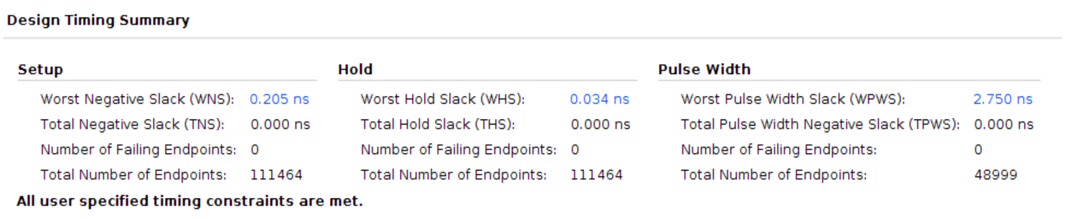
RISCkant is a technology that can be utilized for various purposes. Consequently, its use is not limited to a specific field of technology.

2.6 Weaknesses

Whilst creating a new fundament for research in artificial intelligence, some progresses in technology have to be awaited: GPU and CPU units are mass-produced articles. In comparison, FPGA chips are even fabricated in large scales, but in vastly lower quantities. This means, to take the best advantages out of the RISCkant technology, major-sized FPGAs have to become more affordable.

Another point that has to be mentioned is the funding of RISCkant: As we developed the whole codebase in our free time as a hobby, we are dependent on the generous support of our sponsors. Without them, we would not have been able to go that far. Nevertheless, we are wholeheartedly convinced, that our decision to make our software freely available for everyone was right.

Backed by a great open source community around the earth, we are conceited to have spent many days and nights to help and foster the world of image recognition.



```
Design Timing Summary

 Setup                                       Hold                                        Pulse Width
   Worst Negative Slack (WNS):   0.205 ns      Worst Hold Slack (WHS):       0.034 ns      Worst Pulse Width Slack (WPWS):           2.750 ns
   Total Negative Slack (TNS):   0.000 ns      Total Hold Slack (THS):       0.000 ns      Total Pulse Width Negative Slack (TPWS):  0.000 ns
   Number of Failing Endpoints:  0             Number of Failing Endpoints:  0             Number of Failing Endpoints:              0
   Total Number of Endpoints:    111464        Total Number of Endpoints:    111464        Total Number of Endpoints:                48999
 All user specified timing constraints are met.
```

Figure 9: RISCkant meets all timing constraints

3 Basic concepts

We can build a much brighter future where humans are relieved of menial work using AI capabilities.

(Andrew Ng)

The vast field of Machine learning, especially convolutional neural networks, has sprawled many new innovative concepts and notions. This section provides a brief overview over important aspects of neural networks. These are applied in many AI platforms to provide the ability to create, train and use neural networks. In RISCkant these are implemented both in hardware and in software, which results in a tightly integrated, highly advanced and efficient AI accelerator architecture.

3.1 CNN

CNNs are neural networks inspired by biological processes, trying to recreate the recognition mechanism using a layered structure. The main type of layers are convolutional, subsampling and pooling layers. Convolutional layers repeatedly apply a filter to the previous layers output. Subsampling layers reduce the size of the layer by reducing multiple inputs (usually four) to one output. This reduces the complexity of the following layers. These operations can be performed by multiplying matrices. The shape of the CNNs input tensor usually is

$$(number\ of\ images) \times (image\ width) \times \\ \times (image\ height) \times (image\ depth) \tag{3}$$

3.1.1 Convolutional layer

The convolutional layers are the key part of CNNs. The layer-input interaction occurs as follows: The layer operates a matrix-multiplication with a trainable kernel (also called filter) and a portion of the image. This kernel is applied on each portion of the image and a smaller resulting image matrix is created.

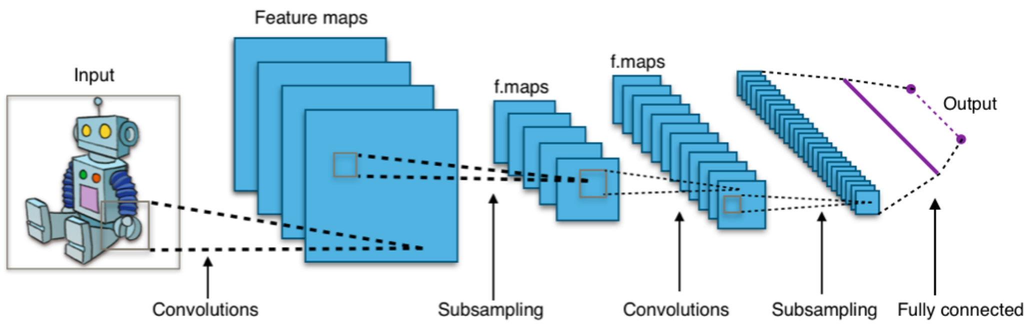


Figure 10: Architecture of a CNN for image classification. Image source: [6]

3.1.2 Pooling layer

A pooling layer is used for reducing the size of the CNN matrix. *Max pooling* is used most often. This means, that the highest value in the area is selected and the others fall out. It is important to reduce the size of the CNN, so that computations get less and speed increases.

3.1.3 ReLU layer

ReLU stands for Rectified Linear Unit and describes a function as follows:

$$f(x) = max(0, x) \tag{4}$$

This gets applied on every value in the layer.

3.1.4 Fully connected layer

A fully connected Layer is used in the last step of the CNN in order to return a precise result. All neurons get connected together.

3.2 Block matrices

A matrix can be partitioned into smaller sub matrices or block matrices.

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & \cdots & M_{1n} \\ M_{21} & M_{22} & \cdots & M_{2n} \\ d\vdots & \vdots & \ddots & \vdots \\ M_{m1} & M_{m2} & \cdots & M_{mn} \end{bmatrix} \tag{5}$$

The RISCkant hardware platform uses matrices 64×64 in size.

3.2.1 Block matrix multiplication

A, B and C shall be block matrices. In this example they are 2×2 matrices. The product

$$C = AB \tag{6}$$

can be defined using the partitioned matrices.

$$C = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \tag{7}$$

$$\begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \end{bmatrix} \tag{8}$$

This can of course be extended to larger matrices with more block matrices.

3.2.2 Block matrix addition

A, B and C shall be block matrices. In this example they are 2×2 matrices. The sum

$$C = A + B \tag{9}$$

can be defined using the partitioned matrices.

$$C = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \quad (10)$$

$$= \begin{bmatrix} A_{11} + B_{11} & A_{12} + B_{12} \\ A_{21} + B_{21} & A_{22} + B_{22} \end{bmatrix} \quad (11)$$

3.3 GEMM

GEMM, or GEneral Matrix Multiplication, describes a mathematical operation which multiplies two input matrices A and B to get an output matrix C.

$$C = \alpha AB + \beta C \quad (12)$$

where α is one and C is an all-zeros matrix. This Operation is especially important in applications such as 3D Graphics, Fully Connected Layers or Convolutional Layers. Because of the heavy utilization of this operation and the great magnitude of the input matrices, the GEMM algorithm is often implemented recursively by subverting A and B into block matrices.

3.3.1 Fully Connected Layers

In a Fully Connected Layer each value in the input layer is multiplied with the corresponding weight and sums the results. Given k input values and n neurons, with a predefined set of weights for the input layer, there are n output values, for each neuron. This is calculated using the dot product of the input values and the weights.

3.3.2 Convolutional Layers

In contrast to the Fully Connected Layer, the convolutional operation produces its output by applying a number of kernels across the input. A Convolutional Layer has a two dimensional image as input with an arbitrary number of channels for each pixel. Each kernel is another three-dimensional array of integers, with the depth being the same as the input image, but with a much smaller width and height. To produce a result, the kernel is applied to a grid across the input image. Each result of the multiplication of the input value and the weights at a specific point are summed and constitute the output for this point. When implementing this functionality on a computer the 3D image first must be converted into a 2D array, that can be treated like a matrix. This is accomplished by patching the 3D matrix and serializing it using an operation called IMG2COL. Now, doing the same with the kernel matrix, results in a second matrix for multiplication, which then, can be used for matrix multiplication using GEMM.

Because of the redundancy and specialization of this operation it can easily be implemented in the Hardware. Those GEMM cores complete functional units, which are the foundation of RISCkant's scalability. Increasing the size of the board, one can easily place more GEMM cores on the chip design further increasing performance.

3.4 FPGA

FPGAs are programmable matrices of logic blocks. Theoretically every logical circuit can be programmed on a FPGA (even processors, etc.). In reality often the size of the FPGA is a limiting factor in implementing highly complex circuits. RISCkant uses a FPGA to implement General Matrix Multiplication in hardware. This allows the operation to be performed in a much faster way than on a processor or a comparable GPU, whilst reducing the power consumption. Applications of FPGAs are in aerospace and defense, ASIC Prototyping, video & image processing and many more [5].

4 Hardware

Computers themselves, and software yet to be developed, will revolutionize the way we learn.
(Steve Jobs)

The previous sections of this documentation were dedicated to explaining the basic concepts and ideas behind RISCkant. In contrast, this section in going to focus on the concrete development and implementation of RISCkant. RISCkant is based on the fundamental idea of distributing the computation load onto two different types of processors: On the one hand a generic CPU and, on the other hand a highly specialized FPGA chip. According to this, FPGA evaluation boards are tailor-made for development on the RISCkant platform. Meaning that the image classification is done heterogeneously, it is possible to execute general tasks on the CPU, which is backed by RISCkant's GEMM core calculating matrix multiplications.

4.1 Choosing a suitable FPGA

Due to the heterogeneous nature of RISCkant we needed a microprocessor and a FPGA as the hardware platform of RISCkant. After some research and help by a expert from Xilinx we decided to use a Xilinx Zynq-7000 SoC. This SoC combines a CPU and a FPGA in one package and provides a fast interconnect between them. In addition there are a lot of documentation and tools provided by Xilinx.

4.2 Development of the GEMM Core

After choosing a hardware platform we wanted to start to design our GEMM core. Xilinx provides traditional design tools in which we could design our chip with VHDL and also high level synthesis (HLS) tools. These tools can convert design written in a subset of C to be converted into a hardware description language (HDL) such as VHDL or Verilog, albeit the resulting design is at lower speed and efficiency. We decided to use the HLS tools instead of the HDL tools as it would allow for faster iteration times which was very important to us as there were many variables in our design such as block matrix size which we needed to optimize.

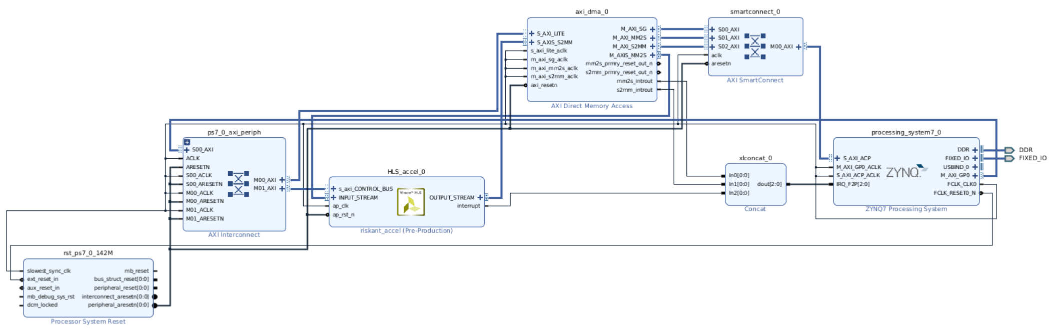


Figure 11: Block design of our hardware

4.3 Development in HLS tools

These tools made implementing the core logic which multiplies matrices quite easy as we just needed to implement them in C (code snippet 1). After this was finished we needed to connect the matrix multiplier to the ARM CPU which is part of the Zynq SoC. This proved to be quite a challenge both in hardware and in software. In the end we came up with the block design seen in Figure 11. The data can be sent from a DMA buffer in the CPU over AXI to a AXI DMA controller. This DMA controller then exposes a transmit and receive AXI Stream interface. These are then used to connect to the GEMM accelerator. We added a receive and transmit state to our GEMM core as can be seen in Figure 13 and code snippet 2. In the code snippet you can see that firstly we receive three matrices from the input stream, then compute the multiplication and then transmit them over the output stream. Then we finished the block design by setting up and connecting the clocks, reset signals and interrupts.

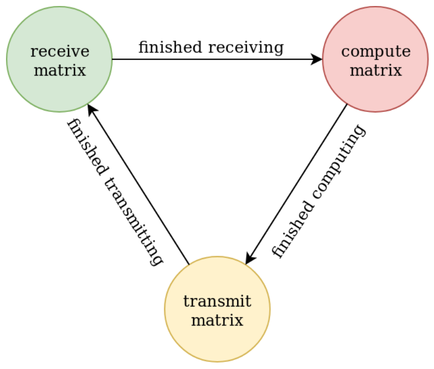
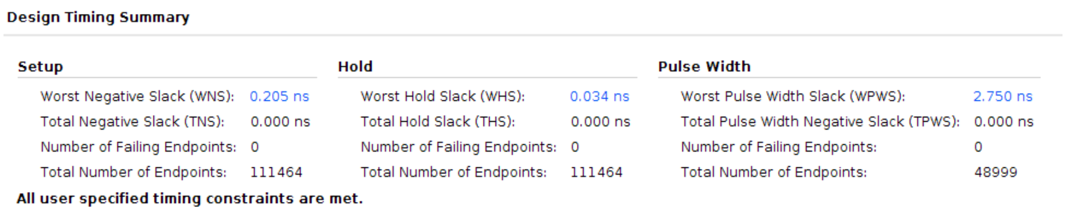


Figure 13: State diagram of our GEMM core

4.4 Development in HDL tools

Originally we planned on replacing the HLS code with HDL code after the prototyping phase of the project because it usually performs better. However after we reevaluated our plans after we got the prototype to work faster then we planned in our project plan we decided that optimising our HLS code and the software would have greater benefits than working on a HDL accelerator because the HLS was surprisingly effective and efficient. In addition one of the main bottlenecks in the design is the interface between the FPGA and the CPU. A HDL accelerator could in no way speed this up. In conclusion we decided on sticking with our accelerator and optimised it instead of rewriting it.



Design Timing Summary

Setup		**Hold**		**Pulse Width**	
Worst Negative Slack (WNS):	0.205 ns	Worst Hold Slack (WHS):	0.034 ns	Worst Pulse Width Slack (WPWS):	2.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	111464	Total Number of Endpoints:	111464	Total Number of Endpoints:	48999

All user specified timing constraints are met.

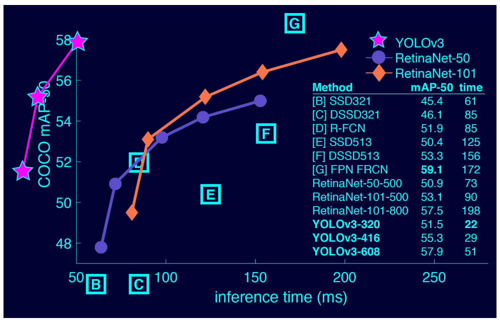
Figure 12: RISCkant meets all timing constraints

5 Software

I was lucky to be involved and get to contribute to something that was important, which is empowering people with software.

(Bill Gates)

RISCkant emerges out of “Darknet”, an open-source framework written in C and CUDA for the usage in Computer Vision and Artificial Intelligence. In Darknet one uses YOLOv3 (standing for “You only look once”) as a “state-of-the-art, real-time detection system” [3]. Figure 14 shows a comparison of different



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

Figure 14: YOLOv3 in comparison to other detectors Image source: [2]

detector systems. The x-axis indicates the time needed for classification of an image and the y-axis indicates the performance scored with a picture of the COCO dataset. The performance is measured with the mean Average Precision (mAP). The reason why RetinaNet and YOLOv3 are lines in the diagram, while other detectors are only single points is the scalability of YOLOv3 and RetinaNet. One can easily change the size of the model and with YOLOv3 there isn't even any retraining needed! [4]

The main difference between orthodox image detectors and YOLO is their different approach on the image. Whilst conventional detectors apply their model on different sections on the pictures and try to detect objects by this, YOLO applies on model on the whole image at once. The networks divides the image into section by itself and predicts bounding boxes and probabilities. As a result of this, the number of required calculations decreases and speed increases tremendously.

One can either use a pre-trained model, or train one by itself. We are going to demonstrate both options with our demonstrators.

In order to accelerate Darknet, one has to find out, which operation is time-consuming and required to be accelerated. For this we modified the `gemm()` function in the `gemm.c` file as seen in listing **??**. An experimental run on a notebook returned a total execution time of 22.382 seconds and a GEMM-execution time of 21.78 seconds.

$$\frac{21.781\,006\,\mathrm{s}}{22.382\,10\,\mathrm{s}} \approx 97.314\,\% \tag{13}$$

Concluding we have to speed up the GEMM-operation. We achieved this by leveraging the RISCkant hardware.

5.1 Software on the Zynq

There are two methods of running software on the Zynq SoC. On the one hand it is possible to write bare metal code for the ARM CPU which allows very low level access to the SoC and therefore makes it very easy to use the GEMM accelerator. On the other hand it is possible to use Linux which has many advanced features but requires a lot of work to set up and it is necessary to write your own kernel driver to access the FPGA.

We decided on using Linux as we needed features like networking or the file system. Furthermore this made porting software to the

Zynq quite easy, however getting it to work was quite a lot of work. In addition the processes of setting up Linux was sparsely documented but we managed to figure it out by a lot of researching.

5.2 Building Linux for the Zynq

Luckily for us Xilinx already provides tools to build Linux which are called "Petalinux". After installing Petalinux which required some patching to get it to work building Linux was quite easy. However being able to access the FPGA from Linux was not.

After a lot of research we figured out how it we could implement it. Firstly we needed to adjust Linux Device Tree. The Device Tree is a object that tells the Linux Kernel at what physical addresses it can find the FPGA (to be exact: the AXI port of the FPGA). The Petalinux tools should be able to configure it automatically but for us they were configured wrongly so we fixed it.

Secondly we needed a Linux Kernel Driver which could interface to our accelerator. This is needed because only the Linux Kernel can access physical memory, the driver therefore needs to map the physical addresses of our accelerator in to virtual user address space. Writing a Kernel Driver is quite complicated but there were existing drivers we could adapt and a presentation by Xilinx which explained some concepts.

5.3 Speeding up Darknet with the FPGA

Because we are using Linux it was quite easy to port Darknet to the Zynq. Now with only using the CPU we can observe Darknet taking 225 seconds to classify and detect one image. To integrate the FPGA we firstly need to be able to upload the matrices to the FPGA, start the computing and after it has completed download the result. Listing 4 shows how we achieved this. This makes it possible to quickly calculate the multiplication of 64×64 (the size of the GEMM core) matrices.

To make it possible to multiply arbitrary sized matrices we use block matrices. As the rest of Darknet uses regular matrices we needed to write a function that creates block matrices and one that would delete them (Listing 5 and 6). With this we could integrate GEMM into Darknet (Listing 7). With this Darknet now only takes 15 seconds to process one image. This represents a speed-up of 1500 %.

$$\frac{225\,\mathrm{s}}{15\,\mathrm{s}} = 1500\,\% \tag{14}$$

6 Demonstrator

Genius is one percent inspiration,
ninety-nine percent perspiration.
(Thomas Alva Edison)

Our project got two stages of demonstration:

1. Demonstration of the acceleration

2. Practical use in robotics

6.1 Acceleration

The strength of acceleration highly depends on the board used, and which processor we reference to. For instance a by comparison cheap board like the ZYNQ XC7Z020-1CLG400C[3] has a 650 MHz dual-core Cortex-A9 processor and the calculation time of darknet is: 230.34s. Meanwhile an Intel Silver N5000 Processor (quad-core, 2.7 GHz) takes a calculation time of about 2.76 seconds.

Same approach applies for the FPGA. A by comparison cheap FPGA like the Artix-7 FPGA in a Zynq-7000 environment takes a

[3] `https://www.xilinx.com/support/documentation/data_sheets/ds187-XC7Z010-XC7Z020-Data-Sheet.pdf`

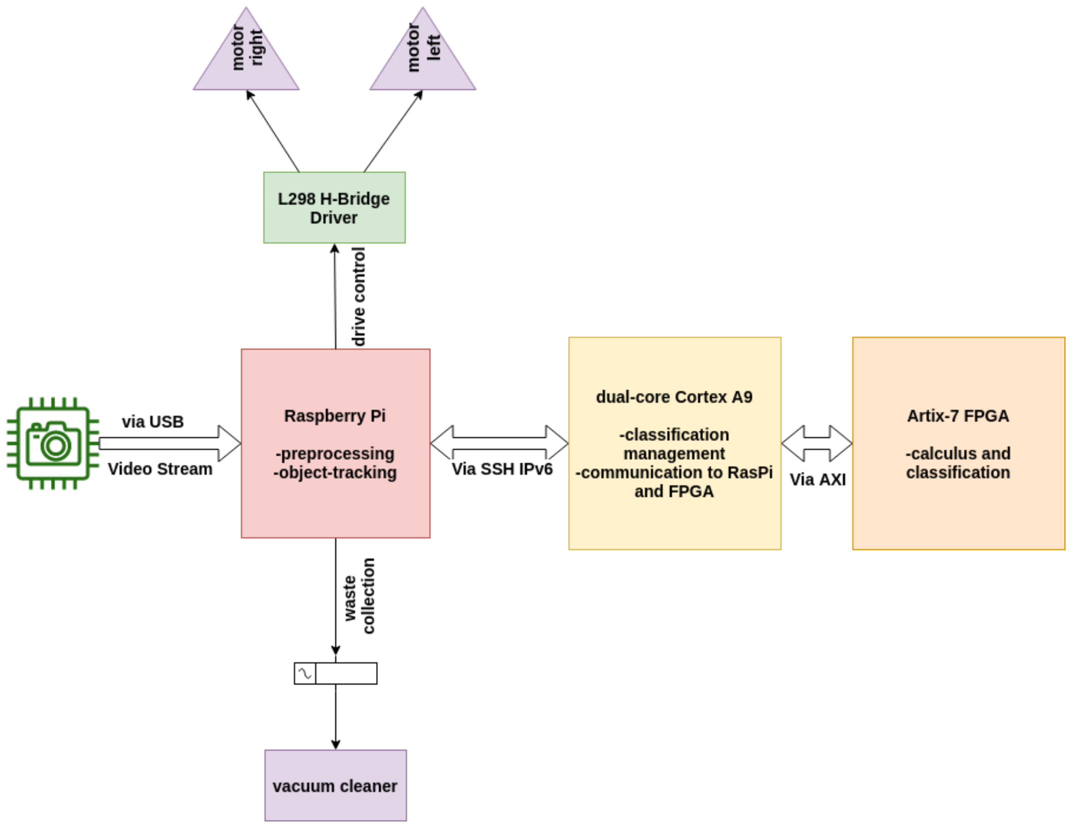


Figure 15: object-orientated and semi-functional structure diagram of nicolAI

calculation time of about 7.9 s. Meanwhile a more expensive and high-end board like the "Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit" with a Zynq-UltraScale+ environment takes a calculation time of about 1.6 s.

So it isn't possible to give absolute numbers, but in general we can say that RISCkant provides an acceleration factor of 30-50x, whilst reducing the power consumption by 3x.

Some may ask: "Why do you compare your accelerator to CPUs instead of GPUs, which are more efficient in the field of AI?". The answer is that the are nearly no low-power GPUs (except of smartphone GPUs – but those aren't suitable for AI) available for constrained environments. In contrast there are many RISC-CPUs like ARM-Cortex etc., which are used on evaluation boards and are suitable for a benchmark comparison.

6.2 Robotics

To provide an appropriate showcase experience, we decided to modify a robot vacuum cleaner. Specifically we used the "Vileda VR 101"[4], a cheap but sufficient model for our project. This decision has proven to work well, because we have not had to build a undercarriage nor a cleaning mechanism, since it also wouldn't have been expedient with regard to the project. The result of this process is called nicolAI, which is short for neural interactive convolutional open-source low-energy Artificial Intelligence experience. Figure 15 shows the structure of our robot. There are three main processing components:

[4]https://www.vileda.com/media/wysiwyg/VR101_Manual_Pt-1_INT_230616.pdf

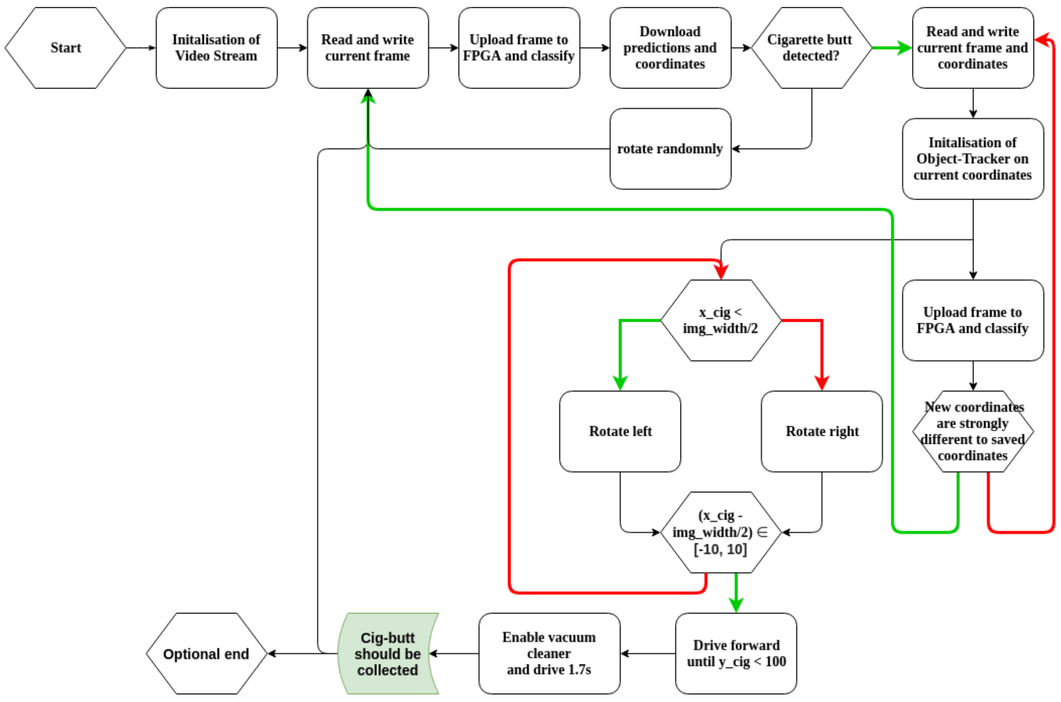

Start
Initalisation of Video Stream
Read and write current frame
Upload frame to FPGA and classify
Download predictions and coordinates
Cigarette butt detected?
Read and write current frame and coordinates
rotate randomnly
Initalisation of Object-Tracker on current coordinates
x_cig < img_width/2
Upload frame to FPGA and classify
Rotate left
Rotate right
New coordinates are strongly different to saved coordinates
(x_cig - img_width/2) ∈ [-10, 10]
Optional end
Cig-butt should be collected
Enable vacuum cleaner and drive 1.7s
Drive forward until y_cig < 100

Figure 16: Flowchart of the robot

1. the FPGA, an Xilinx Artix-7
2. the on-board Zynq-processor, a dual-core Cortex A
3. the Raspberry Pi, for object-tracking and main coordination

Basic information about FPGAs may be found under subsection 3.4. Artix-7 FPGA are characterized through a high performance-per-watt ratio and "are best value for a variety of cost and power-sensitive applications including software-defined radio, machine vision cameras, and low-end wireless backhaul." [1]

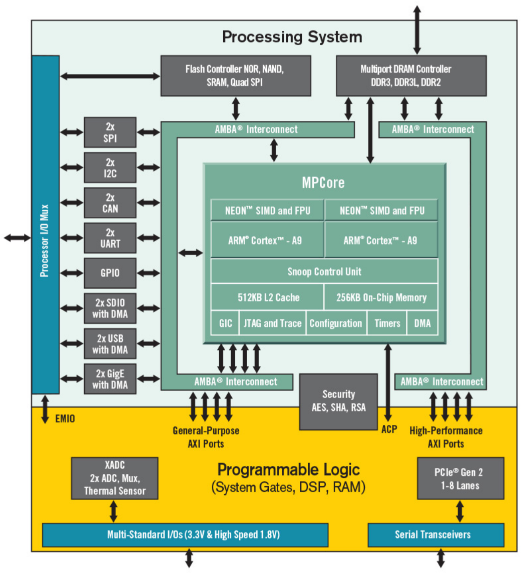


Figure 17: block-diagram of a Zynq-7000 SoC [7]

Zynq is a SoC-family developed by Xilinx in behalf of the combination of powerful ARM-based processors and the hardware opportunities given by a FPGA. Our project uses a Zynq XC7Z020-1CLG400C, a member of the Zynq 7000 family. Zynq-7000 devices are equipped with dual-core ARM Cortex-A9 processors integrated with 28 nm Artix-7 based programmable logic for maximum design flexibility.

The Raspberry Pi is a single-board general-purpose computer. He is energy-efficient and suitable for a energy-constrained environment. We use the Raspberry Pi for object-tracking, motor control, camera pre-processing and task-coordination.

Further components are the camera, the motor-driver, etc. As a camera, we use a Logitech C270 HD-Webcam[5]. The motor driver is a L298n dual full-bridge driver[6]. Furthermore the vacuum cleaner itself is connected to the Raspberry Pi over a relay, because the power of the Pi's GPIO pins wouldn't be sufficient in power.

The program flow of our robot, is displayed in Figure 16. The special feature of our implementation is the symbiosis of Artificial Intelligence and Object Tracking. The AI calculates the initialisation coordinates of our object and the object tracker moves the robot in order to drive to the object. This saves a lot of calculation time, because no real-time AI is needed. The AI is used for tracker verification and initialisation. By this, the robot is able to operate in real-time, although being restricted through its power- and space-constrained environment.

The program flow is implemented in Python using a Raspberry Pi. The code for this can be found in our GitLab repository `nicolAI`[7]. The program flow itself can be found in the file `tracker.py`. The other files are tests, emulations or self-written modules and libraries. More information on each file can be found in the header or in the repo `README`.

[5]`https://www.logitech.com/de-de/product/hd-webcam-c270`
[6]`https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf`
[7]`https://gitlab.com/risckant/nicolai`

7 Code snippets

Programming is the art of algorithm design and the craft of debugging errant code.

(Ellen Ullman)

7.1 Hardware

```
// implements matrix multiplication in hardware
void mmult_hw(float a[DIM][DIM], float b[DIM][DIM], float C[DIM][DIM],
    float out[DIM][DIM])
{
    // can be adjusted for better performance but needs more space on the
     FPGA; this should be ideal for a Zynq-Z7020
    int const FACTOR = DIM/4;
    #pragma HLS INLINE
    #pragma HLS array_partition variable=a block factor=FACTOR dim=2
    #pragma HLS array_partition variable=b block factor=FACTOR dim=1

    // computing A * B + C
    L1:for (int ia = 0; ia < DIM; ++ia)
    {
        L2:for (int ib = 0; ib < DIM; ++ib)
        {
            #pragma HLS PIPELINE II=1
            T sum = C[ia][ib];
            L3:for (int id = 0; id < DIM; ++id)
                sum += a[ia][id] * b[id][ib];
            out[ia][ib] = sum;
        }
    }
    return;
}
```

Listing 1: Matrix multiplication in hardware

```
// pop float from axi stream
float pop_stream(ap_axiu <AXI_SIZE,U,TI,TD> const &e)
{
#pragma HLS INLINE

    // convert from AXI float
    union
    {
        unsigned int ival;
        float oval;
    } converter;
    converter.ival = e.data;
    float ret = converter.oval;

    // receive axi vals
    volatile ap_uint<sizeof(T)> strb = e.strb;
    volatile ap_uint<sizeof(T)> keep = e.keep;
    volatile ap_uint<U> user = e.user;
    volatile ap_uint<1> last = e.last;
    volatile ap_uint<TI> id = e.id;
    volatile ap_uint<TD> dest = e.dest;

    return ret;
}

ap_axiu<AXI_SIZE,U,TI,TD> push_stream(float const &v1, bool last = false)
{
#pragma HLS INLINE
    ap_axiu<AXI_SIZE,U,TI,TD> e;

    // convert from float to AXI
    union
    {
        float ival;
        unsigned int oval;
    } converter;
    converter.ival = v1;
    e.data = converter.oval;

    // set axi stream vals, this is needed for receiving from an AXI
   Stream
    e.strb = -1;
    e.keep = 15;
    e.user = 0;
    e.last = last ? 1 : 0;
    e.id = 0;
    e.dest = 0;
    return e;
}

void axi_mmult_hw (
    AXI_VAL in_stream[3*SIZE],
    AXI_VAL out_stream[SIZE])
{

#pragma HLS INLINE

    float a[DIM][DIM];
    float b[DIM][DIM];
    float c[DIM][DIM];
    float out[DIM][DIM];

    // stream in first matrix
    for(int i = 0; i < DIM; i++)
    {
        for(int j = 0; j < DIM; j++)
        {
#pragma HLS PIPELINE II=1
            int k = i * DIM + j;
            a[i][j] = pop_stream(in_stream[k]);
        }
    }

    // stream in second matrix
    for(int i = 0; i < DIM; i++)
    {
        for(int j = 0; j < DIM; j++)
        {
#pragma HLS PIPELINE II=1
            int k = i * DIM + j + SIZE;
            b[i][j] = pop_stream(in_stream[k]);
        }
    }

    // stream in third matrix
    for(int i = 0; i < DIM; i++)
    {
        for(int j = 0; j < DIM; j++)
        {
#pragma HLS PIPELINE II=1
            int k = i * DIM + j + 2 * SIZE;
            c[i][j] = pop_stream(in_stream[k]);
        }
    }

    // do HW multiplication
    mmult_hw(a, b, c, out);

    // stream out result matrix
    for(int i = 0; i < DIM; i++)
    {
        for(int j = 0; j < DIM; j++)
        {
            #pragma HLS PIPELINE II=1
            int k = i * DIM + j;
            out_stream[k] = push_stream(out[i][j], k == (SIZE-1));
        }
    }

    return;
}
```

Listing 2: Recieving and transmitting matrices from AXI Stream

7.2 Software

```c
#include <time.h>
float total_seconds = 0;
void gemm(int TA, int TB, int M, int N, int K, float ALPHA,
        float *A, int lda,
        float *B, int ldb,
        float BETA,
        float *C, int ldc)
{
    clock_t t;
    t = clock();
    gemm_cpu(TA, TB, M, N, K, ALPHA, A, lda, B, ldb, BETA, C, ldc);
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC; // in seconds
    total_seconds += time_taken; // add to total time;
    printf("gemm() took %f seconds to execute\n", time_taken);
    printf("the total time ist %f seconds\n, total_seconds);
}
```

Listing 3: Testing matrix multiplication in darknet

```c
void gemm_hw(float* A, float* B, float* C)
{
    // start the accelerator
    XHls_accel_Start(&accel);

    // copy matrix to dma buffers
    memcpy(a, A, BLOCK_SIZE);
    memcpy(b, B, BLOCK_SIZE);
    memcpy(c, C, BLOCK_SIZE);

    // get tx and rx channels
    int tx_chan = axidma_get_dma_tx(dma_dev)->data[0];
    int rx_chan = axidma_get_dma_rx(dma_dev)->data[0];

    // transfer matrices to FPGA
    axidma_oneway_transfer(dma_dev, tx_chan, (void*)a, BLOCK_SIZE,true);
    axidma_oneway_transfer(dma_dev, tx_chan, (void*)b, BLOCK_SIZE,true);
    axidma_oneway_transfer(dma_dev, tx_chan, (void*)c, BLOCK_SIZE,true);

    // recieve result from FPGA
    axidma_oneway_transfer(dma_dev, rx_chan, (void*)out, BLOCK_SIZE,true)
    ;

    // copy from dma buffer to output
    memcpy(C, out, BLOCK_SIZE);
}
```

Listing 4: Running matrix multiplication on the hardware

```c
float** create_block_matrix(int M, int N, float* A)
{
    // row and column count of block matrix
    int rows = ceil((float)M / BLOCK);
    int cols = ceil((float)N / BLOCK);

    // allocate block matrix
    float** A_part = malloc(rows * cols * sizeof(float*));

    // iterate over block matrix
    for (int i = 0; i < rows * cols; i++)
    {
        // allocate individual matrices
        A_part[i] = malloc(BLOCK_SIZE);
        float* part = A_part[i];

        // calculate current row and column
        int col = i % cols;
        int row = i / cols;

        // iterate over current matrix
        for (int j = 0; j < BLOCK; j++)
        {
            for (int k = 0; k < BLOCK; k++)
            {
                // if outside of original matrix: pad with 0
                if (col * BLOCK + k >= N || row * BLOCK + j >= M)
                {
                    part[j * BLOCK + k] = 0.0f;
                }
                else // if inside: extract value from original matrix and put into the block
                {
                    part[j * BLOCK + k] = A[col * BLOCK + k + (row * BLOCK + j) * N];
                }
            }
        }
    }

    return A_part;
}
```

Listing 5: Creating a block matrix from a matrix

```c
void release_block_matrix(int M, int N, float** A)
{
    // calculate number of blocks in matrix
    int num_blocks = ceil((float)M / BLOCK) * ceil((float)N / BLOCK);
    for (int i = 0; i < num_blocks; i++)
    {
        // free block
        free(A[i]);
    }
    // free matrix
    free(A);
}
```

Listing 6: Deleting a block matrix

```c
void gemm_block(int M, int N, int K, float** A, float** B, float** C)
{
    // calculate size of individual matrices
    int lm = (int)ceil((float)M / BLOCK);
    int ln = (int)ceil((float)N / BLOCK);
    int lk = (int)ceil((float)K / BLOCK);

    for (int i = 0; i < lm; ++i)
    {
        for (int j = 0; j < ln; ++j)
        {
            for (int k = 0; k < lk; ++k)
            {
                // calculate indices
                int a = i * lk + k;
                int b = k * ln + j;
                int c = i * ln + j;

                // multiply the matrices
                gemm_hw(A[a], B[b], C[c]);
            }
        }
        }
}
```

Listing 7: Multiplying block matrices

8 Glossary

AI	artificial intelligence
ARM	advanced RISC machines
ASIC	application-specific integrated circuits
AXI	advanced extensible interface
CNN	convolutional neural network
CPU	central processing unit
CV	computer vision
DMA	direct memory access
FPGA	field-programmable gate array
GEMM	general matrix multiplication
GPU	graphics processing unit
SoC	System on Chip
UAV	unmanned aerial vehicle
UGV	unmanned ground vehicle
VHDL	very high speed integrated circuit hardware description language
OOTL	human out of the loop control
HRI	human-robot interaction
HDL	hardware description language
HLS	high level synthesis
YOLO	you only look once
nicolAI	neural interactive convolutional open-source low-energy Artificial Intelligence

References

[1] *Artix-7 FPGA Family.* URL: `https://www.xilinx.com/products/silicon-devices/fpga/artix-7.html`.

[2] Joseph Redmon. *Comparison to other detectors.* 2018. URL: `https://pjreddie.com/darknet/yolo/`.

[3] Joseph Chet Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger." In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[4] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018). arXiv: `1804.02767`. URL: `http://arxiv.org/abs/1804.02767`.

[5] Juan José Rodríguez-Andina, María Valdés, and María Jesus Moure. "Advanced Features and Industrial Applications of FPGAs - A Review". In: *IEEE Transactions on Industrial Informatics* 11 (Aug. 2015). DOI: `10.1109/TII.2015.2431223`.

[6] *Typical CNN architecture.* 2015. URL: `https://www.ziiai.com/blog/619`.

[7] *zynq-mp-core-dual.* 2017. URL: `https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png`.